EFFICIENT PERSONALIZED LEARNING TO RANK FROM IMPLICIT FEEDBACK FOR TIME-SENSITIVE RECOMMENDATIONS

by

Arif Murat Yağcı

B.S., Physics Engineering, Istanbul Technical University, 2000

M.S., Computer Engineering, Bahçeşehir University, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University
2019

EFFICIENT PERSONALIZED LEARNING TO RANK FROM IMPLICIT FEEDBACK FOR TIME-SENSITIVE RECOMMENDATIONS

APPROVED BY:

Prof. Sadık Fikret Gürgen .....
(Thesis Supervisor)

Assist. Prof. Tevfik Aytekin .....
(Thesis Co-supervisor)

Assoc. Prof. Albert Ali Salah .....

Prof. Yücel Saygın .....

Prof. Tunga Güngör .....

Prof. Mehmet Alper Tunga .....

DATE OF APPROVAL: 29.05.2019

ACKNOWLEDGEMENTS

With gratitude to my advisor, Prof. Sadık Fikret Gürgen, for believing in me and supervising my research. He has always been guiding through technical and non-technical problems with all his knowledge, patience, and kindness. I have learned a lot from him.

With gratitude to my co-advisor, Prof. Tevfik Aytekin, for advising me while struggling in a sea of ideas. Numerous discussions with him have been both fruitful and enjoyable. His constructive criticism as well as his experience at the intersection of academia and industry have been helpful.

With gratitude to Prof. Albert Ali Salah for his support, constructiveness, and thought-provoking questions during the progress of this thesis. I also learned a lot from his lectures. With gratitude to Prof. Yücel Saygın for his support and constructive criticism during the progress of this thesis. With gratitude to Prof. Tunga Güngör and Prof. Mehmet Alper Tunga for participating in the defense jury and their constructive criticism. With gratitude to Prof. Bert Arnrich, Prof. Devrim Ünay, and Prof. Arzucan Özgür for their support in various phases of my Ph.D. studies.

The school of computer engineering at Boğaziçi University is full of smart and enthusiastic researchers. With gratitude to all my professors, colleagues, and friends for providing insights into advanced topics and their support.

With gratitude to my family for always being there. This thesis is dedicated to them and to all the great people who inspired me in life.

The work in this thesis received partial incentives from Boğaziçi University BAP fund grant no. 17A01P6 and Selçuk Halaç fund.

ABSTRACT

EFFICIENT PERSONALIZED LEARNING TO RANK FROM IMPLICIT FEEDBACK FOR TIME-SENSITIVE RECOMMENDATIONS

This thesis focuses on the problems at the intersection of time-sensitive recommendations, implicit user feedback, and learning to rank. Major challenges for achieving time sensitivity are distinguished, the importance of handling implicit feedback is emphasized, and an overview of learning to rank methods is presented with an emphasis on the models that can learn from implicit feedback for time-sensitive recommendations. Subsequently, novel and improved personalized learning to rank methods are proposed to handle large-scale implicit feedback datasets and streams as well as to defeat the different challenges for achieving time-sensitive recommendations. These proposals comprise: (i) Mining the user feedback stream for collaborative filtering and the SASCF algorithm, (ii) Parallel personalized pairwise learning to rank and the PLtR family of algorithms, (iii) Improving the efficiency of top-N predictions from matrix factorization models and the MMFNN meta-algorithm, (iv) Learning intention in user sessions and the BRF family of algorithms, and finally (v) Timely push recommendations in a cold start setting and a hybrid learning to rank approach. Theoretical as well as extensive empirical analyses of the proposed methods on real-life data show significant performance and trade-off improvements with respect to ranking accuracy, adaptivity, diversity, efficiency, and scalability.

ÖZET

ZAMAN DUYARLI ÖNERİLER İÇİN ÖRTÜK GERİ BİLDİRİMDEN VERİMLİ ŞEKİLDE KİŞİSELLEŞTİRİLMİŞ SIRALAMA ÖĞRENME

Bu tez zaman duyarlı öneriler, örtük kullanıcı geri bildirimi ve sıralama öğrenme kesişimindeki problemlere odaklanmaktadır. Zaman duyarlılığı sağlamak için başlıca zorluklara dikkat çekilmekte, örtük geri bildirimi işleyebilmenin önemi belirtilmekte ve zaman duyarlı öneriler için örtük geri bildirimden öğrenebilen modeller vurgulanarak sıralama öğrenme yöntemlerine genel bir bakış sunulmaktadır. Daha sonra ise büyük ölçekli örtük geri bildirim veri kümeleri ve akışlarını işleyebilen ve zaman duyarlı öneri oluşturmadaki zorlukları aşabilen yeni ve geliştirilmiş kişiselleştirilmiş sıralama öğrenme yöntemleri önerilmektedir. Bu öneriler şunları kapsamaktadır: (i) İşbirlikçi filtreleme için kullanıcı geri bildirim verisi akışı madenciliği ve SASCF algoritması, (ii) Paralel kişiselleştirilmiş çift ögeli sıralama öğrenme ve PLtR algoritma ailesi, (iii) Matris ayrıştırma modellerinden en üst N öge tahmin etmenin verimliliğini artırma ve MMFNN meta-algoritması, (iv) Kullanıcı oturumlarında yönelim öğrenme ve BRF algoritma ailesi, (v) Soğuk başlangıç ortamında doğru zamanlı talep dışı öneriler ve hibrit bir sıralama öğrenme yaklaşımı. Önerilen yöntemlerin hem teorik hem de gerçek hayat verileri üzerinde yapılan deneysel analizlerinin sonuçları, sıralama doğruluğu, uyum sağlayabilme, öge çeşitliliği, verimlilik ve ölçeklenebilirlik kriterleri için önemli performans ve dengeleme iyileşmeleri göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES x
LIST OF TABLES xiv
LIST OF SYMBOLS xv
LIST OF ACRONYMS/ABBREVIATIONS xvi
1. INTRODUCTION 1
2. BACKGROUND AND FOUNDATIONS 6
2.1. Recommender Systems Essentials 6
2.1.1. Common Strategies 7
2.1.2. Implicit Feedback 8
2.1.3. Time Sensitivity 11
2.1.3.1. Efficient and Adaptive Model Learning 11
2.1.3.2. Fast Personalized Predictions 11
2.1.3.3. Session-based Recommendations 12
2.1.3.4. Time Dependency and Time Awareness 13
2.2. Learning to Rank for Recommender Systems 13
2.2.1. Personalized Learning to Rank Framework 13
2.2.2. Learning Similarities 16
2.2.3. Learning User-Item Models 17
2.2.3.1. Latent Factor Models 18
2.2.3.2. Other User-Item Models 18
2.2.4. Learning from Data Streams 19
2.2.4.1. Data Stream Mining 19
2.2.4.2. Reinforcement Learning 20
2.2.5. Evaluation 20
2.2.5.1. Ranking Accuracy 21
2.2.5.2. Other Performance Criteria 22

2.2.5.3. Experimental Settings 24
3. MINING USER FEEDBACK STREAM FOR COLLABORATIVE FILTERING 26
3.1. Introduction 26
3.2. Neighborhood-based Collaborative Filtering 27
3.3. Frequent Items and Frequent Top-k Items in a Stream 28
3.4. Frequent Co-occurrences and SASCF 31
3.4.1. Effective Personalized Recommendations 36
3.4.2. Complexity and Hyperparameterization 37
3.5. Experiments 38
3.5.1. Data and Exploratory Analysis 39
3.5.2. Performance Evaluations 41
3.5.2.1. Experiments Using a Holdout Set 42
3.5.2.2. Sequential Evaluation 44
4. PARALLEL PERSONALIZED PAIRWISE LEARNING TO RANK 49
4.1. Introduction 49
4.2. Personalized Pairwise LtR 50
4.3. Base PLtR Algorithms 52
4.3.1. Block Partitioning and PLtR-B 52
4.3.2. No Partitioning and PLtR-N 55
4.4. Extensions to PLtR Algorithms 58
4.4.1. Different Sampling Strategies 58
4.4.2. Adaptive Gradient Updates 60
4.4.3. Parallel LtR from Streaming Feedback 61
4.5. Experiments 65
4.5.1. Datasets and Evaluation 65
4.5.2. Statistics of Dataset Graphs 66
4.5.3. Evaluation of Ranking Accuracy and Speedup 68
4.5.4. Experiments with Extended Algorithms 70
4.5.4.1. Experiments with Different Sampling Strategies 70
4.5.4.2. Experiments with Adaptive Gradient Updates 73

4.5.4.3. Experiments for Parallel LtR from Streaming Feedback 75
5. EFFICIENT TOP-N PREDICTION FROM MATRIX FACTORIZATION MODELS . . . 77
5.1. Introduction . . . 77
5.2. MF Models for CF from Implicit Feedback . . . 78
5.3. Approximate NN Search in Metric Spaces . . . 80
5.4. A Meta-algorithm for Efficient Prediction from MF Models . . . 81
5.4.1. Enhancing Incremental CF . . . 85
5.4.2. Complexity . . . 87
5.5. Comparison to Related Work . . . 88
5.6. Experiments . . . 90
5.6.1. Datasets and Experimental Setup . . . 90
5.6.2. Experimental Results . . . 92
5.6.2.1. Experiments Using a Random Holdout Set . . . 92
5.6.2.2. Time-split Experiments . . . 96
5.6.2.3. Experimenting with RL Extension . . . 98
6. LEARNING INTENTION IN USER SESSIONS . . . 100
6.1. Introduction . . . 100
6.2. Problem Representation . . . 101
6.3. Learning Models . . . 103
6.3.1. RF for Imbalanced Data . . . 104
6.3.2. Balanced RF for Imbalanced Streams . . . 105
6.3.2.1. The Base Learner . . . 106
6.3.2.2. Online Bootstrap Sampling . . . 107
6.3.2.3. The Ensemble . . . 108
6.3.3. Improving Prediction Efficiency . . . 109
6.4. Experiments . . . 110
6.4.1. Dataset . . . 110
6.4.2. Experimental Results for BRF . . . 111
6.4.3. Experimental Results for BRFIS . . . 116
6.4.4. Experiments for Prediction Efficiency . . . 120

7. TIMELY PUSH RECOMMENDATIONS IN A COLD START SETTING . 121
7.1. Introduction 121
7.2. Problem Definition and Representation 122
7.3. Ranker Ensemble for Push Recommendations 123
7.3.1. Profile-based Rankers 123
7.3.2. Content-based Rankers 126
7.3.3. Top-M and Top-N Selection 126
7.4. Experiments 127
7.4.1. Data and Experimental Setting 127
7.4.2. Experimental Results 128
8. CONCLUSIONS 132
8.1. Mining User Feedback Stream for CF 132
8.2. Parallel Personalized Pairwise LtR 133
8.3. Efficient Top-N Prediction from MF models 134
8.4. Learning Intention in User Sessions 136
8.5. Timely Push Recommendations in a Cold Start Setting 136
REFERENCES 138

LIST OF FIGURES

Figure 1.1. An overview of the thesis 2

Figure 2.1. High-level view of a recommender system 6

Figure 2.2. High-level view of personalized LtR for recommender systems . . . 14

Figure 3.1. Single pass FREQUENT algorithm 29

Figure 3.2. Generic data structure for FREQUENT and SPACESAVING . . . 31

Figure 3.3. Training with SASCF algorithm 34

Figure 3.4. True vs. approximate co-occurrences 35

Figure 3.5. Basic co-occurrence statistics of representative items 40

Figure 3.6. SASCF and ranking accuracy 43

Figure 3.7. Effect of sampling on ranking accuracy 44

Figure 3.8. Sequential evaluation 45

Figure 3.9. Comparative sequential evaluation with OFFLINECF 46

Figure 3.10. Comparative sequential evaluation with ISGD 47

Figure 4.1. Personalized pairwise learning to rank 51

Figure 4.2. Block partitioning 52

Figure 4.3. Deciding the set of chunks in a round 53

Figure 4.4. Parallel personalized pairwise LtR with block partitioning 54

Figure 4.5. Hypergraph representations of LtR methods 56

Figure 4.6. Parallel personalized pairwise LtR with no partitioning 57

Figure 4.7. PLtR-N for streams 62

Figure 4.8. Comparison of vertex degree distributions 67

Figure 4.9. Comparing ranking accuracy and speedup of PLtR algorithms . . 69

Figure 4.10. Comparing different sampling strategies 71

Figure 4.11. Adaptive vs. uniformly at random sampling for PLtR-N 73

Figure 4.12. Comparing adaptive gradient updates 74

Figure 4.13. Comparing parallel LtR for streaming feedback 75

Figure 5.1. Meta-algorithm for efficient top-N prediction from MF models . . 82

Figure 5.2. Exhaustive top-N prediction from MF models 83

Figure 5.3. Example embeddings and cones in the latent space 84

Figure 5.4. MMFNN prediction stage with a MAB 86

Figure 5.5. MMFNN HR@N results 93

Figure 5.6. MMFNN MRR results 94

Figure 5.7. MMFNN AD results 94

Figure 5.8. MMFNN top-N prediction times 96

Figure 5.9. MMFNN MAP@N results in time-split experiments 97

Figure 5.10. MMFNN with MAB experiment 98

Figure 6.1. Session as a timeline 103

Figure 6.2. BRF2 algorithm 105

Figure 6.3. BRFIS algorithm 106

Figure 6.4. Simulation for online bootstrap sampling 108

Figure 6.5. Correlations among features 112

Figure 6.6. Normalized mean decrease impurity 112

Figure 6.7. ROC curves for different models 114

Figure 6.8. Precision-recall curves for different models 115

Figure 6.9. Predictions using BRFIS 117

Figure 6.10. An illustration of BRFIS model hyperparameters 118

Figure 6.11. Comparison of methods for efficient prediction 119

Figure 7.1. Ranker ensemble for push recommendations 123

Figure 7.2. Multi-objective performance measure 129

Figure 7.3. Comparative performance of different rankers 130

LIST OF TABLES

Table 1.1. Algorithmic contributions 4

Table 2.1. Explicit vs. implicit feedback 10

Table 3.1. Comparative complexity and hyperparameterization of SASCF . . 38

Table 3.2. Datasets for testing SASCF 39

Table 4.1. Datasets for testing PLtR algorithms 66

Table 4.2. Statistics of dataset graphs for LtR 67

Table 5.1. Datasets for testing MMFNN 91

Table 5.2. MMFNN learning and prediction times 96

Table 6.1. Representation of a session-item pair 102

Table 6.2. Benchmark dataset for intention learning 111

Table 6.3. A comparison of BRF models 116

Table 7.1. Representation of a user-item pair 125

LIST OF SYMBOLS

$1_A(x)$	Indicator Function
f	Number of Features or Factors
I	Set of Items
I_u^+	Set of Items Known to Be Relevant to User u
L	Loss Function
$\mathcal{M}$	Learning Model
$\mathbf{p}_u$	Vector Representation of User u
$\mathbf{q}_i$	Vector Representation of Item i
$r_c(i)$	Known Rank of Item i under Context c
$\hat{r}_c(i)$	Estimated Rank of Item i under Context c
$\hat{r}_c^{-1}(n)$	Estimated Item at Rank n under Context c
$\mathbf{S}$	Similarity Matrix
U	Set of Users
U_i^+	Set of Users to Whom Item i is Known to Be Relevant
$\mathbf{Y}$	Matrix (Tensor) of Known User-Item Relevances
$\widehat{\mathbf{Y}}$	Matrix (Tensor) of Estimated User-Item Relevance Scores
$\hat{y}_c(i)$	Estimated Relevance Score of Item i under Context c
γ	Learning Rate
λ	Regularization Parameter
σ	Sigmoid Function
ψ	Number of Processing Units

LIST OF ACRONYMS/ABBREVIATIONS

AD	Aggregate Diversity
ALS	Alternating Least Squares
AS	Adaptive Sampling
AUC	Area under Receiver Operating Characteristic Curve
BPR	Bayesian Personalized Ranking
BRF	Balanced Random Forest
BRFIS	Balanced Random Forest for Imbalanced Streams
CBF	Content-based Filtering
CF	Collaborative Filtering
CTR	Click-through Rate
EWMA	Exponentially Weighted Moving Average
GBDT	Gradient-boosted Decision Tree
GF	Graded Relevance Feedback
HR@N	Hit Rate in Top-N Recommendations
ICR	Item Conversion Rate
KMT	K-means Tree
LFM	Latent Factor Model
LtR	Learning to Rank
MAB	Multi-armed Bandit
MAP@N	Mean Average Precision at Cut-off N
MBM	Maximum Bipartite Matching
MF	Matrix Factorization
MMFNN	Meta-algorithm for Efficient Prediction from MF Models
MRR	Mean Reciprocal Rank
NN	Nearest Neighbors
PLtR-B	Parallel Pairwise LtR with Block Partitioning
PLtR-N	Parallel Pairwise LtR with No Partitioning
PLtR-NS	PLtR-N for Streams

RF	Breiman's Random Forest
RKT	Randomized K-d Trees
RL	Reinforcement Learning
RS	Recommender System
SASCF	Scalable and Adaptive Stream-based CF Algorithm
SGD	Stochastic Gradient Descent
SLtR	Sequential Pairwise LtR
SVD	Singular Value Decomposition
WRMF	Weighted Regularized Matrix Factorization

1. INTRODUCTION

Recommender systems [1, 2] are becoming increasingly more decisive in how we consume goods, services, and digital content. Consequently, the ever-increasing scale and diversity of their applications necessitate effective recommender systems which are based on more efficient ranking models with higher predictive power. Such models are typically constructed by some form of learning to rank (LtR) [3] using available user feedback and associated information in the system.

On the contrary to popularized examples [4, 5] based on explicit relevance assessments, many real-life recommender systems are based on implicit feedback [6–8] which is inferred from different types of user behavior and interactions in the system. This is usually because modeling implicit feedback typically requires no extra effort from the user and, additionally, it can capture the user-item relevance information in the system better through time. In this thesis, the recommendation problem is mainly considered as personalized LtR from such feedback. Novel efficient personalized LtR methods are proposed to learn from large-scale implicit feedback datasets or streams.

In addition to scale, recommender systems need to handle time from various aspects. Accordingly, we observe that there are almost always some requirements imposed by the particular application area for time-sensitive recommendations. In this thesis, we also emphasize the importance of time-sensitive recommendations and distinguish the major challenges for achieving them. With each proposed personalized LtR method, we try to address some of these challenges as well. While we provide a detailed view in Section 2.1.3, we list these challenges below:

(i) Efficient and adaptive model learning
(ii) Fast personalized predictions
(iii) Session-based recommendations
(iv) Time dependency and time awareness

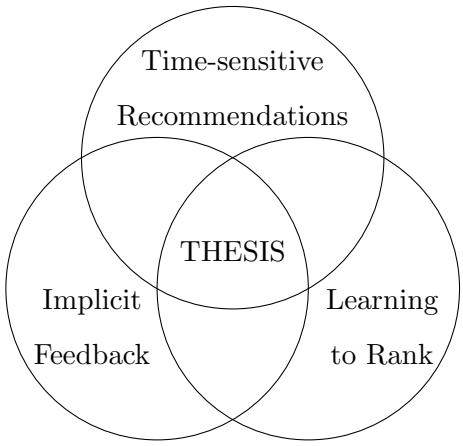

Figure 1.1. An overview of the thesis.

Therefore, in general, the contributions in this thesis can be seen at the intersection depicted in Figure 1.1. Furthermore, in particular, the following contributions are made and presented in their dedicated chapters:

- *Mining user feedback stream for collaborative filtering.* We propose SASCF, a novel scalable and adaptive personalized recommendation algorithm, in which the underlying neighborhood model can be updated with the streaming user feedback. The algorithm uses a space-efficient data structure which can also be used to retrieve an item's top-k co-occurring items in $O(k)$ time. This enables finding similar items quickly with respect to various heuristic similarity measures, and hence, enables fast personalized predictions based on the items known to be relevant to a user through the user's feedback history.
- *Parallel personalized pairwise learning to rank.* Pairwise learning to rank is known to be suitable for personalized recommendation tasks involving implicit feedback. Focusing on collaborative filtering, we show that its efficiency can be greatly improved with parallel stochastic gradient descent schemes. Accordingly, we first propose PLtR-B and PLtR-N algorithms for the pairwise learning to rank problem setting. We then show the versatility of these proposals by showing the applicability of several important extensions commonly desired in practice. Theoretical as well as extensive empirical analyses of our proposals show remarkable efficiency results for pairwise learning to rank in offline and stream learning settings.

- *Efficient top-N prediction from matrix factorization models.* Personalized LtR based on matrix factorization is a prevailing approach. We propose a meta-algorithm, MMFNN, which can employ various common matrix factorization models, drastically improve their prediction efficiency using approximate nearest neighbor search methods, and still perform comparably to standard prediction approaches or sometimes even better in terms of predictive power. Using various batch, online, and incremental matrix factorization models, we present detailed empirical analysis results on many large implicit feedback datasets from different application domains.
- *Learning intention in user sessions.* We propose a way to represent the intention learning problem by converting session information into feature vectors. Then, we propose batch and stream versions of an ensemble learning method which can take into account the high class imbalance due to the rarely occurring purchasing feedback. We also investigate several ways for efficient prediction from this ensemble since the prediction of intention needs to be fast enough for time-sensitive recommendations. We show our experimental results on a recent e-commerce benchmark dataset.
- *Timely push recommendations in a cold start setting.* Push recommendations typically have explicit requirements to satisfy multiple stakeholder objectives while being time sensitive. We propose an effective hybrid personalized LtR approach for large-scale push recommendations especially in an item cold start setting. While the proposed approach can be used for different applications, our case study is a job recommender system, and we work on a recent real-life benchmark.

Table 1.1 summarizes the algorithms contributed in this thesis. These contributions are based on publications by the author during his Ph.D. studies, and they are cited in the first column. The time sensitivity column refers to the four previously listed challenges and shows that an algorithm provides mechanisms for solving the corresponding challenges. Additionally, the table summarizes the types of feedback the algorithm can handle, the LtR approach used, and the recommendation strategy. The details of all columns 2-5 can be found in Chapter 2.

Table 1.1. Algorithmic contributions.

Algorithm	Time sensitivity	Feedback type	LtR methods	Strategy
SASCF [9]	(i), (ii), (iv), useful for (iii)	Implicit or explicit	Similarity learning	CF
PLtR family [10–12]	(i), (iv), useful for (iii)	Implicit or explicit	Pairwise LtR. User-item model.	CF
MMFNN meta-algorithm [13]	(ii), (iv), useful for (iii)	Implicit or explicit	Pointwise, pairwise, or listwise LtR. User-item model.	CF, user-item models using dot product space
BRF family for intention learning [14, 15]	(i), (ii), (iii), (iv)	Implicit	Pointwise LtR. User-item model.	Supports all
Ranker ensemble for push recommendations [16]	(i), (ii)	Implicit	Pointwise LtR. User-item model.	Hybrid, CBF

The thesis is organized as follows: In Chapter 2, the essential background is presented for recommender systems and implicit feedback as well as a clearer view of the challenges for achieving time-sensitive recommendations. The same chapter also provides an overview of LtR for recommender systems in offline and stream learning settings together with the evaluation methods. In Chapter 3, we present our approach for finding neighborhoods by mining user feedback stream and the SASCF algorithm. Then, in Chapter 4, we study efficient parallel pairwise LtR in offline and stream learning settings and present the PLtR-B and PLtR-N algorithms along with their extensions. The meta-algorithm, MMFNN, for improving top-N prediction efficiency

of matrix factorization models is presented in Chapter 5. Our approach for learning intention in user sessions is presented in Chapter 6 and for timely push recommendations in Chapter 7. Finally, we conclude in Chapter 8 with a general discussion of findings and future research directions.

2. BACKGROUND AND FOUNDATIONS

2.1. Recommender Systems Essentials

Recommender systems [1,2] perform information filtering to find the most suitable items for system users among a large number of choices. Since they are effective in dealing with the *information overload* problem [17], they have become an integral part of many web-based systems, e-commerce, and social networks due to the ever-increasing number of users and items in such systems. The term *item* in a recommender system defines a general concept which may refer to, for example, an advertisement, a retail product, or news. Some of these items can be long-standing in the system such as a book whereas others can be shorter-lived such as a job advertisement, and even ephemeral such as recent news. Recommender systems can also recommend other users or a sequence of items instead of individual items.

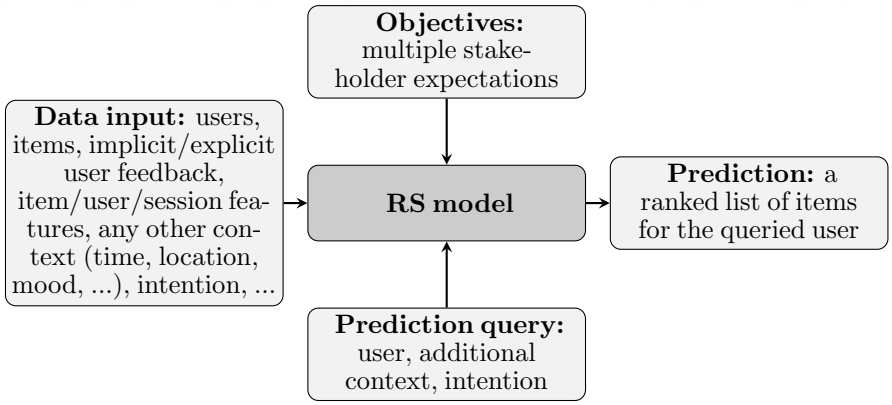

Figure 2.1. High-level view of a recommender system.

A high-level view of a recommender system is given in Figure 2.1. At the core is a recommender system model which learns from the data input to the model. This learning can be guided with the objectives of multiple stakeholders in the system. For example, while the common user objectives may refer to effective personalization, discovery of new serendipitous items, and privacy, the item owner or the service provider may desire to increase sales from recommendations, push under-explored or new items,

and ensure the robustness of the system. Therefore, the ultimate goal of a recommender system can be thought of as optimizing these objectives by making the most out of available input to the system. When a recommender system is queried, it outputs a ranked list of items which attempts to reflect the goal. One or more measures of performance can be used to monitor the achievement of the goal.

2.1.1. Common Strategies

Most approaches for building recommender systems fall under a few common strategies. These strategies can be listed as follows:

- *Weak personalization*: These approaches include recommending items which, for example, are the most recent, the most popular, or have the highest CTR based on some relatively simpler statistics. Association rule extraction from user feedback histories is also popular. Such approaches may impose a weak form of personalization, for example, by excluding the popular items with which the user has already interacted or by finding the most frequently co-purchased items with the latest interacted item. But, they can still be strong baseline heuristics depending on the particular application area.
- *Content-based filtering (CBF)*: The common approach is based on building a user profile from user's past feedback [18]. This profile often amounts to a sparse feature vector which contains normalized scores derived from the contents of the previously interacted items. It can also be mapped to concepts using, for example, latent semantic indexing [19]. In some cases, the user profile can contain demographic features or prespecified features corresponding to some item features [20]. Recommendations can be based on the ranked similarity of an item content with the user profile. Alternatively, user, item, and additional context information can be merged in a vector which is then mapped to some relevance score through pointwise or pairwise LtR [16, 21, 22]. We note that CBF may require extensive application-specific feature selection or extraction, but it is useful especially in cold start scenarios [23].

- *Collaborative filtering (CF)*: This strategy [24] is primarily based on user feedback in the system, and it has several flavors: User-based CF finds users with similar feedback histories and recommends their items to each other based on some weighted similarity scheme. Item-based CF finds items which receive feedback from a similar subset of users, and then the users are recommended similar items to those in their feedback history. These two methods are commonly called neighborhood-based CF. Latent factor models (LFMs) are also prevalent in CF, and they map users and items into a space where users with similar tastes are recommended similar items. In general, CF can be considered an application-agnostic strategy which potentially reduces efforts on feature selection and extraction. Apart from user feedback, additional context such as time and location can also be used in CF models. Furthermore, side information from users and items can be integrated [25]. On the other hand, CF can suffer from cold start. One way to alleviate this problem is using incremental algorithms which can incorporate the newest feedback quickly into the model.
- *Hybrids*: Different approaches to recommender systems have their own strengths and weaknesses. Furthermore, the same approach with different hyperparameterizations or data can yield different results. Therefore, combining different recommenders have been found useful. Common hybridization methods include [26]: Weighted combination schemes, mixed presentation from different recommenders, and meta-level hybrids in which a previous recommendation model is the input of the next. It should be noted that a good trade-off between hybridization and efficiency is often of concern in large-scale systems. Furthermore, since hybrids are inevitable in many applications, individual strategies within an hybrid is often desired to be efficient.

2.1.2. Implicit Feedback

Two commonly distinguished types of user feedback in recommender systems are *explicit* and *implicit* feedback. The former refers to explicit relevance assessments of items by the users. It is still prominent in RS research as popularized by the

historically available benchmark datasets [5] and the Netflix prize challenge [4]. The latter type of feedback is implicit in the sense that it does not require an explicit assessment process by the user. Implicit feedback is extracted from user behavior and any type of user interaction with the system such as a click, view, purchase, or bookmark [27]. It may be inferred as some sort of binary or graded item relevance for the user. Here, the grade refers to a confidence value or information based, for example, on the frequency, duration, or perceived importance of the user feedback [7,28]. Implicit feedback may also facilitate useful additional information such as dwell times and various other session-based [14] or longer-term statistics.

Despite the popularity of explicit feedback in research, real-life recommender systems are often based on implicit feedback [6–8]. One of the main reasons is that it can be infeasible to collect explicit human assessments in many real-life large-scale systems. Another is that implicit feedback can reflect more reliable and up-to-date item relevance to a user [3] when collected over time. In other words, rather than obtaining explicit user feedback for an item at a fixed point in time, it can be more informative to track user's behavior for the item over time. On the other hand, recommender systems based on implicit feedback have their own difficulties. Table 2.1 presents a comparison of explicit and implicit feedback for recommender systems.

Typical examples of recommender systems based on implicit feedback include: video recommendation to users or user groups based on watching behavior, music recommendation based on listening behavior, job recommendation based on click and application behavior, news and social media recommendations based on interaction histories with the system, and online retail recommendations based on click and purchase behavior. Due to dataset scarcity, explicit feedback datasets are sometimes converted into implicit feedback datasets using binarization or some other form of discretization [6, 30]. This is a viable approach since irrespective of the explicit rating, a user's interaction with an item may still indicate some sort of relevance.

Table 2.1. Explicit vs. implicit feedback in RSs.

Explicit	**Implicit**
Refers to explicit user assessments each usually collected at a fixed point in time	Refers to unobtrusively collected and possibly repetitive user behavior and interactions
Relevance typically expressed on a rating scale	Binary or graded relevance can be acquired, for example, through frequency, duration, perceived importance of the feedback. Modeling unknown or negative feedback can be more critical.
A single feedback instance may be perceived as a strong signal of relevance. However, it can also refer to a biased or even misleading user assessment.	A single feedback instance is often a weak signal of relevance. However, since implicit feedback refers to what the users actually "do", accumulating feedback can become quite reliable.
Models are often evaluated using error metrics [29]	Models are often evaluated using ranking-based measures
Harder to collect. People do not tend to assess items explicitly outside a few domains.	More abundant. May arrive as huge log data or massive streams. Algorithm efficiency and adaptivity can be relatively more important.
More popular in research due to historical dataset availability	Most real-life applications are based on implicit feedback. However, availability for research purposes is relatively low.

2.1.3. Time Sensitivity

In a broad sense, time-sensitive recommendations can be described as sufficiently accurate recommendations which have to be delivered in a timely manner. As motivated in Chapter 1, we distinguish major challenges for achieving time-sensitive recommendations and present a detailed view below.

2.1.3.1. Efficient and Adaptive Model Learning. Efficient model learning with a minimal trade-off of predictive power is a major concern in recommender systems. Furthermore, since many real-life recommender systems are hybrids, individual recommenders need to be efficient. Such requirements often necessitate faster batch training schemes or incremental training.

It is also often desired that a model is adaptive so that the changing dynamics of the system are reflected in the model quickly. Model adaptivity is often handled by performing periodic retraining with full or windowed historical data. Therefore, to be able to increase the training frequency, efficient training schemes are desired. Incremental training can also be used to increase adaptivity since it is able to incorporate the latest data into the model without retraining from the beginning. This also fits the nature of feedback data in recommender systems which comes in a streaming fashion, and the learning should naturally be an incremental process. Incremental training can also enable the model to gradually forget the past data and adapt to the latest.

In general, efficient and adaptive model learning can achieve faster adaptation to changing temporal dynamics such as incoming cold start users/items, changing user preferences, and drifting item popularity.

2.1.3.2. Fast Personalized Predictions. Fast predictions from a model are often a key but underestimated requirement in web-based intelligent systems research. Even though the number of prediction queries can be massive, it is often cited that practical systems today should respond to each query well under 100 milliseconds [3]. It should

be noted that personalized queries can even be more complex than non-personalized ones since the predictions are always conditioned on a user. Therefore, apart from the model learning time, real-time predictions from a model have strict response time requirements.

Although the RS research usually focuses on training highly accurate models, costly predictions may prohibit accurate but too complex models. Even linear models which are often relatively faster to yield useful predictions can be problematic when the number of items is large and the number of queries is massive.

One solution for faster personalized predictions is to precompute predictions for each user. The drawback of this approach is that precomputation itself can be costly and result in decreasing the training frequency. Moreover, the predictions are fixed until the next precomputation which may not be desired in dynamic recommendation domains. Therefore, efficient and real-time personalized prediction schemes are always desired in practice.

2.1.3.3. Session-based Recommendations. Recommendations based on session information may pose some specific challenges. If no prior user information is available, personalized predictions must be based on the ongoing session information only. This may also necessitate a quick model learning scheme from incoming session feedback. Such a case is quite common in practice due to anonymous user behavior or when the recommendation service is outsourced [31]. In other cases, it may be desired to incorporate session information into the available user model to merge long- and short-term interests [32].

Session information can also be actively monitored to predict user context or intention [14]. This can facilitate more customized recommendations. On the other hand, modeling context or intention can be challenging due to the limited session information as well as response time constraints.

2.1.3.4. Time Dependency and Time Awareness. Recognizing two concepts, *time dependency* and *time awareness*, has been found useful in RS research [25, 33], and they are also important for time-sensitive recommendations. The former refers to the ability of a model to consider time order of the user feedback. This aligns with our notion of adaptive learning in Section 2.1.3.1. Furthermore, it can sometimes be achieved by filtering recommendations during the prediction phase. The latter refers to taking into account recurrences in time such as seasonality or periodicity of user preferences. While handling time dependency can also be useful for time awareness, time-aware models are often studied under the topic of context-aware recommender systems [34, 35]. Common approaches include incorporating time context as an extra model dimension as well as filtering the input or output of the model for the context of interest.

2.2. Learning to Rank for Recommender Systems

This section provides preliminaries on LtR for recommender systems. The focus is on important modeling approaches which can handle implicit feedback and achieve time sensitivity. We also discuss the evaluation of resulting models.

2.2.1. Personalized Learning to Rank Framework

With appropriate model selection, Figure 2.1 allows us to see the recommendation problem as an LtR [3, 36, 37] problem. Since the recommendations typically require some sort of personalization, that is, the user is (or part of) the context [38], the problem is called *personalized learning to rank* in this thesis.

LtR is a flourishing area of machine learning [39] in which the generic problem is to learn a good overall ranking model from available partial rankings or known relevance information by using suitable objective functions. The problem sometimes transforms into regression or classification in which case the predicted scores are used to rank entities. LtR has important applications especially in information retrieval. Examples include document retrieval for a given query or sentence retrieval for machine translation [36]. In the case of recommender systems, a ranked list of items is retrieved

for a given user or more generally for a given context. Since the number of items are usually quite large, it is computationally expensive to find the best ranking for a user over all possible item permutations. Therefore, personalized LtR models often predict relevance scores for items which are then sorted to find a good ranking.

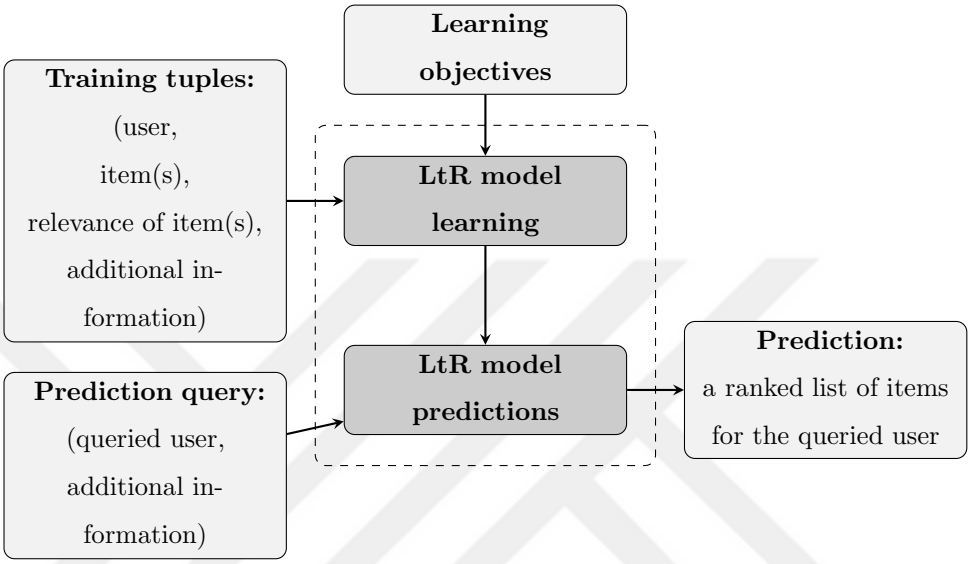

Figure 2.2. High-level view of personalized LtR for recommender systems.

A high-level view of personalized LtR for recommender systems is given in Figure 2.2. The LtR model learns from the training tuples by considering the learning objectives typically through some objective function, such as a loss function. As the training tuples keep coming, the model keeps learning through periodic retraining, incremental learning, or possibly reinforcement learning [39, 40]. In the meantime, the model supports prediction queries and outputs a ranked list of conceivably relevant items for each queried user. A useful taxonomy of LtR approaches is proposed in [3] basically referring to the different types of training tuples and the corresponding objective functions. We consider this taxonomy in our personalized LtR setting as follows:

- *Pointwise LtR*: In this approach, each training tuple contains a user and a single item. The item is known to be relevant to the user with some degree of relevance. The model makes use of a tailored objective function which takes some feature representation of the user and the item as parameters, and it tries to learn

personalized relevance scores for all items by optimizing the associated objective function.

- *Pairwise LtR*: The training tuple contains a user and a pair of items for which the user has a pairwise preference. A pairwise objective function takes this input and facilitates learning to rank the more relevant items above the less relevant. In other words, pairwise LtR works with partial rankings between items. The error is typically based on the number of inversions or how well separated the pairs are [10].
- *Listwise LtR*: The training tuple contains a user and an arbitrarily large list of items with their degree of relevance to the particular user. An objective function considers the partial ranking implied by this item list. In some cases, the objective function can be derived explicitly from a suitable information-retrieval-specific measure, for example, by considering the positional information of items to optimize directly for top-N recommendations [41]. On the other hand, listwise LtR can be computationally expensive since each training tuple may involve updates based on many items. This can especially be problematic for incremental learning. Finding the optimal hyperparameter values for listwise LtR models can also be more costly.

A personalized LtR model can also be a hybrid of different LtR models. If the problem can be transformed into classification or regression, common ensemble methods in machine learning can be used [39]. A hybrid model can also be formed by considering the special cases in recommender systems [26] or rank aggregation [36].

Although designing good objective functions is at the core of personalized LtR, efficiency of models with large-scale data should not be overlooked [3] in modern LtR systems. In Section 2.1.3, the challenges for achieving time-sensitive recommendations are distinguished which cover model efficiency and beyond. Consequently, in Sections 2.2.2 to 2.2.4, we discuss some important modeling approaches which can be suitable for defeating such challenges.

2.2.2. Learning Similarities

Many personalized LtR approaches are primarily based on learning a similarity model for users or items which explicitly or implicitly constructs a similarity matrix, $\mathbf{S}$. These similarities can then be used to decide nearest neighbors and facilitate recommendation queries [42]. Examples include:

$$\hat{y}_u(i) = \sum_{j \in I_u^+} w_{uj} \mathbf{S}_{ij} \qquad \text{and} \qquad \hat{y}_u(i) = \sum_{v \in U_i^+} w_{vi} \mathbf{S}_{uv}, \tag{2.1}$$

for item-based CF or CBF and user-based CF, respectively, where $w_{\cdot}$ is an application-specific weighting term.

While the user feedback for items is not missing at random [43], in real-life systems, it is almost always incomplete. In fact, it is not uncommon that a user-item feedback matrix is well above 99% sparse. Furthermore, the existing feedback may become obsolete or imply less relevance over time. Therefore, user or item similarities based on feedback histories are never exact and rather learned from what is available.

A common objective is to approximate a co-occurrence-based similarity measure such as cosine similarity or conditional probability from available feedback. For example, in the case of item-based CF, for every user, every available co-occurring item pair can be considered to update a normalized score which eventually results in an item similarity matrix through batch or incremental approximation methods. This approach is relatively straightforward, but it has proven to be very effective in practice [44,45]. We note that similarities computed this way are not necessarily based on some distance in a metric space. However, this is not a strict requirement either, since we are usually trying to obtain a ranking as per Equation 2.1.

Similarity learning can also be seen as a form of pairwise learning. Some notable approaches are based on maximum margin classifiers with kernel tricks [46,47] and AUC optimization [48]. These approaches can learn interesting similarity measures from

co-occurring item pairs in the pairwise LtR setting. While such similarity measures increase predictive power, several problems may arise as to learning time efficiency especially if the metric rules are maintained and space efficiency due to learning a much denser or full similarity matrix.

Learning item embeddings in latent spaces through specialized listwise loss functions [49,50] is also possible. These approaches have the potential benefit of discovering interesting similarities while bringing space efficiency due to compact representations. On the other hand, hyperparameter tuning as well as online updates are usually more costly due to more complex loss functions. More recently, recurrent neural networks are extensively researched for predicting the next item for a given sequence of session items [51]. Such networks have the potential of including side information and provide a way to model the time. However, the trade-off between accuracy and efficiency should be justified in different applications [52].

Finally, some methods try to learn highly similar neighbors directly in a less lazy [39] fashion, that is, they try to learn a sparse representation of the similarity matrix which maintains only a set of items highly similar to an item (or possibly users highly similar to a user). For example, elastic net regularization has been used for this purpose [53] with a constrained optimization scheme. However, such an optimization scheme can be prohibitively inefficient for large-scale applications. One possible remedy is relaxing the constraints such as non-negative similarities and solving a simpler problem using SGD [54]. An interesting approach in direct neighborhood learning can be trying to maintain top-k neighbors using data stream mining techniques [9]. We discuss this approach in detail in Chapter 3.

2.2.3. Learning User-Item Models

Rather than concentrating on the similarities between users or items, user-item models try to model users and items in a common feature space using suitable representations. In the following, we categorize these models as latent factor models and others.

2.2.3.1. Latent Factor Models. The most common model is based on estimating a user-item relevance matrix by learning the following low-rank matrix factorization [55],

$$\widehat{\mathbf{Y}}_{ui} = \sum_{k=1}^{f} \mathbf{P}_{ku}\mathbf{Q}_{ki}, \tag{2.2}$$

where $\mathbf{P} \in \mathbb{R}^{f\times|U|}$, $\mathbf{Q} \in \mathbb{R}^{f\times|I|}$ and typically the number of factors, $f \ll |U|, |I|$. This has its roots in the successful application of SVD to some information retrieval tasks [19, 56]. On the other hand, a direct application of SVD is problematic in many personalized LtR scenarios since SVD optimizes a particular type of loss function and requires a dense input matrix. In the more general case, the matrix factorization model transforms into learning a relevance tensor [38] with the possible inclusion of additional contexts in the model,

$$\widehat{\mathbf{Y}}_{c_1c_2\ldots c_m i} = \sum_{k=1}^{f} \mathbf{Q}_{ki} \prod_{j=1}^{m} \mathbf{C}_{kc_j}. \tag{2.3}$$

LFMs usually fit well under the personalized LtR framework. They have several obvious advantages including space efficiency due to low-rank representations and versatility for tailoring different loss functions. Unlike learning from explicit feedback, in the implicit feedback case, the main objective is usually not to complete or approximate a known preference matrix, but rather learn a relevance scoring matrix for good personalized rankings of items. Some successful LFM approaches for implicit feedback can be considered under the pointwise [7, 57], pairwise [48], and listwise [41] LtR framework, and we cover them more extensively in Chapters 4 and 5.

2.2.3.2. Other User-Item Models. If the user and item vectors are in the same space, we can simply look at their similarities. This is the typical scenario in CBF where user profiles are matched with the item content. The vectors can be in the original space using sparse representations, or we can also map these representations to a latent space [16] and measure their similarities there.

Alternatively, a user and an item (or items) together with additional context can be represented using a single feature vector, $\mathbf{x}_{ui}$. This representation allows us to use successful classification or regression models [39] for learning ranking scores [16, 22] as well as some hybrid models [21]. Two examples are:

$$\hat{y}_u(i) = \frac{1}{1 + \mathrm{e}^{-\mathbf{w}^\top \mathbf{x}_{\mathrm{ui}}}} \qquad \text{and} \qquad \hat{p}(r_u(i) < r_u(j)) = \frac{1}{1 + \mathrm{e}^{-(\mathbf{w}^\top \mathbf{x}_{\mathrm{ui}} - \mathbf{w}^\top \mathbf{x}_{\mathrm{uj}})}}, \tag{2.4}$$

for the pointwise and the pairwise [58] LtR cases, respectively. On the other hand, such models typically require an extensive effort for engineering the features of $\mathbf{x}_{ui}$. Furthermore, to be practical for personalized LtR, both the learning and the prediction efficiency of the models are important. Especially, as the number of items gets larger, personalized predictions often necessitate a multi-stage pipeline from a simpler to an increasingly more complex model [16, 59]. We discuss these issues further in Chapter 7.

2.2.4. Learning from Data Streams

User feedback naturally arrives at the system as a data stream. Furthermore, to deal with the challenges for time-sensitive recommendations discussed in Section 2.1.3, LtR models which learn from streaming feedback can be useful. In the following, we mention two different research efforts to learn from such streaming data.

2.2.4.1. Data Stream Mining. In this approach, the common assumption is that there is a massive flow of data into the system so that it is infeasible to maintain all of the data in a fast memory and also to retrain the model from scratch periodically. Rather, data stream mining models try to learn in an incremental fashion from the incoming data and maintain a relatively space-efficient summary of it. This way the model learning process can also use the available computational resources more effectively [60]. Such models are able to approximate many useful statistics of a theoretically infinite stream using elaborate sampling, counting, and filtering techniques [61]. Furthermore, they facilitate incremental versions of common machine learning approaches such as classification, clustering, and association rule learning [62], and more recently LtR [33].

Data stream mining especially considers the time dependency and adaptivity concepts given in Section 2.1.3, and we provide more detailed discussions in Chapters 3 to 6.

2.2.4.2. Reinforcement Learning. Most personalized LtR methods are based on supervised learning. Alternatively, in some applications, user feedback which follows from a previous recommendation list predicted by the system, can be imagined as a reward (or penalty). Such a system can implement reinforcement learning [40] and learn from incoming feedback in a sequential fashion to maximize the total reward.

The state-of-the-art learning approach in this direction is a multi-armed bandit (MAB) in which an arm typically corresponds to an item, and it has some unknown payoff distribution to be learned with a sequential explore-and-exploit scheme [39, 40]. To achieve personalized recommendations, this approach has been extended to contextual bandits [63] in which the expected payoff is a function of both the context (such as a user) and the item.

In general, while reinforcement learning approaches are promising, it can be hard to implement them effectively in real-life applications, for example, when the number of items is large or regarding their prediction efficiency. Furthermore, it is relatively hard to evaluate reinforcement learning approaches especially in an offline fashion [63]. An extended discussion about these issues is given in Chapter 5.

2.2.5. Evaluation

The primary objective in many RSs is user satisfaction. Yet, there can be additional objectives regarding other stakeholders of the system. Given the intricacy of such system objectives, it is often too hard to evaluate everything using a single performance criterion. Therefore, breaking system objectives into smaller performance criteria is found useful for understanding and improving the system [64]. Furthermore, the experimental setting for evaluations is rather important, and it is affected by the

nature of available data. In this section, we introduce important evaluation concepts which are used throughout this thesis.

2.2.5.1. Ranking Accuracy. A commonly used performance criterion is accuracy. When explicit ratings are available, this can be based on some convenient error metrics such as root mean squared error [4] or mean absolute error. In the case of personalized LtR for implicit feedback, we are rather interested in evaluating the accuracy of ranked lists for some test users. Although the user feedback in a system is vastly missing and the number of false positives is prone to overestimation, measures based on precision, recall, or false positive rate in a limited top-N [29] or fully ranked list of items are common. One such widely used measure is hit rate [44] and it is defined as,

$$\text{HR@}N = \frac{1}{|U^\star|}\sum_{u=1}^{|U^\star|} |\{i_u^\star\} \cap \{\hat{r}_u^{-1}(1), \hat{r}_u^{-1}(2), \ldots, \hat{r}_u^{-1}(N)\}|, \tag{2.5}$$

where a hit means that a hidden test item for a test user, $i_u^\star$, appears in the user's top-N predicted recommendations, and $U^\star \subseteq U$ is a set of test users. Another is an estimate of AUC over the test users [38, 65],

$$\widehat{\text{AUC}} = \frac{1}{|U^\star|}\sum_{u\in U^\star} \frac{1}{|I_u^+||I \setminus I_u^+|}\sum_{i\in I_u^+}\sum_{j\in I\setminus I_u^+} H(\hat{y}_u(i) - \hat{y}_u(j)), \tag{2.6}$$

where H is the Heaviside step function and $\hat{y}_u(k)$ is the estimated relevance score which decides ranking of an item k for a user u. In other words, $H(.) = 1$ if and only if $\hat{r}_u(i) < \hat{r}_u(j)$. This measure can be adapted for various experimental settings and also turned conveniently into a differentiable objective function for personalized pairwise LtR [38].

Some other useful accuracy measures are also adapted from Information retrieval [56, 66]. Mean reciprocal rank is defined as,

$$\text{MRR} = \frac{1}{|U^\star|}\sum_{u=1}^{|U^\star|} \frac{1}{\hat{r}_u(i_u^\star)}, \tag{2.7}$$

and takes the rank information of a hidden item into account in a different way complementing measures like HR@N. A truncated version of mean average precision, MAP@N [67, 68], can also be suitable to some experimental settings. MAP is a point estimate of precision and recall [66], and its truncated version is convenient for evaluating personalized LtR algorithms when each test user has an arbitrary number of relevant test items. MAP@N is measured as follows,

$$\begin{aligned}
\mathrm{P}(u,n) &= \frac{1}{n}\sum_{m=1}^{n} 1_{I_u^\star}(\hat{r}_u^{-1}(m)) \\
\mathrm{AP}(u,N) &= \frac{1}{\min(|I_u^\star|,N)}\sum_{n=1}^{N} \mathrm{P}(u,n)\times 1_{I_u^\star}(\hat{r}_u^{-1}(n)) \\
\mathrm{MAP@}N &= \frac{1}{|U^\star|}\sum_{u\in U^\star} \mathrm{AP}(u,N),
\end{aligned} \tag{2.8}$$

where $I_u^\star$ is assumed to be a set of relevant test items for a test user u.

2.2.5.2. Other Performance Criteria. There are other performance criteria beyond ranking accuracy which can be important for different RS applications. These include diversity, novelty, and serendipity of recommendations, adaptivity to changing system dynamics, efficiency, and scalability [64].

Increasing diversity, novelty, or serendipity of recommendations can have a trade-off with accuracy [69]. However, it is still important to be able to evaluate them so that this trade-off can be controlled, or different LtR algorithms can be compared for how they handle the trade-off. For example, *aggregate diversity* [70] can be an important performance criterion to observe the diversity of items across all top-N recommendations, and it can be measured as follows,

$$\mathrm{AD} = \frac{-\sum_{i=1}^{|I|} p(i)\log p(i)}{\log|I|}, \quad p(i) = \frac{\sum_{u=1}^{|U^\star|} |\{i\}\cap\{\hat{r}_u^{-1}(1),\hat{r}_u^{-1}(2),\ldots,\hat{r}_u^{-1}(N)\}|}{\sum_{k=1}^{|I|}\sum_{u=1}^{|U^\star|} |\{k\}\cap\{\hat{r}_u^{-1}(1),\hat{r}_u^{-1}(2),\ldots,\hat{r}_u^{-1}(N)\}|}. \tag{2.9}$$

This equation corresponds to a normalized entropy measure having the maximum entropy obtained from the uniform distribution of items as denominator.

As discussed in Section 2.1.3.1, *adaptivity* is an important challenge for time-sensitive recommendations. It can be evaluated qualitatively by observing whether the proposed LtR model provides mechanisms that are, for example, sensitive to changing user preferences or able to incorporate cold start users/items quickly into the model. Adaptivity can also be evaluated quantitatively, for example, by measuring accuracy of recommendations over time.

While accuracy of a personalized LtR model is always desired, a highly accurate model can be useless if its *efficiency* is inadequate. Both learning and prediction efficiency are important as discussed in Section 2.1.3. In addition to the computational complexity analysis, they can be evaluated by measuring the execution time for the learning process and the query, respectively. Furthermore, speedup patterns can be assessed for comparing improved models to a baseline. Besides time efficiency, space efficiency of an LtR model can also be an important performance criterion regarding the training data it requires as well as the number of parameters it maintains for a large number of users and items. We note here that it is important to use the same experimental setup to be able to compare time and space efficiency of different models. Efficiency is also related to *scalability* which refers to the ability of a model to achieve comparable accuracy and efficiency results with growing number of users, items, and feedback instances. This can be evaluated qualitatively by observing whether the model supports useful mechanisms such as sampling or parallelism to handle growth and quantitatively by comparing accuracy and efficiency results at different scales.

We finally note that different performance criteria or variations of a performance criterion can be combined to obtain an aggregated performance score, for example, to evaluate a multi-objective system [20, 71].

2.2.5.3. Experimental Settings. General machine learning experiment design principles [39] such as forming a testable hypothesis and fixing uncontrolled factors are also valid for personalized LtR models. On the other hand, it is worthwhile to discuss two important experimental settings: online and offline.

In real-life RS applications, it is always of interest to carry out *online experiments* such as A/B testing [72, 73] to evaluate alternative LtR models for the system. Such experiments typically allow the alternative models to produce recommendations for different random subsets of users for a certain period of time. During this period, the user feedback to the models is analyzed comparatively. Since the models are tested on real users in real time and in an exploratory fashion, comparing many of them can cause user dissatisfaction. Therefore, online experiments are usually useful after extensive *offline experiments* so that only the reasonable candidate models can be tested.

Offline experiments are typically based on logged user feedback in the system. Such logs can include binary or graded relevance feedback as well as additional context and content information to form useful datasets for understanding some aspects of the system. Carefully curated datasets for offline experiments also enable reproducibility in research. The validation or test sets in an offline experiment are typically obtained by hiding some of the user-item relevance information. Depending on the particular application and the available information in the dataset, the hiding process may differ. One common approach is forming a holdout set by leaving out a random relevant item for each test user. After training the model with the rest of the data, each test user is recommended a top-N list which can be used to evaluate many accuracy and other performance criteria. While being versatile, this approach may not consider the natural time order of the user feedback. Therefore, when time information is available in the dataset, it can be complemented by further experiments. A useful approach in this direction is to use a time-split in which user feedback up until the split is accepted as training data and the rest is used as test data. This can be used to evaluate accuracy and efficiency of both offline and stream learning for personalized LtR. A difficulty with this approach is that we are left with a set of test users having a variable number of feedback instances after the time split. In the case of accuracy measurements,

MAP@N can be used to handle this situation. An extreme evaluation approach in the case of stream learning can be the first-test-then-train approach [74, 75] in which an incoming feedback instance or tuple is first used for testing the model and updating a performance score, and then it is used for updating the model in an incremental fashion.

3. MINING USER FEEDBACK STREAM FOR COLLABORATIVE FILTERING

3.1. Introduction

This chapter presents a novel scalable and adaptive algorithm for personalized recommendations in which the underlying neighborhood model can be updated with the streaming user feedback. The algorithm does not perform an offline search for finding nearest neighbors in an item similarity matrix. Instead, taking a landmark window over the user feedback stream, a space-efficient summary structure is maintained. This structure corresponds to the result of a standing iceberg query for finding every item's top-k frequently co-occurring items over a specified support threshold. Mining such frequent co-occurrences can facilitate approximate computation of several useful similarity measures. The algorithm offers space efficiency and scalability thanks to the neighborhood summary structure. It also offers adaptivity in the sense that newly arriving users, items, and user feedback on items are quickly integrated into the model. The up-to-date model can readily be used for efficient personalized predictions with the most recent information. In particular, this chapter presents the following contributions:

- Frequent item finding algorithms are extended to efficiently mining top-k frequent co-occurrences in streams. These co-occurrences are also shown to facilitate computation of several useful similarity measures.
- A novel scalable and adaptive neighborhood-based CF algorithm, SASCF, is proposed. The algorithm can be used for personalized recommendations with the most up-to-date information in the system.
- Theoretical and empirical analysis results are shown for the usefulness of the proposed stream summary structure instead of an item similarity matrix. Apart from space efficiency, the structure can maintain top-k frequently co-occurring items always in sorted order.

- It is shown empirically that the co-occurrence frequency of items in ranked order exhibit a power-law relationship and that the support thresholds for finding items' top-k frequent co-occurrences can be fixed to the mode of their distribution.
- Empirical analysis results are reported for comparative accuracy, scalability, and adaptivity.

The remainder of this chapter is organized as follows: The preliminaries for neighborhood-based collaborative filtering and finding frequent top-k items in a stream are discussed in Sections 3.2 and 3.3, respectively. Then, a discussion for mining frequent co-occurrences follows and SASCF is presented in Section 3.4. Experimental results on real-life data are presented and discussed in detail in Section 3.5.

3.2. Neighborhood-based Collaborative Filtering

As discussed in Section 2.1.1, neighborhood-based CF algorithms typically measure similarity among users or items in a user-item relevance matrix, $\mathbf{Y} \in \mathbb{R}^{|U|\times|I|}$, with respect to a similarity function. Our proposal in this chapter is based on a widely used offline item-based CF approach [44]. We call this approach OFFLINECF and describe it next.

In the model building phase, OFFLINECF computes the k most similar items for each item $i \in I$. Exact computation of the similarities typically requires building, updating, and storing an item similarity matrix, $\mathbf{S} \in \mathbb{R}^{|I|\times|I|}$. The primary goal is top-N recommendation which corresponds to predicting a personalized ranking of items, $\hat{I}_u^N = \{\hat{r}_u^{-1}(1), \hat{r}_u^{-1}(2), \ldots, \hat{r}_u^{-1}(N)\}$, such that $\hat{I}_u^N \subset I$. To solve this problem for a queried user u, the set C of candidate items are identified by taking the union of the k most similar items for each $i \in I_u^+$ and by possibly removing from the union the items that are already in I_u^+. Then, for each item $i_c \in C$, the similarity to the set I_u^+ is identified as the sum of similarities between all the items $i \in I_u^+$ and i_c, using only the k most similar items to i. Finally, the items in C are sorted in decreasing order with respect to this sum of similarities, and the first N items are selected as the top-N recommendations.

3.3. Frequent Items and Frequent Top-k Items in a Stream

Discovering frequent and other interesting patterns in sequences of actions is a core research area in data mining and knowledge discovery [76–78]. We focus on the problem of maintaining frequent items in streaming data. This is an interesting problem when the number of items is large, the space is limited, and a single pass over data is allowed. In this direction, FREQUENT algorithm [79, 80] is proposed for identifying in a set of items, the items with frequencies above a support threshold, ϕ. The space requirement of the algorithm is $O(1/\phi)$, and it guarantees to find all true positives, though some false positives may also be included in the resulting set. To get rid of false positives and obtain the exact frequency of true positives, a second pass over the stream is proposed. However, more than a single pass is not always desired in large-scale problems. SPACESAVING algorithm [81] also offers guarantees on space complexity. Additionally, it can produce better approximations to exact frequencies of the top-k frequent items in a single pass, especially if the frequencies of items in ranked order follow a power-law relationship. Such a relationship is common in many problem domains. LOSSYCOUNTING algorithm [82] is also proposed for solving the same problem with controllable error bounds at the cost of increased space complexity. Since they all conceptually use item counters, these three algorithms are sometimes unified under the name *counter-based algorithms*. Extensive comparisons of counter-based algorithms can be found in [83, 84].

For the completeness of discussion, the single pass FREQUENT algorithm is illustrated in Figure 3.1. Let I be a set of $|I|$ distinct items and S be a stream of n item appearances. The frequency count, g_i, of an item $i \in I$ is the number of times it appears in the stream S. It is trivial to maintain all exact item frequency counts if we are allowed to use $|I|$ counters. But, the algorithm in Figure 3.1 can assure that $O(l)$ space ($l = 1/\phi$) is used in the worst case and all items with $g_i > \phi n$ are in the output set K'. Furthermore, $\forall i \in I, g_i - \hat{g}_i \leq n/l$ are obtained where $\hat{g}_i$ is the approximate frequency count of an item. The key operation in the algorithm is deleting one appearance of each item in the counters if $|K'| > 1/\phi$. The update operations assure that the stream summary is always stored in at most l counters, and

$\hat{g}_i$ can be a useful approximation to g_i even if $\hat{g}_i = 0$. The usefulness of this idea for finding frequent item co-occurrences is shown in Section 3.4.

```
Input S: A stream of items, φ: support threshold
Output K′: {i ∈ I : g_i > φn} ⊆ K′
K′ ← ∅, counter ← [ ] ;
for all item s in S do
  if s ∈ K′ then
    counter[s] ← counter[s] + 1 ;
  else
    K′ ← K′ ∪ {s} ;
    counter[s] ← 1 ;
  end if
  if |K′| > 1/φ then
    for all item a ∈ K′ do
      counter[a] ← counter[a] − 1 ;
      if counter[a] = 0 then
        K′ ← K′ \ {a} ;
      end if
    end for
  end if
end for
```

Figure 3.1. Single pass FREQUENT algorithm.

SPACESAVING requires the following modification to the algorithm in Figure 3.1: If the next item, s, is not in K' and $|K'| > 1/\phi$, instead of decrementing every counter by 1, it inserts this item with a value $min + 1$ into the counter having minimum value, min. This provides a way for not missing frequent items by erring on the positive side, although the count of a restored item can actually be an integer in interval $[1, min + 1]$. As a result, the algorithm guarantees to find all items with

$g_i > min$ and $min \leq \lfloor n/l \rfloor$ [81], but with the side effect of possibly overestimated frequency counts for some items. Nevertheless, the approach has still some practical implications such as better approximations to the frequency counts of top-k items in skewed item distributions, because it always alters the item with the minimum count. In the rest of this chapter, we continue to work with both FREQUENT and SPACESAVING since they allow strict bounds on space complexity and useful bounds on approximation errors.

We now show that both FREQUENT and SPACESAVING can be efficiently implemented using the generic data structure illustrated in Figure 3.2. The data structure also has the useful property that it holds items always in sorted order with respect to their approximate frequency counts which enables efficient top-k queries. These properties make the data structure convenient for the collaborative filtering approach proposed in Section 3.4. The data structure uses a hash table which, instead of directly holding counts, points to a doubly linked list of values. FREQUENT uses a linked list with a node value showing numeric difference from the left node value, where the leftmost node shows difference from zero. The item lists attached to nodes facilitate efficient queries. When a list is empty, the node can be deleted. It can be seen that all update operations are performed at most on a single node and its neighboring nodes. The key operation, decrementing every counter, is simply achieved by decrementing the value of the leftmost node. SPACESAVING can use the same structure with the exception that a node value directly shows a count. The min value is always kept in the leftmost node which assures its constant time retrieval and update. Figure 3.2 also illustrates the following example: Assume $l = 4$ and the 4 items have counts 2, 3, 2, 3, respectively. In this case, for FREQUENT, the values are $v_1 = 2$ and $v_2 = 1$. To decrement every counter by 1, v_1 is reduced to 1. For SPACESAVING, the values are $v_1 = min = 2$ and $v_2 = 3$. Two items with the min value are contained in the leftmost node. If a new item comes from the stream, one of them can be randomly removed and the new item is inserted into the second node which has value $min + 1$.

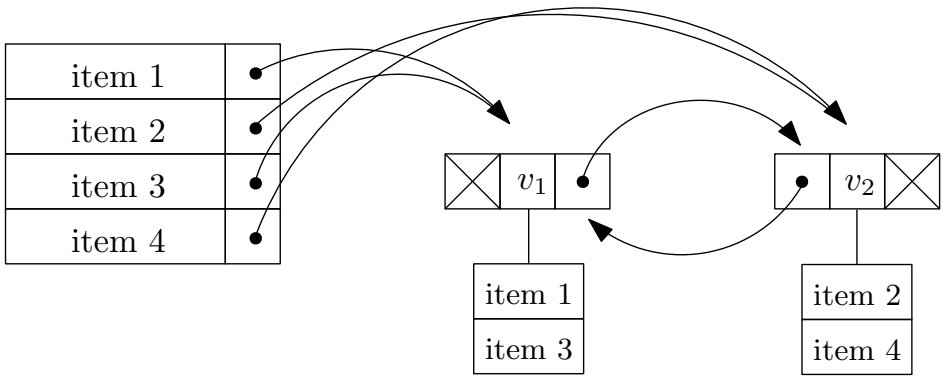

Figure 3.2. Generic data structure for FREQUENT and SPACESAVING for efficient updates and top-k queries.

3.4. Frequent Co-occurrences and SASCF

In this section, we extend the ideas in Section 3.3 to finding frequently co-occurring items. The resulting structure can enable collaborative filtering over the user feedback stream and render the similarity matrix, **S**, unnecessary by maintaining frequently co-occurring items in place of it.

More specifically, we assume a list holding the stream summary structures for every item seen in a user feedback stream. Each summary structure is based on the generic data structure given in Figure 3.2 where the counts now refer to an item's approximate number of co-occurrences with other items. We call this the item co-occurrence list, IL and let IL_i denote the summary structure of an item i. It can be shown that the FREQUENT-based implementation of IL_i has the following guarantees:

Lemma 3.1. *Let $K = \{j \in IL_i : g_{j|i} > \phi n_i\}$ where $g_{j|i}$ is the co-occurrence frequency count of an item j with item i and n_i is the number of all co-occurrences of item i or equivalently the size of its co-occurrence stream, S_i. Then, $|K| < 1/\phi = l = |IL_i|$.*

Proof. Otherwise, there would be more than $1/\phi \times \phi n_i$ co-occurrences of items from K in S_i, which is impossible. □

Lemma 3.2. *Consider an item $j \notin IL_i$. Then, $g_{j|i} < \phi n_i$.*

Proof. Each co-occurrence of j was deleted with $l-1$ other items. Therefore, $g_{j|i} l < n_i$ or $g_{j|i} < \phi n_i$. □

Lemma 3.3. *The upper bound for approximation error $g_{j|i} - \hat{g}_{j|i} \leq n_i/l$, where $\hat{g}_{j|i}$ is the approximate co-occurrence frequency count of item j in IL_i.*

Proof. Assume that $g_{j|i} - \hat{g}_{j|i} \leq d$, where d is the total number of times deletion condition occurs. Each deletion decrements the count of a distinct item by at most 1 and deletes from l distinct items. Therefore, $dl \leq n_i$ or $d \leq n_i/l$. □

Lemma 3.4. *By using the data structure in Figure 3.2, an update operation is constant time and the top-k retrieval operation is $O(k)$ since the linked list holds items in sorted order.*

Theorem 3.5. *FREQUENT-based implementation of IL_i assures the following: At any time, a worst case $O(l)$ space ($l = 1/\phi$) is used and all frequent co-occurrences are maintained. For all items $j \in I$, the approximation error of co-occurrence frequency count is bounded by n_i/l. Each update to IL_i is constant time and the retrieval of top-k co-occurrences from IL_i is $O(k)$.*

Proof. Follows from Lemmas 1, 2, 3, and 4. □

Lemma 3.4 also holds for SPACESAVING-based implementation of IL_i, and we refer to [81] for similar guarantees to the first two lemmas. On the other hand, as mentioned in Section 3.3, some items including the false positives may have overestimated co-occurrence frequency counts when SPACESAVING is used. The case of false positives can be tested using an auxiliary bookkeeping variable and at the expense of increased time complexity [81]. Nevertheless, SPACESAVING-based implementation of IL_i can also achieve good approximations for the top-k co-occurrences in skewed

distributions, and it is still of practical importance. We discuss this situation further in Figure 3.4 at the end of this section.

Now, we assume a stream of tuples (u, i, t) where $u \in U$, $i \in I$. A tuple implies that the item is relevant to the user. A timestamp, t, is used to keep track of temporal order while processing the stream. By making use of the known relevant items to a user and IL, the stream is processed as follows: With the arrival of each (u, i, t), the stream summaries of items known to be relevant to the user u are updated by inserting item i. Furthermore, the stream summary of item i is created or updated by inserting items known to be relevant to the user u. For this purpose, we allow using a set $I_u^{++} \subseteq I_u^+$ for flexibility in different applications. Note that the update operations take $O(|I_u^+|)$ time in the worst case and often in practice $|I_u^+| \ll |I|$. Finally, item i is inserted into I_u^+. Performing the required updates in IL and I_u^+ corresponds to training a scalable and adaptive stream collaborative filtering algorithm which we name as SASCF. The whole procedure is summarized in Figure 3.3.

The hash function $h(.)$ and the hyperparameter α/β in Figure 3.3 constitute an optional scheme to achieve a representative sampling of item co-occurrences. This scheme can be instrumental in managing scalability of the algorithm especially if the stream is too massive. Assume first a naive sampling scheme which generates a random integer in range $[0, 9]$ for each incoming tuple and performs updates with the tuple if the outcome is 0. With a very large stream, the law of large numbers will assure a sample in which about 1/10th of the frequency count of each item is observed. However, this scheme is not very useful for sampling co-occurrences. Assume all users have interacted exactly with two items resulting in a single co-occurrence. Then, the expectation is that only 1/100th of co-occurrences will be sampled. Therefore, we must strive to pick 1/10th of users rather than tuples. To achieve this goal efficiently, we can select a hash function $h(u, 10)$ which maps users in the system randomly to 10 buckets [61]. We sample all tuples from users mapping to a certain single bucket and ignore all other users. More generally, we can obtain a representative sample consisting of any rational fraction α/β of the users by hashing users to β buckets, 0 through $\beta - 1$, and sampling a tuple (u, i, t) if the hash value $h(u, \beta) < \alpha$. Specifically, if $\alpha = \beta$, there will be no

Input S: A stream of tuples (u, i, t), l (or ϕ), α/β: sampling parameter
Output I_u^+ for $u \in U$ and IL
$IL \leftarrow [\ ]$;
for all tuple (u, i, t) in stream S **do**
  if u is new **then**
    $I_u^+ \leftarrow \emptyset$;
  end if
  if there is no IL_i in IL **then**
    Initialize IL_i using data structure in Figure 3.2 with l counters ;
    Add IL_i to IL ;
  end if
  if $h(u, \beta) < \alpha$ and $i \notin I_u^{++} \subseteq I_u^+$ **then**
    for all item $j \in I_u^{++}$ **do**
      Update IL_j with i ;
      Update IL_i with j ;
    end for
  end if
  $I_u^+ \leftarrow I_u^+ \cup \{i\}$;
end for

Figure 3.3. Training with SASCF algorithm.

sampling. We note that in practice α and β are not two different hyperparameters, but together they define a single hyperparameter, that is, the sampling ratio, α/β, for the stream.

For choosing l (or ϕ), our key observation is that the frequency counts of item co-occurrences in ranked order often follow a power-law relationship [85] with respect to the rank. Both FREQUENT and SPACESAVING are expected to yield good approximations for topmost items using a low number of counters when the co-occurrences show this behavior. This situation is illustrated in Figure 3.4 for different datasets.

We see that the true ranks are captured by both FREQUENT- and SPACESAVING-based IL_i. The former captures a more transient relationship at the expense of increased approximation errors for topmost frequent co-occurrences. The latter captures better approximations for topmost frequent co-occurrences, but constantly increasing *min* count results in a thicker tail. More implications of these observations will be discussed in Section 3.5.

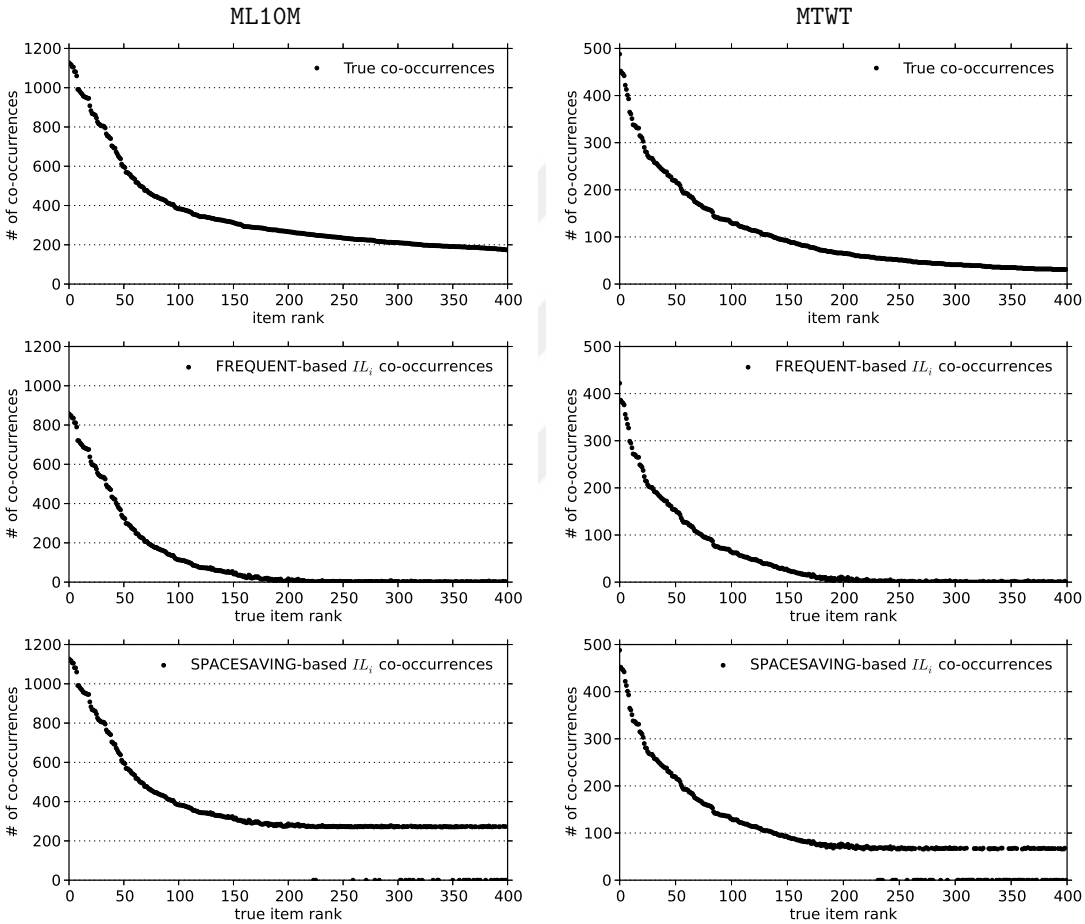

Figure 3.4. (Top row) True co-occurrences in ranked order for representative items. (Middle-Bottom rows) Approximate co-occurrence frequency counts with FREQUENT- and SPACESAVING-based IL_i, respectively.

Finally, we wrap up the potential benefits of using IL instead of a similarity matrix: First, the full similarity matrix can be large and dense, which complicates the nearest neighbor search process. This way, we can fix and compress the size in one dimension keeping interesting co-occurrences only. The expectation is that $l \ll |I|$

for obtaining a useful set of top-k co-occurrences. Second, many uninteresting co-occurrences are automatically filtered in the update process which eases finding top-k frequent co-occurrences, hence the nearest neighbors. Actually, when using the generic data structure in Figure 3.2, querying for top-k frequent co-occurrences is $O(k)$ since the co-occurring items are already kept in sorted order with respect to their approximate frequency counts. Third, counters are always up-to-date with the recent user feedback, and no offline training is necessary.

3.4.1. Effective Personalized Recommendations

SASCF enables effective personalized recommendations by making use of similarities based on the readily available and up-to-date top-k frequent co-occurrences in IL. For recommending to a user u, the set of user's previously known relevant items, I_u^+ or possibly a subset of it is considered. For each item $i \in I_u^+$, its top-k frequently co-occurring items are retrieved from IL_i. Typically, if a retrieved item is not already in I_u^+, it is assumed to be a candidate item, i_c, for recommendation, and its similarity, $s(i, i_c)$, is recorded. Before we carry on, we discuss how such a similarity can be measured: First, we can directly use the approximate frequency count of co-occurrences between the two items, $count(IL_i, i_c)$, as similarity and assume $count(IL_i, i_c) = 0$ when i_c is not in IL_i. In a way, this can be thought of as approximating the dot product between two binary column vectors in a user-item relevance matrix, $\mathbf{Y} \in \mathbb{R}^{|U| \times |I|}$,

$$s(i, i_c) = \mathbf{y}_i^\top \mathbf{y}_{i_c} \approx count(IL_i, i_c). \tag{3.1}$$

Alternatively, similarity can be based on conditional probability (or confidence) [44]. It is easy to obtain this type of similarity because the denominator term corresponds to the frequency count of item i in the stream, and a simple accumulator can keep track of it. Therefore,

$$s(i, i_c) = p(X_1 = i_c | X_2 = i) = \frac{p(X_1 = i_c, X_2 = i)}{p(X_2 = i)} \approx \frac{count(IL_i, i_c)}{count(i)}. \tag{3.2}$$

Yet another similarity measure can be obtained by adding the frequency count of i_c in the denominator to remedy for increased similarity to popular candidate items. One such measure is cosine similarity,

$$s(i, i_c) = \frac{\mathbf{y}_i^\top \mathbf{y}_{i_c}}{\|\mathbf{y}_i\| \|\mathbf{y}_{i_c}\|} \approx \frac{count(IL_i, i_c)}{\sqrt{count(i) \times count(i_c)}}. \tag{3.3}$$

However, maintaining frequent co-occurrences may not fully capture this last similarity. The denominator term should also be considered when updating IL which can be costly in the data stream model. Here, we refer to two stream sampling approaches [86,87] to approximate such similarities. But, we do not investigate them further in this chapter, since it is not straightforward to adapt them in SASCF. Furthermore, their space and time complexities are higher. Instead, we stick to a naive approach and compute cosine similarities of top-k frequently co-occurring items in an IL_i. This is possible since we already have the nominator, and it is cheap to obtain the denominator terms from simple accumulators at any point in time.

By using one of the above similarity measures, the relevance of each candidate item can be predicted as,

$$\hat{y}_u(i_c) = \sum_{i=1}^{|I_u^+|} s(i, i_c). \tag{3.4}$$

Then, personalized recommendations are obtained by sorting the candidate items in descending order with respect to their predicted relevance scores. We note that for various practical reasons, a queried user's known relevant items, I_u^+, can be restricted, for example, to a few most recent feedback instances or to a few feedback instances available in a particular session.

3.4.2. Complexity and Hyperparameterization

In general, SASCF is a scalable and adaptive algorithm which provides a way to work on streaming data. On the other hand, a tidy comparison to its base of-

Table 3.1. Comparative complexity and hyperparameterization.

	SASCF	OFFLINECF
Space complexity	IL requires $O(\|I\| \times l)$ space	$\mathbf{S} \in \mathbb{R}^{\|I\| \times \|I\|}$ and a matrix of $\|I\| \times k$ NN are required
Time complexity (model building)	For each streaming tuple, $O(\|I_u^+\|)$ updates can be performed in IL. Each update is constant time.	$O(\|I\|^2 \times \|U\|)$ offline multiplications to find $\mathbf{S}$ from $\mathbf{Y}$. Then, $O(\|I\|^2 \times \log(\|I\|))$ operations for sorting and finding top-k NN for each item.
Time complexity (personalized top-N prediction)	$O(k \times \|I_u^+\|)$ for a queried user u	$O(k \times \|I_u^+\|)$ for a queried user u with precomputed k NN
Hyperparameters (for personalized top-N prediction)	k, N, l (or ϕ), α/β	k, N
Similarity measures	dot product, conditional probability, cosine (naive)	dot product, conditional probability, cosine

fline algorithm, OFFLINECF, can still be useful. Table 3.1 presents this comparison with respect to the worst case time and space complexities as well as differences in hyperparameterization.

3.5. Experiments

This section presents an empirical analysis of SASCF from various aspects. We begin with exploratory data analysis and then report detailed performance evaluation results in various experimental settings.

Table 3.2. Basic properties of datasets.

Dataset	$\|U\|$	$\|I\|$	# of tuples	Description
`ML10M`	71,567	10,681	10,000,054	User feedback on movies over time [5]
`MTWT`	35,894	20,419	368,490	Snapshot of MovieTweetings [88] social media stream containing user feedback on movies
`KOSARAK`	990,002	41,270	8,019,015	Session-based click stream data from an online news portal [89]. Each session is assumed to be a separate user.
`AMAZON`	2,146,057	1,230,915	5,838,041	User feedback on online shopping products over time [90]

3.5.1. Data and Exploratory Analysis

The real-life public data used in the experiments is summarized in Table 3.2. In all cases, we assume that an item is relevant to a user if the user has feedback for it. Each feedback instance is considered a tuple, and the time order of tuples is preserved.

At this point, it is useful to carry out an exploratory data analysis considering the stream processing model of SASCF. Figure 3.5 illustrates the results of this analysis compactly. The leftmost column shows the frequency counts of co-occurrences in ranked order for different representative items. Together with the middle column, we observe that the co-occurrences follow a power-law relationship with respect to the rank. In accordance with the theoretical analysis [81] for different rank-frequency distributions, as the distribution has a steeper peak, ϕ can be increased and, as a result, l can be smaller. Even if $|I|$ of data is relatively larger, when the frequency counts of co-occurrences in ranked order show a more transient pattern, still a comparable or even a smaller l can be enough to assure all frequent co-occurrences. To further

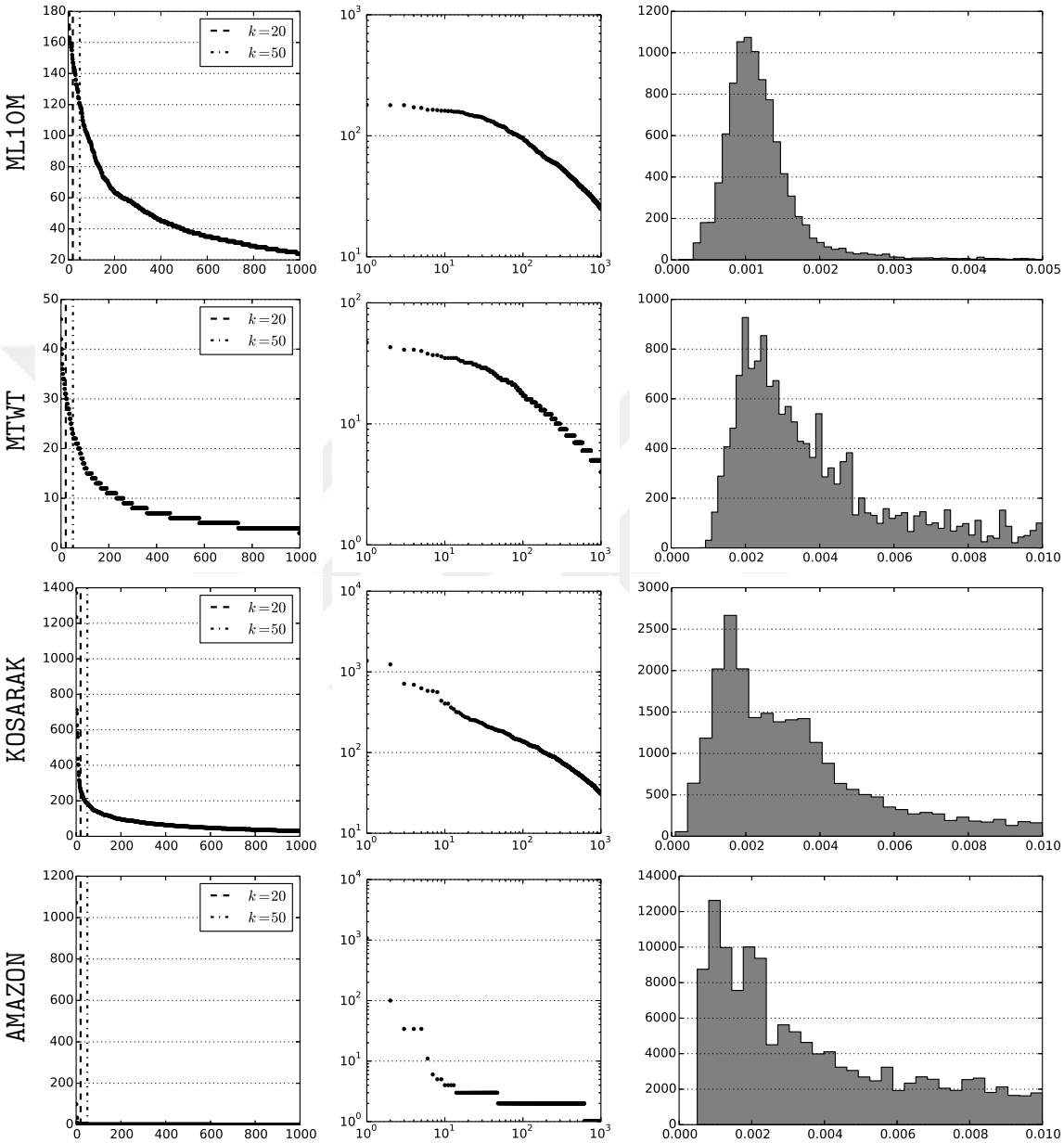

Figure 3.5. (Left column) Rank vs. frequency count of co-occurrences for representative items. top-k-th frequent co-occurrences are also shown for common choices of k. (Middle column) log-log plots of the left column (Right column) ϕ_k distribution over all items for $k = 20$.

justify this observation, we define ϕ_k as the support threshold to guarantee obtaining the top-k-th frequently co-occurring item. The rightmost column of Figure 3.5 shows that the ϕ_k distribution over all items has a distinguishing mode. This is important because it allows usage of a fixed ϕ in SASCF. For example, when we choose to use $\phi = Mo(\text{pmf}(\phi_k))$, items on the right hand side of the mode are guaranteed to capture all of their true positive co-occurrences. Furthermore, items on the left hand side of the mode may not be very interesting since their co-occurrences possibly have a more flat distribution rather than a power-law relationship. In such cases, finding frequent co-occurrences is not very meaningful either. Experimental results in Section 3.5.2 support this further, where we observe that choosing ϕ around the mode often achieves the best performance results, and a smaller ϕ, that is, a larger number of counters, does not bring much improvement, if any. Therefore, we conclude that a fixed value for the hyperparameter ϕ can be estimated through validation or if we have assumptions about the particular power-law behavior. Furthermore, it can be chosen with respect to the available system resources as shown in Section 3.5.2.

3.5.2. Performance Evaluations

Two experimental settings are designed to evaluate the performance of SASCF:

(i) The first setting uses a holdout set with two goals: One is to test the effects of different hyperparameters on a wide range. The tested hyperparameters are ϕ (or the corresponding l), the underlying algorithm for IL (FREQUENT or SPACESAVING), the similarity measure (dot product, conditional probability, or cosine), and the sampling ratio (α/β). The second goal is to compare SASCF to OFFLINECF. In this setting, the expectation is to obtain comparable ranking accuracy results.

(ii) The second setting employs a first-test-then-train sequential scheme which can be more suitable for evaluating stream algorithms. The algorithms are compared using the best hyperparameter values obtained from the former experimental setting. The primary goal is to test the adaptivity of SASCF in comparison to

periodically performed OFFLINECF over the stream and also an incremental single pass matrix factorization algorithm.

3.5.2.1. Experiments Using a Holdout Set. For each user who has feedback for more than two items, we randomly leave one item out from the user's feedback history and put it in a holdout set. If the feedback instances have grade information, the left-out item is chosen among the ones with the maximum grade for the particular user. Then, we use HR@N and MRR to evaluate the test results. The details of these evaluation measures can be found in Section 2.2.5. In the case of MRR, we assume zero reciprocal rank if $\hat{r}_u(i_u^\star) > N$ where $i_u^\star$ refers to the left-out item.

Figure 3.6 presents the experimental results with respect to different similarity measures as explained in Section 3.4.1 and for a wide range of ϕ values. We refer to SASCF-F and SASCF-S as the FREQUENT- and SPACESAVING-based SASCF, respectively. Comparative results for OFFLINECF are illustrated with dashed horizontal lines since they do not depend on ϕ. In the first two columns of Figure 3.6, we observe that the results are often comparable to those of OFFLINECF for a given range of ϕ values. Further increasing the ϕ value enables usage of a smaller number of counters ($l = 1/\phi$), but HR@N and MRR results may also start to decline. In the third column, we observe that our naive approach to cosine similarity is effective but in a narrower range of ϕ values with a faster decline in HR@N values especially when SASCF-S is used. On the other hand, it should be noted that cosine similarity is not always the best choice among others. We observe in all figures except those of `ML10M` that SASCF-F is often better than SASCF-S in terms of both HR@N and MRR. Although SPACESAVING offers to maintain better approximations to frequency counts of the top-k items, FREQUENT's resulting co-occurrence pattern (as shown in Figure 3.4) seems to achieve a more useful weighting when calculating similarities. Finally, `AMAZON` results are provided for the proof of resource awareness concept where we vary the number of counters in IL (hence ϕ) proportional to the available main memory in our testing machine and still observe acceptable performance results. This observation

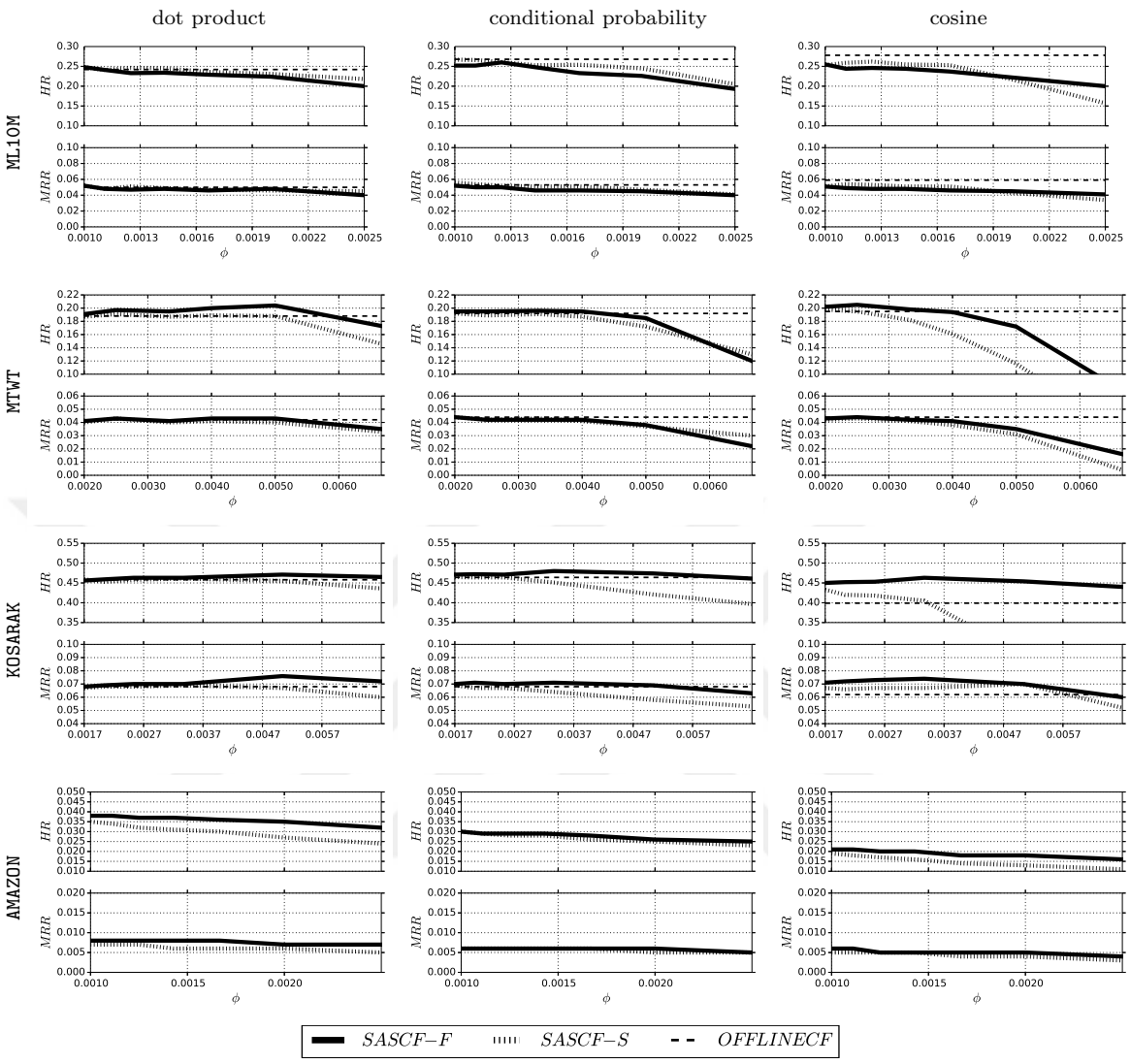

Figure 3.6. HR@N and MRR results for a broad range of ϕ values and different similarity measures. $k = 20$, $N = 10$.

is also valid for the other datasets since comparable performance results are obtained over a wide range of ϕ values.

Figure 3.7 shows the effect of sampling on ranking accuracy for a wide range of sampling ratios (α/β) and different similarity measures. For measuring this effect, other hyperparameter values are fixed to their best and SASCF-F is used. MRR results are not shown for clarity since they are usually highly correlated with the HR@N results. The results in Figure 3.7 are interesting since the accuracy trade-off is often small even when the data streams are heavily sampled. This brings further scalability to SASCF.

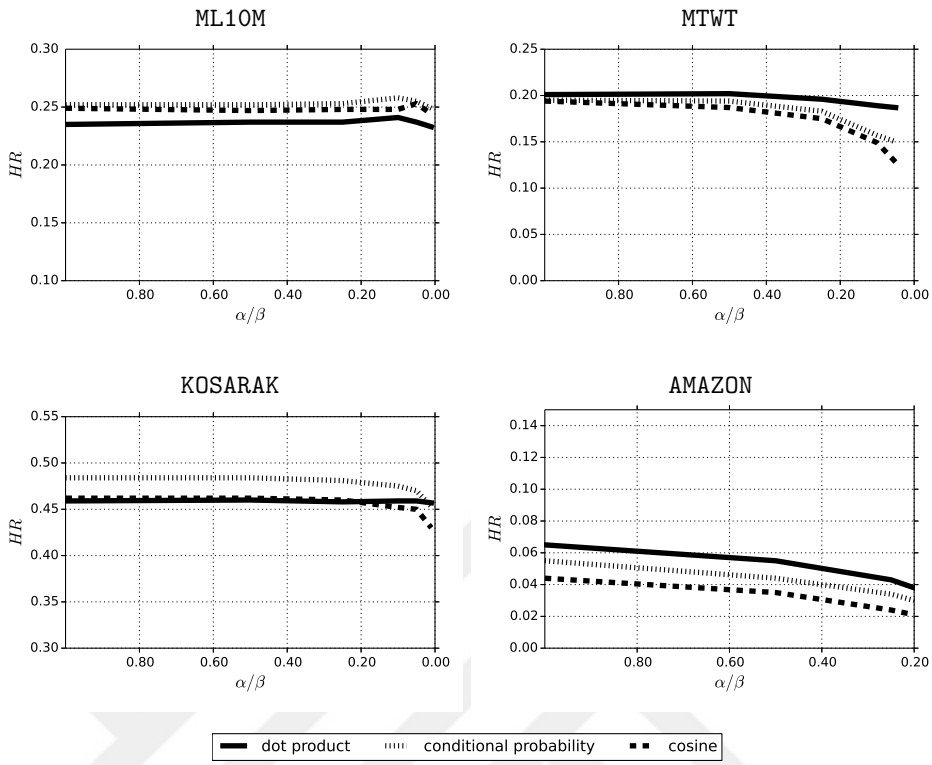

Figure 3.7. Effect of sampling on ranking accuracy. $k = 20$, $N = 10$. Number of counters, l, are 900, 250, 500, and 1,000 for `ML10M`, `MTWT`, `KOSARAK`, and `AMAZON`, respectively.

3.5.2.2. Sequential Evaluation. We use a sequential evaluation scheme similar to [75, 91] and suitable for testing stream algorithms. The procedure is described in Figure 3.8. This time there are no separate training and test sets. All streaming tuples are used for incremental updates in a first-test-then-train fashion [74]. Similar to HR@N, we measure a hit ratio over all streaming tuples whose user has previously provided feedback for another item. In case the stream has graded feedback information, only tuples with a grade above some threshold ρ can be used for testing.

Sequential evaluation results in comparison to OFFLINECF are shown in Figure 3.9. For each experiment, we report two results. The first one is obtained by using SASCF directly through the algorithm in Figure 3.8. The second result is obtained by adapting OFFLINECF to the sequential evaluation setting: Instead of updating the stream summary structures of SASCF, a full item similarity matrix is sequentially

```
Define a sliding window of size W ;
hits ← 0, recommended ← 0, w ← 0 ;
for all tuple (u, i, t) in stream S do
  w ← w + 1 ;
  if I_u^+ ≠ ∅ and tuple grade > ρ then
    Recommend top-N items, Î_u^N ;
    recommended ← recommended + 1 ;
    if i ∈ Î_u^N then
      hits ← hits + 1 ;
    end if
  end if
  if w = W then
    /* Report for the current window */
    hit ratio ← hits / recommended ;
    hits ← 0, recommended ← 0, w ← 0 ;
  end if
  Update SASCF model with (u, i, t) (as per Figure 3.3) ;
end for
```

Figure 3.8. First-test-then-train procedure for sequential evaluation.

updated with the streaming tuples. However, in this case, nearest neighbors of an item are not immediately available for recommendation purposes. Therefore, periodic offline nearest neighbor search is performed on the similarity matrix at regular intervals, and k nearest neighbors of every item are recorded until the next search. The following hyperparameters are used for the experiments: For all streams, $k = 20$ and $N = 10$. Conditional probability is used as the similarity measure. For `ML10M`, $\phi = 0.00\bar{1}$ ($l = 900$) is fixed for SASCF. Periodic retraining for OFFLINECF is performed every 60,000 tuples. $W = 10,000$. For `MTWT`, $\phi = 0.004$ ($l = 250$) is fixed for SASCF. Periodic retraining for OFFLINECF is performed every 2,000 tuples. $W = 1,000$. For `KOSARAK`, $\phi = 0.002$ ($l = 500$) is fixed for SASCF. Periodic retraining for OFFLINECF is performed every

50,000 tuples. $W = 10,000$. For AMAZON, $\phi = 0.001$ ($l = 1000$) is fixed for SASCF. Periodic retraining for OFFLINECF is performed every 30,000 tuples. $W = 10,000$.

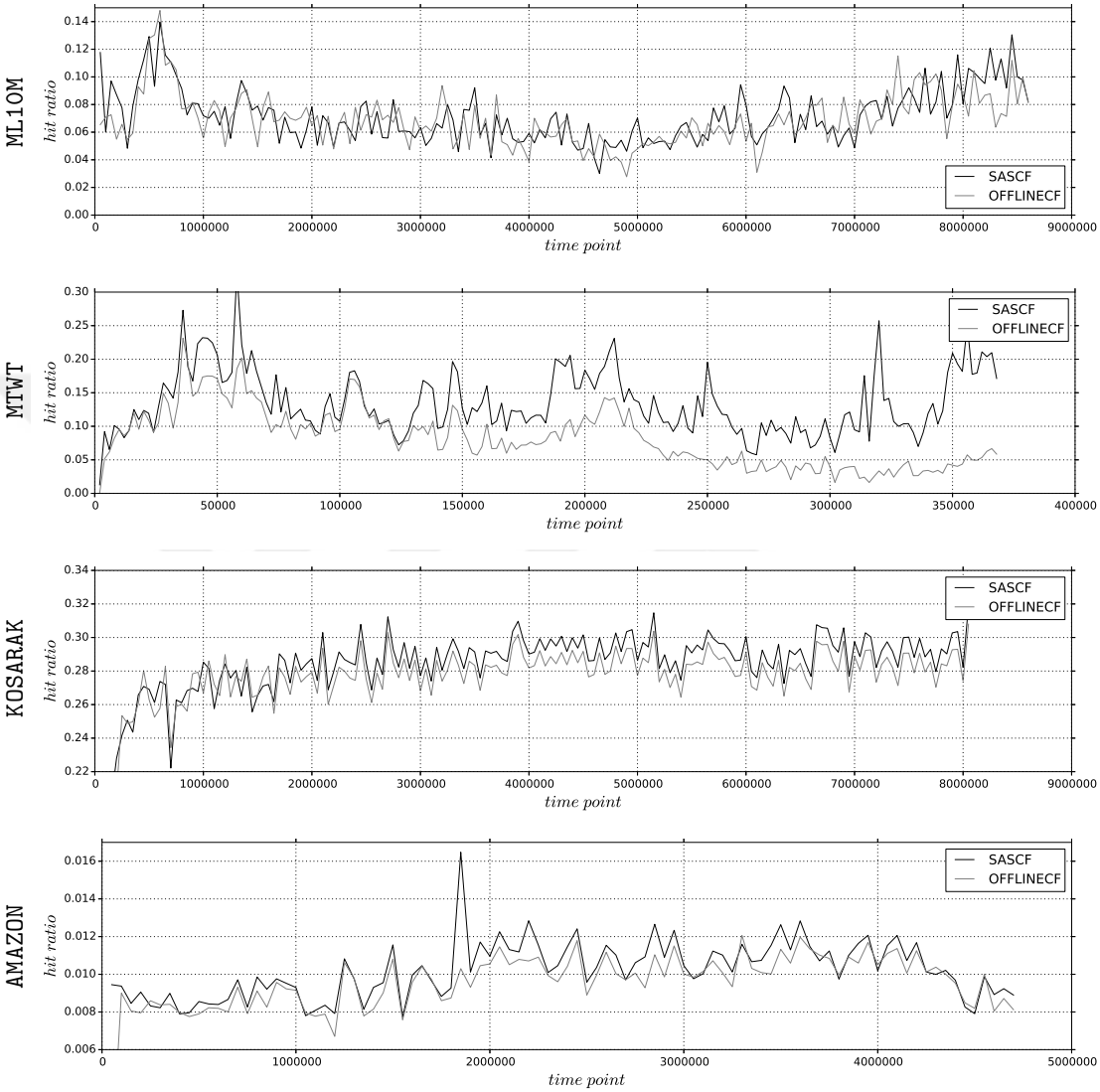

Figure 3.9. Comparative sequential evaluation with OFFLINECF.

The experimental results in Figure 3.9 suggest that SASCF is at least as good as OFFLINECF in terms of hit ratio and often significantly better. This result is expected since SASCF is more adaptive to recent user feedback. OFFLINECF results show various degrees of degradation with MTWT being the strongest. We note that these results are sensitive to the length of training intervals, and the degradations tend to increase even more when the periodic retraining is done less regularly. Although characteristics of each stream may vary, the results suggest usefulness of SASCF in which the latest

user feedback can be more quickly integrated, and the space requirements are more controllable. The statistical significance of the sequential evaluation results is tested using a Wilcoxon signed rank test [92] as follows: At each sliding window, the error of an algorithm is defined as the miss ratio (or $1 -$ hit ratio). The null hypothesis is that paired miss ratios from the two algorithms come from the same distribution. For a significance level of 1%, we fail to reject the null hypothesis for `ML10M`, but we reject it for `MTWT`, `KOSARAK`, and `AMAZON` where rejection means there is statistical significance between the errors of the two algorithms.

We finally compare SASCF to an incremental matrix factorization approach called ISGD (incremental stochastic gradient descent) [75]. Similar to our work, ISGD tries to learn from streaming tuples in a single pass and facilitate top-N recommendations. Figure 3.10 shows comparative results of our experiments. Again, we make use of the evaluation scheme in Figure 3.8, except this time the ISGD model is updated.

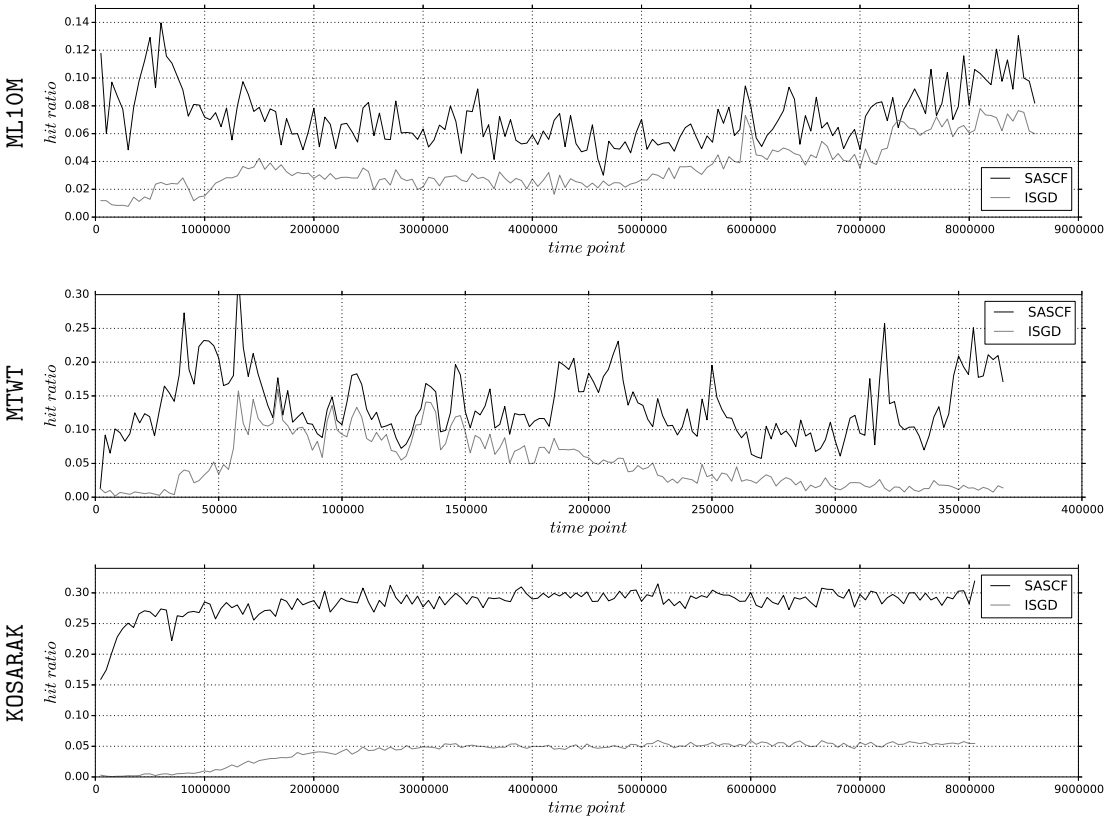

Figure 3.10. Comparative sequential evaluation with ISGD.

We first observe that ISGD takes some time at the beginning of the stream to converge before producing comparable hit ratios. For `ML10M`, we observe that ISGD hit ratios have an improving trend with $\gamma = 0.005$, $\lambda = 0.05$, and $f = 10$. For `MTWT`, $\gamma = 0.04$, $\lambda = 0.01$, and $f = 10$ produce competitive hit ratios after initial convergence, but then the results show a degradation trend beginning towards the middle of the stream as newer users and items are introduced. For `KOSARAK`, $\gamma = 0.008$, $\lambda = 0.002$, and $f = 10$ produce comparatively low hit ratios on session-based user data. These hyperparameter values are found by validation in a limited search space but in general, such values for SGD-based approaches may not be trivial to fine-tune, especially for single pass stream learning. For example, in the case of `AMAZON`, we are unable to observe convergence for ISGD with the hyperparameter values from a limited grid search (SASCF results are shown in Figure 3.9). In all experiments, we observe that ISGD and SASCF results are significantly different both visually and statistically. Overall, we note that the updates of ISGD can be quite fast when single pass updates and a small number of latent factors are used. As in other MF-based approaches, its space complexity is attractive when a small number of latent factors is sufficient. On the other hand, the effect of growing number of items on ISGD needs further investigation since it both affects the comparative hit ratios and the computational cost of prediction. The latter may even necessitate offline precomputation of the recommendation lists although the learning is continuously done from the stream. Since the convergence with many model parameters takes time, the adaptivity of ISGD also needs further investigation for streams especially with very dynamic users and items, and in the cases where users or items have a few feedback instances. While we propose improvements for incremental MF schemes in Chapters 4 and 5, neighborhood-based methods like SASCF can clearly be considered as strong competitors for their ability to handle these situations effectively.

4. PARALLEL PERSONALIZED PAIRWISE LEARNING TO RANK

4.1. Introduction

Pairwise LtR is particularly suitable for learning from implicit feedback. For example, it differs from the typical pointwise LtR approaches to implicit feedback where all user-item relevances are given a positive label, and unknown relevances are either ignored or given a negative class label which may arguably lead to an undesired learning model [48]. Furthermore, it can allow direct rank optimization based on some desired information retrieval measures such as AUC. Pairwise LtR can also be more efficient compared to listwise LtR since it works on item pairs instead of arbitrarily large lists. Therefore, given the usefulness of pairwise LtR for implicit feedback scenarios, we investigate methods to further improve its efficiency and adaptivity in offline and stream learning settings with a focus on collaborative filtering.

As data and model parameters to be learned grow larger, capabilities of machine learning methods are limited by their time [93] and space efficiencies. Consequently, at web scale, many LtR models are based on learning a factorized representation of a user-item relevance matrix. A commonly preferred optimization method for such a matrix factorization is stochastic gradient descent (SGD) [94] due to its ease of use, efficiency, and suitability to online learning. However, SGD also has a sequential nature which can still be problematic due to multiple passes over a large dataset while trying to learn a large number of model parameters. As a remedy, the idea of lock-free parallel SGD has been motivated for different machine learning methods and also applied to matrix completion [95] from sparse input. In this direction, two major approaches are based on *block partitioning* [96] and *no partitioning* [97]. These approaches are especially interesting for multi-core CPU or GPU processing, and they can be preferable for intensive large-scale numerical computation.

In this chapter, we investigate major lock-free parallel SGD approaches in a personalized pairwise LtR setting. With the aim of improving its scalability and speeding up its convergence to a good solution, we first adapt two base schemes to this setting: the PLtR-B algorithm which applies a block partitioning approach and the PLtR-N algorithm which follows the no partitioning approach. We then analyze useful extensions of the base schemes to show their versatility and improve their practicality. Such extensions are commonly desired in practice, and they include the following: Using different sampling strategies for adaptive sampling as well as handling graded relevance feedback, incorporating adaptive gradient update methods, and the PLtR-NS algorithm for efficient learning from streaming user feedback. We show that the applicability of these extensions bring more versatility to the base schemes as well as potential for further improvements on convergence results.

The remainder of this chapter is organized as follows: In Section 4.2, an overview of personalized pairwise LtR is presented. Then, in Section 4.3, the base PLtR-B and PLtR-N algorithms are presented. Extensions of the base algorithms are presented in Section 4.4. Finally, in Section 4.5, comparative experimental results are presented for the ranking accuracy and the speedup patterns of the algorithms.

4.2. Personalized Pairwise LtR

Further elaborating on Section 2.2.1, personalized pairwise LtR methods commonly learn from a dataset, $\mathcal{D}$, of triples (u, i, j), where $u \in U$ is the (or part of the) context and $i, j \in I$. A triple commonly implies a partial personalized ranking such that item i is more relevant to user u than item j. In this analysis, unless otherwise stated, we assume that $i \in I_u^+$, and the dataset of all relevant user-item tuples is denoted by $\mathcal{D}^+$. On the other hand, j can be sampled from a conditional probability distribution with $p(j|u, i)$. The basic and typical sampling strategy is to draw j uniformly at random from $I \setminus I_u^+$. While other sampling strategies are also analyzed in this chapter, we stick to this basic strategy unless otherwise stated. The ultimate goal of personalized pairwise LtR is to find a good estimate of full or topmost personalized rankings from available partial rankings.

The typical pairwise error function is in the form:

$$\mathcal{E}\left(\Theta \mid \mathcal{D}\right) = \sum_{(u,i,j)\in\mathcal{D}} \Big\{ L\left(r_u(i) < r_u(j), \theta_u, \theta_i, \theta_j\right) + \sum_{\theta_x \in \{\theta_u, \theta_i, \theta_j\}} \lambda_{\theta_x} \|\theta_x\|_2^2 \Big\}, \tag{4.1}$$

where Θ represents all model parameters, and λ_{θ_x} is a regularization coefficient. Different loss functions, L, are possible. In this chapter, we stick to sigmoid-smoothed pairwise loss and negative log-likelihood described in the seminal work, Bayesian personalized ranking (BPR) [38, 48]. The same ideas apply to learning from other differentiable pairwise loss functions, for example, based on hinge loss [98].

In the case of BPR with matrix factorization, $L = -\ln \sigma(\mathbf{p}_u^\top \mathbf{q}_i - \mathbf{p}_u^\top \mathbf{q}_j)$ where $\mathbf{p}_u, \mathbf{q}_k \in \mathbb{R}^f$ are column vectors in the low-rank component matrices of the factorized user-item relevance matrix, $\widehat{\mathbf{Y}} = \mathbf{P}^\top \mathbf{Q}$. The resulting error function is differentiable with respect to the model parameters, and it can be optimized with SGD using a suitable learning rate, γ. We summarize the typical procedure in Figure 4.1. Note that due to the typically large number of model parameters and sparsely available feedback, the algorithm usually requires multiple passes over the dataset prior to convergence.

repeat
  Sample a tuple (u, i) from $\mathcal{D}^+$;
  Sample j with $p(j|u, i)$;
  for all $\theta_x \in \{\theta_u, \theta_i, \theta_j\}$ **do**
    $\theta_x \leftarrow \theta_x - \gamma \frac{\partial}{\partial \theta_x} \{ L\left(r_u(i) < r_u(j), \theta_u, \theta_i, \theta_j\right) + \lambda_{\theta_x} \|\theta_x\|_2^2 \}$;
  end for
until convergence

Figure 4.1. Personalized pairwise learning to rank.

4.3. Base PLtR Algorithms

4.3.1. Block Partitioning and PLtR-B

Block partitioning refers to partitioning the input to an algorithm into multiple sets of ideally non-overlapping chunks (blocks). Then, different processing units can perform model updates in parallel, each using the available partial input in one of the blocks of every set and without using locks. This approach has been first applied [96,99] to personalized pointwise LtR scenarios. In that case, the model updates are based on (u, i) tuples, and the tuples can be thought to come from a sparsely filled $|U| \times |I|$ matrix of known user-item relevances. In the case of personalized pairwise LtR, the model updates are based on (u, i, j) triples representing pairwise relevances, and we can consider a $|U| \times |I| \times |I|$ tensor instead. Using $\psi = 2$ processing units and representing different sets of blocks with different colors, Figure 4.2 illustrates two possible block partitioning approaches for personalized pairwise LtR and also compares them to the standard approach for pointwise LtR. While tempting, a closer inspection reveals that the block partitioning in the middle does not guarantee non-overlapping blocks in every set of blocks since the same item may appear as both an i and a j item in different blocks of the same set. Therefore, we concentrate on the rightmost partitioning approach which excludes the problematic sets of blocks. This restriction guarantees mutually exclusive updates to the model parameters of both users and items, and it also makes the partitioning computationally more efficient. We explain its applicability next.

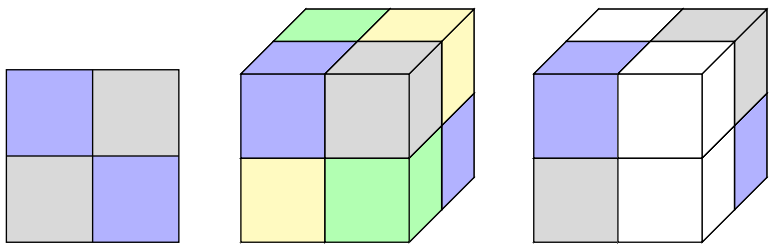

Figure 4.2. (Left) block partitioning for parallel personalized pointwise LtR, (Middle-Right) two different ideas for the case of personalized pairwise LtR (see Figure 4.3 for deciding the sets of blocks for the rightmost partitioning).

By extending the pointwise scheme [96], the following is possible for personalized pairwise LtR: Suppose we have ψ processing units. At each training epoch, first a random permutation of users ($perm_U$) and items ($perm_I$) are generated by shuffling their indexes in place. We then partition (u, i, j) triples into ψ^2 blocks, B_{abc}, where $a, b, c = 1, \ldots, \psi$ such that,

$$a = \left\lfloor \frac{\psi}{|U|} \left(perm_U\left(u\right) - 1\right) \right\rfloor + 1, b = \left\lfloor \frac{\psi}{|I|} \left(perm_I\left(i\right) - 1\right) \right\rfloor + 1,$$
$$c = \left\lfloor \frac{\psi}{|I|} \left(perm_I\left(j\right) - 1\right) \right\rfloor + 1, \tag{4.2}$$

with the requirement that $c = b$. To achieve this, our proposal is to simply sample an item j with $p(j|u, i)$ such that $p(j|u, i) = 0$ if $c \neq b$. Since each sampled (u, i, j) within a block, B_{abc}, is used to perform a sequential SGD update, this is acceptable. Furthermore, due to random permutations, users and items corresponding to a block are different at each training epoch, which makes this restrictive approach viable.

Updates based on each set of blocks are performed in a single round, and each update round in a training epoch can be decided simply by the algorithm in Figure 4.3. Consequently, personalized pairwise learning to rank can be parallelized as in Figure 4.4. We call the resulting algorithm PLtR-B. The algorithm is designed to perform multiple training epochs efficiently by parallelizing its update stage. The partitioning stage can actually be considered as a preprocessing stage. As commented in Figure 4.4, if this stage is not performed in dedicated processing units, the sampling of

```
for z = 1 to ψ  do
  for a = 1 to ψ  do
    c ← b ← (a + z − 1)  mod ψ ;    /* set b ← ψ if b = 0 */
    Add B_abc to Round[z] ;
  end for
end for
```

Figure 4.3. Deciding the set of blocks for each round for arbitrary $\psi \geq 1$.

```
for all training epochs do
  — PARTITIONING STAGE —
  Generate perm_U and perm_I ;
  for all (u, i) in D^+ do
    Get a and b w.r.t. Equation 4.2 ;
    Sample j with p(j|u, i) and place (u, i, j) in corresponding B_abc ;
    /* p(j|u, i) ← 0, if c ≠ b w.r.t. Equation 4.2. Note that sampling of j
    can be postponed until after line 10, if partitioning stage is not performed in
    dedicated processing units. */
  end for
  Assign all B_abc to rounds w.r.t. Figure 4.3 ;
  — UPDATE STAGE —
  for all Rounds do
    for all B_abc in current round in parallel do
      for all (u, i, j) ∈ B_abc do /* line 10 */
        for all θ_x ∈ {θ_u, θ_i, θ_j} do
          θ_x ← θ_x − γ ∂/∂θ_x {L (r_u(i) < r_u(j), θ_u, θ_i, θ_j) + λ_θx ||θ_x||_2^2} ;
        end for
      end for
    end for
    Synchronize processing units ;
  end for
end for
```

Figure 4.4. Parallel personalized pairwise LtR with block partitioning (PLtR-B).

j can be postponed until the update stage for further efficiency. This can also enable the algorithm to sample j more adaptively during the course of actual updates.

Block partitioning can guarantee mutually exclusive updates to the model parameters, but a few practical concerns should be noted: First, the blocks in each round need to have a balanced number of training triples in order to fully benefit from parallelism. Random shuffling as well as a few other tricks [100] for scheduling and better use of memory hierarchy can be useful for improving efficiency in this regard. Second, the partitioning stage can bring some extra computation time overhead which can optionally be overcome at the cost of dedicating separate processing units [96]. Space overhead is also increased due to auxiliary structures. Third, and importantly, block partitioning imposes a slightly restrictive scheme for sampling (u, i, j) triples. Therefore, in the next section, we also investigate the no partitioning approach which can be more versatile for extended pairwise LtR tasks (see Section 4.4), for example, with custom sampling schemes as well as for dynamic stream learning environments.

4.3.2. No Partitioning and PLtR-N

In this approach, parallel processing units can access and update any portion of a shared memory at any time and in a lock-free fashion [97]. Nevertheless, this extreme approach still provides some theoretical guarantees for convergence when the optimized function can be defined as sparse summations, typical of SGD-based personalized LtR. In the case of personalized pairwise LtR, the summation has the sparse form given in Equation 4.1. This summation induces a hypergraph, $G = (V, E)$. However, unlike a bipartite graph for pointwise LtR, we now have a hypergraph which reflects personalized pairwise item relationships. An example of this new hypergraph is illustrated in Figure 4.5.

Both $|V|$ and $|E|$ are typically very large with the latter being especially large in the case of personalized pairwise LtR. However, as seen in Equation 4.1, each summation acts on a single $e \in E$, that is, a very small subset of V, which suggests intuitively

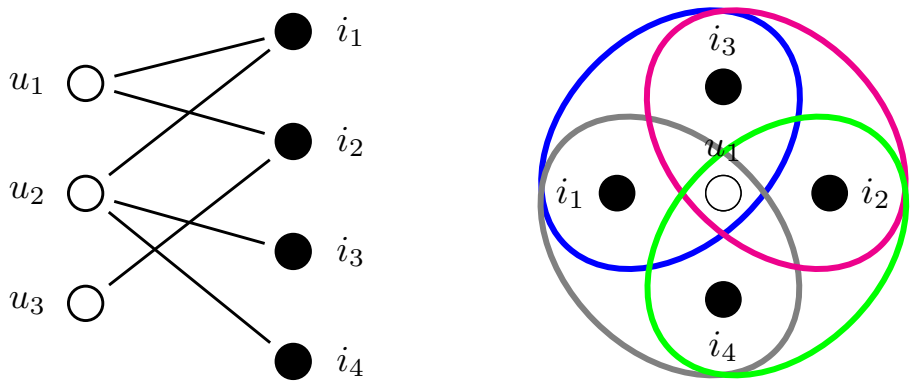

Figure 4.5. Hypergraphs showing user-item relevances for pointwise and pairwise LtR, respectively. For clarity, the latter is shown for a single user only, where the colored hyperedges capture all possible pairwise item relationships for u_1. Here, $i_1, i_2 \in I^+_{u_1}$.

that lock-free parallel updates without any partitioning are viable. This notion can be formalized [97] using the following statistics of a hypergraph, G:

$$\begin{aligned} \Omega &= \max_{e \in E} |e|, \\ \Delta &= \frac{\max_{1 \leq v \leq |V|} |\{e \in E : v \in e\}|}{|E|}, \\ \rho &= \frac{\max_{e \in E} |\{e' \in E : e' \cap e \neq \emptyset\}|}{|E|}, \end{aligned} \tag{4.3}$$

where Δ and ρ are measures of vertex regularity and hypergraph sparsity, respectively. When these values are relatively small, the staleness between parallel access and update of model parameters can be compensated, and the parallel updates can bring a highly effective speedup for convergence. In personalized pairwise LtR, $\Omega = 3$, since each $e \in E$ is made up of a (u, i, j) triple. On the other hand, Δ is very much dependent on the dataset. Under realistic but worst case assumptions, we note the following: $|E| \approx \zeta \times |U| \times |I|^2$ where ζ denotes the density of the known user-item relevances. In many datasets, ζ is typically well under 0.01. Since we have (u, i, j) triples, there can be user and item vertices in the hypergraph. Assuming a user has m relevant items, there can be $m \times (|I| - m)$ edges in which the user exists. This quantity is maximized when $m = |I|/2$. Therefore, it can be that $\Delta \approx 1/(4 \times \zeta \times |U|)$ with respect to the user vertices. An item, on the other hand, can occur in approximately $|U| \times |I|$ edges in the

worst case. Therefore, it can be that $\Delta \approx 1/(\zeta \times |I|)$ with respect to the item vertices. Since $|I| < |U|$ in typical real-life applications, we conclude that the item vertices often determine Δ. We also see that Δ for pairwise LtR is on the same order with Δ for pointwise LtR under similar assumptions. Finally, we can say $\rho \leq 3\Delta$, since $\Omega = 3$.

In real-life applications, the lock-free parallel approach without any partitioning is effective even when the values of Δ and ρ are high and close to the worst case, as shown experimentally for different machine learning problems [97] including matrix completion in a pointwise LtR setting. We observe in our experiments that in the personalized LtR problem setting, a possible reason behind this situation is that user and item vertex degrees often follow a power-law distribution, that is, Δ (maximum normalized vertex degree) can be significantly higher than the values in the modal interval of the normalized vertex degree distribution. In Section 4.5.2, we compute these statistics for various datasets and also show that the statistics in the case of pairwise LtR are quite similar to those in the case of pointwise LtR.

```
/* Perform the following loop in ψ parallel processing units */
for local epoch = 1 to ⌈# of training epochs / ψ⌉ do
  for all iterations do
    Sample a tuple (u, i) from D+ ;
    Sample j with p(j|u, i) ;
    for all θx ∈ {θu, θi, θj} do
      θx ← θx − γ ∂/∂θx {L (ru(i) < ru(j), θu, θi, θj) + λθx ||θx||22} ;
    end for
  end for
  Synchronize processing unit if the learning rate, γ, changes ;
end for
```

Figure 4.6. Parallel personalized pairwise LtR with no partitioning (PLtR-N).

Therefore, we propose to parallelize personalized pairwise LtR without partitioning as given in Figure 4.6. We call the resulting algorithm PLtR-N. As an illustrative

example, if 2ψ training epochs of the learning algorithm are desired, then we see that every parallel processing unit will perform 2 local epochs instead of a single processing unit performing all 2ψ epochs in a sequential algorithm. We note that, in general, there is no need for synchronization among parallel processing units while access and update operations of the model parameters are being performed. Nevertheless, the algorithm provides a piecewise-defined process when there is a need to change the globally-set learning rate over time.

4.4. Extensions to PLtR Algorithms

In this section, we analyze the PLtR-B and PLtR-N algorithms in combination with various important strategies, the applicability of which can bring more versatility to the algorithms as well as potential for further improvements on convergence results. Accordingly, we propose several extensions to the algorithms for offline and stream learning settings.

4.4.1. Different Sampling Strategies

The basic strategy in personalized pairwise LtR in Figure 4.1 is to draw (u, i) uniformly at random from $\mathcal{D}^+$. Then, $p(j|u, i)$ is chosen such that j is drawn uniformly at random from $I \setminus I_u^+$. This simulates an unbiased (u, i, j) sample [68, 101]. More recently, there are some proposals which alter this sampling strategy for various reasons. Many of these fall into one of the two categories below:

(i) *Adaptive Sampling.* This refers to biased sampling in an attempt to choose a relatively more suitable (u, i, j) triple adaptively for each consecutive model update. The common approach is trying to obtain a biased j sample efficiently given the model parameters for u and i at the point of update. In an important follow-up research [68] for BPR (see Section 4.2), $p(j|u, i)$ is biased in a way that j is sampled with a higher probability from top ranking irrelevant items for u at the point of update. It is argued that this approach uses more informative (u, i, j) triples and avoids wasteful gradient computations resulting in faster convergence.

A related approach [102] is to sample j from a random subset of irrelevant items which have higher ranking scores than i with a margin. Yet, in another related approach [98], j is sampled from a random but fixed-sized subset, S_u^-, of irrelevant items for user u. $p(j|u,i) \propto |\mathbf{p}_u\mathbf{q}_i - \mathbf{p}_u\mathbf{q}_j|^{-1}$ if $j \in S_u^-$, and 0 otherwise. This means that the closer an item $j \in S_u^-$ to i, the higher its probability of being sampled. It turns out that choosing $|S_u^-| \ll |I|$ still enables potentially good candidates [103]. Furthermore, a closer i and j pair enables large and useful gradient updates similar to the first proposal [68]. We will refer to this final adaptive sampling strategy for our empirical analysis in Section 4.5. In general, although adaptive sampling strategies can improve convergence speed, it is not hard to see that complex sampling schemes themselves can be costly.

(ii) *Handling Graded Relevance Feedback.* User feedback, whether implicit or explicit, can imply some grade of relevance. This grade can be based, for example, on recency or number of interactions as well as some perceived importance of the feedback, or a rating scale. To handle graded relevance feedback, a viable proposal [28] is to switch between two distributions for $p(j|u,i)$ using a biased coin toss. With this scheme, j is sampled from either irrelevant items, $I \setminus I_u^+$, or relevant items with a lower grade than i. We experiment with this scheme in Section 4.5. More complicated sampling schemes are also possible which, for example [104], enable the usage of information from different types of user feedback more finely. In general, biased sampling for handling graded relevance feedback is shown to improve predictive power of pairwise LtR. However, as the complexity of sampling increases, the computational cost of LtR also increases.

To benefit from different sampling strategies, it can be useful to combine them with parallel SGD to compensate for their efficiency problems. It is usually straightforward to extend PLtR-B with these strategies. The only reservation is that the sampling process is restricted to the items available in each block. PLtR-N can be more versatile in this regard. On the other hand, the effect of not drawing (u,i,j) uniformly at random needs further investigation, and it will be discussed in Section 4.5.4.

4.4.2. Adaptive Gradient Updates

In theory, SGD results in convergence to a minimum almost surely when the learning rate is decreased over time [94]. However, some additional tweaks can be instrumental in achieving faster convergence. More recently, it is often cited, especially in the neural networks literature, that adaptive gradient computations which borrow ideas from second order methods and using momentum terms [105] can bring superior convergence results [106, 107]. Since, some neural networks can be seen as universal approximators [39], and many CF models can be expressed as neural networks [108,109], such computations can also be useful in the general LtR setting for CF. Example applications [110–112] are also supportive of this.

Adaptive gradient methods consider the geometry of the optimized function for each parameter and affect the computation of updates at every iteration. A seminal method is AdaGrad [113] which accumulates squared gradients from previous updates of each parameter and scales a new parameter update with the corresponding accumulated sum. This approach potentially works well with sparse input enabling larger updates for infrequent parameters and smaller updates for the frequent ones. Furthermore, the need for extensive validation for choosing a good learning rate is often eliminated, since the learning process is actually guided by the adaptive gradients. The main drawback is that constantly accumulating the previous gradients can cause the gradient updates to diminish through time. Among various alternatives to solving this problem [106], Root mean square propagation (RMSProp) keeps an exponentially weighted moving average (EWMA) of the squared gradient updates by slightly increasing time complexity and without increasing space complexity compared to AdaGrad. RMSProp can also be useful when learning from streams since its averaging scheme can provide some sort of adaptivity to non-stationary distributions [114].

Adaptive gradient methods can be used within PLtR-B and PLtR-N. We concentrate on AdaGrad and RMSProp which both scale the x'-th individual parameter update with $1/\sqrt{\mathbf{G}^t_{x'x'} + \epsilon}$, where $\mathbf{G}$ is a diagonal matrix containing the sum of squared gradients at a time point t. Moreover, ϵ is a fixed smoothing term. While in AdaGrad

the sums are exact, RMSProp uses EWMA with a typical weight around 0.1. In the following, we note two important points regarding a possible extension to the PLtR algorithms using adaptive gradients:

(i) *Space complexity.* In pairwise LtR for CF, we have a minimum of $|U| \times f + |I| \times f$ individual model parameters. In the case of extensions with AdaGrad or RMSProp, maintaining accumulators ($\mathbf{G}$) for adaptive gradient computations normally requires the same amount of extra space. Therefore, whether we extend the PLtR versions or their sequential counterpart, such an extra space is required.

(ii) *Parallelism.* In the case of PLtR-B, since the block partitioning scheme guarantees mutually exclusive model updates, accumulators are also guaranteed to be updated mutually exclusively. Therefore, adaptive gradient methods are directly applicable. In the case of PLtR-N, parallel processing units may try to update the same parameters, although with very low probability as discussed in Section 4.3.2. Therefore, accumulator updates can also be affected with a similar probability. We provide empirical analysis results for the effectiveness of PLtR-N with AdaGrad and RMSProp in Section 4.5.4.

4.4.3. Parallel LtR from Streaming Feedback

In this section, we revisit the stream learning perspective introduced in Section 2.2.4 and further discussed in Chapter 3. From this perspective, latent factor models, which we investigate in this chapter, can be considered to offer relatively space-efficient representations for LtR. Furthermore, each SGD update to such a model often conveniently involves a small number of model parameters. These updates can also be incremental and in parallel which we detail next.

A first idea is to perform a single update to the model with every captured streaming user feedback instance [75]. However, even if applicable, this can yield a poor factorization mainly due to the large number of parameters in such models and the sparsely available feedback (see also Section 3.5.2.2). A larger learning rate can be useful but then the convergence is not guaranteed. Adaptive gradient update methods

discussed in Section 4.4.2 can also be useful, but need to acquire enough feedback for each parameter. Rather than single updates, we propose incremental parallel SGD updates for the stream learning setting which potentially enables faster convergence and makes better use of computational resources in modern systems.

```
/* Perform the following procedure in the stream producing system process */
procedure PRODUCESTREAM
  for all feedback instance t = (u, i, t, ...) from stream do
    Insert t into a buffer, B ;
    if u is new then Insert u into a buffer, B_u ;
    if i is new then Insert i into a buffer, B_i ;
    if B is full or training is timed then
      Serialize B, B_u, B_i ;
      Initiate new buffers, B, B_u, B_i ;
    end if
  end for
end procedure

/* Perform the following procedure in the stream consuming system process */
procedure INCREMENTALPARALLELUPDATE
  Deserialize B, B_u, B_i ;
  Initialize model parameter, θ_u, for all u in B_u ;
  Initialize model parameter, θ_i, for all i in B_i ;
  Apply PLtR-N in Figure 4.6 using B instead of D^+ and ψ parallel processing
  units (stream consumers) ;
end procedure
```

Figure 4.7. PLtR-N for streams (PLtR-NS).

Our proposal is an extension to PLtR-N in Figure 4.6, and we define it in Figure 4.7. The PLtR-NS algorithm requires maintaining a buffer, $\mathcal{B}$, over the streaming feedback in the system. This buffer is filled with the incoming feedback by a stream

producing system process. When it is full, or the training is timed, the buffer is serialized. Then, whenever available, a stream consuming system process deserializes the buffer, performs model updates using parallel processing units (stream consumers), and then waits for the next buffer. Each such update session warm starts with the model parameter values learned in the previous session, and then the incremental updates are performed. Each stream consumer can perform up to $\lceil \frac{\text{\#of training epochs}}{\psi} \rceil$ epochs over the buffer. For example, if we choose to perform one epoch in each consumer, we still perform ψ parallel epochs in a single pass over the streaming data which is equivalent to ψ sequential epochs. Therefore, the convergence is expected to be faster with parallel updates. We note that, depending on the application, if the PRODUCESTREAM and INCREMENTALPARALLELUPDATE procedures of PLtR-NS work in the same system process, the former procedure can directly invoke the latter instead of performing serialization/deserialization operations.

Theoretical and empirical analyses show that the following strategies can be useful for maintaining a buffer, $\mathcal{B}$:

(i) *Reservoir sampling with exponential time decay.* A possible approach for maintaining a buffer is reservoir sampling [98] which forms a uniformly random subset of the whole stream incrementally. However, this typically requires quite a large buffer, and it is prone to losing valuable feedback information anyway. The latter is because every incoming feedback instance is held with a probability $\frac{l_{\mathcal{B}}}{\text{length of stream}}$ where $l_{\mathcal{B}}$ is the length of the buffer [115]. While this assures that every feedback instance has equal probability of existing in the buffer, the probability value keeps decreasing with increasing stream length resulting in a possibly reduced number of recent feedback instances in the model updates. Therefore, in applications in which the recent feedback is more important, it can be more appropriate to use reservoir sampling with an exponential time decay function [116,117]. We also adopt this strategy as follows: Similar to the former proposal, once the buffer is full, a new incoming feedback instance is placed in the buffer with a probability $1 - l_{\mathcal{B}}/t$ where t is time order of the feedback instance. The new feedback instance can be efficiently replaced with a randomly drawn feedback

instance already existing in the buffer using rejection sampling [118]. We assume the probability of drawing such an existing feedback instance is proportional to $1-\mathrm{e}^{-\alpha(t-t')}$ where t' is the time order of that feedback instance, and α is a scaling term. We sequentially draw samples from this distribution by first sampling a position in the buffer and then accepting the feedback instance at this position using uniform distribution as our proposal distribution.

(ii) *Sliding windows.* Alternatively, we propose to apply non-overlapping sliding windows over the stream and use these windows as buffers. This approach imposes a more strict time order for model updates by always keeping the most recent feedback in the buffer. Furthermore, the probability of never using a buffered feedback instance for model update only depends on sampling the buffer with replacement during training, and it is quite low even with a few processing units each performing a single epoch over the buffer, $(1-1/l_{\mathcal{B}})^{\psi\times l_{\mathcal{B}}} \approx \mathrm{e}^{-\psi}$. Experimental results in Section 4.5 also point to the effectiveness of this approach.

In general, the chosen buffering strategies have potential benefits: First is to keep a predictably-sized space for the incoming streaming feedback. Second is considering the temporal order of feedback while still enabling stochastic optimization. In other words, not all data is treated equally and recently arriving feedback keeps updating the parameters which may enable better adaptation to changing temporal dynamics. Considering the temporal order also changes $p(j|u,i)$ dynamically and in a meaningful way since the user feedback keeps building up through time.

Stream learning also requires incorporating the new users and items incrementally into the model [119]. As seen in Figure 4.7, this can be done by first buffering the new users and items and then initializing the corresponding model parameters. The updates to the new parameters start right away with the processing of the buffered feedback.

PLtR-N in Figure 4.6 provides a mechanism to change the learning rate in a piecewise fashion. However, since a stream is theoretically infinite, it may not be

straightforward how to set the initial learning rate. For PLtR-NS, one possible option is validation using a portion of the stream so that the chosen learning rate is not very large or small. Another is incorporating an adaptive gradient update method into PLtR-N as discussed in Section 4.4.2.

The effectiveness of our proposal in Figure 4.7 is further discussed in Section 4.5.4.

4.5. Experiments

4.5.1. Datasets and Evaluation

We make use of four datasets to present detailed experimental results from various aspects. Basic properties of the datasets are given in Table 4.1. Apart from binary user-item relevance information, the raw datasets contain graded relevance feedback information in the form of ratings, interaction counts, or different feedback types. We use this information to build holdout sets by leaving one random item out with maximum grade for every test user. In time-split-based experiments, we use the timestamp information attached to every feedback instance in the `ML20M` and `XING` datasets. In the `ML20M` and `LASTFM` datasets, every user has feedback for at least 10 items, and in the latter an item is interacted by a user at least twice. `XING` dataset is relatively sparse, and we use it mainly for testing the stream learning algorithms. Furthermore, since it contains much rarer bookmark and reply types of feedback, we oversample each such feedback instance with a factor given in the parenthesis as shown in Table 4.1.

We measure the ranking accuracy of the algorithms using AUC or MAP@N. While we observe that AUC and MAP@N measurements show a correlated pattern, the latter is preferred for time-split-based evaluations since it can handle arbitrarily large lists of relevant items after a time split. Besides evaluation of ranking accuracy, we illustrate speedup achieved by the proposed algorithms graphically. For this purpose, we use execution times divided by the execution time of a single epoch in the sequential counterpart of the algorithm in question.

Table 4.1. Basic properties of datasets.

Dataset	$\|U\|$	$\|I\|$	# of feedback instances or tuples	Description
`ML20M`	138,493	27,278	20,000,263	User feedback on movies over time [5]
`LASTFM`	359,208	159,000	17,177,350	User-item interaction counts for music recommendations [120]
`MSD`	1,019,318	384,546	48,373,586	Yet a larger dataset of interaction counts for music recommendations [121]
`XING`	770,858	1,002,161	8,861,498	Different feedback types: Click (1), Bookmark (2), Reply (3) as well as timestamp for job recommendations [20]

We use a virtual cloud machine having a multi-core Intel Xeon CPU with 12 physical cores, enough main memory, and a 64-bit Linux operating system with the latest kernel [10]. We implement shared memory parallelism (SMP) to exploit the available cores. There are many ways to implement SMP, but our primary setting is C++11 with OpenMP [122] threads through GCC. We also replicate the experiments using Java 1.7 threads and observe similar comparative results. We monitor the effectiveness of our implementations by diagnostic tools. We leave the atomicity of operations to what is available in the system.

4.5.2. Statistics of Dataset Graphs

To complement analysis of the PLtR-N algorithm in Section 4.3.2, we consider the graph representations (see Figure 4.5) of the datasets and collect statistics for vertices

Table 4.2. Statistics of dataset graphs for LtR.

Dataset	LtR approach	Δ	ρ
ML20M	Pointwise	0.0035	$\leq 2\Delta$
	Pairwise	0.0036	$\leq 3\Delta$
LASTFM	Pointwise	0.0045	$\leq 2\Delta$
	Pairwise	0.0046	$\leq 3\Delta$
MSD	Pointwise	0.0023	$\leq 2\Delta$
	Pairwise	0.0023	$\leq 3\Delta$
XING	Pointwise	0.0011	$\leq 2\Delta$
	Pairwise	0.0015	$\leq 3\Delta$

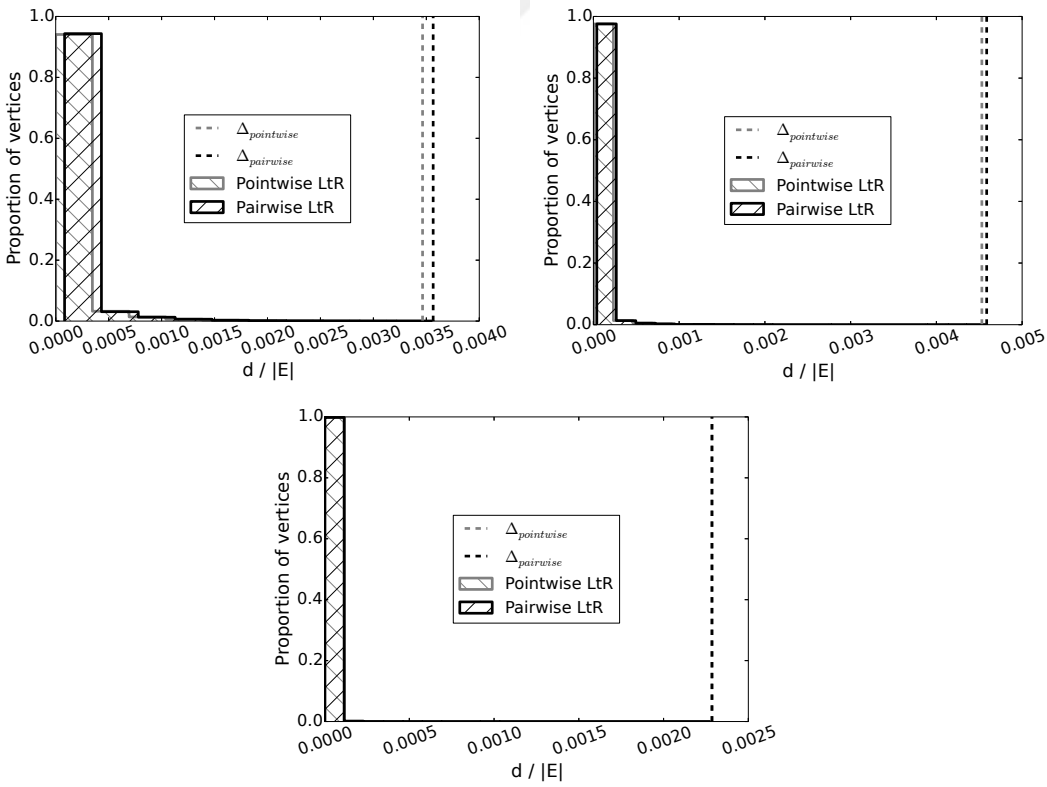

Figure 4.8. Vertex degree (d) distributions in comparison to Δ values for ML20M, LASTFM, and MSD, respectively. Note that, for clarity, only item vertices are shown.

and hyperedges. We first observe that Δ and ρ statistics in the case of pairwise LtR are quite similar to those in the case of pointwise LtR. Table 4.2 shows these comparative statistics which imply similar convergence properties for pointwise and pairwise LtR.

We also investigate vertex degree distributions in graph representations of the datasets. In Figure 4.8, we illustrate a comparison of these distributions together with the Δ values. We observe that although the Δ values define an upper bound for normalized vertex degree distributions, the modal intervals of the distributions correspond to much lower values which further support parallel methods like PLtR-N.

4.5.3. Evaluation of Ranking Accuracy and Speedup

Experimental results for PLtR-N up to 12 training epochs are obtained by performing a single epoch in each processing unit. Since we have 12 available processing units in our experimental setup, we then obtain results for 24 and 48 training epochs by running 2 and 4 epochs at each processing unit, respectively. The results are illustrated in Figure 4.9. The important comparison here is to the algorithm in Figure 4.1 which performs the corresponding number of training epochs sequentially. In our experiments, we call this algorithm SLtR (sequential pairwise LtR). For both SLtR and PLtR-N, the following model hyperparameter values are found by validation: For `ML20M`, $\gamma = 0.01$ and the number of factors, $f = 20$. For `LASTFM`, $\gamma = 0.05$ and $f = 40$. For `MSD` and `XING`, $\gamma = 0.05$ and $f = 50$. Regularization hyperparameter values are found by using a search space around $\lambda_{\theta_u} = \lambda_{\theta_i} = 0.0025$ and $\lambda_{\theta_j} = 0.00025$. Initial values in matrices $\mathbf{P}$ and $\mathbf{Q}$ are sampled from $\mathcal{N}(0, 0.01)$. These hyperparameter values also achieve numerical stability across various algorithms, and they are used as default values for experiments with the extended algorithms in Section 4.5.4.

We observe in all datasets that the difference between the accuracy of SLtR and PLtR-N is not statistically significant which clearly shows the effectiveness of lock-free parallelism with no partitioning. Furthermore, the speedup patterns show that PLtR-N scales quite well whereas, as expected, SLtR has linearly increasing execution time with the number of training epochs. These results suggest that PLtR-N can converge

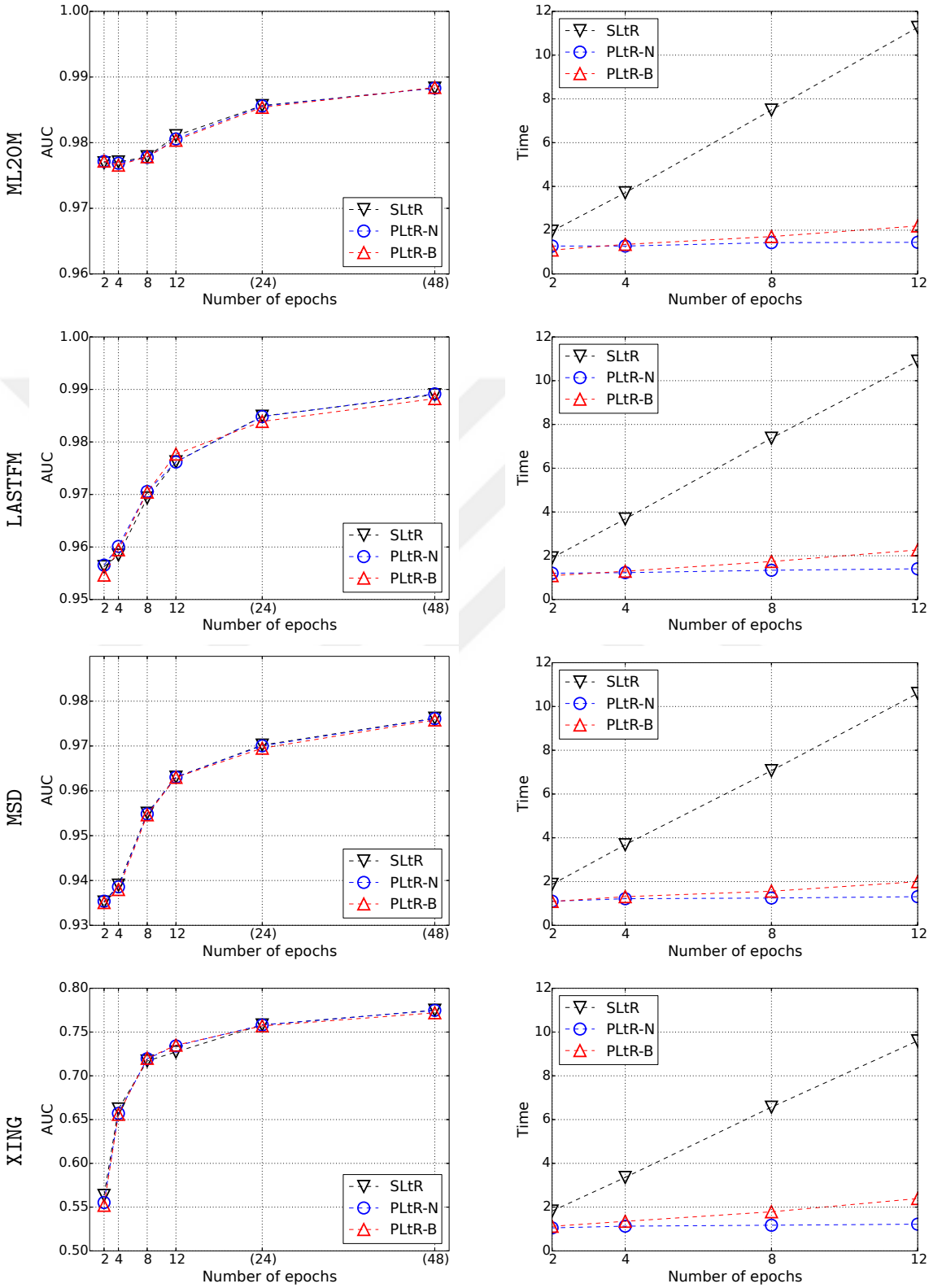

Figure 4.9. Comparing ranking accuracy and speedup of PLtR algorithms.

much faster without loss of accuracy and by making better use of the resources at hand.

Experimental results for PLtR-B up to 12 training epochs are obtained by performing ψ parallel block updates in each round where ψ is equal to the number of training epochs. For 24 and 48 training epochs, $\psi = 12$ due to the experimental setup. The partitioning stage of PLtR-B is performed at the master processing unit and the update stage at all ψ parallel processing units. The rest of the hyperparameter values are the same as those for SLtR and PLtR-N. The AUC pattern shows that the accuracy of PLtR-B is comparable to that of SLtR and PLtR-N. On the other hand, while PLtR-B offers considerable speedup compared to SLtR, its speedup pattern is somewhat worse than that of PLtR-N. The major reason is the partitioning stage performed at the master processing unit. Using separate dedicated processing units for this stage has a potential to improve this speedup.

4.5.4. Experiments with Extended Algorithms

In this section, we present detailed experimental results for the extensions to the PLtR algorithms discussed in Section 4.4.

4.5.4.1. Experiments with Different Sampling Strategies. We begin with illustrating the experimental results for different sampling strategies in Figure 4.10. For all datasets, the adaptive sampling (AS) strategy affects the convergence speed in a positive way. This can be seen by comparing to the SLtR algorithm which draws j uniformly at random from $I \setminus I_u^+$. For adaptive sampling, we choose $|S_u^-| = \lceil \log 0.10 / \log 0.90 \rceil = 22$ and pick with 90% probability at least one candidate item j among the top 10% closest items to i [103]. We observe that both PLtR-B-AS and PLtR-N-AS produce an AUC pattern very similar to their sequential counterpart with adaptive sampling, SLtR-AS, which shows that they can readily be used for efficient adaptive sampling. The comparative speedup patterns of the extended algorithms are similar to those in Figure 4.9 and therefore not illustrated. On the other hand, compared to SLtR which performs

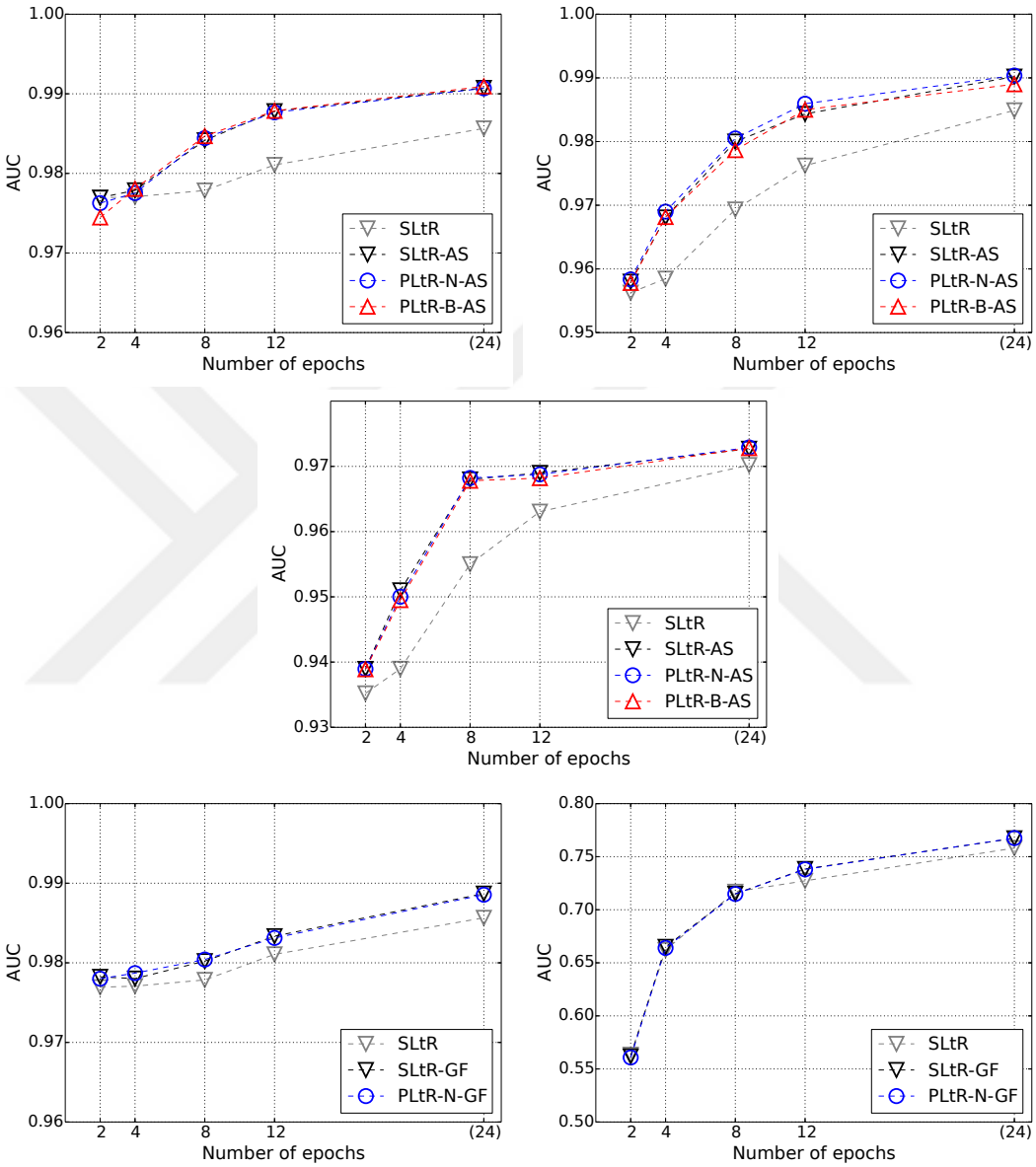

Figure 4.10. Comparing different sampling strategies. (Top-Middle rows) Adaptive sampling results for ML20M, LASTFM, and MSD, respectively. (Bottom row) Handling graded relevance feedback in ML20M and XING, respectively.

sampling uniformly at random, one epoch of SLtR-AS has more than twice the execution time in our experimental setup. This makes combination of PLtR family of algorithms with adaptive sampling a valuable approach.

As pointed out in Section 4.4.1, PLtR-B can incorporate adaptive sampling with a single restriction, that is, j is sampled from the items available in each block. On the other hand, it is useful to further analyze the effect of such biased sampling schemes when incorporated into PLtR-N. Figure 4.11 shows that PLtR-N-AS produces a skewed distribution of item j samples while PLtR-N, which samples uniformly at random, produces a steep peak centered around true uniform distribution over all items. We see in Figure 4.10 that this shift in the distribution does not decrease predictive power of PLtR-N-AS with the chosen level of parallelism. For further analysis, we design and make yet another experiment: First, we observe the rank-frequency distribution of items in $\mathcal{D}^+$ as well as that of j after adaptive sampling in real datasets. It turns out that both distributions can be well approximated with the Zipf-Mandelbrot law [123, 124]. Then, we simulate various scenarios each with a different number of items, $|I|$, while we keep the number of processing units, ψ, fixed. For each scenario, we fit items the two rank-frequency distributions with similar shape characteristics to what is observed in the real-life datasets. We also initialize counters for each item and set them to zero. Parallel counter updates are done at each processing unit by sampling an i and a j from corresponding rank-frequency distributions, reading their counters at the time of sampling, waiting for a short random period to achieve some staleness, and then incrementing their counters. After sufficiently large number of iterations, the percentage of update losses can be decided by comparing the sum of item counters to the true number of updates. We compare each scenario to the case in which j is sampled uniformly at random rather than from the fitted distribution. The result of this experiment is also illustrated in Figure 4.11. It turns out that the comparative update losses quickly converge for even relatively small $|I|/\psi$ hinting us that the negative effect of biased sampling on PLtR-N can be quite limited in many practical scenarios.

In Figure 4.10, we also present experimental results for the sampling strategy described in Section 4.4.1 for handling graded relevance feedback (GF). When used

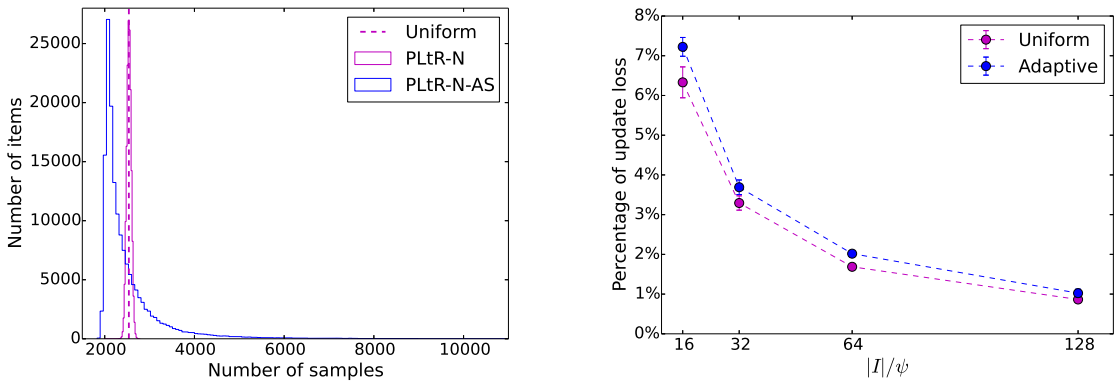

Figure 4.11. Effect of sampling j adaptively vs. uniformly at random for PLtR-N. (Left) Distribution of sampled items j in `LASTFM` after applying the different sampling strategies. (Right) A simulation of comparative update loss.

with PLtR-B, this strategy is applicable but slightly more restrictive since it requires sampling both relevant and irrelevant items from what is available in each block. On the other hand, PLtR-N-GF results compared to its sequential counterpart, SLtR-GF, show that PLtR-N can be conveniently coupled with this sampling strategy. We provide experimental results for `ML20M` using its rating information as grades and `XING` using its different types of feedback given in Table 4.1. The probability of choosing j from relevant items is set to 0.10. To be suitable for this type of sampling, `LASTFM` and `MSD` require some preprocessing for handling their grade information (item interaction counts), and their results are not reported. It turns out that this sampling strategy is computationally less costly compared to adaptive sampling, and the samples are more uniformly distributed. Therefore, the analysis for adaptive sampling is also valid here, and it can be considered as the worst case.

4.5.4.2. Experiments with Adaptive Gradient Updates. Experimental results for the extensions with AdaGrad and RMSProp adaptive gradient update methods are reported in Figure 4.12. We fix the learning rate, γ, to 0.1 for AdaGrad extensions and to 0.01 for RMSProp extensions. We also set $\epsilon = 10^{-8}$, and for RMSProp, EWMA weight is 0.1. Then, we let the adaptive gradient updates guide the learning process. We only report results for the PLtR-N versions since adaptive gradient updates do not

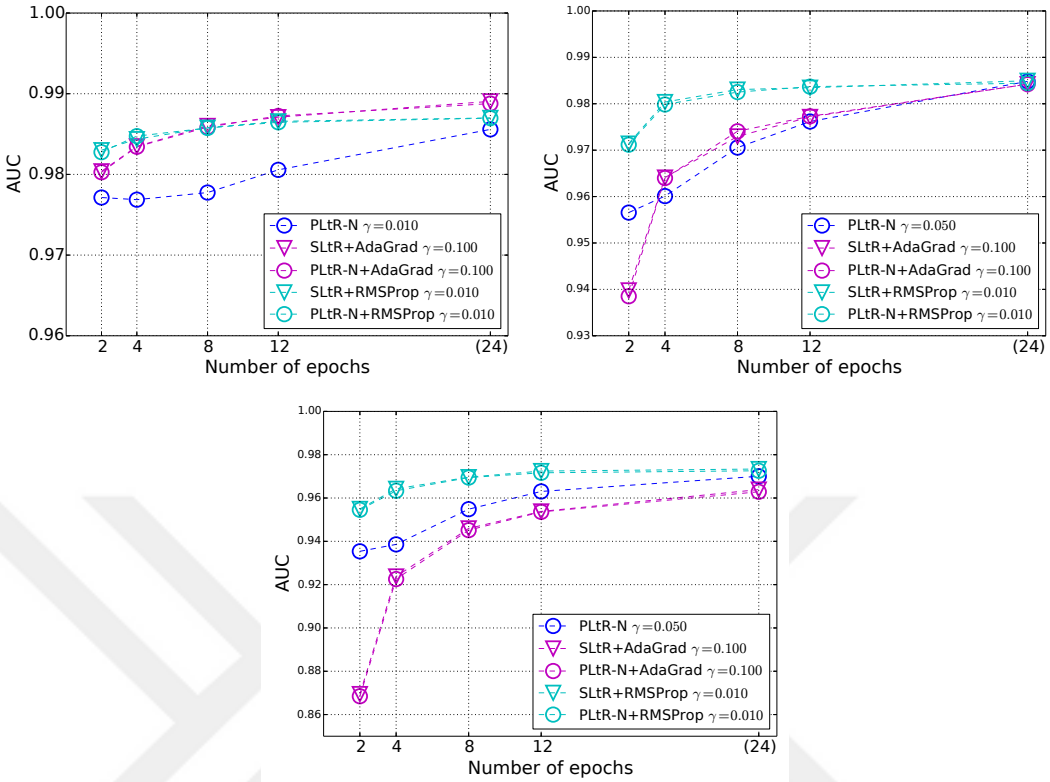

Figure 4.12. Comparative results for PLtR-N extensions with adaptive gradient updates on `ML20M`, `LASTFM`, and `MSD`, respectively.

really pose a challenge for the PLtR-B algorithm which performs mutually exclusive parameter updates. We first observe that PLtR-N extended with adaptive gradient update methods can have an increase in convergence speed compared to the non-adaptive PLtR-N with the best learning rate. Furthermore, SLtR and PLtR-N combined with the same adaptive gradient method have very similar AUC patterns which shows that the parallelism preserves the ranking accuracy. We also note that the comparative speedup patterns of the extended algorithms with adaptive gradient update methods are similar to those in Figure 4.9. On the other hand, in our experimental setting, the execution time for one epoch with adaptive gradient updates is about twice to three times the execution time with non-adaptive updates. Therefore, the efficiency obtained by combining adaptive gradient update methods with PLtR can be quite valuable in practice.

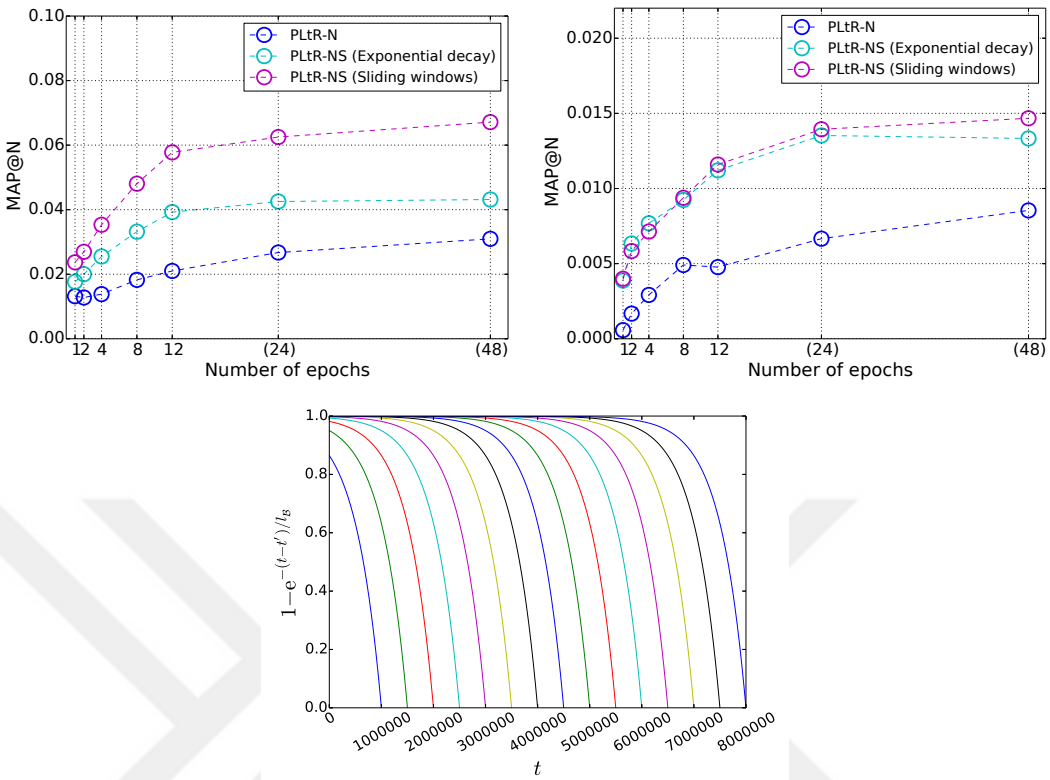

Figure 4.13. (Top row) Comparing ranking accuracy of PLtR-NS on `ML20M` and `XING`, respectively. (Bottom row) Shape of exponential decay at various time points.

4.5.4.3. Experiments for Parallel LtR from Streaming Feedback. We finally report experimental results for variations of the PLtR-NS algorithm proposed in Section 4.4.3 for LtR from streaming feedback. In Figure 4.13, we compare the ranking accuracy of the algorithms on `ML20M` and `XING` datasets which contain timestamp information for every feedback instance. We use a time split where the first 99.5% of a time-ordered dataset is used for training the model, and the last portion is for testing. Since the number of relevant items for each user can be different in the test set, we use MAP@N as our evaluation measure with $N = 50$. Varying N does not change the comparative results. While PLtR-N performs learning using all the training data at once, the PLtR-NS variations implement incremental learning using the different buffering approaches discussed in Section 4.4.3. For `ML20M`, PLtR-NS uses a buffer length, $l_{\mathcal{B}} = 500,000$, and performs incremental learning after every 500,000-th incoming feedback instance. For `XING`, both values are fixed to 100,000. When exponential decay is used, $\alpha = 1/l_{\mathcal{B}}$.

In general, the PLtR-NS variations produce superior results by taking time order of the user feedback into account. It is also observed that the sliding windows buffering approach outperforms the exponential time decay approach in both experiments. In Figure 4.13, we can see the shape of exponential decay for various points in time by choosing $\alpha = 1/l_{\mathcal{B}} = 1/500,000$ and observe that the probability of eliminating an older feedback instance from the buffer is always higher. For potential improvements, it is possible to modify the shape of decay function by further tuning the scaling term, α. Finally, we observe that performing a single pass (one epoch) over the stream yields poorer convergence results compared to performing multiple parallel epochs. This clearly shows the usefulness of the parallelism achieved by the PLtR-NS algorithm.

5. EFFICIENT TOP-N PREDICTION FROM MATRIX FACTORIZATION MODELS

5.1. Introduction

As discussed in Section 2.2.3, latent factor models, and in particular, matrix factorization (MF) models [24] are a major line of research in CF. They allow using versatile loss functions and often produce good low-dimensional latent representations of users and items. In this latent space, the curse of dimensionality is reduced and the nonlinearities can be approximated so that the simpler dot product of a user and an item vector often yields a good predictive score for personalized relevance. Optimizations based on stochastic gradient descent (SGD) or alternating least squares (ALS) [48, 125] as well as their parallel versions [10, 96] are shown to do a good job of training MF models on large user feedback datasets. However, when the item repository is large, personalized prediction queries for finding the top-N items with the highest dot products can still be costly due to dense matrix multiplications and further sorting operations. This may raise practical concerns about MF models such as adaptivity due to longer computation times when the top-N predictions are precomputed for a large number of users or fast retrieval when there is a need for top-N predictions in real time.

Finding nearest neighbors (NN) is a common task in pattern recognition and machine learning [39]. Yet, many large-scale applications enforce approximate nearest neighbor search methods [126] due to the computational cost of exact search. This can bring considerable speedup for a slight loss of precision.

In this chapter, we combine MF with approximate NN search methods to improve the efficiency of top-N prediction queries. Our efforts result in a meta-algorithm, MMFNN, for MF models and the following main contributions:

- The problem of finding the top-N highest dot products is relaxed to finding the highest dot products in an approximate proximity of the items known to be relevant to the user. We show that this is a viable heuristic approach since many MF models try inherently to perform some sort of dot product maximization for these items, and searching their neighborhoods is useful for discovering new top-N items. This approach also allows the proposed meta-algorithm to employ various common MF models without requiring a modification to their loss functions.
- We show that the meta-algorithm can highly improve the time efficiency of top-N prediction queries in the first place. Furthermore, it still performs comparably to standard exhaustive prediction methods or sometimes even better in terms of important performance criteria such as ranking accuracy and diversity.
- We show that the meta-algorithm can also be used to facilitate incremental CF which can learn from streaming user feedback.
- Although the ideas here can be adapted to learning from explicit user feedback as well, we focus on implicit user feedback. We present a detailed analysis of our approach for learning from implicit user feedback together with experimental results on large implicit feedback datasets from different application domains.

The remainder of this chapter is organized as follows: In Section 5.2, we present more background about the MF models for CF from implicit feedback. Then, in Section 5.3, we present preliminaries about approximate nearest neighbor search in metric spaces with a focus on the methods used in this chapter. In Section 5.4, we propose the meta-algorithm, MMFNN, to improve the efficiency of top-N prediction queries and analyze it from various aspects. We also show that it can enhance incremental CF. We present a comparison to the related work in Section 5.5. Detailed empirical analysis results are presented in Section 5.6.

5.2. MF Models for CF from Implicit Feedback

Let U be the set of users and I be the set of items in a CF system. Each user, $u \in U$, is assumed to have provided some feedback to the system and therefore has a

set of relevant items, $I_u^+ \subseteq I$. Typically, $|I_u^+| \ll |I|$. As discussed in Section 2.1.2, the relevance can be binary or graded.

A matrix factorization model, $\mathcal{M}_{MF}$, tries to minimize an error function in the typical form,

$$\mathcal{E}\left(\mathbf{P}, \mathbf{Q} \mid \mathcal{D}\right) = \sum_{\mathbf{t} \in \mathcal{D}} \Big\{ L\left(\mathbf{t}, \mathbf{P}, \mathbf{Q}\right) + \text{regularization term} \Big\}, \tag{5.1}$$

where $\mathbf{t}$ is a tuple in a personalized relevance dataset, $\mathcal{D}$, consisting of a user, $u \in U$, and at least one item, $i \in I$. Depending on the personalized LtR approach (see Section 2.2.1), $\mathbf{t}$ may also contain additional items as well as grade and extra context information such as time. In general, the model parameters correspond to fixed-sized column vectors $\mathbf{p}_u, \mathbf{q}_i \in \mathbb{R}^f$ in the low-rank component matrices of the factorized user-item relevance matrix, $\widehat{\mathbf{Y}} = \mathbf{P}^\top \mathbf{Q}$. Therefore, the number of factors, $f \ll |I|$. Different loss functions, L, are possible for learning from implicit feedback which will be detailed next. The resulting error function is differentiable with respect to the model parameters, and it can be optimized with numerical optimization techniques such as SGD or ALS.

Without loss of generality, in this chapter, we stick to two seminal MF models for implicit feedback and their extensions:

(i) *Weighted regularized matrix factorization* (WRMF) [7, 57] is a pointwise LtR model. The loss function is in the form,

$$L = w_{ui} \left(z_{ui} - \mathbf{p}_u^\top \mathbf{q}_i\right)^2. \tag{5.2}$$

Following [7], given some confidence value, c_{ui}, corresponding to a grade of relevance, each training tuple performs updates by selecting $z_{ui} = 1_{\mathbb{R}_{>0}}\left(c_{ui}\right)$ using an indicator function, and $w_{ui} = 1 + ac_{ui}$ where a is a constant scaling factor.

(ii) *Bayesian personalized ranking matrix factorization* (BPRMF) [48] is a pairwise LtR model. The loss function is based on sigmoid-smoothed pairwise loss and negative log-likelihood,

$$L = -\ln \sigma(\mathbf{p}_u^\top \mathbf{q}_i - \mathbf{p}_u^\top \mathbf{q}_j), \tag{5.3}$$

where basically $i \in I_u^+$ and $j \in I \setminus I_u^+$. Many improvements and extensions for such pairwise models are covered in Chapter 4.

When an $\mathcal{M}_{MF}$ is learned, a common prediction task is finding a set of top-N items, $\{\hat{r}_u^{-1}(1), \hat{r}_u^{-1}(2), \ldots, \hat{r}_u^{-1}(N)\}$, suitable for each queried user, $u \in U$. In general, a top-N query first requires predicting relevance scores by computing the dot product, $\hat{y}_u(i) = \mathbf{p}_u^\top \mathbf{q}_i$, for all $i \in I$. Then, for obtaining the N items, these scores are sorted. In practice, the item repositories can be quite large and these queries can become costly due to practical constraints.

5.3. Approximate NN Search in Metric Spaces

In this chapter, we are interested in finding approximate k nearest neighbors of items in metric spaces which can be formalized as follows:

$$kNN(i_q, I, k) = K, \quad |K| = k, \quad K \subseteq I, \tag{5.4}$$

where $i_q \in I$ is a queried item. In exact nearest neighbor search, $\forall i_+ \in K, i_- \in I \setminus K$, $d(i_q, i_+) \leq d(i_q, i_-)$, where $d : I \times I \to \mathbb{R}_{\geq 0}$ is a metric distance. In approximate search, this is relaxed up to some precision by potentially allowing false positive items in the neighborhood.

There are many methods for finding approximate nearest neighbors in metric spaces. A useful taxonomy [126] is dividing these methods into *partitioning trees*, *hashing-based*, and *nearest neighbor graph* techniques. We carry on with two effective partitioning tree techniques reported to outperform others in the same reference in

terms of efficiency. We briefly describe these techniques below and provide additional details in Sections 5.4.2 and 5.6.

(i) *Randomized k-d trees* (RKT) : k-d tree is a well-known data structure for space partitioning. RKT approach builds multiple independent k-d trees, each partitioning the space recursively into two on randomly chosen dimensions. A modest-sized ensemble of randomized trees improves the precision of a single tree with little computational overhead.

(ii) *K-means tree* (KMT) : This partitioning technique builds a tree by performing recursive k-means clustering in the metric space using all dimensions and a chosen distance measure.

The construction of both partitioning trees is quite efficient when we have a large number of entities with a modest number of features which is typical of low-rank matrices from matrix factorization for CF. Furthermore, both partitioning trees allow efficient tree traversal to find approximate kNN for a given query among a predefined number of candidate neighbors.

Throughout this chapter, we refer to a nearest neighbor search model as $\mathcal{M}_{NN}$.

5.4. A Meta-algorithm for Efficient Prediction from MF Models

In this section, we propose a meta-algorithm, MMFNN, to improve the efficiency of top-N predictions by combining matrix factorization models with approximate nearest neighbor search. To start our discussion, we first present the high-level view of our approach in Algorithm 5.1. We then carry on with the analysis and potential benefits of the proposed algorithm.

The meta-algorithm in Figure 5.1 begins with learning parameters of an MF model, $\mathcal{M}_{MF}$. At this stage, many MF models including those discussed in Section 5.2 are pluggable into the algorithm, and no custom loss functions are applied. Then,

— LEARNING STAGE —

Learn $\mathbf{P}$ and $\mathbf{Q}$ using $\mathcal{M}_{MF}$;

Learn a partitioning of I in $\mathbf{Q}$ space using $\mathcal{M}_{NN}$;

for all item $i \in I$ **do**

  Query and store $kNN(i, I, k)$ from $\mathcal{M}_{NN}$;

end for

— PREDICTION STAGE —

for all queried user u **do**

  Choose $I_u^{++} \subseteq I_u^{+}$;

  $S \leftarrow \bigcup_{i \in I_u^{++}} kNN(i, I, k)$;

  Output $\{\hat{r}_u^{-1}(1), \hat{r}_u^{-1}(2), \ldots, \hat{r}_u^{-1}(N)\}$ with the highest $\hat{y}_u(i) \leftarrow \mathbf{p}_u^{\top}\mathbf{q}_i, i \in S$;

end for

Figure 5.1. MMFNN($\mathcal{M}_{MF}, \mathcal{M}_{NN}$) : Meta-algorithm for efficient top-N prediction from matrix factorization models.

the approximate NN search model, $\mathcal{M}_{NN}$, is learned using the factorized representation of items, $\mathbf{Q}$, from $\mathcal{M}_{MF}$. This means that we learn an $\mathcal{M}_{NN}$ in a latent space rather than the original item space where the item vector size is reduced to a modest number (typically $f \in [10, 100]$ in practical CF tasks). Learning and querying $\mathcal{M}_{NN}$ for obtaining kNN sets are extra preprocessing steps compared to the state-of-the-art exhaustive prediction (EP) given in Figure 5.2 for clarity. However, this is done once and efficiently, and then we are able to perform every top-N prediction using an expectedly much smaller and refined union set, S, rather than the whole set of items, I. For obtaining S, we refer to a set, $I_u^{++} \subseteq I_u^{+}$, which may include all known relevant items to a user or just a smaller subset, for example, based on the most recent user feedback.

Interestingly, learning and querying $\mathcal{M}_{NN}$ often have negligible computational cost for common k, f, and $|I|$ values in large-scale CF tasks. Furthermore, since typ-

Require: $\mathbf{P}$ and $\mathbf{Q}$ from an MF model, $\mathcal{M}_{MF}$
  for all queried user u **do**
    Output $\{\hat{r}_u^{-1}(1), \hat{r}_u^{-1}(2), \ldots, \hat{r}_u^{-1}(N)\}$ with the highest $\hat{y}_u(i) \leftarrow \mathbf{p}_u^\top \mathbf{q}_i, i \in I$;
  end for

Figure 5.2. EP: Exhaustive top-N prediction from matrix factorization models.

ically $|S| \ll |I|$, a remarkable speedup can be observed for top-N queries which deal with a reduced set of items. The selection strategy of S is based on the heuristic idea that relevant unexplored items for a user, u, can be in close proximity to user's previously known relevant items, I_u^+, in the latent space. To put in another way, at least some items from I_u^+ are actually expected to have high dot products with $\mathbf{p}_u$ due to the dot product optimization procedure in an $\mathcal{M}_{MF}$. Therefore, their neighboring items are also expected to have high dot products. A closer inspection of Equations 5.2 and 5.3 supports these assumptions: In the former, previously known relevant items are expected to have higher dot products, and a penalty is applied if the dot product diverges, depending on the weighted confidence, w_{ui}, in the relevance. In the latter, the loss function penalizes more obviously in a way that AUC is directly optimized [48] to rank previously known relevant items above the rest. Therefore, searching neighborhoods of the items in I_u^+ is useful to find unexplored items with high dot products for top-N prediction. This strategy seems to work well and even impose useful additional filtering which results in improved performance with respect to various important performance criteria as shown in Section 5.6.

Figure 5.3 shows example user and item embeddings in the latent space learned from an $\mathcal{M}_{MF}$. For illustrative purposes, three dimensions are given where item factor values have the highest variance. Note that a dot product can be written as $\mathbf{p}_u^\top \mathbf{q}_i = \|\mathbf{p}_u\| \|\mathbf{q}_i\| \cos(\alpha_{\mathbf{p}_u, \mathbf{q}_i})$, and given a specific user, u, its maximization can be simplified to the maximization of $\|\mathbf{q}_i\| \cos(\alpha_{\mathbf{p}_u, \mathbf{q}_i})$ by normalizing the user vector. The latter is possible since the recommendations are personalized and the direction of $\mathbf{p}_u$ suffices for predictions. Therefore, it can be imagined that the most relevant items should

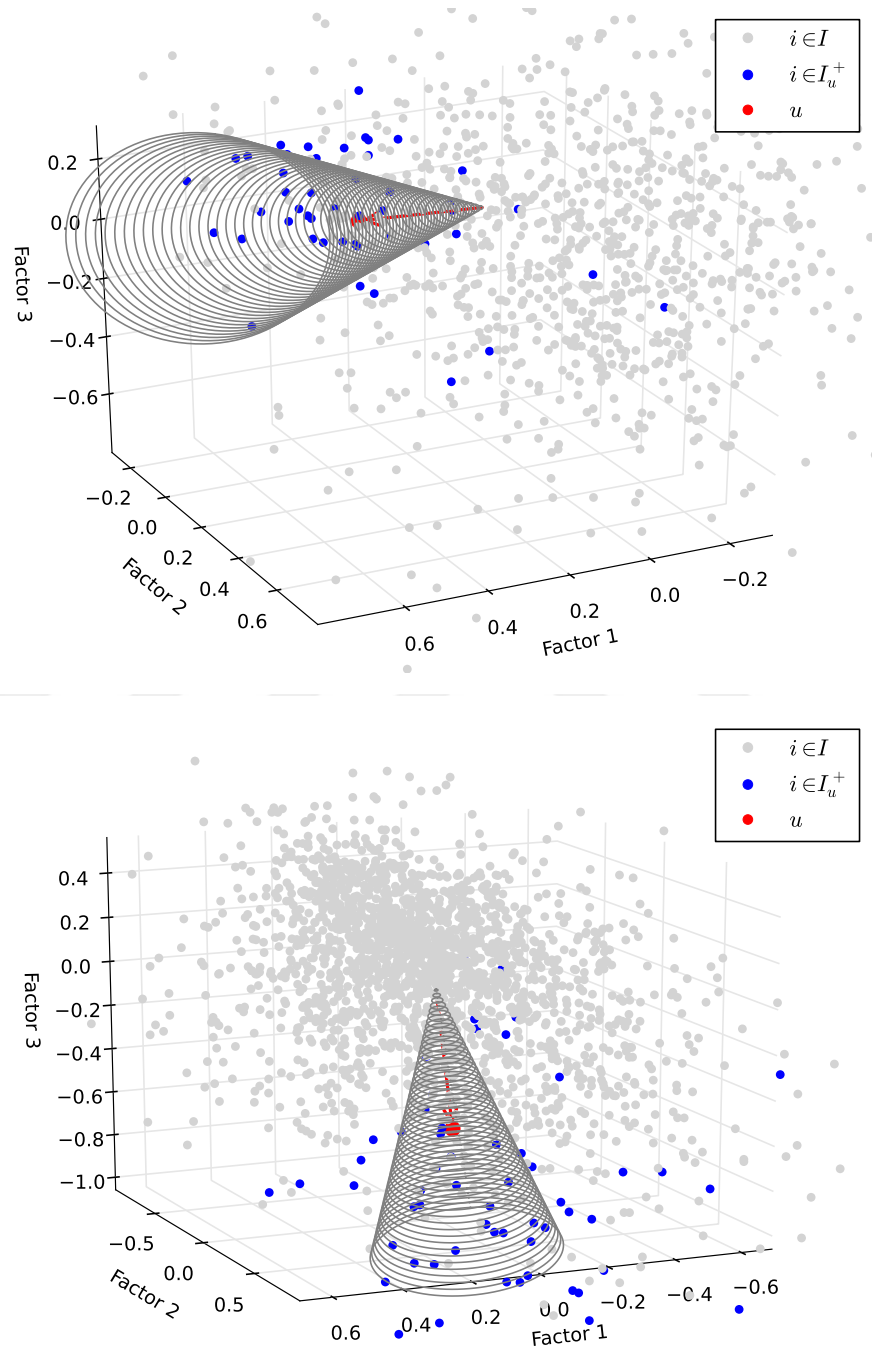

Figure 5.3. Example embeddings and cones in the latent space for `LASTFM2` and `XING` datasets, respectively ($i \in I$ are downsampled).

lie within an open cone around the user vector and as far as possible from the origin maximizing the terms $\cos(\alpha_{\mathbf{p}_u,\mathbf{q}_i})$ and $\|\mathbf{q}_i\|$, respectively [127, 128]. In Figure 5.3, we observe that the items $i \in I_u^+$ are often around such a cone and relatively far from the origin. As a result, their neighbors in a metric space, such as the Euclidean space, are likely to have high dot products.

Therefore, we reason that the proposed meta-algorithm is an efficient heuristic approach for finding top-N items with high dot products. We point to some additional potential benefits of our proposal as follows:

- We provide a way to look for items with high dot products while, at the same time, taking into account useful neighborhood constraints. As supported by the experimental results in Section 5.6, this can bring additional performance improvements compared to finding the top-N highest dot products among all items.
- For predicting the top-N items, a subset of user's known relevant items can be selected, for example, to emphasize recent feedback or to further improve efficiency by sampling a small representative subset.
- When the user is cold start by having a few feedback instances, we can still rely on the neighborhoods of the items in these instances.
- Incremental CF can be facilitated which is detailed next in Section 5.4.1.

5.4.1. Enhancing Incremental CF

While incremental learning offers efficiency and adaptivity, it can be crucial for practicality that the predictions from the associated models are also efficient. In this sense, MMFNN can enhance incremental CF algorithms based on matrix factorization. For example, the PLtR-NS algorithm proposed in Chapter 4 can constitute an $\mathcal{M}_{MF}$ so that the learned model parameters $\mathbf{P}$ and $\mathbf{Q}$ can be used by MMFNN for faster retrieval from the most up-to-date model. In general, by incorporating incremental matrix factorization, MMFNN can make LtR from streams more practical especially with large item repositories. In Section 5.6, we present experimental results towards this direction.

— MODIFIED PREDICTION STAGE —

for all queried user u at a specific time point t **do**

Choose $I_u^{++} \subseteq I_u^{+}$;

$S \leftarrow \bigcup_{i \in I_u^{++}} kNN(i, I, k)$;

$A' \leftarrow$ top-N' items with the highest $\mathbf{p}_u^\top \mathbf{q}_i, i \in S, N < N' \leq |S|$;

for all item $i \in A'$ **do**

if $g_i^t = 0$ **then**

$p_i \leftarrow +\infty$;

else

$p_i \leftarrow \hat{\theta}_i^{MAB} + \sqrt{\frac{2 \ln t}{g_i^t}}$;

end if

end for

$A \leftarrow$ top-N items with the highest $p_i, i \in A'$; /* Break ties arbitrarily */

for all item $i \in A$ **do**

$g_i^t \leftarrow g_i^t + 1$;

Observe a reward ρ_i in range $[0, 1]$;

Update $\hat{\theta}_i^{MAB} \leftarrow \frac{g_i^{t-1}}{g_i^t} \hat{\theta}_i^{MAB} + \frac{1}{g_i^t} \rho_i$;

end for

end for

Figure 5.4. MMFNN prediction stage with UCB-type multi-armed bandit.

In this section, we also investigate a proof of concept extension in which MMFNN adopts an additional reinforcement learning model, $\mathcal{M}_{RL}$. The resulting procedure can be considered as a contextual multi-armed bandit based on UCB-type approaches [129, 130] where UCB refers to an upper confidence bound. We present the extension in Figure 5.4 which modifies the prediction stage of MMFNN. The core idea is reranking the best candidate arms (items) for a certain context (user) with respect to their expected payoff taking into account the confidence in this payoff. Each arm (or item), $i \in I$, is modeled with a payoff distribution, Bernoulli(θ_i^{MAB}). These distributions are unknown at the beginning and learned over time. Arm selection is subject to the expected payoff at a time point $t = 1, 2, \ldots, T$ aggregated with an upper confidence bound. This bound can be thought to add a bonus term if an item is under-explored. g_i^t refers to the frequency count of an item i in top-N predictions at time t. The best N' predictions from an $\mathcal{M}_{MF}$ at time t are reranked by the aggregated score. Since $\mathcal{M}_{MF}$ is a personalized model, the $\mathcal{M}_{RL}$ can be thought to consider personalization by handling only the available set of candidate items from $\mathcal{M}_{MF}$. This approach also brings efficiency for arm selection since we deal with a reduced and quickly retrieved set of items. Finally, the arms are updated with the weighted average of previous payoff expectation and the current reward at an interval [0,1]. An example reward mechanism can be $\rho_i = 1$ if a predicted item for the user u is relevant, and $\rho_i = 0$ otherwise. In Section 5.6, we also present experimental results for this extension.

5.4.2. Complexity

Referring to [126], when $\mathcal{M}_{NN}$ = KMT, the time complexity of tree construction is $O(|I| \times f \times \kappa \times \iota \times (\log |I| / \log \kappa))$ where ι is the number of k-means iterations, and κ is the branching factor (or number of clusters) in recursive k-means. Finding nearest neighbors of a query item requires finding leaf nodes containing potentially good neighboring items. Such leaf nodes are reached by holding a priority queue of the promising branches. The overall search complexity is $O(\nu \times f \times (\log |I| / \log \kappa))$ where ν is a predefined number of leaf node items to examine for finding the k nearest neighbors. On the other hand, $\mathcal{M}_{NN}$ = RKT [131] requires building τ independent k-d

trees, each with a time complexity of $O(|I| \times \log |I|)$. The trees can be built in parallel. To find the nearest neighbors, a common priority queue is maintained across all trees so that the most promising branches are explored first. The search stops after reaching a predefined number, ν', of leaf nodes across the trees, and the k nearest neighbors are retrieved.

Using the algorithm in Figure 5.2, each top-N query requires computing a dot product with a complexity of $O(|I| \times f)$ for finding the predictive scores and $O(|I| \times \log |I|)$ for sorting them. Using a min-heap of size N is effective when $|I|$ is large in which case the sorting operation becomes $O(|I| \times \log N)$ by feeding each scored item to the heap. Using the algorithm in Figure 5.1, each top-N query requires computing the set S. Inserting a single item to S or checking its existence has $O(1)$ average time complexity, and a total of $|I_u^{++}| \times k$ of such operations are done. Computing the dot products has a complexity of $O(|S| \times f)$ whereas sorting them is $O(|S| \times \log |S|)$ or $O(|S| \times \log N)$ depending on the sorting strategy.

We note that the top-N queries can also be parallelized for different users or different $i \in S$ which are comparable to possible parallel processing options for exhaustive top-N search over I. However, parallel prediction is not always a desired option in practice [128].

5.5. Comparison to Related Work

Compared to the training task, prediction from an MF model has less research focus. However, due to increasingly larger datasets used in practice, leading industrial research points to its importance, for example, for top-N prediction [132] and deciding item similarities [133].

In [134], the authors formulate a custom loss function for MF which yields user and item embeddings directly in the Euclidean space. This enables search for potentially relevant items in this space rather than using dot product optimization. In other words, given a queried user's embedding, the most relevant item embeddings are ex-

pected to have the smallest Euclidean distance to it. The authors apply brute-force nearest neighbor search and report improved results on explicit feedback datasets with relatively small number of items. A good formalization of dot product optimization for MF is provided in [128]. The authors then propose a metric tree for partitioning the item vectors in the Euclidean space. For retrieving items for a queried user with the highest dot products, their proposal is a branch-and-bound algorithm which traverses the tree in a specialized way. To further improve the efficiency, they propose to cluster the user vectors with respect to their angular similarity and decide the items for each cluster based on highest dot products with the representative user of the cluster. This proposal is viable since user vectors with similar directions are likely to have similar taste profiles. Using large-scale explicit feedback datasets, the authors report useful speedup with various degrees of precision loss compared to exhaustive prediction. Another seminal tree-based approach is reported in [132]. The authors first propose a simple but elegant reduction of the retrieval problem in the dot product space to a problem in the Euclidean space. Then, they propose to use a PCA-tree which partitions the items and enables direct user queries to find relevant items. The authors report useful speedup patterns against acceptable losses in prediction quality as well as improvements over [128]. We name this reduce-and-partition approach R-PCA and present comparative results in Section 5.6.

Another recent and interesting approach is LEMP [135,136] in which the authors propose to partition the item vectors according to their lengths. For finding higher dot products above a threshold, each partition is treated differently: Given a queried user, many partitions are often automatically neglected due to the threshold. For others, their algorithm solves smaller cosine similarity search problems. For finding the top-N items, they also propose a way to adjust the threshold adaptively. The authors report good speedup patterns on different variations of their algorithm and also provide an efficient implementation of their ideas. In Section 5.6, we provide a comparison to this approach as well.

Other notable approaches include: a directly indexable probabilistic MF model for efficient prediction [137] and a locality-sensitive hashing scheme [138] which can enable efficient retrieval in the dot product space with tight guarantees.

All in all, there is growing interest in improving efficiency of predictions from MF models. Differently from the above-mentioned approaches, our approach brings together the following properties: It defines a meta-algorithm which can combine various common matrix factorization models with various approximate nearest neighbor search methods. This offers the ability to use different well-established models without any modifications. The meta-algorithm is based on a low-cost but powerful heuristic for finding top-N items with high dot products within an intuitively preselected neighborhood. Therefore, it offers the potential benefits pointed out in Section 5.4, works well with implicit user feedback, and facilitates non-incremental and incremental CF.

5.6. Experiments

We perform extensive experiments to test the effectiveness of MMFNN. We begin with describing the datasets used in this study and the experimental setup. We then present comparative experimental results with respect to various performance criteria in multiple experimental settings.

5.6.1. Datasets and Experimental Setup

Throughout our experiments, we make use of five real-life datasets from various recommender system application domains. These datasets are summarized in Table 5.1, and further details are provided in [13] and throughout the rest of this section.

Our experimental setup is a commodity PC having an Intel Ivy Bridge Pentium processor with 2 cores each at 2.40 GHz, 8-GB main memory, and 64-bit Linux operating system with the latest kernel [13]. Our results are obtained using C++11 through GCC.

Table 5.1. Basic properties of datasets.

Dataset	$\|U\|$	$\|I\|$	# of feedback instances or tuples	Description
XING	770,858	1,002,161	8,861,498	Different feedback types: Click (1), Bookmark (2), Reply (3) as well as timestamp for job recommendations [20]
LASTFM2	359,349	268,772	17,559,127	User-item interaction counts for music recommendations [120]
MSD	1,019,318	384,546	48,373,586	Yet a larger dataset of interaction counts for music recommendations [121]
TMALL	424,170	1,090,390	54,925,330	Different feedback types: Click (1), Add-to-cart (2), Purchase (3), and Add-to-favorite (4) as well as timestamp for online retail recommendations [52]
AMZBK	8,026,324	2,330,066	22,507,155	User feedback on books [139]

5.6.2. Experimental Results

5.6.2.1. Experiments Using a Random Holdout Set. In this set of experiments, we leave one item out for every test user. These items are chosen randomly from the users' known relevant items among those with the maximum confidence value or grade. For example, the confidence value of a purchase is higher than a click, or a higher interaction count implies higher confidence in relevance. The remaining feedback forms a training set.

We experiment with both WRMF and BPRMF as an MF model, $\mathcal{M}_{MF}$. Our implementations are based on [7] and [10], respectively. In all experiments, we compare five different predictive models over an $\mathcal{M}_{MF}$: $\mathcal{M}_{MF}$+EP uses the exhaustive top-N prediction in Figure 5.2 with min-heap-based sorting, $\mathcal{M}_{MF}$+R-PCA and $\mathcal{M}_{MF}$+LEMP-LI follow from Section 5.5, and two versions of MMFNN($\mathcal{M}_{MF}, \mathcal{M}_{NN}$) use KMT and RKT, respectively. We observe that, with proper choice of hyperparameter values through validation, WRMF and BPRMF often produce comparable results with respect to the tested performance criteria. Therefore, we only report the results of the slightly better $\mathcal{M}_{MF}$ for each dataset, that is, $\mathcal{M}_{MF}$ = WRMF for `LASTFM2`, `MSD`, `TMALL` and $\mathcal{M}_{MF}$ = BPRMF for `XING`, `AMZBK`. We choose the number of factors, $f = 50$, for each $\mathcal{M}_{MF}$. To show the results are far from trivial, we also compare to a most popular (MP) baseline in which the most popular items not existing in a user's feedback history are recommended as top-N predictions.

For training the neighborhood model, $\mathcal{M}_{NN}$, in an MMFNN($\mathcal{M}_{MF}, \mathcal{M}_{NN}$), we refer to [126] and its accompanying C++ implementations [140] for KMT and RKT which are also integrated into our implementation. To decide the nearest neighbor search approximation level for each dataset, we try to achieve predictive power at least as good as that of $\mathcal{M}_{MF}$+EP. For this purpose, we first find the best hyperparameter values for KMT and RKT to achieve on the order of 80-90% precision in nearest neighbor search on latent item vectors, $\mathbf{Q}$. We further refine these hyperparameter values through validation for finding a good speedup trade-off for the top-N performance criteria used in this chapter. In general, we find using $\iota = 5$ and $\kappa = 32$ for KMT

and $\tau = 8$ kd-trees for RKT adequate, and we fix these hyperparameter values for all experiments. We also choose ν and ν' from $\{64, 128, 256, 512\}$.

In principle, the accuracies of R-PCA and LEMP are upper-bounded by that of EP since their primary aim is to find the exact highest dot products. For R-PCA, we obtain the results by setting the tree depth hyperparameter value to 3. For some datasets, a more shallow tree slightly improves the accuracy, but then the prediction time becomes almost the same as EP. LEMP-LI is often the best performing variation of LEMP as reported in [135]. We find its suggested default hyperparameter values suitable for our datasets as well. We adapt the efficient C++ implementations [141] of R-PCA and LEMP-LI to our experimental setup.

We first measure HR@N and present the results in Figure 5.5 for $N = 10$. In our experiments, selecting N between 5 and 50 does not change the general trend in the results. For all datasets, we observe that HR@N for MMFNN versions is equivalent or better compared to exhaustive prediction. We also observe that MMFNN with RKT is slightly better than the KMT version. As expected, the HR@N results for R-PCA and LEMP-LI are comparable to that of exhaustive prediction with the former being sometimes slightly worse in order to benefit from some speedup.

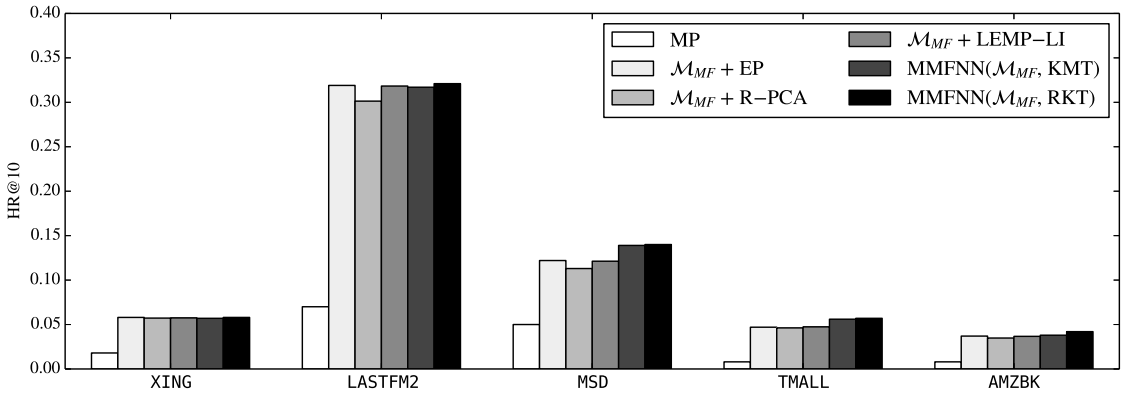

Figure 5.5. HR@N results for different datasets.

To quantify the positions of the holdout set items in the top-N predictions, we also measure the mean reciprocal rank (MRR). We assume zero reciprocal rank if

$\hat{r}_u(i_u^\star) > N$ where $i_u^\star$ refers to an item in the holdout set. MRR results are presented in Figure 5.6. We observe that MMFNN versions have an effect towards improving the ranks of items in the holdout set compared to exhaustive prediction. On the other hand, R-PCA and LEMP-LI results are aligned similarly with the exhaustive prediction results as in the HR@N experiments.

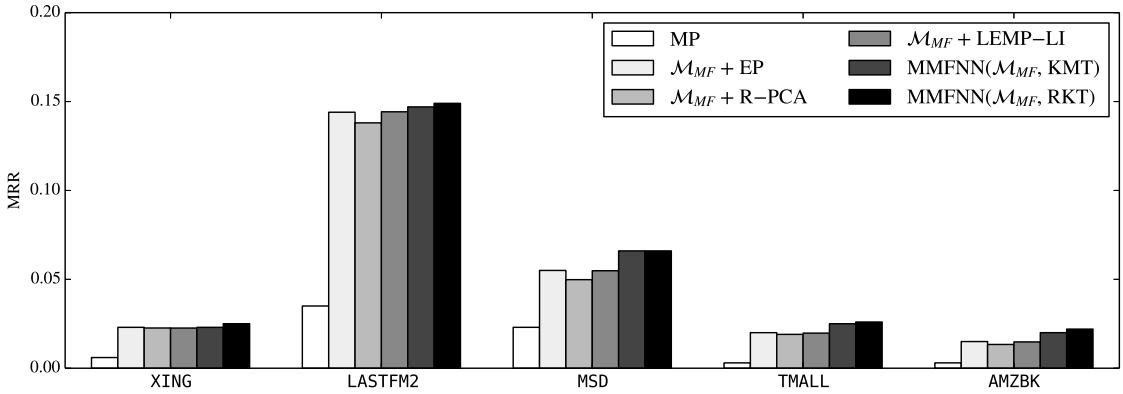

Figure 5.6. MRR results for different datasets.

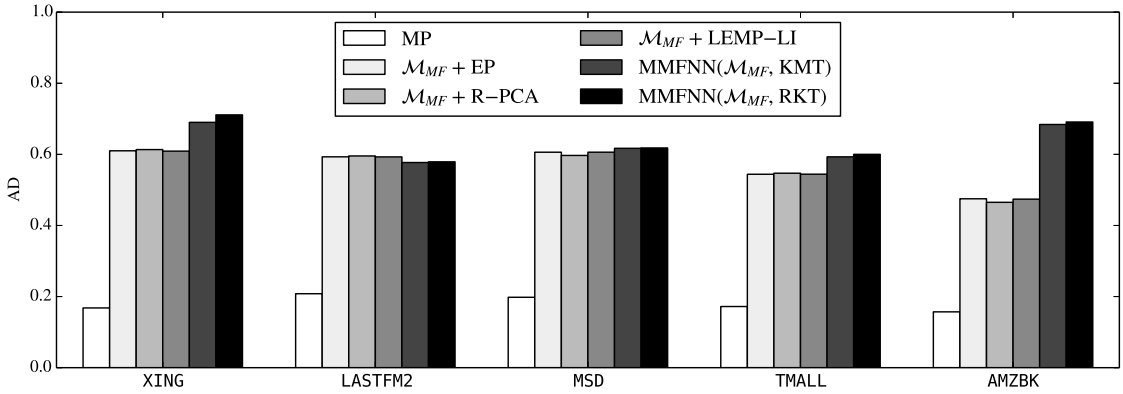

Figure 5.7. AD results for different datasets.

As mentioned in Section 2.2.5, ranking accuracy can have a trade-off with diversity of predictions. Therefore, we also measure aggregate diversity (AD) across top-N predictions and present the results in Figure 5.7. We observe that the level of AD usually persists and even increases when MMFNN versions are used. This is especially observable for the datasets with a larger number of items, $|I|$. Therefore, MMFNN

versions do not necessarily suppress AD while maintaining or improving HR@N and MRR.

The experimental results for ranking accuracy and diversity show that MMFNN can perform comparably and even better than exhaustive top-N prediction. They also show that both R-PCA and LEMP-LI can match the results of exhaustive prediction. We now present efficiency comparisons between different approaches to have a better idea of their effectiveness. We refer to $T_{learn}^{\mathcal{M}_{NN}}$ as the time to learn an $\mathcal{M}_{NN}$ on $\mathbf{Q}$ and $T_{predict}^{\mathcal{M}_{NN}}$ to find $kNN(i, I, k)$ for all $i \in I$. We also refer to $T_{predict}^{\text{top-}N}$ as the average time for top-N prediction for a single queried user. The time comparisons for training and predicting from an $\mathcal{M}_{NN}$ are given in Table 5.2. We apply no parallelism for $T_{learn}^{\mathcal{M}_{NN}}$ while we apply 2-core parallelism for $T_{predict}^{\mathcal{M}_{NN}}$. We first observe in Table 5.2 that RKT has a better $T_{learn}^{\mathcal{M}_{NN}}$ than KMT. While $T_{predict}^{\mathcal{M}_{NN}}$ for KMT and RKT are comparable, we observe that KMT can spend more time for finding nearest neighbors as $|I|$ gets very large. As seen in Section 5.4.2, this can be further improved by selecting a larger κ at the expense of increased $T_{learn}^{\mathcal{M}_{NN}}$ or by selecting a smaller ν. Nevertheless, the time spent for $T_{learn}^{\mathcal{M}_{NN}}$ and $T_{predict}^{\mathcal{M}_{NN}}$ can often be considered negligible since these operations are done only once in a training session, and given how they later facilitate top-N predictions. For R-PCA, $T_{learn}^{\mathcal{M}_{NN}}$ is quite short since it builds a single tree efficiently. For LEMP-LI, we do not have $T_{learn}^{\mathcal{M}_{NN}}$, but an efficient partitioning of items by their lengths which takes well under one minute. $T_{predict}^{\mathcal{M}_{NN}}$ applies to neither R-PCA nor LEMP-LI. Finally, the time comparisons for top-N prediction queries are given in Figure 5.8. We apply no parallelism for obtaining $T_{predict}^{\text{top-}N}$ results. The main observation here is that both MMFNN variants can bring a time efficiency improvement on the order of hundreds of times compared to exhaustive prediction. This clearly facilitates faster computation times when the top-N predictions are precomputed and fast retrieval when there is a need for top-N computations in real time. We also observe that both R-PCA and LEMP-LI methods are quite promising for top-N prediction. On the other hand, we find that despite the somewhat increased precomputation times given in Table 5.2, MMFNN variants can bring a more remarkable speedup while preserving and even improving the results with respect to various other performance criteria such as ranking accuracy and diversity. This makes the MMFNN approach a

highly considerable solution for improving prediction efficiency of matrix factorization models.

Table 5.2. $T_{learn}^{\mathcal{M}_{NN}}$ and $T_{predict}^{\mathcal{M}_{NN}}$ in minutes for different datasets.

Dataset	R-PCA		$\mathcal{M}_{NN}$ = KMT		$\mathcal{M}_{NN}$ = RKT	
	$T_{learn}^{\mathcal{M}_{NN}}$	$T_{predict}^{\mathcal{M}_{NN}}$	$T_{learn}^{\mathcal{M}_{NN}}$	$T_{predict}^{\mathcal{M}_{NN}}$	$T_{learn}^{\mathcal{M}_{NN}}$	$T_{predict}^{\mathcal{M}_{NN}}$
XING	0.1	-	0.7	1.7	0.3	1.9
LASTFM2	0.1	-	0.2	0.3	0.1	0.3
MSD	0.1	-	0.3	0.8	0.1	0.7
TMALL	0.1	-	0.9	1.6	0.3	1.5
AMZBK	0.3	-	1.9	8.8	0.7	3.4

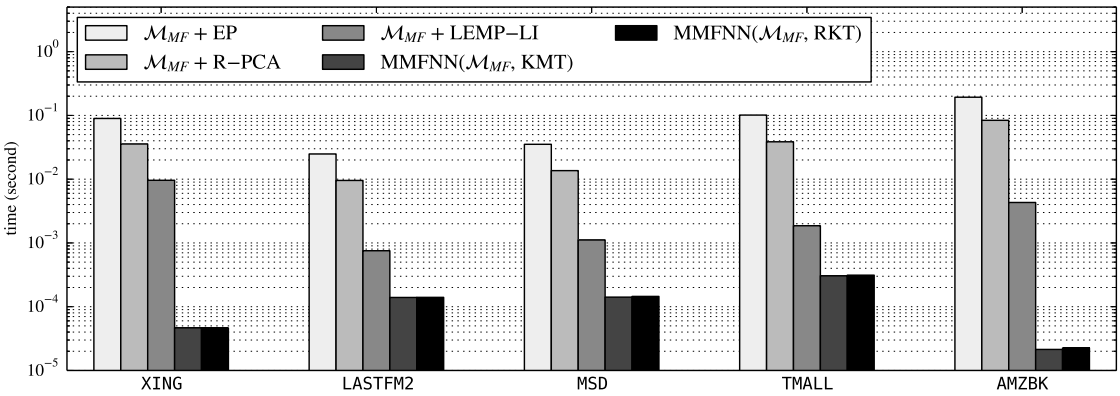

Figure 5.8. Log plot of top-N prediction times for a single user, $T_{predict}^{\text{top-}N}$.

5.6.2.2. Time-split Experiments. In this type of experiment, we use the time information available in XING and TMALL datasets. We make a time split for each dataset such that the user feedback before the split constitutes a training set whereas the user feedback after the split constitutes a test set. This allows us to experiment with various matrix factorization models including the incremental ones.

We compare four algorithms for each dataset: We first look at the performances of BPRMF+EP and MMFNN(BPRMF,RKT). Then, we look at incremental learning

performance using two algorithms: PLtR-NS+EP and MMFNN(PLtR-NS,RKT). All algorithms including the incremental PLtR-NS algorithm are based on BPRMF so that they are comparable. All $\mathcal{M}_{NN}$ are based on RKT due to its slightly superior performance in the experiments with a random holdout set.

The experimental results are given in Figure 5.9. We use MAP@N to compare performance of the algorithms. For the reported results, a time split is made at about 99.5% of each time-ordered dataset so that a portion of the last day user feedback is used as the test set. The buffer ($\mathcal{B}$) size in the incremental algorithms is fixed to 100,000 for each dataset, and sliding windows are used. All other hyperparameter values are the same as those validated for the random holdout set experiments. We first observe that BPRMF+EP and MMFNN(BPRMF,RKT) are comparable to each other with the latter having slightly better results. These results are also in accordance with those in the random holdout set experiments. Similarly, the results for the two incremental algorithms are also comparable with MMFNN having somewhat better accuracy for the second dataset. These results imply that the meta-algorithm is effective in all learning settings by preserving the ranking accuracy while producing the top-N predictions faster. Additionally, we observe that the incremental algorithms perform better than the non-incremental ones. Therefore, MMFNN can be very useful for their practicality by improving their prediction efficiency.

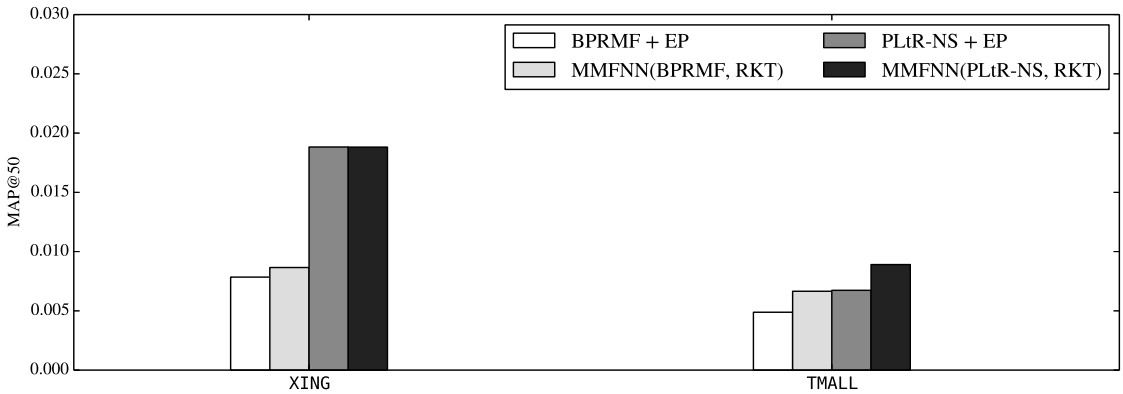

Figure 5.9. MAP@N results of time-split experiments with non-incremental and incremental algorithms.

5.6.2.3. Experimenting with RL Extension. In this section, we present an experiment with the MMFNN extension given in Figure 5.4. Unfortunately, it is not always easy to design robust offline experiments for MAB algorithms [63]. Online experiments are needed but usually not accessible for academic research. The difficulty in offline experiments is because the offline data is the result of another policy or algorithm, and how this policy exploits and explores is often unknown. An unbiased off-policy evaluation scheme is proposed in [142]. However, it is based on reject sampling and ignores many input tuples, which poses problems in MF models where every user is a distinct context and data is already sparse.

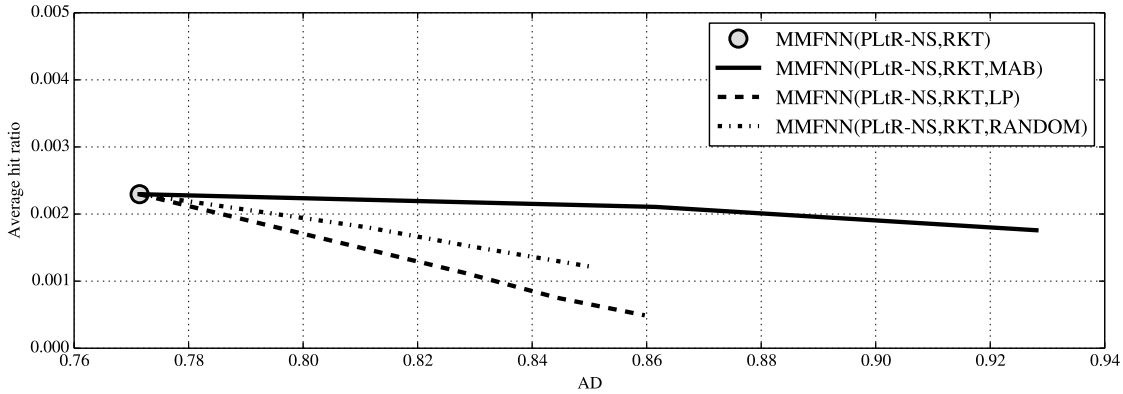

Figure 5.10. MMFNN with MAB on XING.

Therefore, rather than a regret analysis, we present an experiment on how the extension affects ranking accuracy and, at the same time, diversity through reranking [70]. Although, in this experiment, the upper bound of ranking accuracy results is expected to be comparable to the ranking accuracy of MMFNN($\mathcal{M}_{MF}, \mathcal{M}_{NN}$), it can still be interesting to observe how MAB can increase aggregate diversity through exploration when incorporated into MMFNN. The results are given in Figure 5.10. We compare the MAB extension to two other possible extensions for increasing aggregate diversity: LP reranks the N' items with respect to popularity in ascending order, and RANDOM takes a random subset of size N from the N' items. We report the average hit ratio based on the sliding windows of PLtR-NS in comparison to achieved aggregate diversity. We use $N = 1$ and test for N' within range $(1, 20]$. We observe that the MAB

approach has a much better accuracy-diversity trade-off. Consequently, we see that N' can be used as a convenient exploration hyperparameter, and it can be fine-tuned to increase diversity effectively while maintaining comparable ranking accuracy.

6. LEARNING INTENTION IN USER SESSIONS

6.1. Introduction

Learning and predicting user intention in a session can be a challenging task due to limited available information as well as response time constraints. In this chapter, we propose powerful and efficient methods for learning users' purchasing intention from session-based implicit feedback. Our proposal fits the personalized learning to rank setting in two ways:

(i) Using a pointwise scheme, we estimate a probability of purchase for session items similar to a ranking score.

(ii) Good estimates may facilitate better recommendations using this extra piece of information about the session. For example, if session indicates a high probability of purchase, we can narrow down diversity of recommendations to a specific set of items [45]. Similarly, if there is low purchasing intention, we can switch between recommendation algorithms.

In particular, we first propose a way to represent the intention learning problem by converting session information into feature vectors. Then, we propose batch and stream versions of an ensemble learning method which can take into account the high class imbalance due to the rarely occurring purchasing feedback. We also propose several ways for efficient prediction from this ensemble since the prediction of intention can be time sensitive. We show our experimental results on a recent e-commerce challenge benchmark dataset.

The remainder of this chapter is organized as follows: In Section 6.2, we represent the intention learning problem in a feature vector space. Then, we propose suitable models in Section 6.3 for learning from this representation as well as methods for improving model prediction efficiency. In Section 6.4, we present experimental results for the predictive power of the proposed models.

6.2. Problem Representation

Let I be the set of items in the system. A session can be formalized as a sequence of events, $s = (e_1, e_2, \ldots, e_\ell)$, with arbitrary length. At a minimum, each event is represented by a feedback instance $e = (i, t, a)$ where $i \in I$, t is a timestamp, and a is an event action such as a click, view, or purchase. Notice that a session brings context to a sequence of events happening in the system. If the session user is unknown to the system, personalization can only rely on the session information. Otherwise, known user information can also be incorporated into the context.

Given a dataset of sessions, $\mathcal{D}_s$, the ultimate goal of an intention model is to predict the intentions of interest in each session accurately and efficiently. That said, an important predictive goal in an RS is to assess the purchasing intention in a session and possibly which items in the session are likely to be purchased [31]. Then, effective predictions may enable better personalized recommendations. While the discussed models in this chapter can also be useful in other similar problems, we carry on with this predictive goal.

We observe that even without using specific user or item content features, $\mathcal{D}_s$ contains plenty of information which can be inferred from implicit feedback and used to formulate the problem as a pointwise learning task. Accordingly, we propose to represent each session-item pair with a feature vector based on temporal features as well as additional session- and item-based statistics inferred mainly from implicit feedback. For the benchmark dataset used in this chapter, we summarize these features [14] in Table 6.1 with further clarifications in Figure 6.1. While most of the features are self-evident, simple exploratory data analysis on individual features helps to reveal significant differences between session-item pairs which result in a purchase or not. We provide more detail on the benchmark dataset in Section 6.4. Nevertheless, we note that similar features can be derived in many typical intention learning scenarios. Furthermore, the choice of such a feature vector space model enables inclusion of user and item content features whenever they are available.

Table 6.1. Representation of a session-item pair.

	Explanation of feature
F_s	Session ID
F_i	Item ID
F_1	Weekday when first click to item with Item ID happened during that session
F_2	Hour when first click to item with Item ID happened during that session
F_3	*Duration in seconds the session lasts*
F_4	Duration in seconds between the item with Item ID is first clicked and last clicked in that session
F_5	*Total number of clicks in the session*
F_6	Number of clicks to item with Item ID in the session
F_7	*Number of distinct categories session items belong to*
F_8	Category that the item with Item ID belongs to
F_9	*Average of purchases of distinct items in the session*
F_{10}	Number of purchases of item with Item ID
F_{11}	*Average of ICRs of distinct items in the session. Item conversion rate (ICR): Ratio of the number of sessions in which an item is purchased to the number of sessions in which it is clicked.*
F_{12}	ICR of item with Item ID
F_{13}	Average of intervals between clicks to the item with Item ID in the session
F_{14}	Composite feature: F_4/F_3
F_{15}	Composite feature: F_6/F_5
F_{16}	Composite feature: F_{10}/F_9
F_{17}	Composite feature: F_{12}/F_{11}
F_{18}	*Number of different subsequent items in click sequence divided by number of all subsequent item pairs. (This is some sort of energy measurement to see how clicks in the session alternate)*
F_{19}	Sum of durations in seconds spent on the item with Item ID until a different item is clicked
F_{20}	True if the item with Item ID is the first clicked item in session
F_{21}	True if the item with Item ID is the last clicked item in session
F_{22}	*Number of distinct items in the session*
F_{23}	Composite feature: F_{19}/F_3

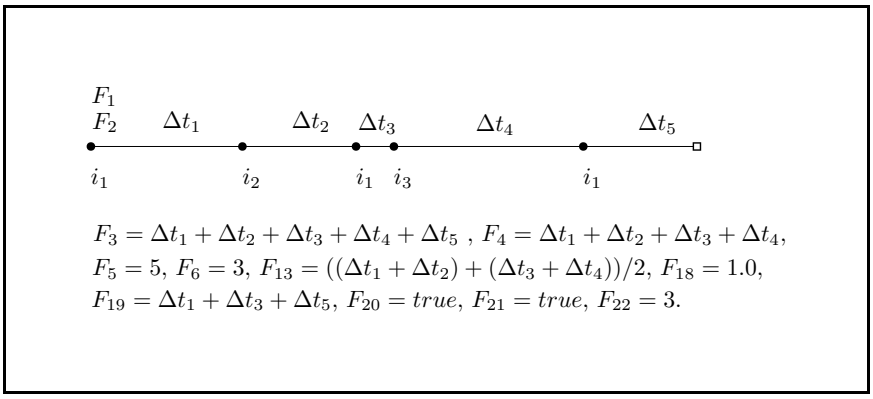

Figure 6.1. Session as a timeline. Dots represent click events involving certain items. Some features for the session item i_1 are computed as an example.

Therefore, our intention learning model uses a suitable feature representation and tries to find a good estimate of the probability, $p(\text{purchase}|s, i, \mathcal{D}_s)$. In principle, it is possible to obtain a prediction for any (s, i) where $i \in I$. More practically, we perform predictions for $i \in I_s^+$ where I_s^+ is the set of items interacted during the session s. Then, for each session, the predictions can be thought to result in a set, $\hat{Y}_s$, of items with a $p(\text{purchase}|s, i, \mathcal{D}_s)$ higher than a given threshold. Oftentimes, the expectation is that $\hat{Y}_s = \emptyset$.

6.3. Learning Models

Estimation of $p(\text{purchase}|s, i, \mathcal{D}_s)$ can be modeled with a binary classifier [39]. Here we note that a more general intention model can be based on multi-label classification which can still be learned using multiple binary classifiers [143]. Therefore, we assume a training set of $(\mathbf{x}, y)$ where $\mathbf{x}$ is an f-dimensional feature vector representing a pair (s, i) and $y \in \{0, 1\}$ is a class label. Among various alternatives for solving the classification problem, we focus on Breiman's random forest (RF) [144] and propose suitable extensions for learning and prediction tasks. These extensions are detailed in the rest of this section.

The basic RF is an ensemble of decision trees which offers various sources of randomness to break correlations among the features and also the trees in the ensemble.

Typically, each tree works on a bagged version of the input dataset. Furthermore, each node split in each tree is decided by considering a random subset of features rather than the whole feature set. The expectation is that the variance of the final learner is lower than individual learners while its bias is compensated. Besides being a powerful non-linear classifier and regressor, RF also has additional benefits in our case: First, it can directly handle a mixture of categorical and real features without necessitating extra encoding schemes. Furthermore, feature selection is inherent. Second, the training of the ensemble is embarrassingly parallel which facilitates working with large datasets. Third, it can output class posterior probabilities which can be used for post-processing the results.

6.3.1. RF for Imbalanced Data

Typically, the number of sessions with a purchase is much smaller which indicates a high class imbalance problem for the learner. Often, a proper strategy should be selected to balance the majority and minority class examples in the learning process to achieve useful predictions. To this end, several strategies are proposed in the machine learning and data mining literatures as a preprocessing step or embedded logic in the classifier ensemble [145]. Accordingly, we observe that the approaches based on downsampling the majority class are quite suitable for handling the class imbalance problem in RFs compared to oversampling and hybrid approaches in terms of predictive power and efficiency with large datasets [14]. Downsampling the majority class for RF is also proposed in [146] and tested on smaller datasets from different application areas.

For our problem, we propose to use the following approaches which combine RF with downsampling to handle imbalanced data:

(i) Before training an RF, preprocess the dataset by keeping all the minority class examples, X^+, and a proportional number, $\eta \times |X^+|$, of majority class examples by random downsampling. For example, when $\eta = 1$, the classes are equally balanced. Merge the two sets as a fixed training set. Each tree in the RF takes a

Initialize an ensemble of M trees ;
for all learner h_m, $m = 1 \ldots M$ (in parallel) **do**
  $X^{++} \leftarrow$ Choose a bootstrap sample of size n_+ from minority class ;
  $X^{--} \leftarrow$ Draw $\eta \times n_+$ examples from majority class with replacement ;
  $X' \leftarrow \text{merge}(X^{++}, X^{--})$;
  Learn h_m on X' by deciding each node split through $f' < f$ random features ;
end for

Figure 6.2. BRF2 algorithm.

bootstrap sample from this training set. We call the resulting balanced random forest BRF1.

(ii) Differently from BRF1, before training each tree in an RF, first take a bootstrap sample of size $n_+ \leq |X^+|$ from the set of all minority class examples. Then, take a proportional number of majority class examples using sampling with replacement. This approach assumes access to a larger portion of the entire dataset. However, it can gather useful extra information while providing more randomness for the RF. We call the resulting random forest BRF2 and show it in Figure 6.2 for clarity. In the next section, we also propose an RF which makes use of this scheme and tries to learn directly from streaming data.

6.3.2. Balanced RF for Imbalanced Streams

In this chapter, we also propose [15] a variation of BRF2 which can learn in an incremental fashion. Such an algorithm can sometimes be preferred for learning user intention since it can efficiently and adaptively learn from streaming data. The BRFIS algorithm given in Figure 6.3 performs online sampling for balancing examples from every class as well as online bootstrap sampling for each learner in the ensemble. Each learner is a Hoeffding (or VFDT) tree [147] which can learn from continuously incoming data without the need for periodic retraining. We allow each decision node in each tree to work on a random subset of the features with a cardinality $f' < f$. The trees are

independent, and they can learn in parallel. In the following, we explain the building blocks of the BRFIS algorithm in more detail.

```
Initialize M Hoeffding trees with hyperparameters δ, τ, n_min, and such that each
decision node can choose f' < f random features ;
for all (x, y) from stream do
  for all learner h_m, m = 1 ... M (in parallel) do
    Draw a ~ U(0, 1) ;
    if a < ρ_y then
      Accept example (x, y) ;
    end if
    if (x, y) is an accepted example then
      Draw k ~ Poisson(1) ;
      Update h_m with k instances of (x, y) or (x, y) weighted by k ;
    end if
  end for
end for
```

Figure 6.3. BRFIS algorithm.

6.3.2.1. The Base Learner. The main problem of incrementally-learning trees is to decide when to split a decision node since we do not have all the data at the beginning of the learning process. A Hoeffding tree is based on the observation that a relatively small number of data examples which arrive at a node often bear enough statistics for a split decision. This observation enables a family of algorithms which can learn from massive streams using fewer computational resources and with a performance similar to batch decision trees given enough examples. The basic Hoeffding tree algorithm works as follows: When a new labeled example from the stream arrives, it traverses the tree down to a leaf node with respect to its feature values, and the required statistics at that node are updated. If there is enough support in favor of a split, the leaf node is transformed into a decision node. The Hoeffding bound is useful to determine the sample size to observe before this transformation is made. Assume n independent

observations of a random variable Z with range R. The Hoeffding bound states with confidence $1-\delta$ that the true mean of Z is at least $\bar{z}-\epsilon$ without making any assumptions about the probability distribution. In this case,

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}, \tag{6.1}$$

and $\bar{z}$ is the sample mean. Let $\mathcal{I}$ be a function which can evaluate the difference before and after a node split by a feature. Suppose after observing n examples in a leaf node, F_A and F_B represent the features with the first and second best results with respect to $\mathcal{I}$. Let $\Delta\bar{\mathcal{I}} = \bar{\mathcal{I}}(F_A) - \bar{\mathcal{I}}(F_B)$ represent a random variable for the observed difference. In case $\mathcal{I}$ is information gain, $R = \log_2(c)$ where c is the number of classes. Then, if $\Delta\bar{\mathcal{I}} > \epsilon$ holds for a given δ, because the difference is greater than zero, we can choose F_A as the splitting feature with δ confidence according to the Hoeffding bound. If the leaf node is not pure enough, the algorithm frequently checks the value ϵ to decide for a split. For efficiency, the checks are performed every time when $(n \mod n_{min}) = 0$ for a given n_{min}. If the top features continuously exhibit a similar behavior preventing the split decision, then the algorithm declares a tie condition. This situation is handled by choosing the best feature for split when $\epsilon < \tau$ for a given threshold τ. Other variations of this basic learning algorithm can be found in [62,147]. We note that Hoeffding trees are unstable which make them good candidates for ensemble approaches [74].

6.3.2.2. Online Bootstrap Sampling. Given a dataset of $|X|$ examples, a bootstrap sample for each learner in an ensemble is created by drawing $|X|$ examples with replacement. Then, every learner's training set contains k copies of each of the original training examples, where k is binomially distributed with $k \sim B(|X|, 1/|X|)$. In the stream learning setting, a practical and useful assumption is that $|X| \to \infty$. In this case, the distribution of k tends to $Poisson(1)$ distribution [148],

$$P(k = \kappa) = \frac{\mathrm{e}^{-1}}{\kappa!} \tag{6.2}$$

as also shown in Figure 6.4. Therefore, every learner in the ensemble can simply accept $k \sim Poisson(1)$ copies of every example coming from the stream irrespective of $|X|$.

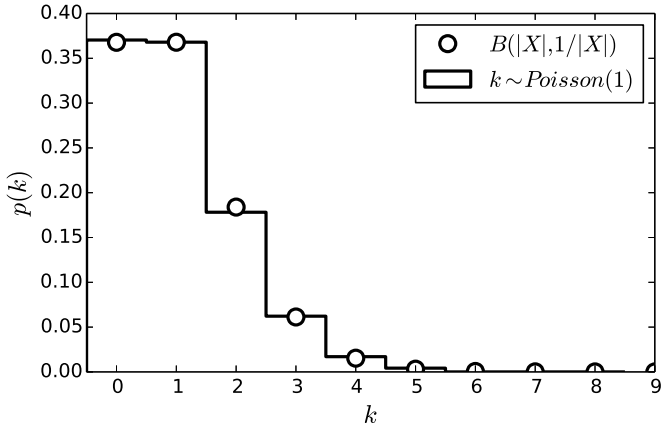

Figure 6.4. Simulation of distribution of k for $|X| = 5,000$.

6.3.2.3. The Ensemble. The BRFIS algorithm in Figure 6.3 can learn an RF from streaming data as follows: For each learner in the ensemble, a balanced number of different class examples are randomly sampled using a class-based sampling ratio, ρ_y. For instance, in binary classification ($y \in \{0, 1\}$), if we observe 1000 majority and 50 minority class examples, we can expect that choosing $\rho_0 = 0.05$ and $\rho_1 = 1.00$ yields roughly the same number of examples from each class. While these values can be decided by observing the stream, they can also be changed adaptively as the stream flows. Subsequently, online bootstrap sampling is used to decide how many copies from each example are to be taken. For instance, the probability of resampling 0 copies is approximately 0.368 and equals that of a single copy as also illustrated in Figure 6.4. Then, the examples are fed into the learner which works on f' random features. Choosing $f' = \sqrt{f}$ or $f' = \log_2 f$ is often practical [144]. The learners can work in parallel. While the algorithm in Figure 6.3 passes streaming examples to learners one by one, it is also possible that the learners work on moderately-sized buffers of streaming examples.

Incremental learning from streams incorporates the latest examples fast into the model. To make the model even more adaptive, some helper mechanisms can be useful. One obvious mechanism is to discard the ensemble periodically and start training a new one. It is possible that the subsequent ensembles are allowed to have some overlapping examples. Another mechanism is to monitor accuracy of the learners. For example, it is proposed [149] to monitor the out-of-bag (OOB) error in each tree through time and discard a tree with a probability proportional to its age and OOB error. Notice that the OOB error comes from the misclassified examples that are not in the bootstrap sample (bag) for that tree and it can be decided online. In our case, online bootstrap sampling assures that approximately 1/3 of the balanced examples are out of bag.

6.3.3. Improving Prediction Efficiency

After training an RF model, predicting $p(\text{purchase}|s, i, \mathcal{D}_s)$ for a single (s, i) requires $O(M \times d_{max})$ worst-case computations where M is the number of trees, and d_{max} is depth of the deepest tree in the ensemble. This suggests that the RF algorithm should be carefully implemented and parameterized to find a balance between predictive power and the allowed d_{max}. It is shown that increasing M does not cause overfitting [144], and it is usually useful for improving predictive power. On the other hand, as M becomes larger, the prediction time becomes longer which may be problematic when predictions are time sensitive.

One solution to deal with a large M in prediction tasks, is to use a subset of trees in the ensemble with respect to the available computational budget. To this end, a straightforward and viable approach is to use a random subset with as many trees as the budget allows. Moreover, greedy forward and backward tree selection approaches are proposed [150]. The former adds sequentially to the subset the next tree which brings the best performance improvement while the latter excludes the next tree whose exclusion causes the least performance loss. These approaches are shown to potentially improve over random subset selection at the cost of extra computation time.

Another solution can be based on the idea of distilling or compressing a complex model into a much simpler model [151] for prediction. The main idea is concentrating on the output of the complex model rather than its learned parameters and transferring this information into the simpler model. In our case, the complex model is an RF with an output of class probabilities. Assuming binary classification, our approach is to first train a single decision tree regressor (DT1) using the minority class probabilities, $p(y_{\mathrm{RF}} = 1|\mathbf{x})$, obtained from the RF as target variables. Similar to the suggestions in [151], we find it useful to further soften the output probabilities by a scalar, β, in $[0, 1]$, that is, we use $\beta \times p(y_{\mathrm{RF}} = 1|\mathbf{x})$ as the target variable. Then, we also train a separate decision tree classifier (DT2) based on the hard class labels available in the dataset. The final prediction using the distilled model is given by the following combination:

$$p(y_{\mathrm{distilled}} = 1|\mathbf{x}) = \alpha \times p(y_{\mathrm{DT1}} = 1|\mathbf{x}) + (1 - \alpha) \times p(y_{\mathrm{DT2}} = 1|\mathbf{x}), \tag{6.3}$$

where α is a weighting term. Another motivation for using the distillation idea is that it produces compact models which can greatly improve space requirements.

6.4. Experiments

6.4.1. Dataset

We experiment with a challenge benchmark dataset [31]. The dataset contains 6-month session-based event data (between months 4-9) from a large consumer goods e-commerce platform. Each session has a session ID corresponding to a single anonymized user. Each session event has an item ID, a timestamp, and an event action. The event actions mainly refer to user clicks in a session while some relatively fewer sessions contain a purchasing action as well. Table 6.2 shows basic properties of the benchmark dataset. We note that the same item can be clicked multiple times in a session. Overall, the dataset statistics indicate that the sessions with and without purchases are highly imbalanced.

Table 6.2. Basic properties of the benchmark dataset.

Action	Number of events	Number of sessions	Number of items
Click	33,033,944	9,249,729	52,739
Purchase	1,150,753	509,696	19,949

Exploratory data analysis reveals that some feedback characteristics such as distributions of clicks, ICRs, and associated temporal information demonstrate distinguishing patterns for sessions with and without purchases. This analysis leads us to derive a mixture of categorical and numeric features [14] for every item interacted in a session as illustrated in Table 6.1. Global session features are highlighted with italics, and we also clarify some of the features in Figure 6.1.

We use the original data without any cleaning. We do not perform feature discretization or variable transformation. It is usually unclear when a session ends after the last item click. Therefore, we perform a single imputation for the duration after the last click (Δt_5 in Figure 6.1): If in a session, there are click intervals greater than zero second, we take the average of click intervals and accept it as Δt_5. Otherwise, we sample this duration from the distribution of all intervals in the dataset using inversion sampling [118].

6.4.2. Experimental Results for BRF

We apply monthly time windows over the dataset. This approach is somewhat superior to handling the entire dataset at once since, for example, it can capture transient feature values such as item purchases and ICR at a finer granularity.

Typical correlations among features and with the response variable are illustrated in Figure 6.5. We rely on our RF-based approaches to compensate for the effects of higher correlations.

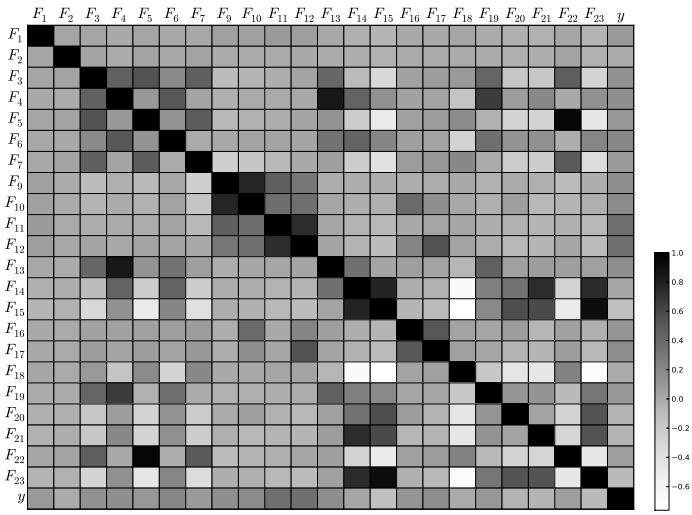

Figure 6.5. Correlations among features and with the response variable for month 8.

Figure 6.6 illustrates mean decrease impurity [152] for each feature which is one way to measure feature importance in an RF. In each tree, importance of a feature is evaluated by adding up the weighted impurity decreases for all nodes where the feature is used for splitting the node. This is averaged over all trees in the ensemble and then normalized over all features for visualization. We see a similar pattern except for the first month. We also observe that ICR-related features are often very important which validates working within a time window to catch important items in that particular time period.

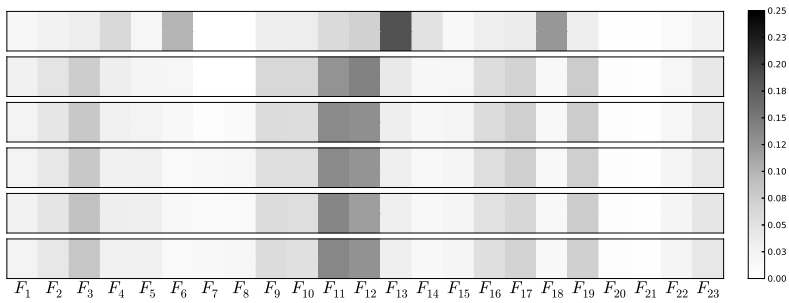

Figure 6.6. Normalized mean decrease impurity for each feature and over the 6 months, respectively. Results are based on BRF1.

To evaluate our models, we use session-based performance criteria [31]. If a given session contains a purchased item, it is assumed to have a positive (or minority) class label. When a dataset is highly imbalanced, measuring classifier accuracy is usually not very informative since classifying most of the examples as a majority class example will produce high accuracy anyway. Furthermore, classifying every example as a minority class example achieves a perfect recall (true positive rate) suggesting that we should also compare to other measures like precision and false positive rate. Therefore, we apply the following evaluation scheme: We handle the sessions on a monthly basis. We make a time split such that the sessions in the last day of the month constitutes a test set whereas the rest constitutes a training set. This way, we preserve the natural time order of the data, the integrity of the sessions, and the actual class imbalance. All our models are trained and tested on the same datasets through which we observe comparative predictive power in terms of ROC and precision-recall curves. Additionally, we observe two point estimates: The first is the F1 score which is the harmonic mean of precision and recall. The second is the Jaccard similarity, $J(s) = |Y_s \cap \hat{Y}_s|/|Y_s \cup \hat{Y}_s|$, between the set of predicted item purchases, $\hat{Y}_s$, and the set of true item purchases, Y_s, averaged over all true positive sessions in the test set. A preliminary version of this evaluation scheme using 2-fold cross-validation is reported in our previous work [14].

We show experimental comparisons of the algorithms in Figures 6.7 and 6.8 as well as in Table 6.3. Our baseline is a single decision tree (DT) which works on the same set of features but decides the best split over all features rather than a subset. BRF1 and BRF2 decide each split on a random subset of $\sqrt{f}$ features. As explained in Section 6.3.1, each tree in BRF1 takes a bootstrap sample from a fixed set of examples while in BRF2 each tree is able to take a bootstrap sample from the whole dataset. For BRF2, we choose $n_+ = |X^+|$. When creating the ROC and precision-recall curves, we use the downsampling ratio, η, as decision thresholds. The chosen thresholds are $1/8, 1/4, 1/2, 1, 2, 4$. DT also uses the same downsampling scheme. However, it uses the sampled data directly without bootstrap sampling. We base our implementations on the scikit-learn [153] package. In all implementations, we use a base CART tree and Gini index as the splitting criterion. The trees are fully grown and not pruned. Positive

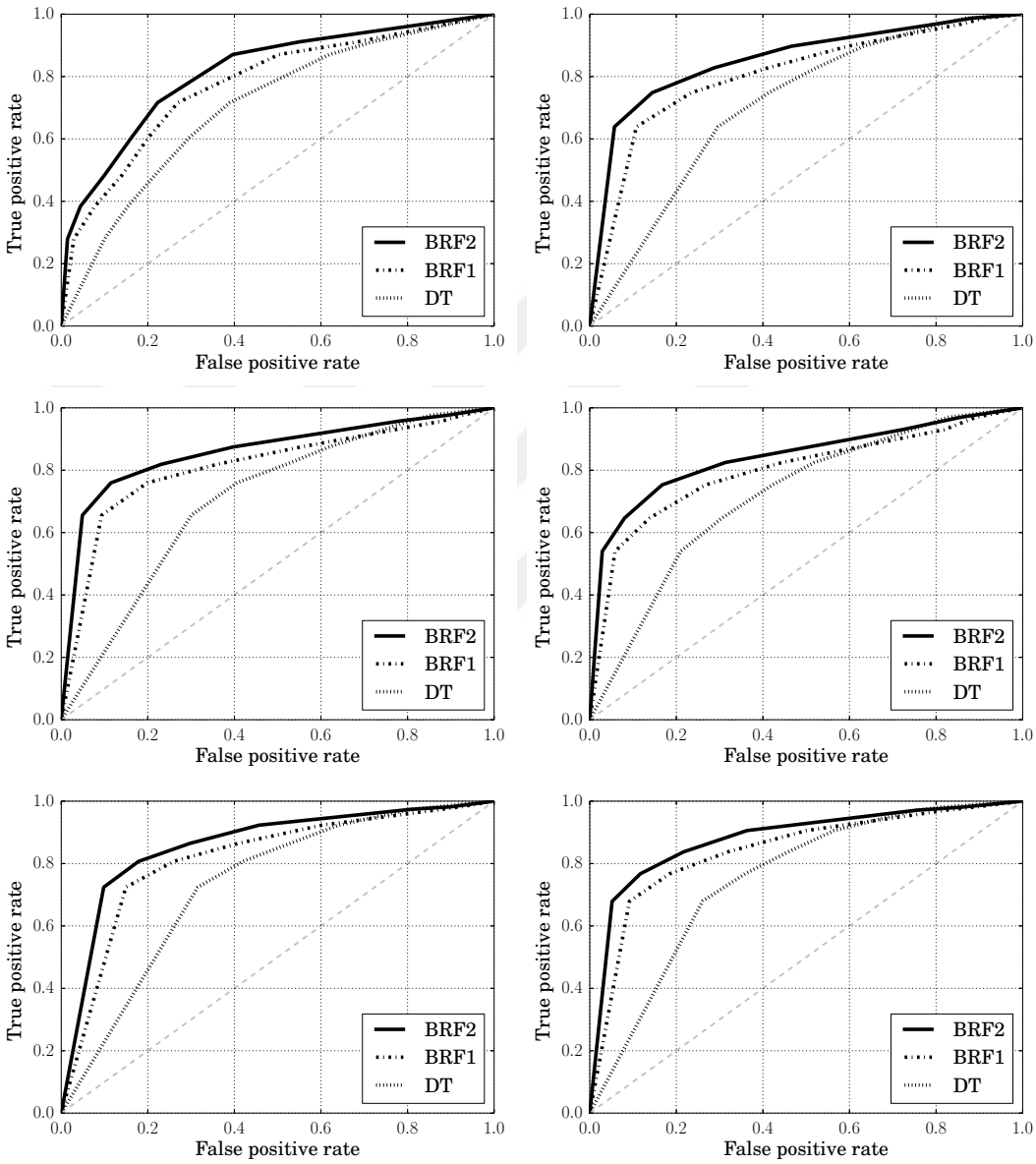

Figure 6.7. ROC curves for different models regarding (Top row) months 4 and 5, (Middle row) months 6 and 7, and (Bottom row) months 8 and 9.

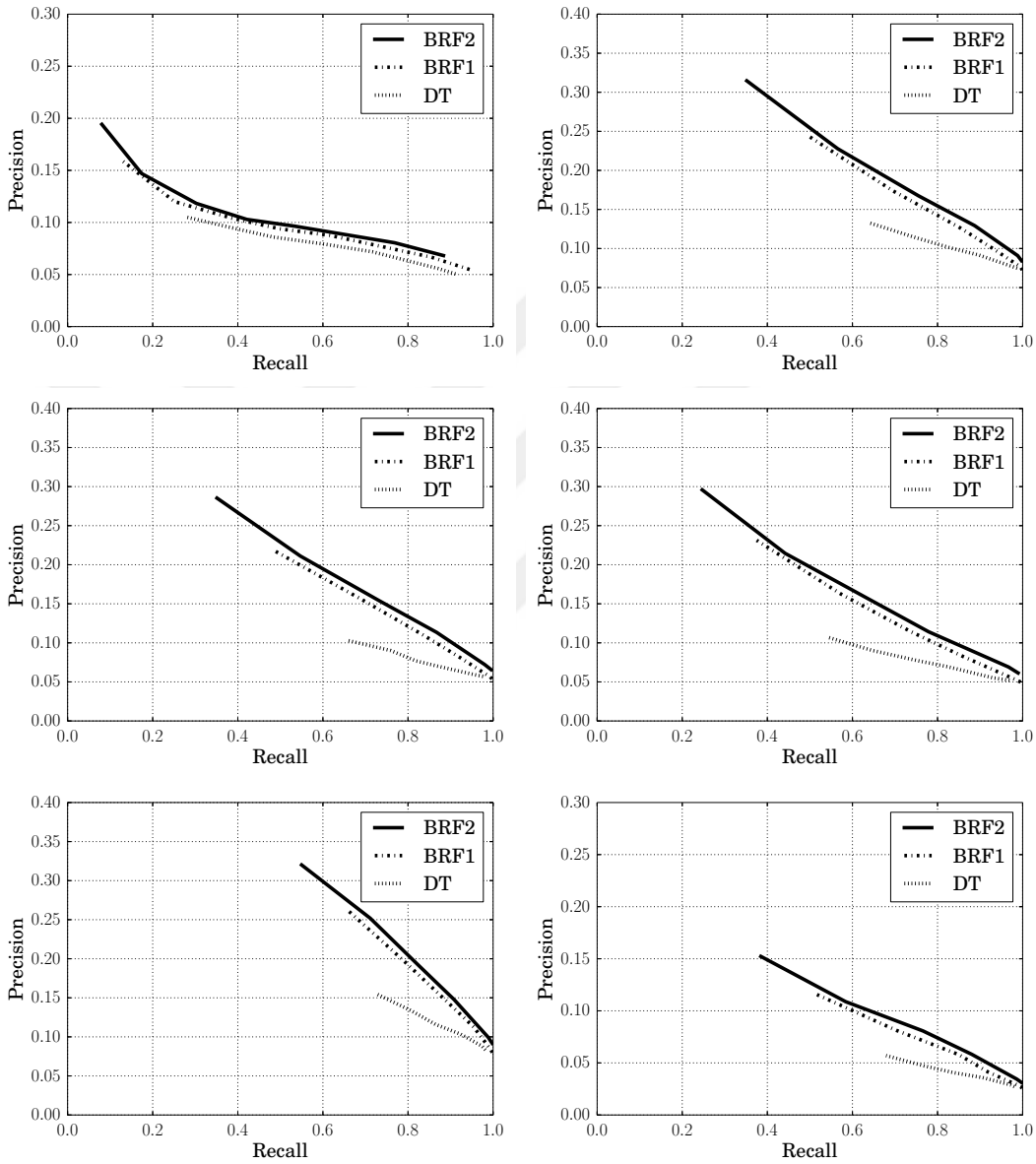

Figure 6.8. Precision-recall curves for different models regarding (Top row) months 4 and 5, (Middle row) months 6 and 7, and (Bottom row) months 8 and 9.

class posterior probability cut-off is chosen to be 0.50. Unless otherwise stated, we use 50 trees in both BRF1 and BRF2. It should be noted that if the number of trees in BRF1 and BRF2 is increased proportional to the available computational resources, the results are expected to improve.

Table 6.3. F1 scores and average Jaccard similarities for different models.

	F1 score			Jaccard similarity		
Timeline	DT	BRF1	BRF2	DT	BRF1	BRF2
Month 4	0.140	0.158	0.159	0.567	0.689	0.688
Month 5	0.180	0.232	0.261	0.616	0.707	0.712
Month 6	0.141	0.204	0.240	0.575	0.667	0.670
Month 7	0.140	0.213	0.241	0.594	0.682	0.683
Month 8	0.204	0.264	0.298	0.589	0.657	0.663
Month 9	0.078	0.119	0.138	0.563	0.655	0.647

In terms of the area under ROC and precision-recall curves, both Figures 6.7 and 6.8 show that BRF versions have a superior predictive power compared to a single DT model. Furthermore, compared to BRF1, BRF2 results are at least as good and usually better. Our first proposal [14] is based on BRF1 with $\eta \approx 1$ and positive class posterior probability cut-off at 0.50. Accordingly, in Table 6.3, we show the results of different models with $\eta = 1$ and a cut-off at 0.50 using F1 score and Jaccard similarity.

6.4.3. Experimental Results for BRFIS

We also experiment with the dataset using the BRFIS algorithm for imbalanced streams. We choose to exclude item popularity features F9, F10, and F16 since they are updated unboundedly as the stream flows. We apply a practical variation of the first-test-then-train procedure mentioned in Sections 2.2.5 and 3.5.2 as follows: The data arrives on a timeline according to the timestamp information. After a short warmup training period, we start testing the incoming $(\mathbf{x}, y)$ pairs from the sessions. Rather than immediately training the model with a tested pair, we buffer the pair within a

time window and perform the training when the buffer is ready. The time windows are roughly the same size and preserve integrity of the sessions in it. We report F1 scores for every time window. Figure 6.9 shows the experimental results.

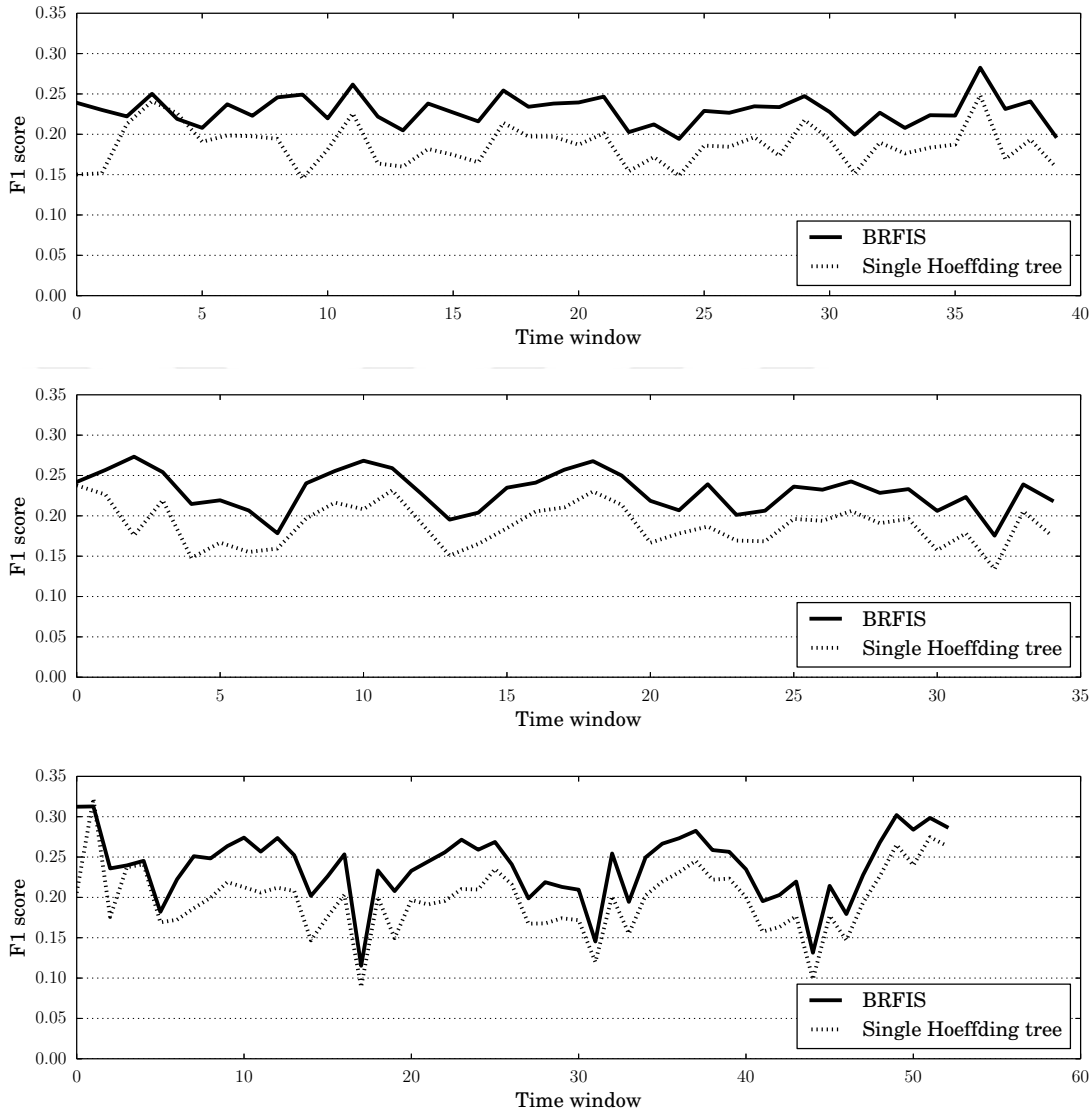

Figure 6.9. Predictions using BRFIS. Rows show months 6, 7, and 8, respectively.

In our experimental setting, the warmup period and the time windows are set to a maximum of 100,000 pairs. We use $\rho_0 = 0.05$ and $\rho_1 = 1.00$. $f' = \sqrt{f}$. We compare BRFIS to a single Hoeffding tree which learns from the same downsampled stream. However, it uses all f features to decide a split and does not perform bootstrap sampling. All trees use the values $\delta = 10^{-7}$, $\tau = 0.05$, and $n_{min} = 200$. We use a

separate ensemble for each month, but we do not apply any other pruning or discard procedure within a month. Our Java-based BRFIS implementation uses the MOA framework [154] API for the robust Hoeffding tree implementation.

We use BRFIS with $M = 20$ trees which can be a practical setting for parallel stream processing. We first observe a statistically significant increase in the comparative F1 scores compared to the single Hoeffding tree. Second, we observe that the results are comparable to batch learning BRF2 in Section 6.4.2. This suggests that incremental learning can be preferred for time and space efficiency. Our problem representation enables inclusion of more session and item features as well as personalized features whenever available. This has a potential to further improve the predictive power of the incremental model.

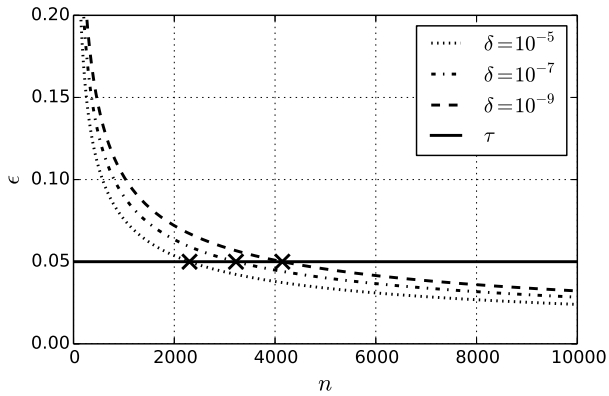

Figure 6.10. Relation of some model hyperparameters to ϵ.

Finally, in Figure 6.10, we clarify some of the hyperparameters tuned in the experiments. We show the change in ϵ of the Hoeffding bound with respect to the change in the number of examples, n, in a decision node. Since, we perform binary classification, $R = 1$. The expansion of a Hoeffding tree also depends on δ and τ. When δ changes, the transiency of the function changes with respect to n and the choice of τ affects the node split slightly differently. On the other hand, our validation results within this range of hyperparameter values show that the BRFIS model is not overly sensitive to small changes in δ given the current experimental setting.

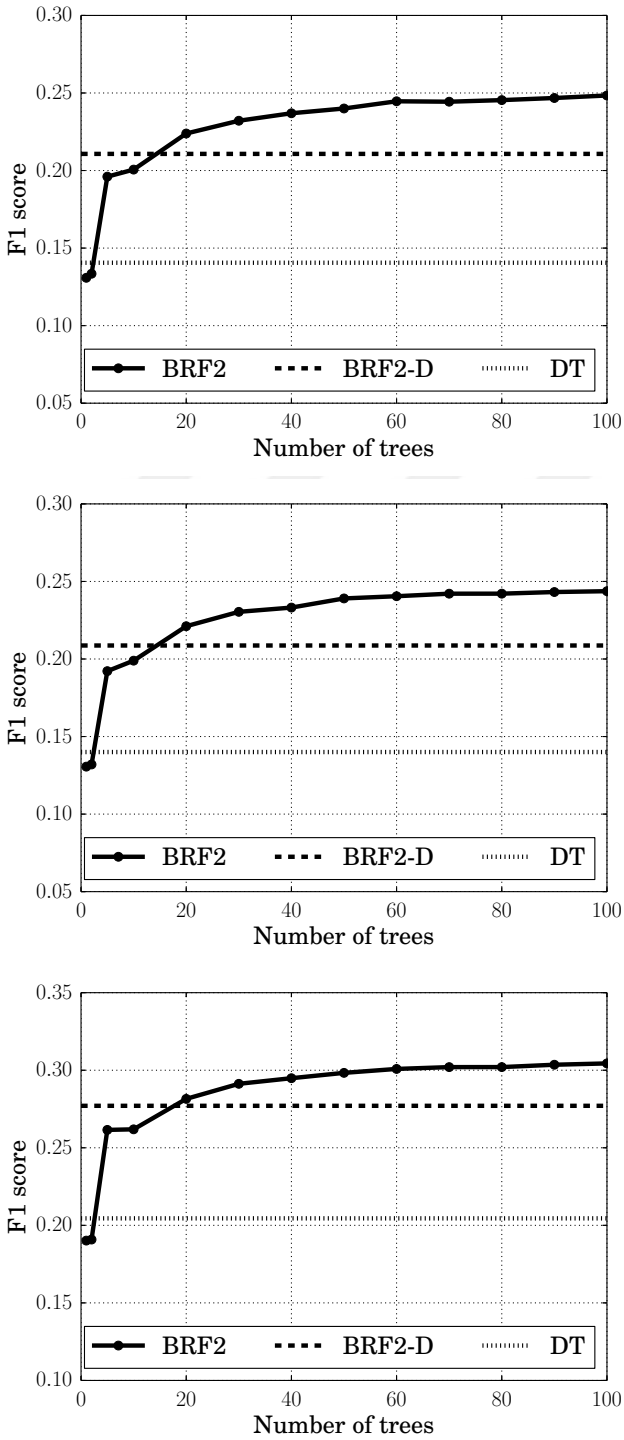

Figure 6.11. Comparison of methods for efficient prediction for months 6, 7, and 8, respectively.

6.4.4. Experiments for Prediction Efficiency

In this section, we experiment with the two different approaches discussed in Section 6.3.3 to improve prediction efficiency of the proposed models. This can be an important concern when predictions are sensitive to time as in the intention learning problem.

We begin with the random subset selection method. We first train a BRF2 model with 100 trees, and then perform predictions using random subsets of trees having 1 to 100 trees. Figure 6.11 shows that selecting a random subset with respect to the available computational budget is a practical approach. Even if we can allow a relatively small number of trees with respect to the full ensemble, the predictive power is still useful. We also observe that selecting even a few trees quickly outperforms the baseline single DT model.

We secondly experiment with the distillation idea. Following our discussion in Section 6.3.3, we distill the 100-tree BRF2 model into a simpler DT regression model using its output probabilities. We find the values $\beta = 0.7$ and $\alpha = 0.8$ useful for deciding the final predictions. In Figure 6.11, we call this predictive model BRF2-D. We observe, in the current experimental setting, that the model is comparable to using about 15 to 20 random trees by using 2 trees only. Furthermore, it outperforms the baseline single DT model significantly. Therefore, we conclude that the distillation idea is a promising alternative in practical situations.

7. TIMELY PUSH RECOMMENDATIONS IN A COLD START SETTING

7.1. Introduction

The delivery of recommendations can have a pull or push nature [155]. In the former, the user can be thought to make a specific request to "pull" recommendations. While this may refer to submitting a query, in today's large-scale applications, the user's interactions and behavior in the system may also be perceived as a request for possibly immediate personalized recommendations so that a wide range of choices can be narrowed down. On the other hand, "push" recommendations are delivered even if a user does not make a request for recommendations, and they are especially popular with relatively recent forms of human-computer interaction such as in mobile computing [71, 156, 157]. While the pull and push nature can sometimes be intermingled, push recommendations typically have more explicit requirements to satisfy multiple stakeholder objectives in the system. These include being timely and diverse regarding the shared interests of the stakeholders while being useful and minimally disturbing for the targeted users.

In this chapter, we propose a hybrid personalized LtR approach for large-scale push recommendations especially in an item cold start setting. While the proposed approach can be used for different applications, our case study is a job recommender system, and we work on a recent real-life challenge benchmark [71]. The cold start items are the job postings which enter the system and previously received no user feedback. The goal is to predict effective push recommendations for job seekers considering expectations of the multiple stakeholders, that is, job seekers, recruiters, and service providers. The push recommendations are time sensitive in the sense that the cold start items are to be recommended to a sufficient number of appropriate job seekers within a short period of time so that they are better disseminated to the system.

Our proposal works on user and item content information as well as different types of implicit feedback from users and other stakeholders. We perform offline experiments on historical data using a combined multi-objective performance criterion. With suitable adaptations, our proposal is also tested with online experiments in real time.

The remainder of this chapter is organized as follows: In Section 7.2, we define the problem and present a high-level view of our hybrid solution. The details of this solution are presented in Section 7.3 referring to the component rankers and different top-M and top-N selection strategies. We present the offline and online experimental results in Section 7.4.

7.2. Problem Definition and Representation

Assume a set of users, U, and a set of items, I, in a recommender system. Let $M, N \in \mathbb{Z}_{>0}$. Given a target subset of users $U' \subseteq U$ for push recommendations and (typically cold start) target items $I' \subseteq I$, the problem can be defined as finding a $\hat{U}_i'^M \subseteq U'$ subject to $0 \leq |\hat{U}_i'^M| \leq M$ for each $i \in I'$, and finding a $\hat{I}_u'^N \subseteq I'$ subject to $0 \leq |\hat{I}_u'^N| \leq N$ for each $u \in U'$ to maximize a suitable multi-objective performance criterion (see Section 2.2.5 for a discussion of performance evaluations).

A solution to this problem can be based on modeling user and item content features as well as the available feedback in the system. Given the multi-objective nature of such problems, the solution is typically a hybrid ensemble of rankers each estimating a relevance score for a given user-item pair considering different aspects of the multi-objective performance criterion. Two groups of rankers can especially be useful: The first group is based on user profiles obtained from user's past feedback and item content. The second group can be based on user and item content only. This type of ranker can be suitable if the user is also cold start with no or very limited feedback in the system. For push recommendations in a job RS, we propose such an ensemble in Figure 7.1 which blends rankers from both groups [16]. Then, the best recommendations from the rankers are further refined through different top-M and top-

N selection strategies to predict the final push recommendations. In the next section, we discuss the details of this hybrid ensemble.

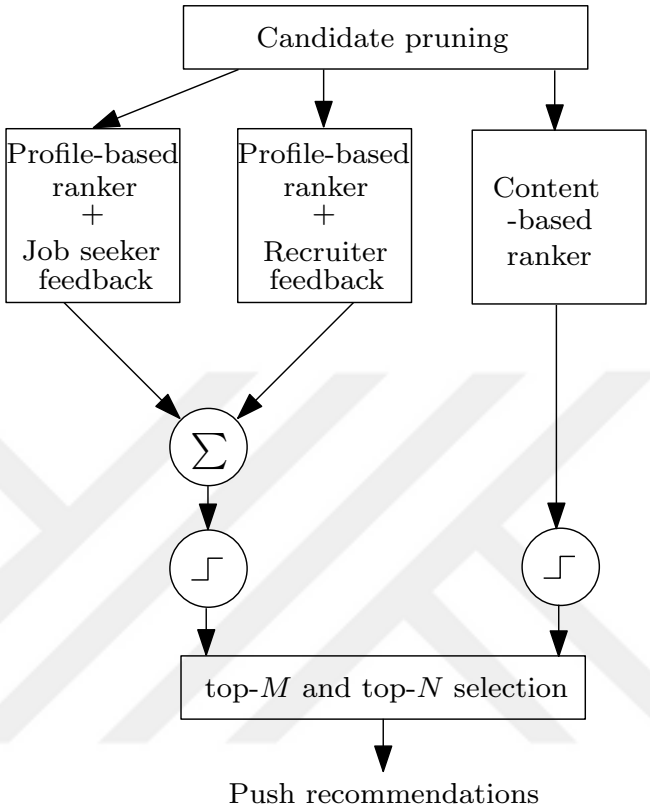

Figure 7.1. High-level view of ranker ensemble for push recommendations in a job RS.

7.3. Ranker Ensemble for Push Recommendations

7.3.1. Profile-based Rankers

In case it exists, user feedback data can often be more reliable for building models than user content features. For example, in the case of job recommendations, content features based on a résumé or form-based data can be misleading due to several reasons like being outdated or incomplete. On the other hand, job posting content is often more elaborate and reliable. Based on this observation, a job seeker's feedback data can be used to build a user *profile*. More specifically, the items for which the user provides feedback can contribute their content to the user profile. Consequently, each item content feature can have a weight proportional to its presence in user's feedback

history. For example, if a job seeker has positive feedback for 5 distinct job postings, and 3 of these have the career level feature "Professional", then the weight of this feature in the user's profile can be accepted as 3/5.

The user feedback can often be considered positive with varying degrees of confidence. However, some feedback types should be carefully handled. For example, if a user deletes a push recommendation, this may have negative semantics and a considerable impact on the system performance. On the other hand, when training a ranking model, we observe that labeling delete feedback as negative examples is not always helpful either, since deleted items may not usually have discriminative content. Instead, delete feedback can be used to decide, for example, if a user is prone to deleting push recommendations, and the negative class examples can be chosen by pairing the user with random items not existing in the user's feedback history.

We consider each profile-based ranker in Figure 7.1 as a pointwise LtR model (see Section 2.2.3.2). Therefore, the rankers take training input in the form of (u, i) pairs where $u \in U$, and $i \in I$. A pair may refer to a relevant or an irrelevant item for the user depending on the existing user feedback. Regarding job recommendations, each pair can be represented with a feature vector combining a job seeker profile with the content of a job posting. In this case, we highlight relatively important features in these vectors in Table 7.1. Each match-based and latent semantic feature refers to a similarity between a corresponding job seeker profile and a job posting content, and more detail can be found in our work [16]. Then, to obtain a relevance score, $\hat{y}_u(i)$, from this model, one straightforward option is logistic regression. However, in this chapter, we adhere to a more powerful alternative, gradient boosted decision trees (GBDT) [158, 159], since they are able to show superior performance in various ranking problems [160, 161] with a moderate number of shallow trees, and they can handle mixed-type and unscaled features without the need for preprocessing.

Since user and item repositories can be quite large, a general issue in web-scale ranking problems is the computational cost of predicting $\hat{y}_u(i)$ for all possible (u, i) pairs and typically sorting them to find the topmost rankings. As mentioned in Section 2.2.3,

Table 7.1. Features representing a job seeker and a job posting pair (see Section 7.4.2 for explanation on feature importances).

Feature	Importance
Match-based features	
Job title	0.322
Career level	0.041
Discipline	0.116
Industry	0.041
Country	0.028
Region	0.104
Employment	0.028
Tags	0.082
Latent semantic features	
SVD-based similarity using job title/tags	0.118
Job seeker features	
Number of positive feedback instances	0.052
Number of negative feedback instances	0.002
Is willing to change job	0.001
Last feedback time	0.041
Has feedback last week	0.001

a multi-stage pipeline of simpler to more complex models can be used to narrow down candidate pairs. Therefore, we find the following strategies useful for candidate pruning prior to prediction with rankers:

(i) *Rule-based pruning.* This requires using domain knowledge. For example, it is usually inappropriate to recommend student jobs to managers, and vice versa. Similarly, a (u, i) pair can be ignored if $u \in U'$ and $i \in I'$ have no match on some important features like job title and/or discipline. However, extensive rule-based pruning may also cause losing less obviously suitable pairs.

(ii) *Target item similarity.* This strategy refers to finding content-based similarity between I' and $I \setminus I'$ items, and then pairing an $i \in I'$ with U' users who provided positive feedback to the topmost similar $I \setminus I'$ items to it. This is quite effective, but it can become costly with very large $|I'|$.

Finally, to incorporate some sort of *reciprocity* [162], the rankers can be trained based on feedback data from different stakeholders. In the ranker ensemble in Figure 7.1, this is performed by considering job seeker and recruiter feedback separately. Since the predicted relevance scores are compatible, given a new (u, i) pair for prediction, scores from different rankers can be blended using simple aggregation methods like weighted average or harmonic mean.

7.3.2. Content-based Rankers

Despite the usage of less informative user content features, content-based rankers can compensate for the cold start users in U' with no or very limited feedback. In the case of job recommendations, this ranker is similar to the profile-based ranker except the match-based features and latent semantic features in Table 7.1 now depend on bilateral content features instead of user feedback profiles. Furthermore, the feedback-based job seeker features are not always applicable. It is also possible to consider reciprocity in content-based rankers by training separate rankers for different stakeholders of the system. However, this alternative is not further investigated in this chapter.

7.3.3. Top-M and Top-N Selection

Irrelevant push recommendations may be disturbing for users and lead to dissatisfaction for all stakeholders. Therefore, an $i \in I'$ should be recommended to a $u \in U'$, if, for example, the relevance score, $\hat{y}_u(i)$, is above some predefined threshold. Furthermore, we need strategies for top-M and top-N selection since the budgets M and N are limited as defined in Section 7.2. In the following, we discuss several effective strategies for this purpose:

(i) *Simple aggregation.* Select top-M users for each $i \in I'$ by aggregating topmost predictions from rankers in the ensemble. Each ranker can equally contribute or stronger rankers can be favored. Push all top-M recommendations. While it can still be practical, this strategy implicitly assumes that $N \leq |I'|$.

(ii) *Round robin.* For each $i \in I'$, select top-αM users from each ranker in the ensemble where $\alpha \geq 0$ and can be different for each ranker. Concatenate all selections to a list starting from the strongest ranker. Iterate through the resulting lists for all $i \in I'$ in a round robin fashion. Accept the next push recommendation if the budgets N and M for the corresponding user and item are not yet exhausted.

(iii) *Improving aggregate diversity.* This strategy is a more sophisticated alternative to round robin. Given the concatenated user lists for all $i \in I'$, this time we assume an unweighted bipartite graph where there is an edge between each user in the item's list and the item. Each edge can be thought to represent relevance of an item to a user. Then, we iteratively find maximum bipartite matchings (MBMs) [163] each time excluding the users and items from the graph whose budgets N and M are exhausted. The MBMs found at each iteration can be immediately used as push recommendations. Finding an MBM can be efficiently implemented, and it is also proposed in [164] for improving diversity of top-N recommendations. In the case of push recommendations, several iterations are performed while monitoring both N and M. This can be effective in increasing diversity of push recommendations in practical situations where, for example, N is small.

7.4. Experiments

7.4.1. Data and Experimental Setting

We present experimental results on a recent job recommendation challenge benchmark data [71]. This consists of an offline part for ideation and developing solutions and another part for online experiments. The offline dataset spans a period of 3 months. It contains about 1.5 million job seekers and 1.3 million job postings with content infor-

mation, of which 74,840 job seekers and 46,559 cold start job postings are targets for push recommendations. Furthermore, the dataset contains different types of job seeker feedback in the form of a click, bookmark, or reply which are assumed to have positive semantics, and also delete feedback. In addition, there is relatively sparse feedback showing recruiter interest on job seekers for a certain job posting. About 70% of target job seekers have positive feedback while the rest are cold start. About 23% of target job seekers are premium and 27% of target job postings are paid. The online part involves 5-week A/B/n testing of competitive proposals from qualified participants. Again, a long-term dataset is provided with comparable descriptive statistics to the offline version. However, new target cold start job postings (up to about 13,000) and new target job seekers (about 50,000) are given to each participant daily. Incremental updates to the long-term dataset are also made during the course of online testing.

In both offline and online parts, the same multi-objective performance measure [16] is used. This measure is summarized in Figure 7.2. While it is pretty much self-explanatory, the measure expresses different expectations of the stakeholders as a single score by aggregating two core components: user success and item success. It can be observed that positive feedback on push recommendations is quite valuable, and the score is further boosted if there is reciprocal interest. On the other hand, pushing irrelevant recommendations may incur high costs if user deletes a push recommendation or stops giving positive feedback. As an additional performance criterion, the online part has a time limit for receiving daily predicted push recommendations and submitting them to the system.

Our experimental setup is a commodity computer with a 4-core CPU, 8-GB main memory, and 64-bit Linux operating system. The implementations are based on the Python ecosystem and scikit-learn [153].

7.4.2. Experimental Results

We split the offline training dataset into two using a time split at the beginning of the last week. Cold start items of that last week together with their associated

$$\text{score} = \sum_{i \in I'} \left(\mathit{itemSuccess}\left(\hat{U}_i'^M\right) + \sum_{u \in \hat{U}_i'^M} \mathit{userSuccess}\,(i, u) \right)$$

$\mathit{userSuccess}(i, u) = ($

1 if (u clicked i) else 0

+5 if (u bookmarked or replied i) else 0

+20 if (recruiter of i is interested in u) else 0

−10 if (u only deleted i) else 0

) ×2 if (u is premium)

$\mathit{itemSuccess}(\hat{U}_i'^M) = ($

50 if (userSuccess$(i, u) > 0$ for some $u \in \hat{U}_i'^M$ and i is paid)

25 if (userSuccess$(i, u) > 0$ for some $u \in \hat{U}_i'^M$ and i is not paid)

0 otherwise

)

Figure 7.2. Multi-objective performance measure.

feedback constitute a validation set which allows us to tune hyperparameters and finally experiment with the provided test set. The following hyperparameter values achieve the results in this chapter: All rankers use a GBDT with 100 trees, cross entropy loss, and a learning rate of 0.10. For training, we use all available positive user-item examples and a balanced number of negative examples as discussed in Section 7.3.1. The number of latent dimensions is 50 for SVD-based similarity feature. Relevance score thresholds are chosen in a range between 0.80-0.95 before top-M and top-N selection, and they are the same irrespective of a user being premium or not, or an item being paid or not. As discussed in Section 7.3.3, the top-M and top-N selection strategy is task dependent and further explained below for offline and online experiments.

The importance of chosen features for the profile-based ranker is reported in Table 7.1 in Section 7.3.1. These values correspond to how much each feature contributes to the reduction of impurity in all nodes across all trees in GBDT. Similar feature importance values are also observed for the content-based ranker.

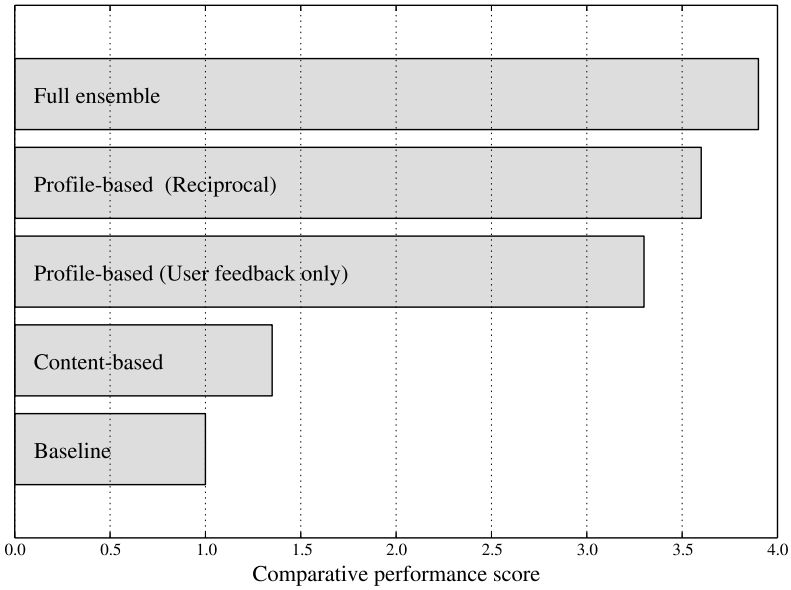

Figure 7.3. Comparative performance of different rankers.

Comparative performance of different ranking options in offline experiments are given in Figure 7.3. In the offline part, $M = 100$ whereas N is unrestricted, and we use the simple aggregation strategy in Section 7.3.3 in all rankers. The performances are based on the multi-objective measure in Figure 7.2, and they are relative to the performance of the baseline content-based ranker [71] which is scaled to 1. We experiment with the ranker ensemble in Figure 7.1 componentwise and as a whole. We first observe that the performance of the content-based ranker can be slightly improved with respect to the baseline, but it is still limited. In accordance with the discussion in Section 7.3.1, the actual significant performance increase is observed by using profile-based rankers which are based on user feedback profiles. Combined profile-based rankers for reciprocity are observed to improve this performance even further by up to 10%. Since there are also a significant number of complete cold start target job seekers (about 30%) and those with very few feedback, the final ensemble combines profile-based rankers

with a content-based ranker as illustrated in Figure 7.1. It is observed that the final ensemble can quadruple the baseline performance given the current features and experimental setup.

After offline performance evaluations, 25 out of 103 active participants with competitive solutions are invited to take part in an online experiment in real time [71] which is a rare setting for academic research. We use some of the best hyperparameter values found in the offline experiments also for online experiments. However, in the online part, $M = 250$ and $N = 1$, that is, each target user can be more realistically pushed at most one recommendation. In this case, we opt for improving aggregate diversity in top-M and top-N selection as follows: For every target job posting, we first pick the topmost ranked job seekers from each ranker ($\alpha M \approx 2 - 10$) and improve the aggregate diversity by applying a few iterations (typically 2-5) to find MBMs as described in Section 7.3.3. We then use the round robin strategy with a larger αM to enrich the final set of push recommendations. In the end, the online testing performance of our proposal shows up to 30% increase in aggregate diversity compared to using only the round-robin strategy, 10% improvement compared to the offline part with respect to the topmost performance score, and gets the 8th position. Furthermore, our available experimental setup is able to produce predictions in about an hour for the daily online testing using the proposed ensemble of rankers, low-cost blending schemes, and effective strategies for top-M and top-N selection. This makes our proposal scalable to millions of users and items in the training phase and able to serve daily load of target users and items in the prediction phase. We note that a more extensive feature selection and extraction as well as adding more rankers to the ensemble have a potential to further improve the performance.

8. CONCLUSIONS

In many real-life recommender systems, the user feedback is implicit meaning that it is inferred from different types of user behavior and interactions in the system. This is usually because implicit feedback can be collected unobtrusively, and it can capture the user-item relevance information better through time. Furthermore, there are almost always some requirements imposed by the particular application area for time-sensitive recommendations, and we distinguish four major challenges for achieving them: (i) Efficient and adaptive model learning, (ii) Fast personalized predictions, (iii) Session-based recommendations, and (iv) Time dependency and time awareness.

From the perspective above, we present an overview of LtR models which can learn from implicit feedback effectively and support time-sensitive recommendations. We then propose novel efficient personalized LtR methods to learn from large-scale implicit feedback datasets and streams while trying to address the challenges for achieving time-sensitive recommendations. In the following, we present our conclusions about these proposals including a discussion of our findings and highlighting some open research directions.

8.1. Mining User Feedback Stream for CF

In Chapter 3, we propose a novel scalable and adaptive personalized recommendation algorithm, SASCF. The algorithm works on streaming implicit feedback and provides mechanisms to solve different challenges for time-sensitive recommendations. The core idea is continuously maintaining a compact summary of frequently co-occurring items instead of computing an offline item similarity matrix. For this purpose, SASCF extends two frequent item finding algorithms from streams and maintains co-occurrences in sorted order with respect to their approximate frequency counts. This enables efficient top-k queries which subsequently facilitates finding similar items quickly with respect to various heuristic similarity measures. In this sense, the approach can be considered as direct neighborhood learning. The algorithm is also combined with

an efficient sampling scheme for further efficiency. All these properties enable scalable and adaptive learning and fast personalized predictions based on a user's past feedback also taking into account time dependency. Furthermore, they make SASCF useful for session-based recommendations.

We perform an empirical analysis of SASCF on real-life datasets. First, our exploratory data analysis shows that the power-law behavior of rank-frequency distributions of item co-occurrences supports SASCF and its hyperparameterization. Second, experimental results using an offline holdout set are reported. Given the selection of hyperparameters in a broad range, the results comply with offline collaborative filtering in terms of ranking accuracy. Third, we perform experiments using a first-test-then-train sequential scheme. The sequential evaluation results show adaptivity of the proposed approach in comparison to offline NN search and a single pass MF approach.

We note some possible future research directions as follows: First, various strategies for finding frequent items in streams [83, 84, 165, 166] can be further investigated for applicability to the SASCF framework. Second, while the mechanisms provided in SASCF can be effective in RS applications, efficiently and accurately finding nearest neighbors in streaming data with respect to an arbitrary similarity measure [86, 87] is an underexplored problem. Third, in addition to the built-in sampling scheme, coupling of SASCF with parallel processing schemes such as map-reduce [61] is a promising direction and deserves further investigation.

8.2. Parallel Personalized Pairwise LtR

Pairwise LtR is especially suitable for CF from implicit feedback. In Chapter 4, we show that its efficiency can be greatly improved using state-of-the-art parallel SGD schemes. In this direction, we first propose two base algorithms: PLtR-B and PLtR-N. The former follows a block partitioning scheme which enables mutually exclusive model parameter updates. The latter applies no particular partitioning and relies on random sampling from a hypergraph. Our analysis as well as extensive comparative experimental results show that both PLtR-B and PLtR-N can preserve the ranking

accuracy of their sequential counterpart without any significant loss due to parallel learning. Furthermore, their speedup patterns indicate a very good exploitation of the available processing units in the system, and hence remarkable learning efficiency. They are also lightweight with minimal extra space requirements and easy to implement. All these qualities can make them desirable for efficient LtR in web-scale applications.

We then discuss and propose some important extensions to PLtR-B and PLtR-N algorithms. We first show that the algorithms can be combined with various biased sampling strategies. These strategies include adaptive sampling to further improve the convergence speed and sampling to handle available graded relevance feedback information. Second, we show that the algorithms can also be combined with adaptive gradient update methods which can bring advantages like faster convergence as well as enabling easier selection of the learning rate. Finally, we propose an extension, PLtR-NS, which builds upon PLtR-N and learns from streaming user feedback efficiently. This extended algorithm considers the time order of the incoming user feedback and provides a mechanism to adapt to the changing temporal dynamics of the system more quickly. We present detailed experimental results for all extensions which both show their usefulness and the versatility of the PLtR-B and PLtR-N algorithms.

We note some possible future research directions as follows: To extend our guiding empirical analysis, experiments with more processing units can be performed, for example, using GPU-based parallel processing. Furthermore, our separate preliminary analysis hints that the proposed algorithms can be useful even when additional context apart from the user and time are used.

8.3. Efficient Top-N Prediction from MF models

MF models are commonly used in personalized LtR, especially in CF scenarios. They usually attempt to map a large number of users and items into a low-dimensional latent space where relevant items for a user are expected to have high dot products. Then, a very common prediction task is to find the top-N items in this space having the highest dot products with a queried user. In Chapter 5, we propose a meta-algorithm,

MMFNN, which can combine various common MF models with approximate NN search methods to improve their top-N prediction efficiency. Our approach is based on an efficient heuristic which searches top-N items among the nearest neighbors of items known to be relevant to the queried user. We show that this is a viable approach since such relevant items are expected to have high dot products and so are their neighbors in a latent metric space. Furthermore, we show that finding the highest dot products in such a preselected neighborhood has the potential effect of improving various other performance results compared to highest dot product search among all items.

Intensive top-N queries from large item repositories are not uncommon in RSs. Reducing the query time is often of concern since it can improve the system's adaptivity by increasing training frequency when the top-N predictions are precomputed, or it can facilitate fast retrieval when there is a need for top-N predictions in real time. Furthermore, in practice, the top-N predictions are often produced by a hybrid of several models and efficient retrieval from individual models is always desired. Therefore, MMFNN can be a valuable meta-algorithm in many different scenarios requiring more efficient predictions.

The effectiveness of MMFNN is tested using various state-of-the-art batch, online, and incremental MF models and two approximate NN search methods based on partitioning trees. Extensive empirical analysis on large implicit feedback datasets shows that the proposed approach can bring a drastic time efficiency compared to exhaustive top-N queries, while, at the same time, preserving or sometimes even improving ranking accuracy and diversity.

A possible future research direction can be using MMFNN on top of other specialized approximate NN search methods [126, 167] and comparing their effects on the efficiency of predictions from MF models. Additionally, the practicality of the proof of concept RL extension as well as its evaluation can be further investigated.

8.4. Learning Intention in User Sessions

Prediction of user intention in a session can enable better recommendations during or short after the session ends. In Chapter 6, we propose a way to represent the intention learning problem by turning implicit user feedback in a session into a feature vector. The vector can also incorporate conveniently the user and item content features in case they are available. We then propose powerful and efficient ensemble methods based on RFs which can learn in batch or incremental fashion and handle class imbalance due to the rarely occurring event of interest. We also investigate several ways for efficient prediction from such an ensemble since the prediction of intention needs to be fast enough for time-sensitive recommendations.

We work on a recent challenge benchmark dataset for predicting purchasing intention in a session. Extensive experimental results show usefulness of our proposals in terms of predictive power and efficiency.

We note that learning intention in user sessions is a relatively underexplored area in academic research. Beyond their practicality, our proposals can serve as strong baselines for further research.

8.5. Timely Push Recommendations in a Cold Start Setting

Push recommendations typically have explicit requirements to satisfy multiple stakeholder objectives while being time sensitive. In Chapter 7, we work on large-scale push recommendations especially in an item cold start setting. We first formalize the problem and then propose an effective hybrid personalized LtR approach which leverages both implicit feedback and content information through different component rankers and effective top-M and top-N selection strategies.

While the proposed approach can be used for different applications, our case study is a job recommender system, and we work on a recent real-life benchmark. We first experiment with the proposed ensemble in an offline setting using a multi-

objective performance measure and obtain much superior ranking accuracy compared to the baseline as well as the component rankers. Especially, the usage of profile-based rankers based on implicit user feedback are observed to bring a huge improvement in the results. Then, we justify these results in an online experimental setting with further improvements in ranking accuracy, diversity, and competitive efficiency.

Push recommendations are an underexplored area in academic research. Therefore, our formalization and proposals can be useful for future research especially when there is increased dataset availability.

REFERENCES

1. Ricci, F., L. Rokach and B. Shapira, *Recommender Systems Handbook*, Springer, 2nd edn., 2015.

2. Bobadilla, J., F. Ortega, A. Hernando and A. Gutierrez, "Recommender Systems Survey", *Knowledge-based Systems*, Vol. 46, pp. 109–132, 2013.

3. Liu, T.-Y., *Learning to Rank for Information Retrieval*, Springer, 2011.

4. Koren, Y., *The BellKor Solution to the Netflix Grand Prize*, 2009, `https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf`, accessed at October 2018.

5. Harper, F. M. and J. A. Konstan, "The MovieLens Datasets: History and Context", *ACM Transactions on Interactive Intelligent Systems*, Vol. 5, No. 4, pp. 1–19, 2015.

6. Hidasi, B. and D. Tikk, "Fast ALS-Based Tensor Factorization for Context-aware Recommendation from Implicit Feedback", *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, ECML PKDD'12, pp. 67–82, 2012.

7. Hu, Y., Y. Koren and C. Volinsky, "Collaborative Filtering for Implicit Feedback Datasets", *Proceedings of the 8th IEEE International Conference on Data Mining*, ICDM '08, pp. 263–272, 2008.

8. Pilaszy, I., A. Sereny, G. Dozsa, B. Hidasi, A. Sari and J. Gub, "Neighbor Methods vs Matrix Factorization - Case Studies of Real-life Recommendations", *ACM Recsys Workshop on Large-scale Recommender Systems*, 2015.

9. Yağcı, A. M., T. Aytekin and F. S. Gürgen, "Scalable and Adaptive Collaborative

Filtering by Mining Frequent Item Co-occurrences in a User Feedback Stream", *Engineering Applications of Artificial Intelligence*, Vol. 58, pp. 171–184, 2017.

10. Yağcı, A. M., T. Aytekin and F. S. Gürgen, "Parallel Pairwise Learning to Rank for Collaborative Filtering", *Concurrency and Computation: Practice and Experience*, (Accepted and Published Online) 2019.

11. Yağcı, M., T. Aytekin and F. Gürgen, "On Parallelizing SGD for Pairwise Learning to Rank in Collaborative Filtering Recommender Systems", *Proceedings of the 11th ACM Conference on Recommender Systems*, RecSys '17, pp. 37–41, 2017.

12. Yağcı, M., T. Aytekin, H. Türen and F. Gürgen, "Parallel Personalized Pairwise Learning to Rank", *Proceedings of the 3rd EURO Mini Conference: From Multiple Criteria Decision Aid to Preference Learning*, DA2PL '16, pp. 53–58, 2016.

13. Yağcı, A. M., T. Aytekin and F. S. Gürgen, "A Meta-algorithm for Improving Top-N Prediction Efficiency of Matrix Factorization Models in Collaborative Filtering", *International Journal of Pattern Recognition and Artificial Intelligence*, (Accepted) 2019.

14. Yağcı, A. M., T. Aytekin and F. S. Gürgen, "An Ensemble Approach for Multi-label Classification of Item Click Sequences", *Proceedings of the 9th ACM Conference on Recommender Systems Challenge*, RecSys '15, 2015.

15. Yağcı, A. M., T. Aytekin and F. S. Gürgen, "Balanced Random Forest for Imbalanced Data Streams", *Proceedings of the IEEE 24th Signal Processing and Communication Applications Conference*, pp. 1065–1068, 2016.

16. Yağcı, M. and F. Gürgen, "A Ranker Ensemble for Multi-objective Job Recommendation in an Item Cold Start Setting", *Proceedings of the 11th ACM Conference on Recommender Systems Challenge*, RecSys '17, 2017.

17. Aljukhadar, M., S. Senecal and C.-E. Daoust, "Using Recommendation Agents to

Cope with Information Overload", *International Journal of Electronic Commerce*, Vol. 17, No. 2, pp. 41–70, 2012.

18. de Gemmis, M., P. Lops, C. Musto, F. Narducci and G. Semeraro, "Semantics-Aware Content-Based Recommender Systems", F. Ricci, L. Rokach and B. Shapira (Editors), *Recommender Systems Handbook*, pp. 119–159, Springer, 2015.

19. McCarey, F., M. Ó. Cinnéide and N. Kushmerick, "Recommending Library Methods: An Evaluation of the Vector Space Model (VSM) and Latent Semantic Indexing (LSI)", M. Morisio (Editor), *Reuse of Off-the-shelf Components*, pp. 217–230, Springer, 2006.

20. Abel, F., A. Benczúr, D. Kohlsdorf, M. Larson and R. Pálovics, "RecSys Challenge 2016: Job Recommendations", *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pp. 425–426, 2016.

21. Rendle, S., "Factorization Machines with libFM", *ACM Transactions on Intelligent Systems and Technology*, Vol. 3, No. 3, pp. 57:1–57:22, 2012.

22. Freno, A., "Practical Lessons from Developing a Large-Scale Recommender System at Zalando", *Proceedings of the 11th ACM Conference on Recommender Systems*, RecSys '17, pp. 251–259, 2017.

23. Park, S.-T., D. Pennock, O. Madani, N. Good and D. DeCoste, "Naïve Filterbots for Robust Cold-start Recommendations", *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pp. 699–705, 2006.

24. Koren, Y. and R. Bell, "Advances in Collaborative Filtering", F. Ricci, L. Rokach and B. Shapira (Editors), *Recommender Systems Handbook*, pp. 77–118, Springer, 2015.

25. Shi, Y., M. Larson and A. Hanjalic, "Collaborative Filtering Beyond the User-Item Matrix: A Survey of the State of the Art and Future Challenges", *ACM Computing Surveys*, Vol. 47, No. 1, pp. 1–45, 2014.

26. Burke, R., "Hybrid Web Recommender Systems", P. Brusilovsky, A. Kobsa and W. Nejdl (Editors), *The Adaptive Web: Methods and Strategies of Web Personalization*, pp. 377–408, Springer, 2007.

27. Kelly, D. and J. Teevan, "Implicit Feedback for Inferring User Preference: A Bibliography", *SIGIR Forum*, Vol. 37, No. 2, pp. 18–28, 2003.

28. Lerche, L. and D. Jannach, "Using Graded Implicit Feedback for Bayesian Personalized Ranking", *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, pp. 353–356, 2014.

29. Cremonesi, P., Y. Koren and R. Turrin, "Performance of Recommender Algorithms on Top-N Recommendation Tasks", *Proceedings of the 4th ACM Conference on Recommender Systems*, RecSys '10, pp. 39–46, 2010.

30. Koren, Y., "Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model", *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pp. 426–434, 2008.

31. Ben-Shimon, D., A. Tsikinovsky, M. Friedmann, B. Shapira, L. Rokach and J. Hoerle, "RecSys Challenge 2015 and the YOOCHOOSE Dataset", *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys '15, pp. 357–358, 2015.

32. Jannach, D., L. Lerche and M. Jugovac, "Adaptation and Evaluation of Recommendations for Short-term Shopping Goals", *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys '15, pp. 211–218, 2015.

33. Vinagre, J., A. M. Jorge and J. Gama, "An Overview on the Exploitation of Time in Collaborative Filtering", *Wiley Interdisciplinary Reviews: Data Mining*

and Knowledge Discovery, Vol. 5, No. 5, pp. 195–215, 2015.

34. Adomavicius, G. and A. Tuzhilin, "Context-aware Recommender Systems", F. Ricci, L. Rokach and B. Shapira (Editors), *Recommender Systems Handbook*, pp. 191–226, Springer, 2015.

35. Campos, P. G., F. Diez and I. Cantador, "Time-aware Recommender Systems: A Comprehensive Survey and Analysis of Existing Evaluation Protocols", *User Modeling and User-Adapted Interaction*, Vol. 24, No. 1, pp. 67–119, 2014.

36. Li, H., *Learning to Rank for Information Retrieval and Natural Language Processing*, Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers, 2nd edn., 2014.

37. Karatzoglou, A., L. Baltrunas and Y. Shi, "Learning to Rank for Recommender Systems", *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 493–494, 2013.

38. Rendle, S., *Context-Aware Ranking with Factorization Models*, Vol. 330 of *Studies in Computational Intelligence*, Springer, 2011.

39. Alpaydin, E., *Introduction to Machine Learning*, The MIT Press, 3rd edn., 2014.

40. Sutton, R. S. and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 2nd edn., 2018.

41. Shi, Y., A. Karatzoglou, L. Baltrunas, M. Larson, N. Oliver and A. Hanjalic, "CLiMF: Learning to Maximize Reciprocal Rank with Collaborative Less-is-more Filtering", *Proceedings of the 6th ACM Conference on Recommender Systems*, RecSys '12, pp. 139–146, 2012.

42. Ning, X., C. Desrosiers and G. Karypis, "A Comprehensive Survey of Neighborhood-Based Recommendation Methods", F. Ricci, L. Rokach and

B. Shapira (Editors), *Recommender Systems Handbook*, pp. 37–76, Springer, 2015.

43. Marlin, B. M., R. S. Zemel, S. Roweis and M. Slaney, "Collaborative Filtering and the Missing at Random Assumption", *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, UAI'07, pp. 267–275, 2007.

44. Deshpande, M. and G. Karypis, "Item-based Top-n Recommendation Algorithms", *ACM Transactions on Information Systems*, Vol. 22, No. 1, pp. 143–177, 2004.

45. Smith, B. and G. Linden, "Two Decades of Recommender Systems at Amazon.com", *IEEE Internet Computing*, Vol. 21, No. 3, pp. 12–18, 2017.

46. McFee, B. and G. Lanckriet, "Metric Learning to Rank", *Proceedings of the 27th International Conference on Machine Learning*, ICML'10, pp. 775–782, 2010.

47. Hsieh, C.-K., L. Yang, Y. Cui, T.-Y. Lin, S. Belongie and D. Estrin, "Collaborative Metric Learning", *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pp. 193–201, 2017.

48. Rendle, S., C. Freudenthaler, Z. Gantner and L. Schmidt-Thieme, "BPR: Bayesian Personalized Ranking from Implicit Feedback", *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, UAI '09, pp. 452–461, 2009.

49. Barkan, O. and N. Koenigstein, "Item2Vec: Neural Item Embedding for Collaborative Filtering", *arXiv*, Vol. 1603.04259, 2016.

50. Koenigstein, N. and Y. Koren, "Towards Scalable and Accurate Item-oriented Recommendations", *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 419–422, 2013.

51. Hidasi, B., A. Karatzoglou, L. Baltrunas and D. Tikk, "Session-based Recommendations with Recurrent Neural Networks", *CoRR*, Vol. abs/1511.06939, 2016.

52. Jannach, D. and M. Ludewig, "When Recurrent Neural Networks Meet the Neighborhood for Session-Based Recommendation", *Proceedings of the 11th ACM Conference on Recommender Systems*, RecSys '17, pp. 306–310, 2017.

53. Ning, X. and G. Karypis, "SLIM: Sparse Linear Methods for Top-N Recommender Systems", *Proceedings of the IEEE 11th International Conference on Data Mining*, ICDM'11, pp. 497–506, 2011.

54. Levy, M. and K. Jack, "Efficient Top-n Recommendation by Linear Regression", *Proceedings of the ACM RecSys Large Scale Recommender Systems Workshop*, Recsys '13, 2013.

55. Markovsky, I., *Low Rank Approximation: Algorithms, Implementation, Applications*, Springer, 2011.

56. Baeza-Yates, R. and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison-Wesley Publishing Company, 2nd edn., 2011.

57. Pan, R., Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz and Q. Yang, "One-Class Collaborative Filtering", *Proceedings of the 8th IEEE International Conference on Data Mining*, ICDM '08, pp. 502–511, 2008.

58. Burges, C. J., *From RankNet to LambdaRank to LambdaMART: An Overview*, Tech. Rep. MSR-TR-2010-82, Microsoft Research, 2010.

59. Covington, P., J. Adams and E. Sargin, "Deep Neural Networks for YouTube Recommendations", *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pp. 191–198, 2016.

60. Miner, D. and A. Shook, *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*, O'Reilly, 1st edn., 2012.

61. Leskovec, J., A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, Cam-

bridge University Press, 2nd edn., 2014.

62. Gama, J., *Knowledge Discovery from Data Streams*, Chapman & Hall/CRC, 2010.

63. Li, L., W. Chu, J. Langford and R. E. Schapire, "A Contextual-bandit Approach to Personalized News Article Recommendation", *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pp. 661–670, 2010.

64. Gunawardana, A. and G. Shani, "Evaluating Recommender Systems", F. Ricci, L. Rokach and B. Shapira (Editors), *Recommender Systems Handbook*, pp. 265–308, Springer, 2015.

65. Herschtal, A. and B. Raskutti, "Optimising Area Under the ROC Curve Using Gradient Descent", *Proceedings of the 21st International Conference on Machine Learning*, ICML '04, pp. 49–56, 2004.

66. Manning, C. D., P. Raghavan and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press, 2008.

67. McFee, B., T. Bertin-Mahieux, D. P. Ellis and G. R. Lanckriet, "The Million Song Dataset Challenge", *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pp. 909–916, 2012.

68. Rendle, S. and C. Freudenthaler, "Improving Pairwise Learning for Item Recommendation from Implicit Feedback", *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pp. 273–282, 2014.

69. Ge, M., C. Delgado-Battenfeld and D. Jannach, "Beyond Accuracy: Evaluating Recommender Systems by Coverage and Serendipity", *Proceedings of the 4th ACM Conference on Recommender Systems*, RecSys '10, pp. 257–260, 2010.

70. Adomavicius, G. and Y. Kwon, "Improving Aggregate Recommendation Diversity Using Ranking-Based Techniques", *IEEE Transactions on Knowledge and Data*

Engineering, Vol. 24, No. 5, pp. 896–911, 2012.

71. Abel, F., Y. Deldjoo, M. Elahi and D. Kohlsdorf, "RecSys Challenge 2017: Offline and Online Evaluation", *Proceedings of the 11th ACM Conference on Recommender Systems*, RecSys '17, pp. 372–373, 2017.

72. Kohavi, R., R. Longbotham, D. Sommerfield and R. M. Henne, "Controlled Experiments on the Web: Survey and Practical Guide", *Data Mining and Knowledge Discovery*, Vol. 18, No. 1, pp. 140–181, 2009.

73. Kohavi, R., "Online Controlled Experiments: Lessons from Running A/B/n Tests for 12 Years", *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, 2015.

74. Bifet, A., G. Holmes, R. Kirkby and B. Fahringer, *Data Stream Mining, A Practical Approach*, 2011, `https://moa.cms.waikato.ac.nz/`, accessed at October 2018.

75. Vinagre, J., A. M. Jorge and J. Gama, "Fast Incremental Matrix Factorization for Recommendation with Positive-only Feedback", *User Modeling, Adaptation, and Personalization*, Vol. 8538 of *Lecture Notes in Computer Science*, pp. 459–470, Springer, 2014.

76. Han, J., M. Kamber and J. Pei, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, 3rd edn., 2011.

77. Moens, S., E. Aksehirli and B. Goethals, "Frequent Itemset Mining for Big Data", *Proceedings of the IEEE International Conference on Big Data*, pp. 111–118, 2013.

78. Lin, J. C.-W., W. Gan, P. Fournier-Viger and T.-P. Hong, "RWFIM: Recent Weighted-Frequent Itemsets Mining", *Engineering Applications of Artificial Intelligence*, Vol. 45, pp. 18–32, 2015.

79. Karp, R. M., S. Shenker and C. H. Papadimitriou, "A Simple Algorithm for Finding Frequent Elements in Streams and Bags", *ACM Transactions on Database Systems*, Vol. 28, No. 1, pp. 51–55, 2003.

80. Liberty, E., "Simple and Deterministic Matrix Sketching", *Proceedings of 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pp. 581–588, 2013.

81. Metwally, A., D. Agrawal and A. El Abbadi, "Efficient Computation of Frequent and Top-k Elements in Data Streams", *Proceedings of 10th International Conference on Database Theory*, ICDT'05, pp. 398–412, 2005.

82. Manku, G. S. and R. Motwani, "Approximate Frequency Counts over Data Streams", *Proceedings of 28th International Conference on Very Large Databases*, VLDB '02, pp. 346–357, 2002.

83. Cormode, G. and M. Hadjieleftheriou, "Finding Frequent Items in Data Streams", *Proceedings of VLDB Endowment*, Vol. 1, No. 2, pp. 1530–1541, 2008.

84. Liu, H., Y. Lin and J. Han, "Methods for Mining Frequent Items in Data Streams: An Overview", *Knowledge and Information Systems*, Vol. 26, No. 1, pp. 1–30, 2011.

85. Adamic, L. A., *Zipf, Power-laws, and Pareto - A Ranking Tutorial*, 2000, `http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html`, accessed at October 2018.

86. Campagna, A. and R. Pagh, "Finding Associations and Computing Similarity via Biased Pair Sampling", *Knowledge and Information Systems*, Vol. 31, No. 3, pp. 505–526, 2012.

87. Zadeh, R. B. and A. Goel, "Dimension Independent Similarity Computation", *Journal of Machine Learning Research*, Vol. 14, No. 1, pp. 1605–1626, 2013.

88. Dooms, S., T. De Pessemier and L. Martens, "MovieTweetings: A Movie Rating Dataset Collected from Twitter", *Proceedings of the ACM Recsys Workshop on Crowdsourcing and Human Computation for Recommender Systems*, RecSys '13, 2013.

89. Bodon, F., *Kosarak Online News Dataset*, `http://fimi.ua.ac.be/data/`, accessed at October 2018.

90. Liu, B., *Amazon Ratings Dataset*, `http://konect.uni-koblenz.de/networks/amazon-ratings`, accessed at October 2018.

91. Gama, J., R. Sebastiao and P. P. Rodrigues, "On Evaluating Stream Learning Algorithms", *Machine Learning*, Vol. 90, No. 3, pp. 317–346, 2013.

92. Lowry, R., *Concepts and Applications of Inferential Statistics*, 2015, `http://www.vassarstats.net/textbook`, accessed at October 2018.

93. Bottou, L., F. E. Curtis and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning", *arXiv*, Vol. 1606.04838, 2016.

94. Bottou, L., "Stochastic Gradient Descent Tricks", G. Montavon, G. B. Orr and K.-R. Müller (Editors), *Neural Networks: Tricks of the Trade: Second Edition*, pp. 421–436, Springer, 2012.

95. Candes, E. J. and B. Recht, "Exact Matrix Completion via Convex Optimization", *Foundations of Computational Mathematics*, Vol. 9, No. 6, pp. 717–772, 2009.

96. Recht, B. and C. Re, "Parallel Stochastic Gradient Algorithms for Large-scale Matrix Completion", *Mathematical Programming Computation*, Vol. 5, No. 2, pp. 201–226, 2013.

97. Niu, F., B. Recht, C. Ré and S. J. Wright, "Hogwild: A Lock-Free Approach to

Parallelizing Stochastic Gradient Descent", J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira and K. Q. Weinberger (Editors), *Advances in Neural Information Processing Systems*, NIPS, pp. 693–701, 2011.

98. Diaz-Aviles, E., L. Drumond, L. Schmidt-Thieme and W. Nejdl, "Real-time Top-n Recommendation in Social Streams", *Proceedings of the 6th ACM Conference on Recommender Systems*, RecSys '12, pp. 59–66, 2012.

99. Gemulla, R., E. Nijkamp, P. J. Haas and Y. Sismanis, "Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent", *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pp. 69–77, 2011.

100. Zhuang, Y., W.-S. Chin, Y.-C. Juan and C.-J. Lin, "A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems", *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 249–256, 2013.

101. Zhao, P. and T. Zhang, "Stochastic Optimization with Importance Sampling for Regularized Loss Minimization", *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ICML'15, pp. 1–9, 2015.

102. Weston, J., H. Yee and R. J. Weiss, "Learning to Rank Recommendations with the K-order Statistic Loss", *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 245–248, 2013.

103. Ertekin, S., J. Huang, L. Bottou and L. Giles, "Learning on the Border: Active Learning in Imbalanced Data Classification", *Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pp. 127–136, 2007.

104. Loni, B., R. Pagano, M. Larson and A. Hanjalic, "Bayesian Personalized Ranking with Multi-Channel User Feedback", *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pp. 361–364, 2016.

105. Qian, N., "On the Momentum Term in Gradient Descent Learning Algorithms", *Neural Networks*, Vol. 12, No. 1, pp. 145–151, 1999.

106. Ruder, S., "An Overview of Gradient Descent Optimization Algorithms", *arXiv*, Vol. 1609.04747, 2016.

107. Dean, J., G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang and A. Y. Ng, "Large Scale Distributed Deep Networks", *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1223–1231, 2012.

108. He, X., L. Liao, H. Zhang, L. Nie, X. Hu and T.-S. Chua, "Neural Collaborative Filtering", *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pp. 173–182, 2017.

109. Yi-Lei, W., T. Wen-Zhe, Y. Xian-Jun, W. Ying-Jie and C. Fu-Ji, "An Efficient Method for Autoencoder-based Collaborative Filtering", *Concurrency and Computation: Practice and Experience*, Vol. Online, pp. 1–7, 2018.

110. Johnson, C. C., "Logistic Matrix Factorization for Implicit Feedback Data", *Proceedings of the NIPS 2014 Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

111. Chin, W.-S., Y. Zhuang, Y.-C. Juan and C.-J. Lin, "A Learning-Rate Schedule for Stochastic Gradient Methods to Matrix Factorization", T. Cao, E.-P. Lim, Z.-H. Zhou, T.-B. Ho, D. Cheung and H. Motoda (Editors), *Advances in Knowledge Discovery and Data Mining*, pp. 442–455, Springer, 2015.

112. Li, D., C. Chen, Q. Lv, H. Gu, T. Lu, L. Shang, N. Gu and S. M. Chu, "AdaError: An Adaptive Learning Rate Method for Matrix Approximation-based Collaborative Filtering", *Proceedings of the 27th International Conference on World Wide Web*, WWW '18, 2018.

113. Duchi, J., E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, Vol. 12, pp. 2121–2159, 2011.

114. Kingma, D. P. and J. Ba, "Adam: A Method for Stochastic Optimization", *arXiv*, Vol. 1412.6980, 2014.

115. Vitter, J. S., "Random Sampling with a Reservoir", *ACM Transactions on Mathematical Software*, Vol. 11, No. 1, pp. 37–57, 1985.

116. Chen, C., H. Yin, J. Yao and B. Cui, "TeRec: A Temporal Recommender System over Tweet Stream", *Proceedings of the VLDB Endowment*, Vol. 6, No. 12, pp. 1254–1257, 2013.

117. Osborne, M., A. Lall and B. Van Durme, "Exponential Reservoir Sampling for Streaming Language Models", *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, Vol. 2, pp. 687–692, 2014.

118. Cemgil, A. T., "A Tutorial Introduction to Monte Carlo Methods, Markov Chain Monte Carlo and Particle Filtering", P. S. Diniz, J. A. Suykens, R. Chellappa and S. Theodoridis (Editors), *Academic Press Library in Signal Processing*, Vol. 1, pp. 1065 – 1114, Elsevier, 2014.

119. Rendle, S. and L. Schmidt-Thieme, "Online-updating Regularized Kernel Matrix Factorization Models for Large-scale Recommender Systems", *Proceedings of the 2nd ACM Conference on Recommender Systems*, RecSys '08, pp. 251–258, 2008.

120. Celma, O., *Music Recommendation and Discovery in the Long Tail*, Springer, 2010.

121. Aiolli, F., "Efficient Top-n Recommendation for Very Large Scale Binary Rated Datasets", *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pp. 273–280, 2013.

122. OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*, 2013, `www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`, accessed at October 2018.

123. Mouillot, D. and A. Lepretre, "Introduction of Relative Abundance Distribution (RAD) Indices, Estimated from the Rank-Frequency Diagrams (RFD), to Assess Changes in Community Diversity", *Environmental Monitoring and Assessment*, Vol. 63, No. 2, pp. 279–295, 2000.

124. Mandelbrot, B. B., *The Fractal Geometry of Nature*, Freeman, 1982.

125. Koren, Y., R. Bell and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems", *Computer*, Vol. 42, No. 8, pp. 30–37, 2009.

126. Muja, M. and D. G. Lowe, "Scalable Nearest Neighbor Algorithms for High Dimensional Data", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 36, No. 11, pp. 2227–2240, 2014.

127. Ram, P. and A. G. Gray, "Maximum Inner-product Search Using Cone Trees", *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pp. 931–939, 2012.

128. Koenigstein, N., P. Ram and Y. Shavitt, "Efficient Retrieval of Recommendations in a Matrix Factorization Framework", *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pp. 535–544, 2012.

129. Auer, P., N. Cesa-Bianchi and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem", *Machine Learning*, Vol. 47, No. 2, pp. 235–256, 2002.

130. Wang, Y., J.-Y. Audibert and R. Munos, "Algorithms for Infinitely Many-Armed Bandits", D. Koller, D. Schuurmans, Y. Bengio and L. Bottou (Editors), *Advances in Neural Information Processing Systems 21*, pp. 1729–1736, 2009.

131. Silpa-Anan, C. and R. Hartley, "Optimised KD-trees for Fast Image Descriptor Matching", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, 2008.

132. Bachrach, Y., Y. Finkelstein, R. Gilad-Bachrach, L. Katzir, N. Koenigstein, N. Nice and U. Paquet, "Speeding Up the Xbox Recommender System Using a Euclidean Transformation for Inner-product Spaces", *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, pp. 257–264, 2014.

133. Bernhardsson, E., *Annoy Software*, Spotify, `https://github.com/spotify/annoy`, accessed at October 2018.

134. Khoshneshin, M. and W. N. Street, "Collaborative Filtering via Euclidean Embedding", *Proceedings of the 4th ACM Conference on Recommender Systems*, RecSys '10, pp. 87–94, 2010.

135. Teflioudi, C., R. Gemulla and O. Mykytiuk, "LEMP: Fast Retrieval of Large Entries in a Matrix Product", *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pp. 107–122, 2015.

136. Teflioudi, C. and R. Gemulla, "Exact and Approximate Maximum Inner Product Search with LEMP", *ACM Transactions on Database Systems*, Vol. 42, No. 1, pp. 5:1–5:49, 2017.

137. Fraccaro, M., U. Paquet and O. Winther, "Indexable Probabilistic Matrix Factorization for Maximum Inner Product Search", *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 1554–1560, 2016.

138. Neyshabur, B. and N. Srebro, "On Symmetric and Asymmetric LSHs for Inner Product Search", *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 1926–1934, 2015.

139. McAuley, J., C. Targett, Q. Shi and A. van den Hengel, "Image-Based Recommendations on Styles and Substitutes", *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pp. 43–52, 2015.

140. Muja, M. and D. G. Lowe, *Flann Software*, `https://www.cs.ubc.ca/research/flann/`, accessed at October 2018.

141. Teflioudi, C., R. Gemulla and O. Mykytiuk, "LEMP Software", `https://dws.informatik.uni-mannheim.de/en/resources/software/lemp`, accessed at October 2018.

142. Li, L., W. Chu, J. Langford and X. Wang, "Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms", *Proceedings of the 4th ACM International Conference on Web Search and Data Mining*, WSDM'11, pp. 297–306, 2011.

143. Tsoumakas, G., I. Katakis and I. Vlahavas, "Mining Multi-label Data", R. L. Maimon O. (Editor), *Data Mining and Knowledge Discovery Handbook*, pp. 667–685, Springer, 2009.

144. Breiman, L., "Random Forests", *Machine Learning*, Vol. 45, No. 1, pp. 5–32, 2001.

145. Galar, M., A. Fernandez, E. Barrenechea, H. Bustince and F. Herrera, "A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches", *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, Vol. 42, No. 4, pp. 463–484, 2012.

146. Chen, C., A. Liaw and L. Breiman, *Using Random Forest to Learn Imbalanced Data*, Tech. Rep. 666, Department of Statistics, University of Berkeley, 2004, `https://statistics.berkeley.edu/tech-reports/666`.

147. Hulten, G., L. Spencer and P. Domingos, "Mining Time-changing Data Streams", *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pp. 97–106, 2001.

148. Oza, N., "Online Bagging and Boosting", *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, Vol. 3, pp. 2340–2345, 2005.

149. Saffari, A., C. Leistner, J. Santner, M. Godec and H. Bischof, "On-line Random Forests", *Proceedings of the 12th IEEE International Conference on Computer Vision (ICCV) Workshops*, pp. 1393–1400, 2009.

150. Heutte, L., S. Adam and S. Bernard, "On the Selection of Decision Trees in Random Forests", *IEEE - INNS - ENNS International Joint Conference on Neural Networks*, IJCNN '09, pp. 302–307, 2009.

151. Hinton, G., O. Vinyals and J. Dean, "Distilling the Knowledge in a Neural Network", *arXiv*, Vol. 1503.02531, 2015.

152. Louppe, G., L. Wehenkel, A. Sutera and P. Geurts, "Understanding Variable Importances in Forests of Randomized Trees", *Advances in Neural Information Processing Systems*, NIPS, pp. 431–439, 2013.

153. Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, Vol. 12, pp. 2825–2830, 2011.

154. Bifet, A., G. Holmes, B. Pfahringer, H. Kremer, T. Jansen and T. Seidl, "MOA: Massive Online Analysis, A Framework for Stream Classification and Clustering", *Journal of Machine Learning Research - Proceedings Track*, Vol. 11, pp. 44–50, 2010.

155. Garcia-Molina, H., G. Koutrika and A. Parameswaran, "Information Seeking: Convergence of Search, Recommendations, and Advertising", *Communications of the ACM*, Vol. 54, No. 11, pp. 121–130, 2011.

156. Zhao, H., L. Si, X. Li and Q. Zhang, "Recommending Complementary Products in E-Commerce Push Notifications with a Mixture Model Approach", *Proceedings of the 40th International ACM Conference on Research and Development in Information Retrieval*, SIGIR '17, pp. 909–912, 2017.

157. Tan, L., A. Roegiest, J. Lin and C. L. Clarke, "An Exploration of Evaluation Metrics for Mobile Push Notifications", *Proceedings of the 39th International ACM Conference on Research and Development in Information Retrieval*, SIGIR '16, pp. 741–744, 2016.

158. Hastie, T., R. Tibshirani and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer, 2nd edn., 2009.

159. Friedman, J. H., "Greedy Function Approximation: A Gradient Boosting Machine", *The Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232, 2001.

160. Burges, C. J. C., K. M. Svore, P. N. Bennett, A. Pastusiak and Q. Wu, "Learning to Rank Using an Ensemble of Lambda-gradient Models", *Proceedings of the International Conference on Yahoo! Learning to Rank Challenge*, YLRC'10, pp. 25–35, JMLR.org, 2010.

161. Pacuk, A., P. Sankowski, K. Wkegrzycki, A. Witkowski and P. Wygocki, "RecSys Challenge 2016: Job Recommendations Based on Preselection of Offers and Gradient Boosting", *Proceedings of the ACM Recsys '16 Challenge*, RecSys '16, pp. 10:1–10:4, 2016.

162. Li, L. and T. Li, "MEET: A Generalized Framework for Reciprocal Recommender Systems", *Proceedings of 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pp. 35–44, 2012.

163. Cormen, T. H., C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, The MIT Press, 3rd edn., 2009.

164. Adomavicius, G. and Y. O. Kwon, "Maximizing Aggregate Recommendation Diversity: A Graph-theoretic Approach", *Proceedings of the ACM Rescys Workshop on Novelty and Diversity in Recommender Systems*, RecSys '11, pp. 3–10, 2011.

165. Cormode, G., F. Korn and S. Tirthapura, "Exponentially Decayed Aggregates on Data Streams", *Proceedings of IEEE 24th International Conference on Data Engineering*, pp. 1379–1381, 2008.

166. Woodruff, D. P., "New Algorithms for Heavy Hitters in Data Streams", *ArXiv*, Vol. arXiv:1603.01733, 2016.

167. Wang, J., T. Zhang, J. Song, N. Sebe and H. T. Shen, "A Survey on Learning to Hash", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 13, No. 9, pp. 1–21, 2017.