STATISTICAL LEARNING IN RECOMMENDATION SYSTEMS: PERSONALIZED RANKING AND DECENTRALIZED LEARNING SYSTEMS

James Lee

Bachelor of Arts, Rice University, 2017

Master of Science, University of Virginia, 2021

A Dissertation submitted to the Graduate Faculty
of the University of Virginia in Candidacy for the Degree of
Doctor of Philosophy

Department of Statistics

University of Virginia

May 2024

Statistical Learning in Recommendation Systems: Personalized Ranking and Decentralized Learning Systems

James Lee

(ABSTRACT)

Recommendation systems are integral to various domains, including content-based platforms and e-commerce. Despite extensive research efforts in designing diverse recommendation systems in recent years, popular approaches based on explicit ratings often struggle to perform effectively in practical recommendation-making scenarios. This dissertation comprises three main sections. Firstly, we will present an overview of recommendation systems from a statistical perspective. Secondly, we propose a personalized ranking-based model that utilizes pairwise preference information from explicit feedback, demonstrating superior performance compared to conventional approaches in real-world settings. Finally, addressing growing concerns over privacy and data safety within recommendation systems, we introduce a novel decentralized federated learning framework. This framework operates without relying on centralized data aggregation or costly model merging procedures.

Acknowledgments

I am deeply grateful to my advisor, Dr. Xiwei Tang, whose invaluable guidance and kindness have been instrumental in making this journey not only fruitful but also enjoyable. Additionally, I extend my sincere appreciation to my esteemed committee members, Dr. Jordan Rodu, Dr. Lingxiao Wang, Dr. Shan Yu, and Dr. Sheng Li, for their valuable time and contributions to my work. I am also indebted to my wife, Dr. Jasmine Lee, for her unwavering support throughout this endeavor. Lastly, I express my heartfelt thanks to my children, Sophie and Emmanuel, for their constant presence and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recommendation systems have dramatically increased in usage and effectiveness throughout the past two decades. With the overarching goal of providing useful connections between users and items, many organizations both in industry and academia have pushed to improve model performance, evaluation criteria, and the ecosystem surrounding recommendation system research. Recommendation systems have proliferated in all technological corners of the world thanks to the power of the internet and exponential growth of personalized data and they touch many aspects of our lives. Large companies like Google or Amazon use these systems to suggest ads or products to a user that they think the user will like. Social media platforms like Facebook and X (formerly Twitter) leverage them by promoting connections or posts they think a user would most likely interact with. By finding patterns in user demographics, historical trends, and most importantly in our case, user-item interactions, recommender systems can guide user behavior in specific directions.

Such user-item data comes in two general formats: explicit feedback, and implicit feedback (Koren, Bell, and Volinsky 2009). As the names suggest, explicit feedback is user data where the user explicitly shares how they feel about a certain item. For example, a movie rating from 1 to 5 stars would be considered explicit feedback. On the other hand, implicit feedback is user data where the user does not share their exact preference for an item they interacted with. Given the fact that a user bought a specific item, it does not tell us whether the user truly liked the item or not. Things like clicks or views can also fall into the implicit feedback umbrella.

Though the difference between the two feedback types may be subtle, there are large differences in how the data are used and how the corresponding models are evaluated. The explicit feedback scenario often deals with the **rating prediction** case, where recommendation system models are tasked with predicting the exact rating that a user would give to an item. These models are evaluated using well known evaluation methods like Root-Mean-Squared error (RMSE) or Mean Absolute Error (MAE). In contrast, **top-k recommendation** is more common with implicit data, where the assumption is that given a user, all items have a true preference (like/dislike). And because the user will only ever see k recommendations, only the top k ranked items matter for a user. This requires alternative evaluation methods such as Recall at k, Precision at k, or Normalized Discounted Cumulative Gain at k. Both types of recommender systems have their merits and it is ultimately a matter of data availability and which evaluation metrics are most important in your use case.

Of course, the proliferation of recommender systems finding patterns in vast amounts of data mined from users has not been without its critics. At the core of recommendation systems is the accumulation and analysis of large amounts of user data. When used judiciously with proper data protocols, they can provide a valuable experience to users of a product. However, such large amounts of personal data collection may seem invasive to users and pose a security and privacy risk. The data can be misused by those collecting the data as in the infamous Facebook and Cambridge Analytica scandal (Confessore 2018), or through secondary means as a data breach (Perlroth 2016). Furthermore, even if precautions were taken to keep the personal data safe, pooling seemingly innocuous data such as movie watched with various sources can even reveal sensitive personal information such as political ideology (Narayanan and Shmatikov 2008). All these issues point to the fact that it is challenging to reconcile traditional recommendation systems with with the important values of user data security and privacy.

In this backdrop a new class of recommendation systems, called Federated Learning, was developed Bonawitz et al. 2016; Konečný et al. 2017. The core idea is to be able to train a model where the data is not directly accessible by the model itself. For example, the data may live on client machines while the model lives in a central server. This way the central server does not need access to the potentially sensitive data present on the client machines. This separation of data and model added complexity in other areas such as communication costs, model aggregation, and synchronization, but much research has been done to improve machine learning in this distributed manner.

The rest of the dissertation covers these important topics in more detail and is structured as follows. The rest of Chapter 1 introduces recommendation systems, the types of data used, as well as widely used evaluation methods. In Chapter 2, we propose an pairwise model that bridges the typical divide in recommendation research between explicit and implicit feedback. Explicit BPR extends the foundational Bayesian Personalized Ranking (BPR) model to the explicit feedback scenario in an efficient and extendable way. Then in Chapter 3 we introduce a novel decentralized federated learning method where each client node shares training information with its neighbors given some underlying network structure.

1.1 Recommendation Systems

The high level goal of any recommender system is to suggest items to a user that it believes the user will connect with. “Connect” is a general description which may mean, enjoy, purchase, watch, or evoke a strong response from, even if it is a negative response. In our research we will use “connect” to typically mean “like” or “want to purchase”, but in practice a multitude of definitions may be used.

1.2 Different Data Frameworks

Data used by recommender systems can be described as observations where each row is a (user, item, value) tuple, with the possible addition of a timestamp of when the interaction occurred. If the timestamp is ignored, the data can also be represented as a utility matrix, $\boldsymbol{Y}$, where each element is the interaction between the corresponding user row and item column. The interaction can be a rating, such a value from 1 to 5 or an up-vote; or an indicator of an action such as a purchase, a click, or a view. An interaction that conveys the actual preference of a user such a rating from 1 to 5 is called explicit feedback, while an interaction that merely indicates the presence of some action is called implicit feedback. The interaction is usually a rating from 1 to 5 for explicit feedback and a 0 or 1 for implicit feedback. Explicit and implicit feedback often have different properties and slightly different use cases.

1.2.1 Explicit Feedback

Explicit feedback provides a clear positive or negative signal, and tends to result in sparse data with as much as 95% of the user-item matrix missing. This is because the majority of user usually only provide explicit feedback for a small proportion of the items available. Figure 1.1 shows an example of explicit feedback in the form of movie ratings. Such data is often used to predict the ratings of each item and models are evaluated using pointwise loss functions like RMSE.

Alice	**5**	**4**	**?**	**?**
Bob	**?**	**2**	**3**	**4**
Chris	**1**	**?**	**?**	**5**
Dan	**?**	**4**	**?**	**1**

Figure 1.1: Example ratings matrix showing sparse explicit feedback.

Common datasets that contain explicit feedback are the MovieLens datasets (Harper and Konstan 2016), which are collections of anonymized ratings that users of the MovieLens website users gave to movies. The ratings were integers from 1 to 5 prior to version 3, but changed to half-integer increments from 0.5 to 5.0 starting in February 2003. These datasets come in various sizes such as 100,000 ratings, 1 million ratings, and 25 million ratings. The Netflix Prize dataset was another dataset of anonymized movie ratings that consisted of 100 million ratings on an integer scale from 1 to 5(Bennett and Lanning 2007).

1.2.2 Implicit Feedback

On the other hand, implicit feedback provides valuable information that is not unambiguously positive or negative. For example, knowing that a user watched a certain movie does not necessarily tell you that they user enjoyed the movie. Because of this ambiguity, the binary data of implicit feedback is often treated as weak positive and negative signals (Steffen Rendle 2021), such that the feedback is weakly positive if the interaction exists and weakly negative if it does not exist. Implicit feedback is often more abundant than explicit feedback, but the sparsity found in explicit feedback is lost due to the fact that all non-interactions are considered to be weak negative signals. Figure 1.2 shows an example of implicit feedback.

Alice	**1**	**1**	**0**	**0**
Bob	**0**	**1**	**1**	**1**
Chris	**1**	**0**	**0**	**1**
Dan	**0**	**1**	**0**	**1**

Figure 1.2: Example utility matrix showing dense implicit feedback.

It is not uncommon to convert explicit feedback data such as item ratings into an implicit one by changing all the ratings to 1 and having all the unobserved data become 0 (Steffen Rendle et al. 2009; Belal et al. 2022), however, by doing so, one loses potentially valuable information about how the user feels about the specific item. Furthermore, by converting the dataset in such manner, any classification model will be predicting whether or not a user has **rated** an item, and not necessarily whether the user “likes” an given item. This distinction may be important depending on what precise outcome you want the system to optimize. Other widely used implicit datasets include the Yahoo! news dataset, some which provides user view/click data on new articles, as well as the Last.fm dataset which contains user and artist listening counts

(*Last.FM* 2011). In the next section we will cover the mathematical formulation for rating-based recommendation, where the goal is to accurate predict user-item ratings.

1.3 Rating-based Recommendation

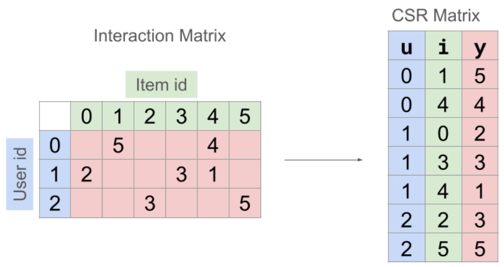
Given some set of users, $u \in \mathcal{U}$ and items, $i \in \mathcal{I}$, let $m = |\mathcal{U}|$ and $n = |\mathcal{I}|$. We denote the $m \times n$ matrix of interactions between users and items as $\boldsymbol{Y}^{m \times n}$. $\boldsymbol{Y}_{u,i}$ or also y_{ui} represents the interaction between user u and item i. In a more general form, the context, c, replaces the user u. Context is general information related to the query and can include things like user, personalized information about the user, and time. For simplicity, the context we use will only include the user, $u \in \mathcal{U}$.

Another representation of the interaction data may be as a Compressed Sparse Row (CSR) matrix where the row and column index array correspond to the user and item respectively and the data array being the value of the interaction. This is equivalent to the set of (u, i, y) tuples which we will call $(u, i, y) \in \mathcal{X}$. If $\boldsymbol{Y}$ is sparse - such as explicit feedback - then $|\mathcal{X}| \ll (m \times n)$. If $\boldsymbol{Y}$ is dense like in implicit feedback, then $|\mathcal{X}| = (m \times n)$. An example of changing $\boldsymbol{Y}$ into a CSR matrix can be seen in Figure 1.3.

The scoring function denotes the preference of a user for a particular item and can be described as

$$\hat{y}_{\boldsymbol{\theta}}(u, i) : U \times I \to \mathbb{R} \tag{1.1}$$

where $\boldsymbol{\theta}$ are the model parameters, $u \in U$ is a user, and $i \in I$ is an item. For convenience we will also write the output of a scoring function for a given user-item pair as $\hat{y}_{ui}$. The exact form of the scoring function, as well as the optimization



User id \ Item id	0	1	2	3	4	5
0		5			4	
1	2			3	1	
2			3			5

u	i	y
0	1	5
0	4	4
1	0	2
1	3	3
1	4	1
2	2	3
2	5	5

Figure 1.3: Utility matrix to CSR matrix.

algorithm for it depends on the task the recommendation system is trying to perform. As mentioned before, the two broad tasks that we will discuss are rating prediction and item recommendation (top-k recommendation). Section 1.3 discuss the rating prediction problem and Section 1.4 covers the item recommendation case.

In the rating prediction paradigm, the goal of a recommendation system is to predict the exact score a user would hypothetically give to each item. Rating prediction has been a popular goal used in the literature (Herlocker et al. 2004; Koren, Bell, and Volinsky 2009; Sedhain et al. 2015). The data used in rating prediction is usually explicit feedback, such as movie ratings or product review scores. For simplicity, we assume that the ratings are integers from 1 to 5. This is relatively common in practice but may also include half intervals like 0.5, 1.5, etc. (Harper and Konstan 2016).

Given these true ratings y, the problem is often formulated by minimizing the magnitude of the difference between the actual scores and the predicted scores $\hat{y}$. There are many ways to model the ratings, and we will cover a few of the popular approaches.

1.3.1 Content-based Recommender Systems

Recommender systems for rating prediction can be broadly categorized into two different approaches: content-based filtering and collaborative filtering. Content-based filtering approaches provide recommendations based on the profiles of items and users. Item profiles could include information such as genre, lead cast, and length of the movie. User profiles could include demographic information. Then, based on the profiles, the recommender system would find movies profiles that are similar to ones the user has already watched. The drawback to this approach is that the information from the profiles must be sourced from external sources. However, this approach does not require a large amount of user-item interaction data.

One example of a content-based model is the Music Genome Project Koren, Bell, and Volinsky 2009 used by music applications such as Pandora. To create song profiles, trained analysts score songs a variety of musical attributes, which can then be used to determine song similarity. Then, if a user likes a song with a certain profile, the system could recommend other songs that have a similar profile. A drawback to content-based models is that the extra information must be obtained for each new item or user. As apparent in the Music Genome Project, this can be a tremendous amount of work.

1.3.2 Collaborative Filtering

An alternative to content-based filtering is called collaborative filtering, which was first introduced by Xerox‘s Tapestry system (Goldberg et al. 1992). This method analyzes the history of user-item interactions and makes predictions based on those interactions. Additional information such as user or item profiles is not required, making collaborative filtering a domain free method. However, new users and items do not have an interaction history, which makes prediction for them difficult. This is known as the cold start problem (Koren, Bell, and Volinsky 2009). This also means that a company that wants to create a recommender system based on collaborative filtering needs lots of data pertaining to the history of user-item interactions before accurate recommendations can be made.

Collaborative filtering can be broken down into two types of models: neighborhood methods and latent factor models (also known as embedding models). Neighborhood methods rely on the principle that users (or items) with similar behaviors will have similar tastes in items (users) and use the user item relationship directly using a similarity metric such as cosine similarity. For example, consider Figure 1.4. Say that Bob likes the three movies he is connected to. A user-centric neighborhood model would find other users who also liked the three movies, and then make recommendations based on what those other users also liked. In this example we see that because Alice, Chris, and Dan like Tenet, the model will recommend Tenet to Bob.

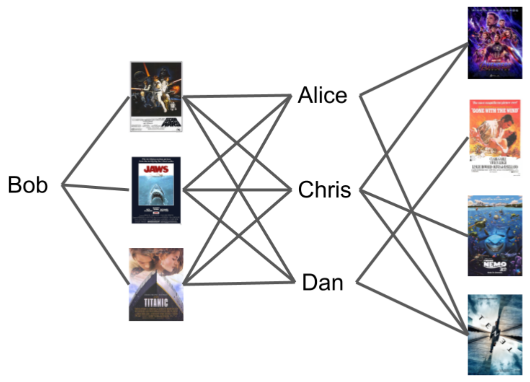


Figure 1.4: User-based neighborhood method.

Given the $m \times n$ utility matrix $\boldsymbol{Y}$ with m users and n items, the similarity metric can be calculated on the rows of $\boldsymbol{Y}$ for a user-based model or on the columns for a item-based model. Then the cosine similarity, s_{ij} between item i and item j would be

$$\begin{aligned} s_{ij} &= \frac{\sum_{k=1}^{m} y_{ki} y_{kj}}{\left(\sum_{k=1}^{m} y_{ki} y_{ki}\right)\left(\sum_{k=1}^{m} y_{kj} y_{kj}\right)} \qquad (1.2) \\ &= \frac{\boldsymbol{y}_{:i} \cdot \boldsymbol{y}_{:j}}{||\boldsymbol{y}_{:i}|| \times ||\boldsymbol{y}_{:j}||}. \end{aligned}$$

Then one way to make predictions for $\hat{y}_{ui}$ is to identify the k items rated by u that are most similar to i and take the weighted average of those weightings (Koren 2008). Let $\mathcal{N}$ be the set of neighbor items, then

$$\hat{y}_{ui} = \frac{\sum_{j \in \mathcal{N}} s_{ij} y_{ui}}{\sum_{j \in \mathcal{N}} s_{ij}}. \qquad (1.3)$$

Some examples for neighborhood based models include random walk methods, SVM's, and k-nearest neighbors.

1.3.3 Latent Factor Models

In contrast to neighborhood methods, latent factor methods learn patterns in the data through latent model parameters. The widely used latent factor models, also called embedding models, embed the user-item interaction information into some embedding space that can be used to generate predictions. A common method is to embed each user and each item into the same latent embedding space, i.e. a k dimensional vector space. Then the scores can be generated by taking the inner product of a user embedding with an item embedding. This is the core idea behind the Matrix Factorization model (Koren 2008) which is based on low-rank singular value decomposition (SVD), but tailored to sparse matrices and without the requirement

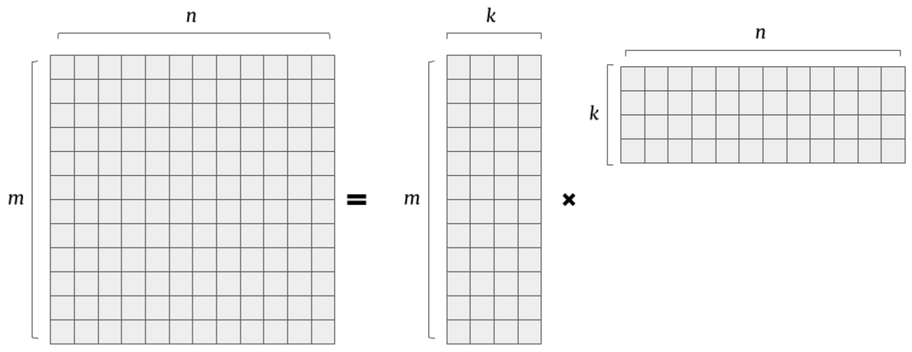


Figure 1.5: Factorization of the user-item matrix

of orthogonality.

Matrix factorization methods are widely employed in recommender systems and gained prominence during the Netflix Prize competition, where they dominated the top scoring spots (Bennett and Lanning 2007). The winning technique of the competition's final team utilized numerous matrix factorization models in an ensemble fashion. These methods aim to decompose the matrix of user-item interactions, Ω, into the product of two matrices: one for the user embeddings, $\boldsymbol{P}$ and one for the item embeddings, $\boldsymbol{Q}$, also known as the latent factors. Figure 1.5 shows the decomposition.

The scoring function for matrix factorization looks like

$$\hat{y}_{ui} = \boldsymbol{p}_u^T \boldsymbol{q}_i + \boldsymbol{b}_u + \boldsymbol{b}_i + \mu, \tag{1.4}$$

where $\boldsymbol{b}_u$ and $\boldsymbol{b}_i$ are user and item biases and μ is a global bias. The matrix factorization method is central to our work, and we discuss it in more detail in Chapter

2.

Various improvements and extensions to matrix factorization have been developed including the SVD++ model, Bayesian probabilistic matrix factorization, as well as Mixture Rank matrix approximation. The SVD++ model smoothly merges aspects of latent factor models and neighborhood models creating a more accurate combined model. The model showed that error improvements could be found by combining elements of both explicit and implicit data (Koren 2008). The scoring function for the SVD++ model is

$$\hat{y}_{ui} = b_{ui} + \boldsymbol{q}_i^T \left(\boldsymbol{p}_u + |N(u)|^{-\frac{1}{2}} \sum_{j \in N(u)} z_j \right), \tag{1.5}$$

where we now see that the user factor is extended with $|N(u)|^{-\frac{1}{2}} \sum_{j \in N(u)} z_j$. Here $N(u)$ contains all items that user u interacted with and z_j is an additional item factor.

The Bayesian probabilistic matrix factorization model introduced a fully Bayesian approach to probabilistic matrix factorization and showed the feasibility of Markov chain Monte Carlo (MCMC) methods for large dataset recommender systems (Salakhutdinov and Mnih 2008). The likelihood of the observed ratings is given by

$$p(\boldsymbol{Y}|\boldsymbol{P}, \boldsymbol{Q}, \alpha) = \prod_{i=1}^{m} \prod_{j=1}^{n} \left[\mathcal{N}(\boldsymbol{Y}_{ij} | \boldsymbol{P}_i^\top \boldsymbol{Q}_j, \alpha^{-1}) \right]^{I_{ij}}, \tag{1.6}$$

where $\mathcal{N}(\boldsymbol{Y}_{ij}|\mu, \alpha^{-1})$ denotes a Normal distribution with mean μ and precision α, and I_{ij} is an indicator variable that is equal to 1 if $(i, j) \in \Omega$. The user and item feature vectors come from Normal prior distributions. And the hyper-parameters for the user and item features are assumed to be Gaussian-Wishart distributions.

The Mixture Rank Matrix Approximation method does not represent the user-item ratings as a low-rank matrix approximation of a fixed rank but as a mixture of low-rank matrix approximations with different ranks (D. Li, C. Chen, W. Liu, et al. 2017). The slightly different likelihood function used by the MRMA model is as the following

$$p(\boldsymbol{Y}|\boldsymbol{P},\boldsymbol{Q},\alpha,\beta,\sigma^2) = \prod_{i=1}^{m}\prod_{j=1}^{n}\left[\sum_{k=1}^{K}\alpha_i^k\beta_j^k\mathcal{N}(\boldsymbol{Y}_{ij}|\boldsymbol{P}_i^\top\boldsymbol{Q}_j,\sigma^2)\right]^{I_{ij}}. \tag{1.7}$$

The difference here is now the K, which denotes the maximum rank for all internal structures of the user-item matrix. Then α_i^k and β_j^k are weights for the rank k model for user i and item j. Gaussian priors are placed on the user and item factors, while Laplacian priors are placed on the α_i^k and β_j^k which makes them sparse. The model is optimized using iterated conditional modes.

Furthermore, an adaptive learning rate method was developed for matrix factorization that adjusted the learning rate based on the noisiness of the rating (D. Li, C. Chen, Lv, et al. 2018). Instead of a constant learning rate, the AdaError method gives entries with large training errors smaller steps and entries with small training errors larger steps. Given a learning rate λ and observation $(u,i)\in\Omega$, the learning rate at the $t-$th iteration is

$$\lambda_{ui}^{(t)} = \frac{\lambda}{\sqrt{E_{ui}^{(t-1)}+\epsilon}}+\beta, \tag{1.8}$$

where $E_{ui}^{(t-1)} = \sum_{x=0}^{t-1}(\boldsymbol{Y}_{ui}-\hat{y}_{ui}^{(x)})^2$, ϵ is a small constant to prevent dividing by zero, and β is a constant to prevent $\lambda_{ui}^(t)$ from becoming 0 after many iterations. The continued extensions and variations of the well-established matrix factorization model show that it is still an effective and popular model in the recommendation system space, despite its relative simplicity.

Neural network methods often extend the embedding model approach but may use other methods to embed user and item interaction data (H. Wang, N. Wang, and Yeung 2015; He et al. 2017). The Restricted Boltzmann Machine was one of the first successful implementation of a neural network based model in recommender systems (Salakhutdinov, Mnih, and Hinton 2007). And AutoRec proposed an autoencoder framework (Sedhain et al. 2015) for the rating prediction problem. An autoencoder is an auto-associative neural network with a single k-dimensional hidden layer that is trained to recreate its input data after embedding it into a lower dimensional space. Given a set S of vectors in $\mathbb{R}^d$ and $k \in \mathbb{N}_+$, an autoencoder minimizes the following

$$\min_{\theta} \sum_{\boldsymbol{r} \in S} ||\boldsymbol{r} - h(\boldsymbol{r}; \theta)||_2^2, \tag{1.9}$$

where $h(\boldsymbol{r}; \theta)$ is the reconstruction of input $\boldsymbol{r} \in \mathbb{R}^d$,

$$h(\boldsymbol{r}; \theta) = f(\boldsymbol{W} \cdot g(\boldsymbol{V}\boldsymbol{r} + \mu) + \boldsymbol{b}) \tag{1.10}$$

for activation functions $f(\cdot)$, and $g(\cdot)$ where $\theta = \{\boldsymbol{W}, \boldsymbol{V}, \mu, \boldsymbol{b}\}$ for transformation $\boldsymbol{W} \in \mathbb{R}^{d \times k}$, $\boldsymbol{V} \in \mathbb{R}^{k \times d}$, and biases $\mu \in \mathbb{R}^k, \boldsymbol{b} \in \mathbb{R}^d$. In contrast to MF which learns a linear latent representation, AutoRec learns a non-linear latent representation of either the items or users.

1.3.4 Drawbacks of Rating-based Framework

Evaluating recommender systems in the rating prediction paradigm is straightforward. A test set is withheld during training and used exclusively for evaluation. Commonly used evaluation metrics are error metrics such as Root Mean Squared Error (RMSE) and Mean absolute Error (MAE). For both metrics, lower values equate to higher prediction accuracy. Given user-item pairs withheld in the test set, $\Omega' = \{(u, i)\}$, the formula for Root Mean Squared Error is

$$\text{RMSE} = \sqrt{\frac{1}{|\Omega'|} \sum_{(u,i)\in\Omega'} (y_{ui} - \hat{y}_{ui})^2}. \tag{1.11}$$

The formula for Mean Average Error is

$$\text{MAE} = \frac{1}{|\Omega'|} \sum_{(u,i)\in\Omega'} |y_{ui} - \hat{y}_{ui}|. \tag{1.12}$$

Such evaluation metrics have been used widely in the recommendation literature, particularly for the rating prediction task (Koren 2008; Salakhutdinov, Mnih, and Hinton 2007; Bi 2017; Sedhain et al. 2015). RMSE in particular is not only a popular evaluation metric but also commonly used as part of the loss function for rating prediction models.

Rating prediction is a popular recommendation system problem setting but it is not without its criticism. The three main issues we cover are the following:

1. The missingness and rating distribution assumptions
2. The difficulty of evaluating baselines
3. The disconnect from user experiences

Missingness and Rating Distribution Assumptions

One concern is that the distribution of ratings in the observed data may be different than the distribution of ratings the unobserved data. Since the goal is to suggest **novel** items to a user that they will like, one approach is to predict the ratings of all unseen items and pick the highest rated items. This approach assumes that the

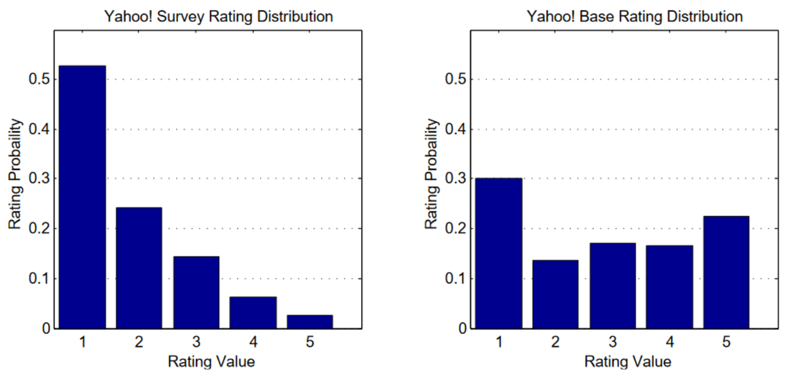


Figure 1.6: Comparison randomly assigned item ratings vs the existing rating distribution (Marlin et al. 2007).

unobserved data is missing at random (MAR). A previous study using music ratings showed that the MAR assumption may not hold (Marlin et al. 2007). A large number of users surveyed in that study described their opinion of a song affected whether or not they rated it. Figure 1.6 compares the distribution of ratings between the random selection and the existing selection.

Much research has been done to address this missingness problem. Ideally, one would evaluate a model by suggesting unobserved items to the user and observing their ratings in a live fashion. However, this is not typically possible in a research setting with static datasets. Instead, there has been research to quantify and reduce the biases that arises from the data being missing not at random (MNAR). One approach is by modeling the binary missingness data directly, such as by using the nuclear-norm-constrained matrix completion algorithm (Ma and G. H. Chen 2019; Davenport et al. 2014), and then using the calculated propensity scores to weight data. Other approaches include causal inference techniques and imputation (J. Chen et al. 2021). This body of research shows that it may be beneficial to include the unobserved data in some way to ensure that the model isn't solely learning patterns in the observed data that do not generalize to the unobserved data.

Difficulty of Evaluating Baselines

State-of-the-art rating prediction results have steadily been improving on benchmark dataset as newer sophisticated models were developed. However, studies showed that the original baseline results used in previous papers may have been sub-optimal (Steffen Rendle, L. Zhang, and Koren 2019) and on potential problems of reproducibility with the deep learning approaches (Dacrema, Cremonesi, and Jannach 2019). Their results showed that the much older baseline models such as matrix factorization, Bayesian Probabilistic Matrix Factorization, and SVD++ outperformed nearly all of the more recent state-of-the-art models when the baselines were carefully set up. This highlights the difficulty of properly setting up baselines and demonstrates that simpler models like matrix factorization are still very competitive in the current landscape. Hence in our work we are comfortable using MF as a central part of our work as it has stood the test of time.

Disconnect from User Experience

Finally, the rating prediction paradigm along with its popular evaluation metrics has been criticized that it is too far removed to the user experience of recommender systems (McNee, Riedl, and Konstan 2006). In practice, users often view recommendations as lists due to restrictions such as space on a screen. Therefore, it may better to tailor models and evaluations toward the task of predicting the best k items that will be shown to the user. This is the item recommendation scenario, also known as top-k recommendation, which has seen an emergence of research over the past decade alongside tradition rating prediction approaches.

Rating-based recommendation, where the goal is to accurately predict the rating a user would give to an item, has been a popular area of research for the last 2 decades. During that time, many new methods were developed ranging from neighborhood models, latent factor models, and neural networks. However, explicit feedback is not easy to collect as it requires active input from users. Implicit feedback on the other hand has grown exponentially as it can be collected passively from the actions of users. In this backdrop another type of recommendation research has been gaining traction - the top-k recommendation problem. In the next section we discuss the top-k item recommendation paradigm in more detail.

1.4 Top-k Recommendation

In the top-k item recommendation problem space, items are typically categorized as 'relevant' or 'irrelevant,' and the goal is to provide k item suggestions that a user should find 'relevant.' It is often treated as a two-class classification problem with the classes coming from the implicit feedback which is abundant compared to explicit feedback. This often means that the observed data are the relevant class, and the unobserved data are considered the "weakly-negative" irrelevant class.

Models originally created for rating prediction, such as KNN and MF, can also be used in the top-k recommendation scenario. However, error metrics like RMSE which are dominant in rating-based recommendation, are not necessarily a good indicator of performance on accuracy metrics used in item recommendation (Cremonesi, Koren, and Turrin 2009). Furthermore, because the nature of implicit feedback, the sparsity in the data is lost and the data space becomes much larger. Due to this, many model tailored to the item recommendation task have been developed. But first, we discuss the evaluation metrics common in top-k recommendation.

1.4.1 Ranking-based Recommendation

Evaluation metrics in the top-k item recommendation scenario are based on the item rankings rather than the item ratings that are used in rating prediction. Some metrics are truncated versions of traditional classification based metrics such as recall or precision. The central idea for these metrics is to answer the two questions for each user:

1. What are the items to rank?
2. What are the predicted ranks of the relevant items?

Assume there are n items to recommend from. For each user, u, a ranked list of those n items is generated by the recommender system. The predicted ranks of the withheld relevant items for the user is denoted as $R(u) \subseteq \{1, \ldots, n\}$, and these predicted ranks are used for evaluation. For example, $R(1) = \{2, 5\}$ means that two relevant items were predicted to be at ranks 2 and 5. A metric M is applied to each $R(u)$ to transform it into a single number and they averaged for the final value

$$\frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} M(R(u)). \tag{1.13}$$

M only uses the predicted ranks below k to contribute to the metric. Next, we cover some commonly used metrics for evaluating the quality of the rankings.

Recall at k

Recall at k measures the fraction of all items that were present in the top k ranks.

$$\text{Recall}(R)_k = \frac{|\{r \in R : r \leq k\}|}{|R|}$$

Precision at k

Precision at k measures the proportion of the top k items that are relevant.

$$\text{Precision}(R)_k = \frac{|\{r \in R : r \leq k\}|}{k}$$

Discounted Gain at k

Discounted Gain at k measures the sum of the item relevance in the top k ranks. For general relevance the formula is

$$\text{DG}(R)_k = \sum_{i=1}^{k} rel_i.$$

In the case of binary relevance (our case) it becomes:

$$\text{DG}(R)_k = \sum_{i=1}^{k} \delta(i \in R)$$

Cumulative Discounted Gain at k

Cumulative Discounted Gain at k measures the sum of the item relevance that is weighted by the item rank. For general relevance the formula is

$$\text{DCG}_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)},$$

and for binary relevance it becomes

$$\text{DCG}(R)_k = \sum_{i=1}^{k} \delta(i \in R) \frac{1}{\log_2(i+1)}$$

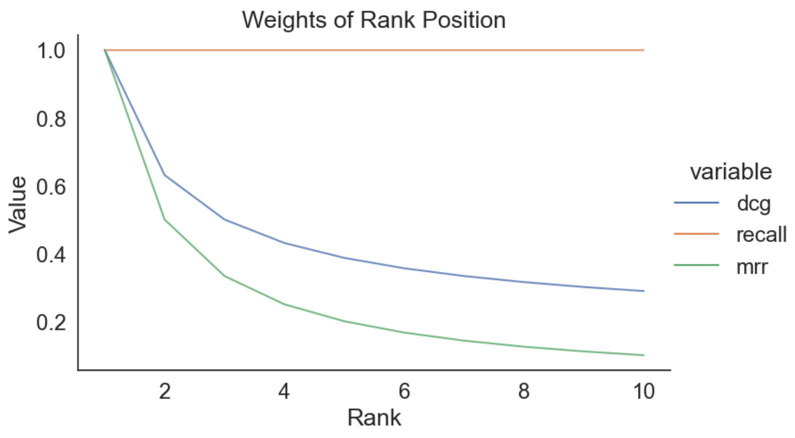


Figure 1.7: Comparing ranking evaluation weights of positions.

Normalized Discounted Cumulative Gain at k

Normalized Discounted Cumulative Gain at k (NDCG@k) normalizes the CDG at k by the Ideal Cumulative Discounted Gain (ICDG), which is the best CDG possible where all the relevant items are ranked the highest. The equation for NDCG at k is

$$\text{NDCG}(R)_k = \frac{\text{DCG}(R)_k}{\text{IDCG}(R)_k},$$

where

$$\text{IDCG}(R)_k = \sum_{i=1}^{\min(|R|,k)} \frac{1}{\log_2(i+1)} \quad .$$

Figure 1.7 shows how the different evaluation metrics weight the various ranks. At their core, the ranking evaluation metrics (accuracy metrics) are different ways to answer the question : “How well are the relevant items represented in the predicted top k ranked items?”

1.4.2 Item Recommendation Literature Review

The evaluation metrics for top-k item recommendation are more involved than their rating prediction counterparts. Furthermore, the discrete nature of ranking lists means that it difficult to use them in the loss functions directly. This means that a large variety of loss functions have been proposed such as pairwise loss functions, would compare two data points, and even list-wise loss functions, more common in the learning-to-rank literature, which calculate the loss on an entire set of items. This this section we cover a variety of recommender systems developed for the top-k recommendation problem.

Bayesian Personalized Ranking

The Bayesian Personalized Ranking (BPR) model serves as one of the inspirations of our work in Chapter 2. The BPR model takes a Bayesian approach to the ranking problem by optimizing the area under the curve AUC (Steffen Rendle et al. 2009). It uses a pairwise loss function comparing relevant to irrelevant items in the implicit feedback scenario to maximize the probability that a relevant item is ranked higher than an irrelevant item. The BPR likelihood function is

$$\begin{aligned}\prod_{u\in\mathcal{U}} p(>_u|\Theta) &= \prod_{(u,i,j)\in D_S} p(i >_u j|\Theta) && (1.14)\\ &= \prod_{(u,i,j)\in D_S} \sigma(\hat{y}_{ui} - \hat{y}_{uj}), && (1.15)\end{aligned}$$

where p is the logistic function, $>_u$ is the item ordering for user u, $\mathcal{U}$ is the set of all users, and $D_S : \mathcal{U} \times \mathcal{I} \times \mathcal{I} := \{(u,i,j)|i \in \mathcal{I}_u^+ \wedge j \in \mathcal{I} \setminus \mathcal{I}_u^+\}$. $\sigma(y_{ui} - y_{uj})$ can be thought of as the probability that user u would prefer item i over item j.

One of the challenges has been the problem of selecting the irrelevant item to pair with the relevant item. The original model uses a uniform sampling process but other informative sampling methods were shown to improve performance (Lian, Q. Liu, and E. Chen 2020).

Sparse Linear Methods

The Sparse Linear Methods model (SLIM) is a linear model that learns a sparse item-to-item similarity matrix W by minimizing the following optimization function

$$\begin{aligned} \min_{W} \quad & \frac{1}{2}||A - AW||_F^2 + \frac{\beta}{2}||W||_F^2 + \lambda||W||_1 && (1.16) \\ \text{subject to} \quad & W \geq 0, && (1.17) \\ & \operatorname{diag}(W) = 0. && (1.18) \end{aligned}$$

A is the utility matrix, $|| \cdot ||_F$ is the Frobenius norm, and $||W||_1 = \sum_{i=1}^{n} \sum_{j=1}^{n} |w_{ij}|$ is the element wise ℓ_1-norm of W. SLIM generated fast recommendations like item neighborhood methods that were high quality like model-based methods by learning a sparse representation of the item-to-item similarity matrix (Ning and Karypis 2011).

Neural Collaborative Filtering

Neural Collaborative Filtering (NCF) was a neural network based model that generalized matrix factorization by adding nonlinear interactions between the user and item latent factors (He et al. 2017). It extended the matrix factorization model with additional nonlinear layers and replaced the commonly used dot product with a multilayer perceptron.

Variational Autoencoder for Collaborative Filtering

The Variational Autoencoder for Collaborative Filtering (VAECF) model introduces a multinomial variational autoencoder. It is based on the variational autoencoder but uses a multinomial likelihood and Bayesian inference for parameter estimation (Liang et al. 2018). Like an autoencoder, it learns to reconstruct its inputs but instead of a deterministic lower dimensional latent embedding, it embeds to a parameter space of a Gaussian distribution which is then sampled from to generate its targets.

Embarrassingly Shallow Autoencoders

The EASE^R model is a simple linear model geared toward implicit data whose training objective has a closed form solution (Steck 2019). Surprisingly, it achieves better ranking accuracy than many other more complicated models (Steck 2019). It is similar to SLIM but uses different constraints for optimization is able to be trained much faster. The parameters of the model are the item-item weight matrix $\boldsymbol{B} \in \mathbb{R}^{|\mathcal{I}|\times|\mathcal{I}|}$. Given the utility matrix $\boldsymbol{Y}$, the predicted score is calculated as

$$\hat{y}_{ui} = \boldsymbol{Y}_{u,\cdot} \cdot \boldsymbol{B}_{\cdot,i}, \tag{1.19}$$

where $\boldsymbol{Y}_{u,\cdot}$ refers to row u and $\boldsymbol{B}_{\cdot,i}$ refers to row i.

The objective function for EASE^R is:

$$\min_{B} \quad ||\boldsymbol{Y} - \boldsymbol{Y}\boldsymbol{B}||_F^2 + \lambda \cdot ||\boldsymbol{B}||_F^2 \tag{1.20}$$

$$\text{s.t.} \quad \text{diag}(\boldsymbol{B}) = 0. \tag{1.21}$$

The closed from of the weights of $\boldsymbol{B}$ are given by:

$$\hat{\boldsymbol{B}}_{i,j} = \begin{cases} 0 & \text{if } i = j \\ -\frac{\hat{P}_{ij}}{\hat{P}_{jj}} & \text{otherwise,} \end{cases} \tag{1.22}$$

where $\hat{P} \coloneqq (\boldsymbol{Y}^T\boldsymbol{Y} + \lambda I)^{-1}$ for sufficiently large λ. EASE^R is like an autoencoder in that it recreates its inputs, but instead of a hidden layer it uses a linear transformation with the constraint that the self-similarity of each item in the input and output layer is constrained to zero.

1.4.3 Challenges in Top-k Item Recommendation

The top k recommender field has exploded with new models and state-of-the-art claims and many novel deep learning approaches to the problem. However two areas of improvement include:

1. The inconsistencies involving evaluation metrics (Krichene and Steffen Rendle 2020; Tamm, Damdinov, and Vasilev 2021)
2. The difficulty of reproducing many of the state of the art deep learning approaches (Dacrema, Boglio, et al. 2021).

Inconsistency in Evaluation Metrics

As mentioned before, the prevalence of sampled evaluation metrics as well as the inconsistent sampling sizes make comparing top k recommender systems difficult and can allow for spurious discoveries. Furthermore, there are differences in literature and popular libraries on how common metrics are implemented. This means that two different libraries could provide different numbers for the same metric on the exact same model and data. This is due to the nuances of how the items to be ranked are selected, and how edge cases may be handled.

Competitive Baseline Performance and Reproducibility

Also, in the reproducibility study in (Dacrema, Boglio, et al. 2021), it was found that only a small number of published state-of-the-art models had provided enough code and the data to reproduce the method with reasonable effort. In addition, many of the deep learning based approaches when reproduced actually performed more poorly than even much older and simpler baselines like item KNN, SLIM, and graph based models. Furthermore, many of the deep learning models look orders of magnitude longer to train than their simpler counterparts. Despite these many challenges, research in the top k continues to mature and improve.

As seen in matrix factorization, SLIM, and EASE, sometimes, simple models and ideas and perform surprisingly well compared to much more complex models. Rating-based models, which use explicit feedback, struggle to perform well in top-k evaluation metrics. On the other hand, ranking-based models ignore the valuable information found in explicit feedback. The key idea of the next chapter is the following: Can we perform well in the top-k recommendation scenario, which includes users' missing items in ranking, by solely utilizing observed explicit feedback and ignoring the unobserved data altogether? We propose a pairwise model, Pairwise Personalized Ranking, to address this question.

Chapter 2

Pairwise Personalized Ranking

Our method, Pairwise Personalized Ranking (PPR), is a pairwise model that fully utilizes available explicit feedback to perform well in ranking-based evaluation methods while retaining high performance and computational efficiency. Top-k recommendation models treat the data as a two class problem and the split the data so that the observed data is 'relevant' and the missing data is 'irrelevant'. The observed data is treated as a weak positive signal and the missing data is treated as a weak negative signal. Although effective, when used on binarized explicit feedback, it completely ignores the rich information provided in the explicit feedback itself. Furthermore, items with low ratings, should not be considered as weak positive feedback.

We hone in on this idea and develop a recommender that uses a pairwise loss function on explicit feedback. Instead of assuming weakly positive and negative implicit data, we directly use the explicit feedback and split the data into "true positive" and "true negative" interactions. The key idea we center on is as follows: Instead of training a pairwise model on implicit data and assume weak positive and negative signals, can we get strong performance comparing within the explicit feedback itself without using the unobserved data at all? Since our PPR model uses the matrix factorization model as its scoring function, we first go over matrix factorization in more detail.

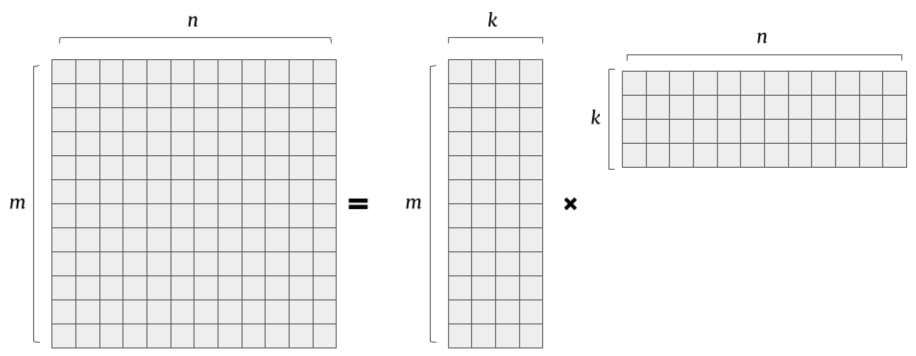


Figure 2.1: Factorization of the user-item matrix.

2.1 Matrix Factorization Model

Let $\boldsymbol{Y} \in \mathbb{R}^{m\times n}$ be the sparse matrix of user-item ratings, where there are m users, and n items. Then, $y_{ui} \in \boldsymbol{Y}$ is the rating that a user u gave to item i, where $u \in \{1, \ldots, m\}$ and $i \in \{1, \ldots, n\}$. Matrix factorization decomposes $\boldsymbol{Y}$ into the product of two low-rank matrices: the user and item embeddings $\boldsymbol{P}$ and $\boldsymbol{Q}$ respectively. Figure 2.1 shows an example of how the user-item matrix is factorized . Then, the predicted rating $\hat{y}_{ui}$ can be calculated by

$$\hat{y}_{ui} = \boldsymbol{p}_u^\top \boldsymbol{q}_i, \tag{2.1}$$

which is similar to the singular value decomposition of a matrix.

However, because the ratings matrix $\boldsymbol{Y}$ is sparse, a normal SVD method is not applicable. A naive way to correct for this would be to train on only the observed data, but this would likely result in overfitting. To avoid this, it is common to regularize

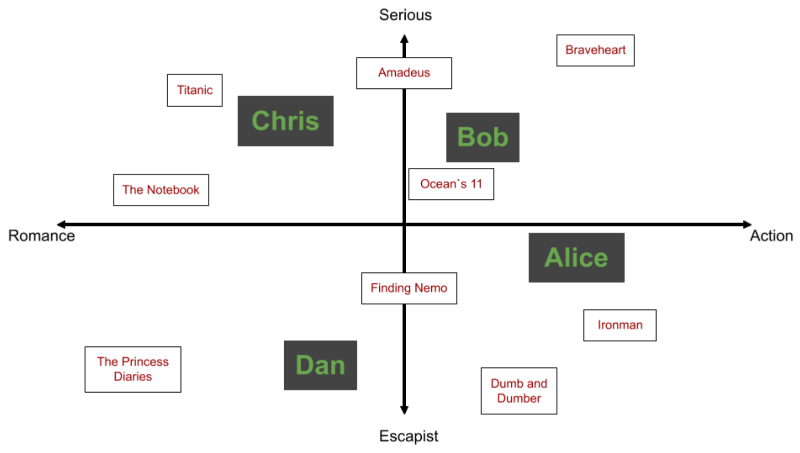


Figure 2.2: Interpretation of embeddings.

the parameters of the latent factors. The loss function to be minimized then becomes

$$L(\boldsymbol{p}, \boldsymbol{q}, \Omega) = \sum_{(u,i)\in\Omega} (y_{ui} - \boldsymbol{p}_u^\top \boldsymbol{q}_i)^2 + \lambda(||\boldsymbol{p}_u||^2 + ||\boldsymbol{q}_i||^2), \tag{2.2}$$

where Ω is the set of observations and λ is the regularization parameter.

One interpretation for the the model is that the rating for a specific user-item pair depends on how similar the user embeddings are to the item embeddings. The meaning of the embeddings themselves are not explicitly known, but may be inferred after the fact by the way the items are distributed along certain factors. For example, the embeddings may encode the affinity or amount of specific genres, and if both the user embedding and item embedding contain large values for that element, then it would positively impact the predicted rating $\hat{y}_{ui}$.

In practice, matrix factorization often includes additional bias parameters such that the $\hat{y}_{ui}$ depends on a global effect, μ, a user effect, $\boldsymbol{b}_u$, an item effect, $\boldsymbol{b}_i$, as well as the inner product of the user and item embeddings $\boldsymbol{p}_u^\top \boldsymbol{q}_i$. The scoring function then becomes

$$\hat{y}_{ui} = \mu + \boldsymbol{b}_u + \boldsymbol{b}_i + \boldsymbol{p}_u^\top \boldsymbol{q}_i, \tag{2.3}$$

where the $\boldsymbol{p}_u, \boldsymbol{q}_i \in \mathbb{R}^k$.

Two methods to solve the loss function in Equation 2.2 are Stochastic Gradient Descent and alternating least squares.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a widely used iterative optimization algorithm. The key idea for SGD is to approximately travel along the gradient of the loss function in a noisy manner by updating model parameters based on a single training observation. A small subset of training data, called a batch, may be used instead of a single data point, and this is often referred to as mini-batch or batch gradient descent. However, we use the term SGD to refer to both interchangeably. By stepping in the gradient direction after each small batch, SGD can avoid the computation burden of traditional gradient descent which requires the gradient to be calculated over the entire training data. This makes SGD a particularly good optimization method when dealing with the large data sets common in recommender systems. Figure 2.3 highlights the difference between SGD and traditional gradient descent (*sgd-vs-gd* n.d.).

For a given training observation, the loss function error is used to calculate the stochastic gradient. For the simple matrix factorization model in Equation 2.1, the

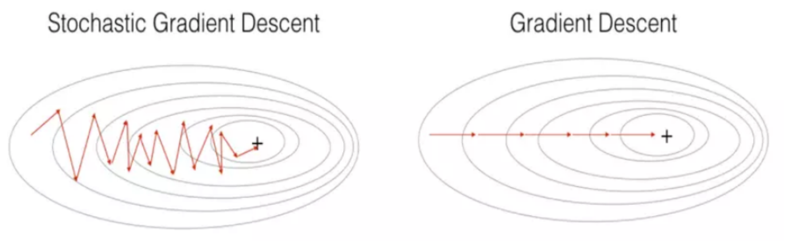


Figure 2.3: Stochastic Gradient Descent vs Gradient Descent.

gradient is shown in Equation 2.2.

$$\begin{aligned} \nabla \boldsymbol{p}_u &= (y_{ui} - \boldsymbol{p}_u^\top \boldsymbol{q}_i) + \lambda \boldsymbol{p}, \\ \nabla \boldsymbol{q}_u &= (y_{ui} - \boldsymbol{p}_u^\top \boldsymbol{q}_i) + \lambda \boldsymbol{q}. \end{aligned} \tag{2.4}$$

Then the approximate gradient and a learning rate γ are used to step to the updated model parameters. Given that the error term is

$$e_{ui} \equiv y_{ui} - \boldsymbol{p}_u^\top \boldsymbol{q}_i, \tag{2.5}$$

the parameter update step equations are

$$\begin{aligned} \boldsymbol{p}_u &\leftarrow \boldsymbol{p}_u + \gamma(e_{ui}\boldsymbol{q}_i - \lambda \boldsymbol{p}_u), \\ \boldsymbol{q}_i &\leftarrow \boldsymbol{q}_i + \gamma(e_{ui}\boldsymbol{p}_u - \lambda \boldsymbol{q}_i). \end{aligned} \tag{2.6}$$

In practice, SGD performs well in many optimization problems. But in general, convergence for SGD is not guaranteed. However, when the learning rate γ decreases at an appropriate rate, the loss function is Lipschitz, and the noise in the stochastic gradient has bounded support, SGD will converge almost surely to a local minima in the non-convex case. Furthermore, modifications to SGD such as Nesterov momentum, acceleration, and others can speed up convergence.

Alternating Least Squares

Another popular approach to solve Equation 2.2 is called Alternating Least Squares (ALS). While the loss function in Equation 2.2 is non-convex, if one of $\boldsymbol{p}$ or $\boldsymbol{q}$ is fixed, it becomes a quadratic optimization problem. Then the approach will be to fix $\boldsymbol{P}$ and optimize $\boldsymbol{Q}$, then fix $\boldsymbol{Q}$ and optimize $\boldsymbol{P}$, and repeat until convergence.

$$\boldsymbol{p}_u = \left(\sum_{y_{ui} \in \Omega} \boldsymbol{q}_i \boldsymbol{q}_i^\top \right)^{-1} \sum_{y_{ui} \in \Omega} y_{ui} \boldsymbol{q}_i \quad \text{for } u = 1, \ldots, m, \tag{2.7}$$

$$\boldsymbol{q}_i = \left(\sum_{y_{ui} \in \Omega} \boldsymbol{p}_u \boldsymbol{p}_u^\top \right)^{-1} \sum_{y_{ui} \in \Omega} y_{ui} \boldsymbol{p}_u \quad \text{for } i = 1, \ldots, n \; . \tag{2.8}$$

ALS can be computed in a distributed manner which allows it work very well for extremely large datasets (Zhou et al. 2008) and parallel computation.

In our experiments in Chapter 2 and Chapter 3, we use SGD and its variants to optimize the non-convex loss functions. One thing to highlight is the calculation of the stochastic gradient is a important part that will be used in Chapter 3. Next we cover how the PPR model leverages explicit feedback for training.

2.2 Pairwise Personalized Ranking Model

One of the distinguishing features of explicit feedback to implicit feedback is that the user provides finer resolution on the interaction it had with an item. The existence of such information allows us to make more informed decisions when determining whether or not a specific user-item interaction was truly positive or negative. When using rating data converted into implicit feedback, the recommender treats any rating as positive. This is problematic since it is highly unlikely that users would want to treat items that were rated as 1 star to be treated the same as items that were rated as 5 stars. Hence, we use the rating data itself to split the observed data into a true positive and true negative class.

After partitioning the observed data from the ratings dataset into true positive and true negative sets based on some criteria, PPR compares those positive and negative items for a given user to learn to differentiate between the two. This criteria can be a global one, such as a global constant, or a user-specific criteria such as users' rating quantiles. The pairwise loss function optimizes the ordering of the true positive and true negative items.

Recall that the set of observed data, Ω, contains the user-item tuples, (u, i), of all observed data - both true positives and true negatives. This set of tuples, Ω, will be used to generate user, positive item, negative item triplets, (u, i, j), used for training the Pairwise Personalized Ranking model. First, we partition the data into true positives and true negatives using the global or user-specific criteria from before. Given a value c_u that may be user-specific, let the set of user, true positive, and true negative item triplets be:

$$T = \{(u, i, j) : (u, i) \in \mathcal{K},\ (u, j) \in \mathcal{K},\ y_{uj} < c_u \leq y_{ui}\}. \tag{2.9}$$

This set contains the tuples of all possible user id, positive item id, and negative item id triplets and allows us to train the model to properly order the true positives and the true negative items for each user. It can be much larger than than Ω, and is on the order of $|\Omega|^2$. The quadratic increase in data points can cause computational challenges if not handled appropriately.

Given that we want to maximize the probability that a positive item is ranked higher than a negative item, the optimization function using a pairwise loss function is:

$$\min_{\theta} \sum_{(u,i,j)\in T} -\log(\sigma(\hat{y}_{u,i,\theta} - \hat{y}_{u,j,\theta})) + \frac{\lambda}{2}(||\theta||), \tag{2.10}$$

where θ are the model parameters and $\hat{y}_{u,i,\theta}$ is a generic scoring function with parameters θ. This optimization function wants to maximize the probability that a true positive item will be ranked higher than a true negative item for a user.

Using a matrix factorization model with item biases as the underlying scoring function for PPR, gives us

$$\hat{y}_{ui} = \boldsymbol{p}_u^\top \boldsymbol{q}_i + \boldsymbol{b}_i. \tag{2.11}$$

Since the optimization function in Equation 2.10 takes the difference within a users items, any user-specific bias will be cancelled out. Then, the optimization function from Equation 2.10 becomes:

$$\min_{\boldsymbol{P},\boldsymbol{Q},\boldsymbol{b}} \left(\sum_{(u,i,j)\in T} -\log(\sigma(\hat{y}_{uij})) \right) + \frac{\lambda}{2}\left(||\boldsymbol{P}||_F^2 + ||\boldsymbol{Q}||_F^2 + ||\boldsymbol{b}||_2^2\right), \tag{2.12}$$

where $\hat{y}_{uij} = \hat{y}_{ui} - \hat{y}_{uj}$, $||\cdot||_F^2$ is the squared Frobenius norm, and $||\cdot||_2^2$ is the squared L_2 norm. Conceptually, this optimization problem wants to ensure that true positive items score higher than true negative items for a given user i.e. maximize the probability that a true positive item will be ranked higher than a true negative item.

2.2.1 Optimization Strategy

To optimize Equation 2.12, we use stochastic gradient descent (SGD). The gradients with respect to $\boldsymbol{p}, \boldsymbol{q}$, and $\boldsymbol{b}_i$ simplified as θ are:

$$\begin{aligned}
\frac{\partial}{\partial \theta} L &= -\frac{\partial}{\partial \theta} \log(\sigma(\hat{y}_{uij})) + \frac{\partial}{\partial \theta} \frac{\lambda}{2} ||\theta||_2^2 && (2.13) \\
&= -\frac{1}{\sigma(\hat{y}_{uij})} \frac{\partial}{\partial \theta} \sigma(\hat{y}_{uij}) + \lambda \theta && (2.14) \\
&= -\frac{1}{1 + e^{\hat{y}_{uij}}} \frac{\partial}{\partial \theta} \hat{y}_{uij} + \lambda \theta. && (2.15)
\end{aligned}$$

Recall that $\hat{y}_{uij} = \boldsymbol{p}_u^\top \boldsymbol{q}_i + \boldsymbol{b}_i - (\boldsymbol{p}_u^\top \boldsymbol{q}_j + \boldsymbol{b}_j)$. By substituting $z_{uij} = -\frac{1}{1+e^{\hat{y}_{uij}}}$, we see the gradients will be

$$\frac{\partial}{\partial \theta} L = \begin{cases}
z_{uij}(\boldsymbol{q}_i - \boldsymbol{q}_j) + \lambda \boldsymbol{p}_u & \text{if } \theta = \boldsymbol{p}_u \\
z_{uij} \boldsymbol{p}_u + \boldsymbol{q}_i & \text{if } \theta = \boldsymbol{q}_i \\
-z_{uij} \boldsymbol{p}_u + \boldsymbol{q}_j & \text{if } \theta = \boldsymbol{q}_j \\
z_{uij} + \boldsymbol{b}_i & \text{if } \theta = \boldsymbol{b}_i \\
-z_{uij} + \boldsymbol{b}_j & \text{if } \theta = \boldsymbol{b}_j.
\end{cases} \qquad (2.16)$$

These gradients are used to update the model parameters at every iteration. Algorithm 1 shows the full algorithm including the SGD steps for the Pairwise Personalized Ranking model. In particular, we show two examples of possible implementations of the GenerateTriplets function.

The first implementation is the naive one, where all possible (u, i, j) tuples are created and then iterated over. As we cover later, it is not particularly efficient, but it does allow the triplet dataset to be created offline.

Algorithm 1 Pairwise Personalized Ranking

```
Require: $\Omega$ observed data, $\boldsymbol{Y}$ utility matrix, $c_u$ rating threshold.
 1: Create Negative Item Mapping $\mathcal{N}(u) = \mathcal{I}_u^-$ for $u \in \mathcal{U}$ where
        $$\mathcal{I}_u^- := \{i \in \mathcal{I} : (u,i) \in \Omega,\ y_{ui} < c_u\} \qquad (2.17)$$
 2: for $i = 1, \ldots, N$ do
 3:     $T \leftarrow$ GenerateTriplets$(\Omega, \mathcal{N})$
 4:     for all $(u,i,j) \in T$ do
 5:         $z_{uij} = -\left(\frac{1}{1+e^{\hat{y}_{uij}}}\right)$
 6:         Update parameters:
 7:         $\boldsymbol{p}_u \leftarrow \boldsymbol{p}_u - \gamma\left(z_{uij} \cdot (\boldsymbol{q}_i - \boldsymbol{q}_j) - \lambda \boldsymbol{p}_u\right)$
 8:         $\boldsymbol{q}_i \leftarrow \boldsymbol{q}_i - \gamma\left(z_{uij} \cdot \boldsymbol{p}_u + \lambda \boldsymbol{q}_i\right)$
 9:         $\boldsymbol{q}_j \leftarrow \boldsymbol{q}_j - \gamma\left(-z_{uij} \cdot \boldsymbol{p}_u + \lambda \boldsymbol{q}_j\right)$
10:         $\boldsymbol{b}_i \leftarrow \boldsymbol{b}_i - \gamma\left(z_{uij} + \lambda \boldsymbol{b}_i\right)$
11:         $\boldsymbol{b}_j \leftarrow \boldsymbol{b}_j - \gamma\left(-z_{uij} + \lambda \boldsymbol{b}_j\right)$
12:     end for
13: end for
```

Algorithm 2 Naive GenerateTriplets

```
  function GenerateTriplets$(\Omega, \mathcal{N})$
Require: $\Omega$ observed $(u,i)$ pairs, $\mathcal{N}$ negative item mapping
      for all $(u,i) \in \Omega$ do
          $T_{ui} = \{(u,i,j) : j \in \mathcal{N}(u)\}$
      end for
      return $T$
  end function
```

The next implementation is a more efficient one and does not require the entire triplet set T to be stored in memory. It uses a two-step sampling procedure, the first to sample the (u, i) and the second to sample the corresponding j. We note that the specific sampling mechanism can be adjusted. We use a uniform randomly sampling method, but as seen in (Lian, Q. Liu, and E. Chen 2020), a more sophisticated sampling scheme may improve performance.

Algorithm 3 Random Sampling GenerateTriplets

```
  function GenerateTriplets($\Omega, \mathcal{N}$)
Require: $\Omega$ observed $(u, i)$ pairs, $\mathcal{N}$ negative item mapping, $n_T$ number of triplets
    for $k = 1, \ldots, n_T$ do
        $(\tilde{u}, \tilde{i}) \sim \Omega$ uniformly without replacement.
        $\tilde{j} \sim \mathcal{N}_{\tilde{u}}$ uniformly with replacement.
        $T_k = (\tilde{u}, \tilde{i}, \tilde{j}$
    end for
    return $T$
  end function
```

One thing to note is that the resulting distributions of T from Algorithm 2 and Algorithm 3 are slightly different. Algorithm 2 creates T using the entire triplet space which means that each user will have $\propto \frac{|\Omega_u| \times |\mathcal{N}(u)|}{|T|}$ triplets while in Algorithm 3 each user will have $\propto \frac{|\Omega_u|}{|\Omega|}$ triplets.

2.3 Computational Challenges

There were significant computational challenges faced during the development of this model. The first was due to the large data size. This meant often meant that it was difficult to keep the entire set of data triplets T in memory as the memory increased quadratically with the number of ratings. The second was due to runtime on such large amounts of data. As mentioned in previous work, training recommendation

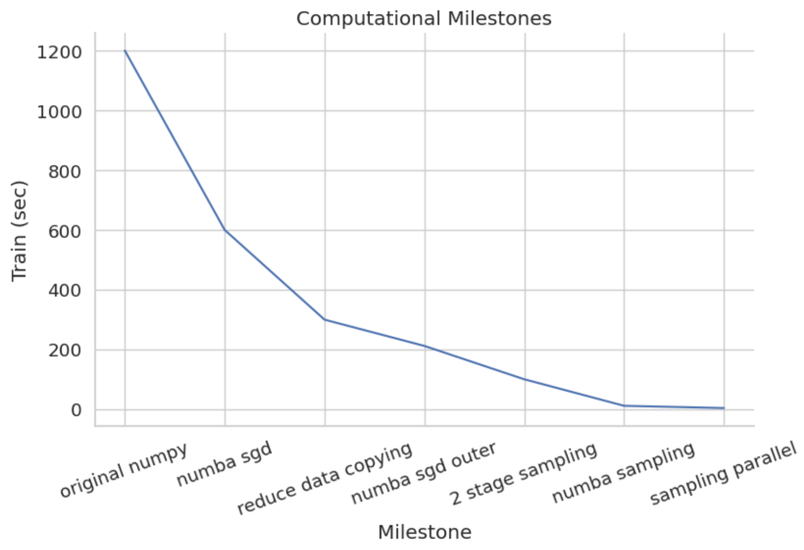


Figure 2.4: Computational milestones.

models can take significant amounts of time, and well set up hyper-parameter tuning may not be feasible if the runtime is too onerous (Dacrema, Boglio, et al. 2021). The complexity of some previous methods meant that hyper-parameter tuning could take days or even weeks. With this in mind we wanted to create a model that was able to be trained efficiently.

The entire model and training code base was written in Python as it was the language the author was most experienced in. The initial implementation of the Pairwise Personalized Ranking model was purely using the `numpy` library, which is a well known numerical multidimensional array library (Harris et al. 2020). However, due to the complex nature of the SGD loop, performance was subpar as it required for loops, which are much slower than vectorized operations.

To remedy the SGD hot spot, we turned to `numba`, a open source Just-In-Time compiler that can accelerate computation in Python. As shown in Figure 2.4, converting portions of the code to leverage `numba` allowed significant speed ups. Converting the inner SGD loop, line 9 in Algorithm 1, to `numba` halved the training time. Further improvements were made by eliminating unnecessary data copying as well as converting the outer SGD loop, line 4, to `numba` as well.

Another large runtime improvement was transitioning from a single sampling process to a two stage sampling process. Previously, all possible data triplets, (u, i, j), were generated in the beginning and then looped through. However, by only sampling the unique (u, i) pairs in the observed dataset and then sampling a negative time j, we found significant performance improvements. This approached is described in line 6 in Algorithm 1.

The final computational milestones were achieved by converting the second sampling stage to `numba` as well as sampling in parallel. Through all these updates we were able to improve the runtime of the code by more than a factor of 1000. This allowed us to efficiently tune hyper-parameters as it ran on the scale of hours rather than weeks. The final version of the model could train in a similar amount of time as the existing BPR model, which was mostly coded in C++ and had much simpler negative pair sampling mechanism. A discussion regarding the computational complexity can be found in Appendix A.1.

2.4 Extensions to Pairwise Personalized Ranking

The Pairwise Personalized Ranking model exclusively uses the explicit feedback to learn user item preferences. In this section we cover 2 extensions to the PPR model. Previous research indicated that utilizing both explicit and implicit feedback could improve accuracy (Koren, Bell, and Volinsky 2009). With this in mind, in the first extension, we leverage both explicit feedback and implicit feedback in a simple way. The

Alternating Personalized Ranking model alternates between comparing true positive to true negative items and comparing observed items to unobserved items.

The simplified corresponding loss function is equivalent to

$$\min_{\theta} -\left(\sum_{(u,i,j)\in T} \log(\sigma(\hat{y}_{u,i} - \hat{y}_{u,j})) + \sum_{(u,i,j)\in D} \log(\sigma(\hat{y}_{u,i} - \hat{y}_{u,j})) \right) + \frac{\lambda}{2}(||\theta||), \quad (2.18)$$

where the underlying scoring models share the same embedding space. In this manner the model is able to include information from missing items into its predictions while keeping the strong signals present in the explicit feedback. The optimization approach is similar to the PPR optimization strategy where we use SGD.

For the second extension to the PPR model, we note that ratings data can be partitioned into 3 classes: true positive, true negative, and missing. One way to expand on the Alternating Personalized Ranking idea is to use a triplet loss function that compares those three classes to each other. We assume that an general ordering exists between the three classes such that

$$\text{True Negative} < \text{Missing} < \text{True Positive}. \quad (2.19)$$

Then the goal would be to optimize the ordering of the three classes simultaneously

$$\min_{\theta} -\left(\sum_{(u,i,j,k)\in\tilde{T}} \log(\sigma(\hat{y}_{u,i,j})) + \log(\sigma(\hat{y}_{u,i,k})) + \log(\sigma(\hat{y}_{u,k,j})) \right) + \frac{\lambda}{2}(||\theta||), \quad (2.20)$$

where (u, i, j, k) corresponds to the user, true positive item, true negative item, and missing item respectively. We denote this model as the Triple Personalized Ranking model (TPR). Again, we use SGD to optimize the Triple Personalized Ranking.

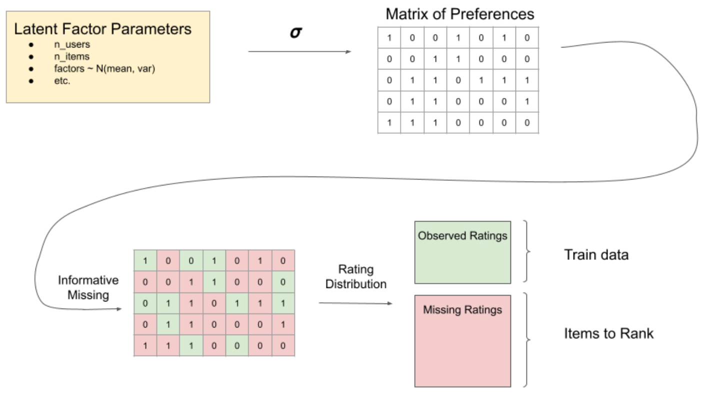


Figure 2.5: Simulation diagram.

2.5 Simulations

We conduct a variety of experiments on simulated data to compare the PPR and APR methods to other baselines. The baselines for the simulations include BPR, EASE, VAECF, MF, I-KNN, and U-KNN.

2.5.1 Data Generating Process

At a high level, our goal is to generate realistic simulated data from a set of user and item factors whose ratings and probabilities of being missing depend on whether or not the user-item interaction is a true positive or a true negative. True positive items should have a higher probability than a true negative item to be observed. The ratings of true positives should be on average higher than the ratings of true negatives.

To generate the simulated data, first the latent factors $\boldsymbol{P}$, $\boldsymbol{Q}$, $\boldsymbol{b}_u$, and $\boldsymbol{b}_i$ are generated by sampling repeatedly from a normal distribution. Then the scores, $\boldsymbol{Z}$ are calculated by taking $\boldsymbol{P}\boldsymbol{Q}^\top$ and adding $\boldsymbol{b}_u$ row-wise and $\boldsymbol{b}_i$ column-wise. The scores, $\boldsymbol{Z}$, are passed

Algorithm 4 Simulated Data Generation

```
Require: True latent parameters: $\boldsymbol{\mu}, \boldsymbol{\Sigma}, k, n, m$
1: for Each user and item factor do
2:     $\boldsymbol{p}_u, \boldsymbol{b}_u, \boldsymbol{q}_i, \boldsymbol{b}_i \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
3: end for
4: $\boldsymbol{R}_{ui} \sim \text{Bern}(\sigma(\boldsymbol{p}_u \boldsymbol{q}_i))$
5: $\boldsymbol{M}_{ui} \sim \text{Bern}(p)$ where $p = \begin{cases} p_+ & \text{if } R_{ui} = 1 \\ p_- & \text{otherwise} \end{cases}$
6: $\boldsymbol{Y}_{ui} \sim \text{Cat}(\boldsymbol{R}_{ui})$ where $\text{Cat}(\boldsymbol{R}_{ui}) = f(x = k|\boldsymbol{p})$ for $k \in \{1, 2, 3, 4, 5\}$
   where $\begin{cases} p_1 + p_2 + p_3 < p_4 + p_5 & \text{if } R_{ui} = 1 \\ p_1 + p_2 + p_3 > p_4 + p_5 & \text{if } R_{ui} = 0 \end{cases}$
7: $\mathcal{X}_{train} = \{(u, i, \boldsymbol{Y}_{ui}) | \boldsymbol{M}_{ui} = 1\}$
8: $\mathcal{X}_{test} = \begin{cases} 80\% - 20\% \text{ random split withheld from } \mathcal{X}_{train} & \text{if not full test} \\ \{(u, i, \boldsymbol{Y}_{ui}) | \boldsymbol{M}_{ui} = 0\} & \text{if full test} \end{cases}$
```

to the logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$ and the resulting probabilities are used as p for the Bernoulli distribution. This generates the binary true relevance matrix, R which determines whether items are true positive or true negative for each user.

Then R is used to generate both the missing matrix mask M and the rating matrix $\boldsymbol{Y}$. We generate M using a pair of Bernoulli distributions, Bern(p_+) and Bern(p_-), one for the true positives and one for the true negatives. This allows true positives to have a higher probability of being observed compared to true negatives, i.e. $p_- < p_+$. To generate the ratings matrix $\boldsymbol{Y}$ from R, we use a categorical distribution with five classes corresponding to the integer ratings from 1 to 5. We assume that $c_u = 4$ is the true cutoff criteria for all users. This means that true positive items will be either a 4 or a 5 rating and true negatives will be a 3 or lower. The missing mask M is then used to select elements of $\boldsymbol{Y}$ that are observed and in the training set.

We utilize two test sets. The first test set is a small test set and is created by further splitting the training set 80-20 train-test. This mirrors what typically occurs when you do not have the true ratings of the unobserved items. However, since these are

Dataset	Users	Items	k	μ	σ	sparsity	$\frac{p_+}{p_-}$	$\boldsymbol{p}_+$	$\boldsymbol{p}_-$
Medium	400	800	15	-2	2	0.02	2	(0,0,0,.5,.5)	(.35,.35,.3,0,0)
Large	2000	4000	20	-1	1	0.02	2	(0,0,0,.5,.5)	(.35,.35,.3,0,0)
MF	943	1682	20	-	-	0.02	1	-	-
Hard	2000	4000	40	-3	1	0.02	3	(0,.02,.03,.5,.45)	(.3,.32,.35,.03,0)

Table 2.1: Simulated dataset parameters

simulations, we have the true ratings of the unobserved items. The unobserved items are used as the second test set. This set has a slightly different distribution than the observed dataset and is much larger than the observed dataset.

2.5.2 Recommendation Accuracy

The parameters used for the three simulated dataset are found in Table 2.1. A lower μ means that there is overall a larger proportion of negative ratings in the entire dataset. A larger $\frac{p_+}{p_-}$ means that positive items are much more likely to be observed than negative items.

Simulation Medium

Figure 2.7 shows the distribution of ratings for both the training and test set for the medium simulated dataset. Figure 2.6 compare the performance of various models on two test sets. One test set is withheld from the smaller observed data while the full test set contains the ratings of all the missing data. We see that although performance on the smaller test set is inconclusive, we see strong performance of both Alternating Personalized Ranking and Pairwise Personalized Ranking on the full test set. They both perform better than nearly all the baselines and are on par with the EASE model.

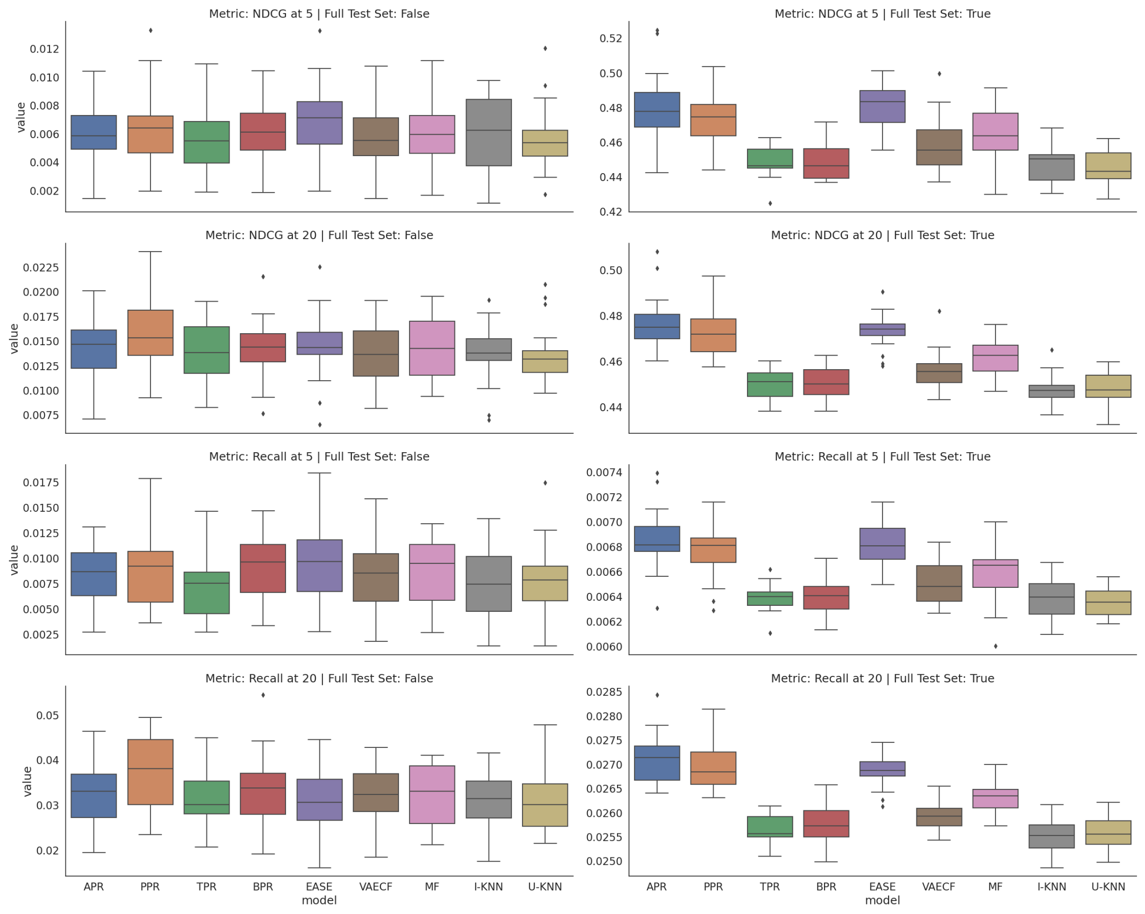


Figure 2.6: Evaluation Metrics on Medium simulated dataset. The metrics used for each row starting from the top are NDCG@5 (top), NDCG@20, Recall@5, and Recall@20 (bottom). Using the small test set (left) shows little difference between models, but the full test set (right) clearly shows performance differences.

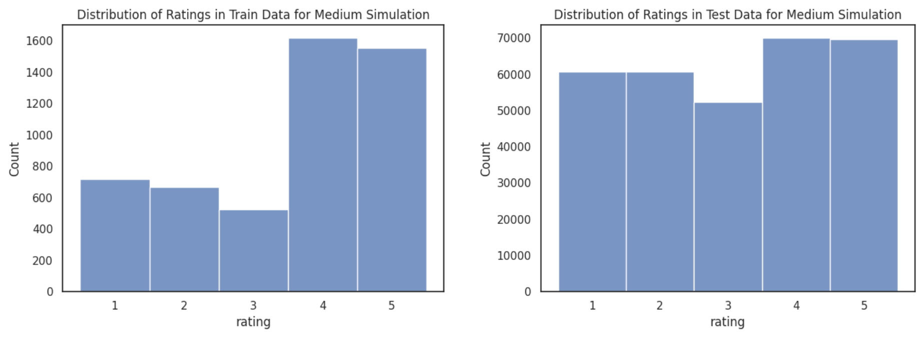


Figure 2.7: Histogram of ratings for the Medium dataset with the training set (left) and test set (right).

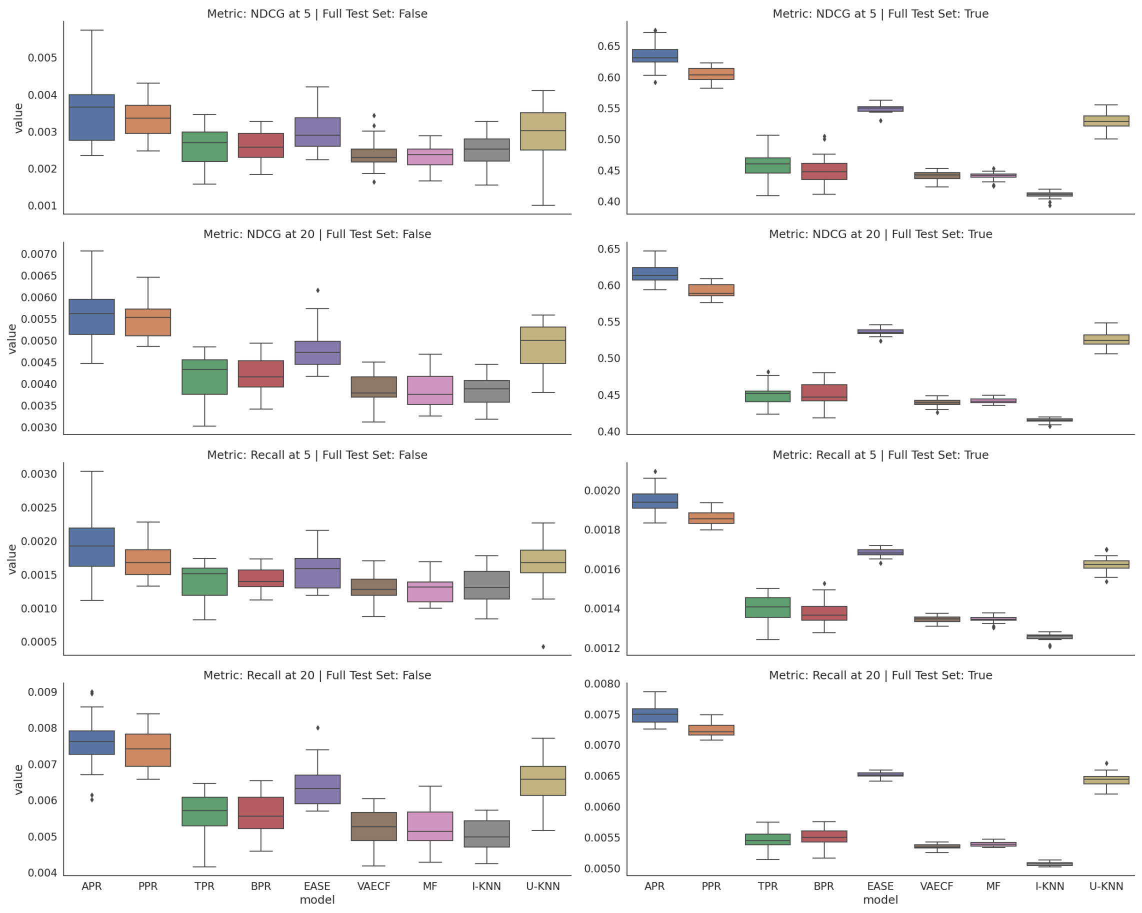


Figure 2.8: Evaluation Metrics on Large simulated dataset for the small test set (left) and the full test set (right). The metrics used for each row starting from the top are NDCG@5 (top), NDCG@20, Recall@5, and Recall@20 (bottom). APR and PPR outperform other models with EASE right behind.

Simulation Large

In this simulation we use a much larger dataset (large) with a slightly different mean and standard deviation. The large mean value means that there will be a higher proportion of positive items in the dataset compared to the medium dataset. Figure 2.9 shows the distribution of ratings for both the train and test set and Figure 2.8 compares evaluation metrics between various models. In this larger simulated dataset, we see that our method out performs all the other methods on the full test set.

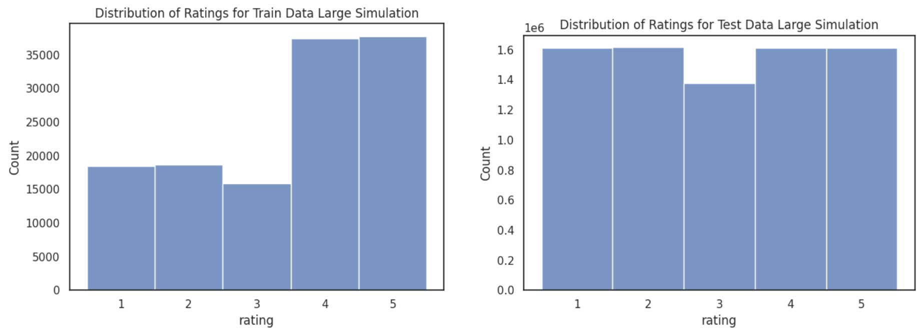


Figure 2.9: Histogram of ratings for the Large dataset with the training set (left) and test set (right).

Simulation MF

In this simulation, we recreate the ratings of the MovieLens 100K dataset. We first fit the vanilla matrix factorization with 20 factors on the MovieLens 100K. Then the learned user and item factors are used to create the simulated full dataset by multiplying them, truncating to 1 to 5, and rounding to the nearest integer. As expected, the MF model improves relative to the other simulations as it it closely matches the data generating process. Even so, we see that our proposed methods are on par with the best models.

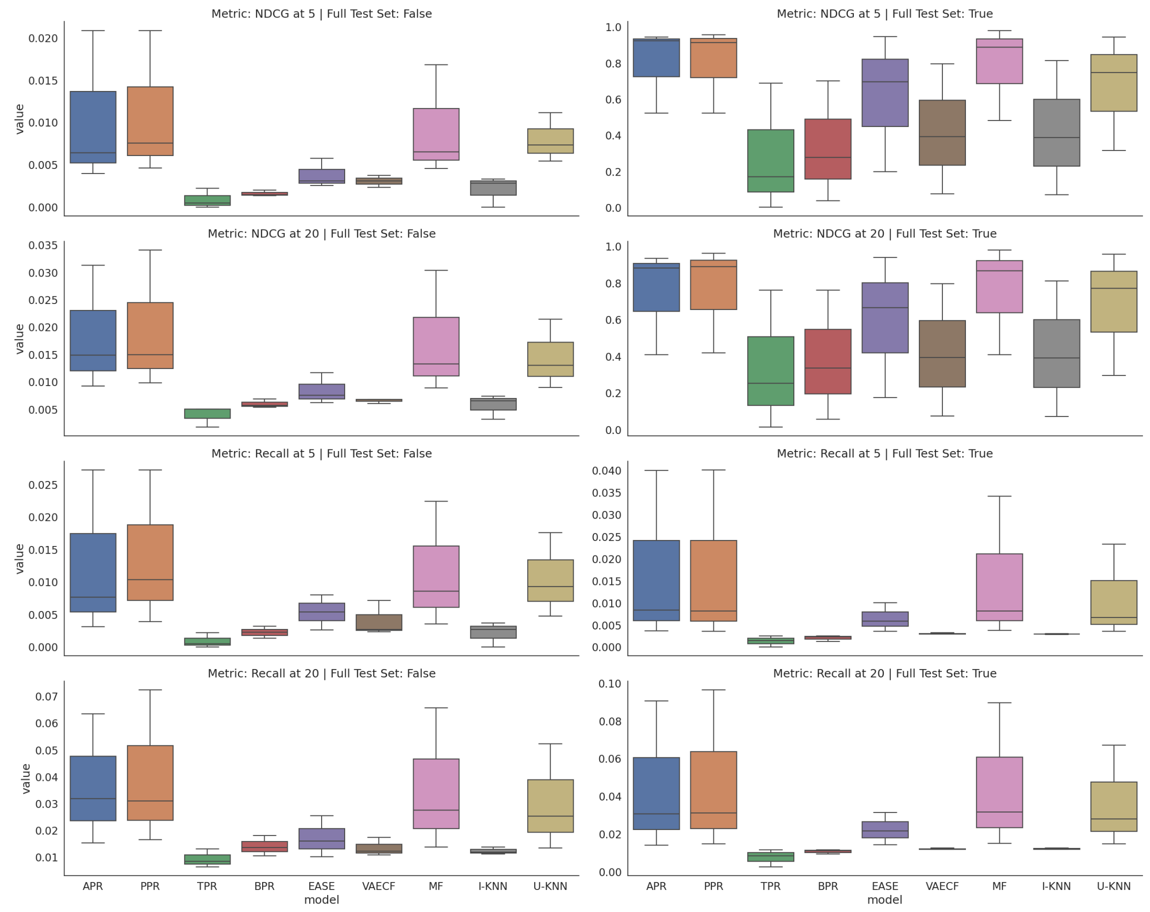


Figure 2.10: Evaluation Metrics on MF simulated dataset for the small test set (left) and the full test set (right). The metrics used for each row starting from the top are NDCG@5 (top), NDCG@20, Recall@5, and Recall@20 (bottom). APR, PPR, and MF show the strongest performance on the full test set.

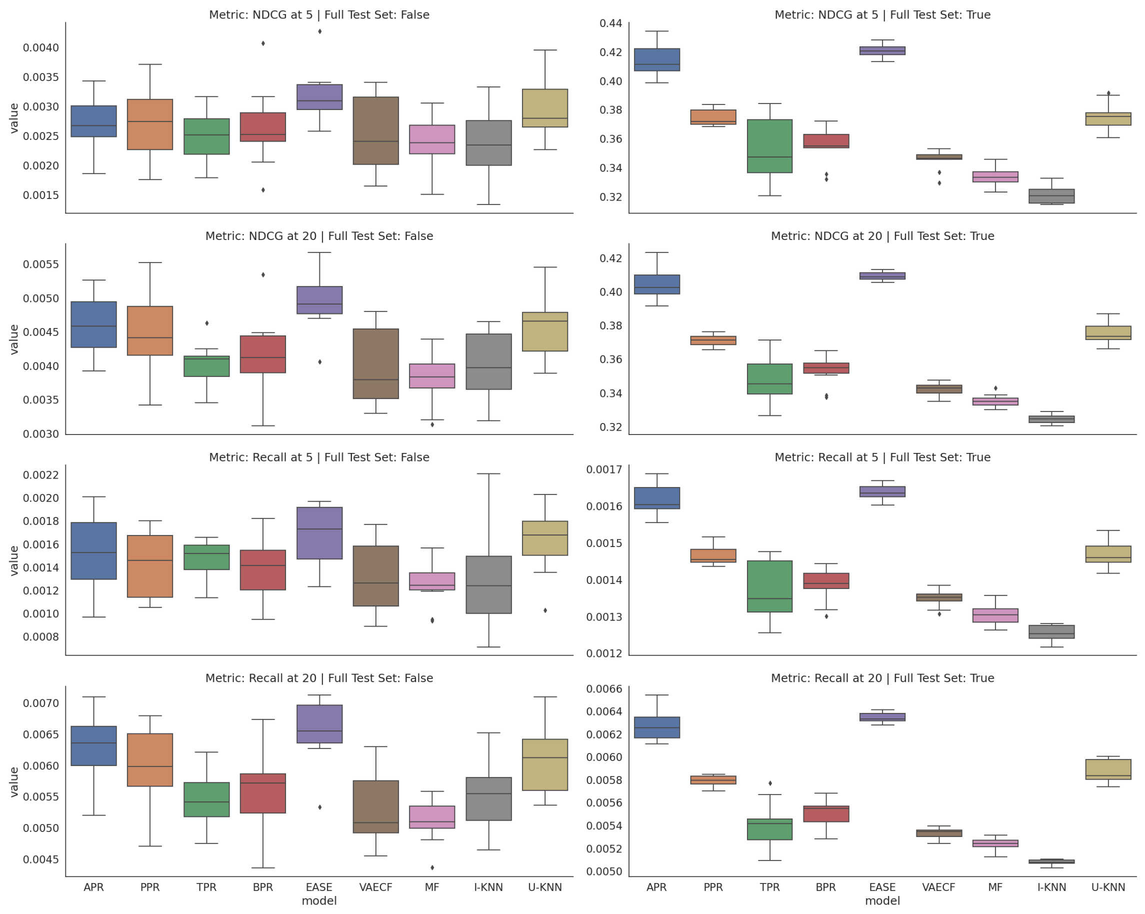


Figure 2.11: Evaluation Metrics on Hard simulated dataset for the small test set (left) and the full test set (right). The metrics used for each row starting from the top are NDCG@5 (top), NDCG@20, Recall@5, and Recall@20 (bottom). Here we see that APR and EASE are the top performing models on the full test set.

Simulation Hard

For the last simulation, the dataset is the similar to the Large simulation but we adjust the parameters to make it more challenging for our model. By lowering the mean, increasing $\frac{p_+}{p_-}$, and allowing for some rating noise, we increase the number of negative items, push the train and test distributions apart, and overall make it more challenging for our model. Even in this scenario, we see that our models, particularly the APR model performs on par with EASE.

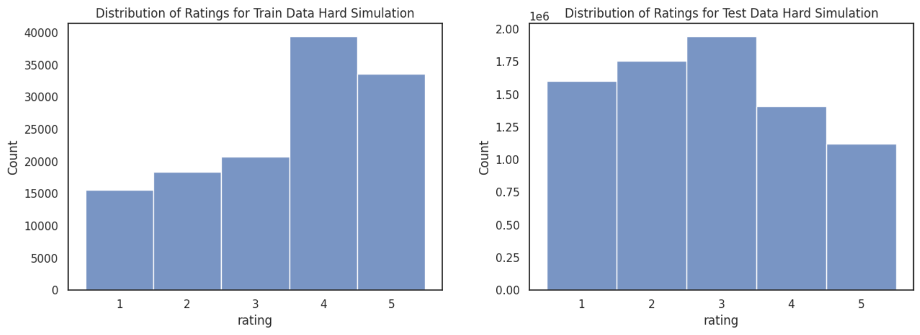


Figure 2.12: Histogram of ratings for the Hard dataset with the training set (left) and test set (right).

2.5.3 Computational Analysis

Figure 2.13 shows the training and testing time of the various models on all three simulated datasets. We see that APR and PPR take longer to train than simpler models like MF and KNN variants. In particular the MF model is written in C++ and trains very quickly. The EASE model also trains extremely quickly as it has a efficient closed form solution. However, the APR and PPR models have the best test time, which is typically more important in real world recommender systems as it directly affects the usability.

The outliers on the training time results for our models are due to the Just-In-Time compilation used by the `numba` accelerator. The first time a `numba` accelerated code is run, it will take longer as it needs to be compiled. Subsequent runs are much faster. In the test times plots we removed the much slower MF, and KNN variants to see more detail in the faster models. Again, our proposed models show extremely fast inference times.

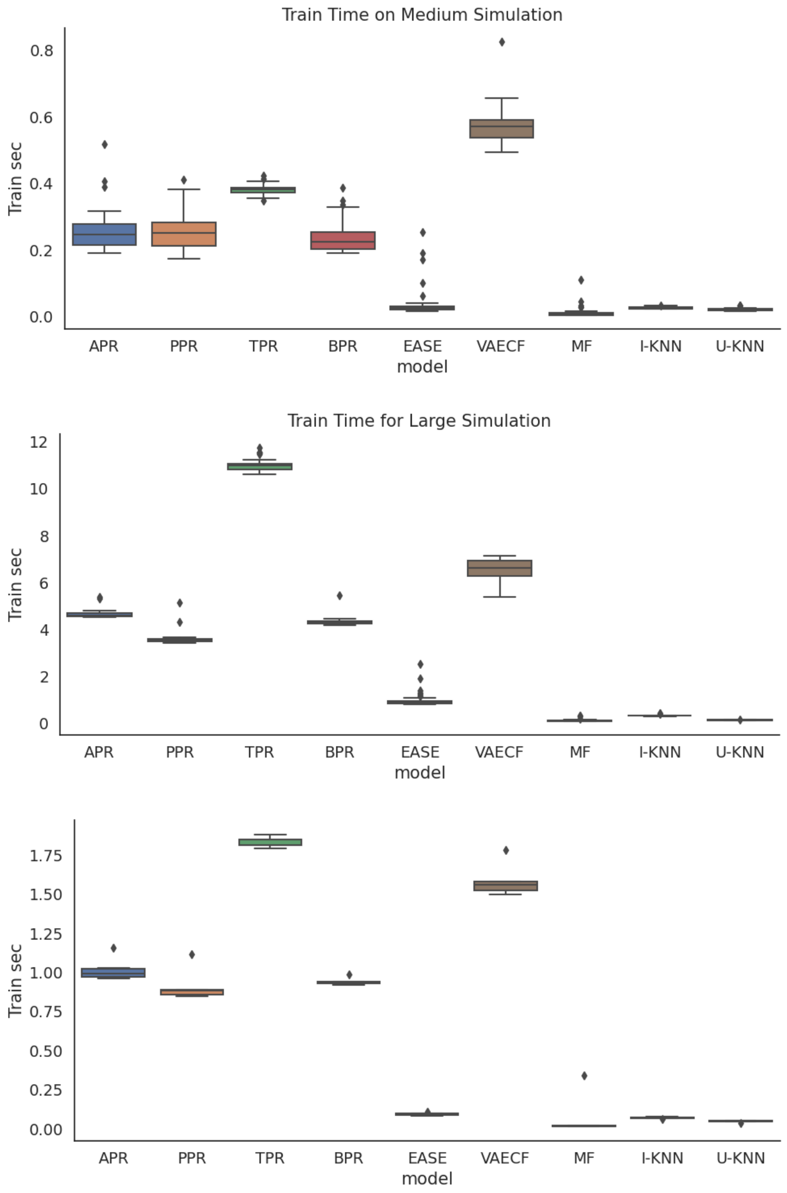


Figure 2.13: Training times for Medium (top), Large (middle), and MF (bottom) simulated datasets. We see that the APR and PPR models train faster than VAECF but slower than EASE.

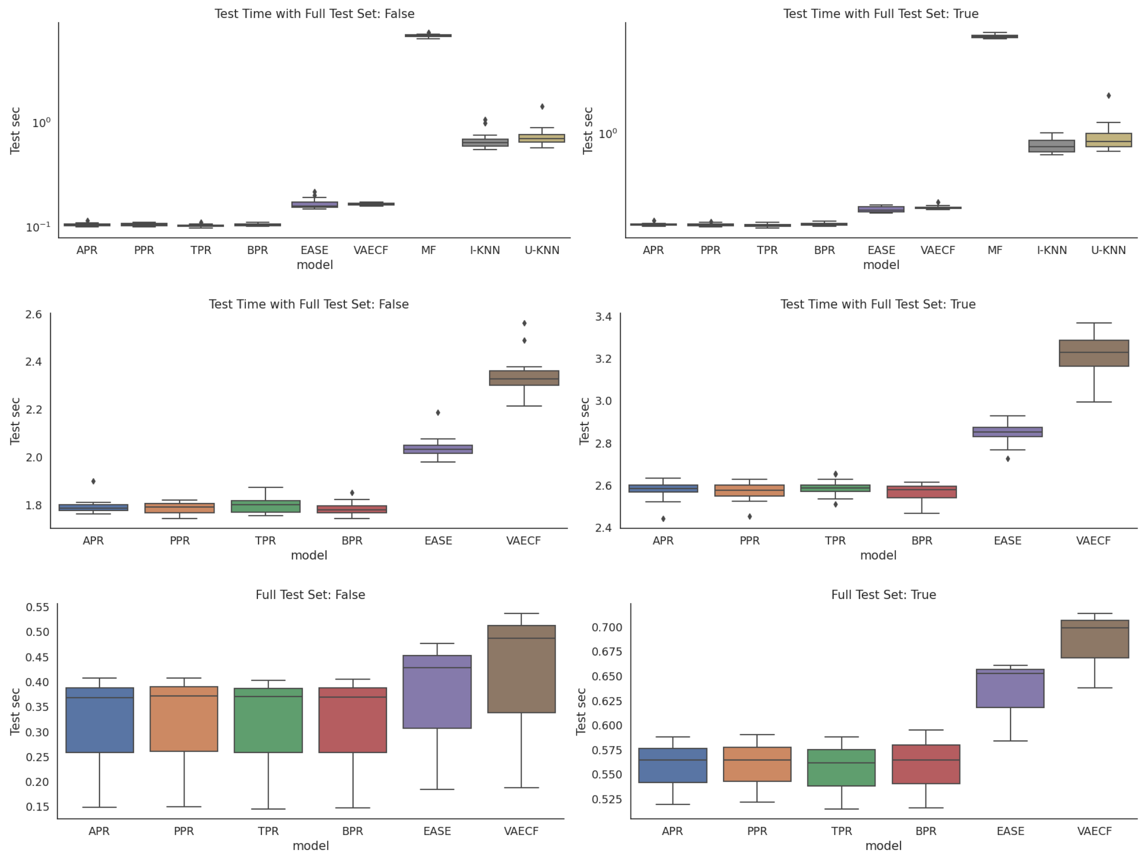


Figure 2.14: Test times for Medium (top), Large (middle), and MF (bottom) simulated datasets. We see that the APR and PPR have a higher performance for testing compared to the other models.

Table 2.2: Details for real-world datasets

dataset	n users	n items	n ratings	ratings	sparsity
MovieLens 100K	943	1682	100,000	integers 1 to 5	0.937
MovieLens 1M	6040	3706	1,000,209	integers 1 to 5	0.955

2.6 MovieLens Datasets

Experiments comparing our Pairwise Personalized Ranking model and Alternating Personalized Ranking model s to other baselines were conducted on the MovieLens 100K and MovieLens 1M dataset. The baselines we compare against are the Bayesian Personalized Ranking model (BPR), matrix factorization model, item k nearest neighbors, and user k nearest neighbors. We did try other models like Neural Collaborative Filtering and Non-Negative Matrix Factorization but found them to have much longer training times as well as poor initial performance. NDCG and Recall at both $k = 5$ and $k = 10$ were used to evaluate performance of all models.

2.6.1 Data Exploration

As mentioned in Section 1.4.1, one of the difficulties of properly evaluating top k item recommendation is that the set of items to rank for each user, S_u is not consistent between researchers. Due to the large item corpus, it is often computationally expensive to rank the all the items for each user. Researchers resorted to sampled metrics, where the m relevant items withheld from the training set are mixed with n randomly sampled unobserved items to create the set of items to be ranked, S_u. Then the models would try to rank the relevant items higher than the irrelevant items.

Our experimental set up is slightly different in that instead of relevant and irrelevant

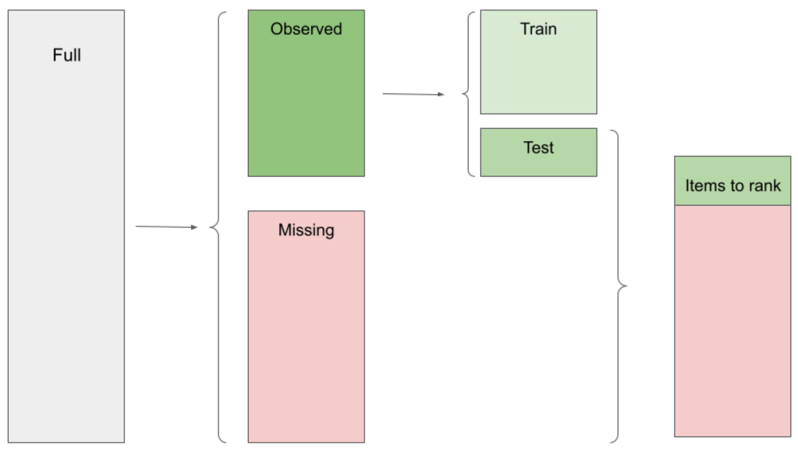


Figure 2.15: Diagram of how data is partitioned into train, test, and items to rank. The set of items to rank for each user is created by taking their test set and optionally combining with their missing items.

items, we have positive, negative, and missing items. Due to this, we use a variety of S_u to see how the performance of our model changes as more unobserved items are added. Figure 2.15 shows how our datasets are partitioned to become the training set, the test set, as well as the set of items to rank S_u.

First, the theoretical full dataset is partitioned into the set of observed ratings data, $\mathcal{X}$ and unobserved data. The observed data, $\mathcal{X}$ is then split into a train set, $\mathcal{X}_{train}$ and test set, $\mathcal{X}_{test}$ in a 80-20 ratio stratified by users. Given the cutoff criteria c_u, the training data is used to train PPR. The test data for user u, $\mathcal{X}^u_{test}$, contains the user's positive and negative items. The items in $\mathcal{X}^u_{test}$ are then optionally combined with the user's missing data to create the final set of items to rank S_u used for evaluation.

One problem is that users have different amounts of ratings and the c_u used for partition may mean that a user has a very small test set or a skewed test set. We propose 3 variations of generating S_u that range from the minimal S_u to the maximal S_u. Given each user, the "mixes" of test data and missing data are as follows:

- Maximal S_u: The entire set of the user's missing items is mixed in with the user's test items. This ranks the entire item corpus while excluding positive and negative items in the training set.
- S_u^N: If $|S_u| < N$, then $N - |S_u|$ items are randomly sample from the user's missing items and mixed with their test items such that all $|S_u| \geq N$. Values $N = 10$ and $N = 20$ are used.
- Minimal S_u: Only the user's observed test items are used for evaluation.

The four types of S_u are used to evaluate the model performance to see if there were any changes in evaluation between the varying S_u sizes as more missing items were added to the set.

2.6.2 Analysis of Cutoff Criteria

Different values of the cutoff criteria c_u will affect the performance of the PPR model as it uses c_u to determine whether an item should be considered a true positive or a true negative. Therefore, it is important to understand how different criteria affect the distribution of TP and TN items. Figure 2.16 shows how the distribution of the proportion of true positive items per user changes as the cutoff criteria changes. We see that at both $c_u = 5$ and $c_u = 3$ the mean proportion indicates that an majority of users have either all positive or all negative items, which is sub-optimal for model. On the other hand, $c_u = 4$ and c_u being the 85th quantile show a more balanced distribution of positive and negative items per user.

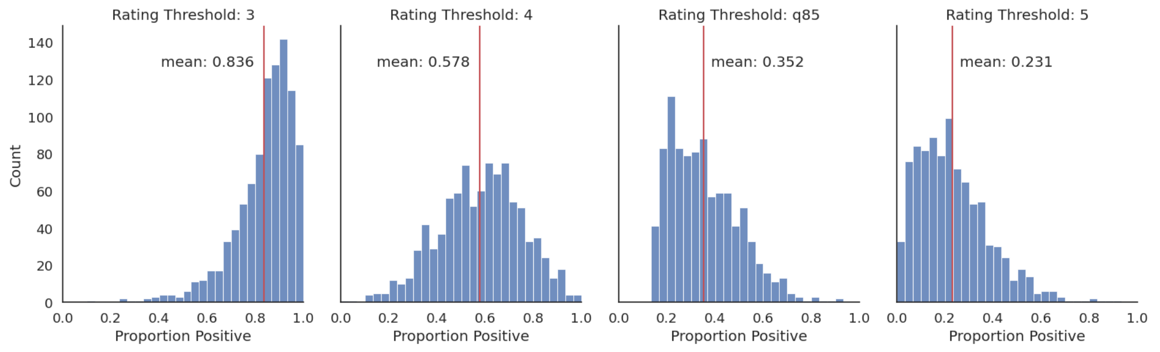
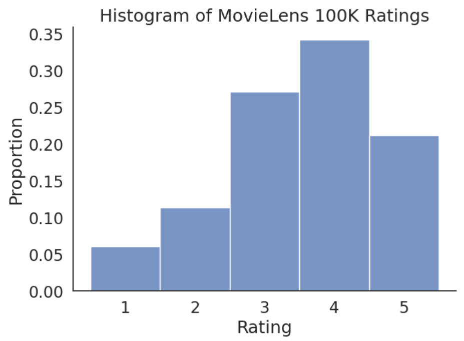
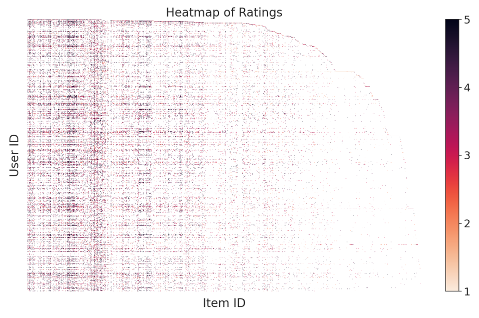


Figure 2.16: Distribution of proportion of positive items per user. A rating threshold of 4 or q85 give the most balanced distribution of postive to negative items while rating threshold of 5 gives the strictest cutoff.



(a) Histogram of ML100K ratings.



(b) Heatmap of ML100K ratings.

2.6.3 MovieLens 100K

The first of the two real world data sets used is the MovieLens 100K dataset (Harper and Konstan 2016) which contains integer ratings from 1 to 5 that users gave to various movies. All users rated at least 20 movies. In these experiments we compare the PPR and APR model against BPR, MF, ItemKNN, and UserKNN models. In particular we highlight the comparison against the BPR model as it is also a pairwise model and is most similar to our own. We include results with the EASE and VAECF models in Appendix A.2.1 where we see that outperform the pairwise model approaches on the ML100k dataset.

The results in Figure 2.18 show performance on 3 cutoff values for metrics NDCG at 5 and NDCG at 10. We see that APR and PPR outperformed all other methods using NDCG at 5 regardless of cutoff. Using NDCG at 10 we see that BPR performs best when the cutoff is 3. As seen in Section 2.16, a $c_u = 3$ results in the majority of items being considered negative, which would negatively impact our PPR model.

The general trend is that using out PPR approach is especially effective in the top few item rankings and provides the best results when the c_u used results in a balanced or positive skewed training set. Furthermore, APR which combines aspects of both PPR and BPR performs the best overall. The full table results including 85th quantile can be found in the appendix in Table A.1.

Results of other sizes of S_u

Varying the size of S_u tells a slightly different story. As S_u decreases from $N = 20$ to the minimal S_u, the number of missing items in S_u decreases to zero. Since BPR is differentiating between observed and unobserved data, it performs relatively worse

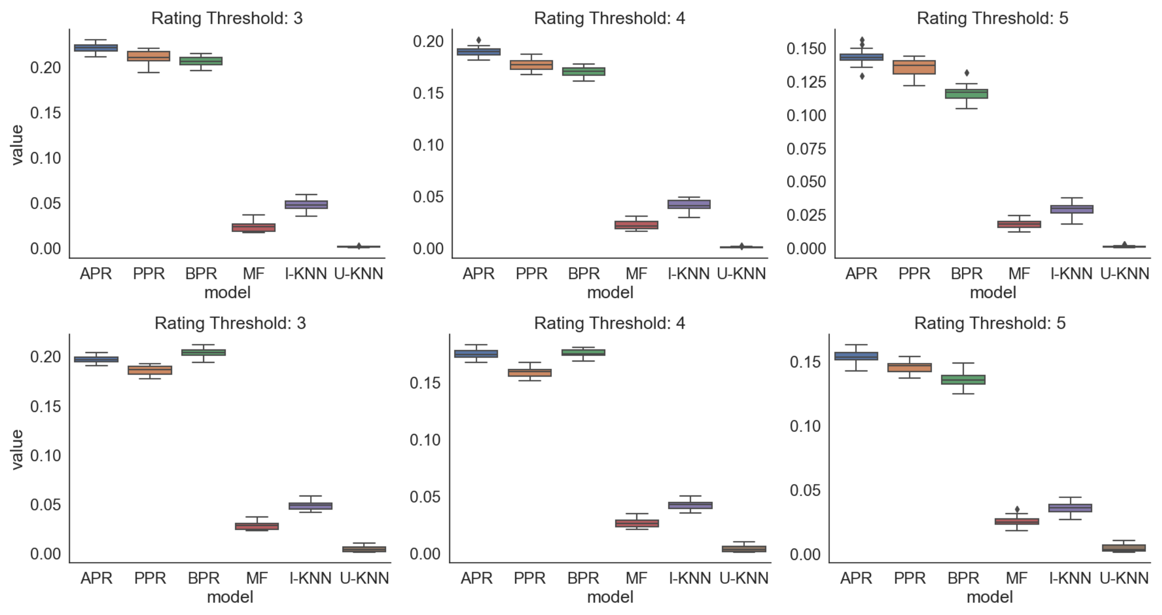


Figure 2.18: Results using NDCG at 5 (top) and NDCG at 10 (bottom) for various cutoff criteria using the Maximal S_u. APR and PPR outperform other models on most metrics.

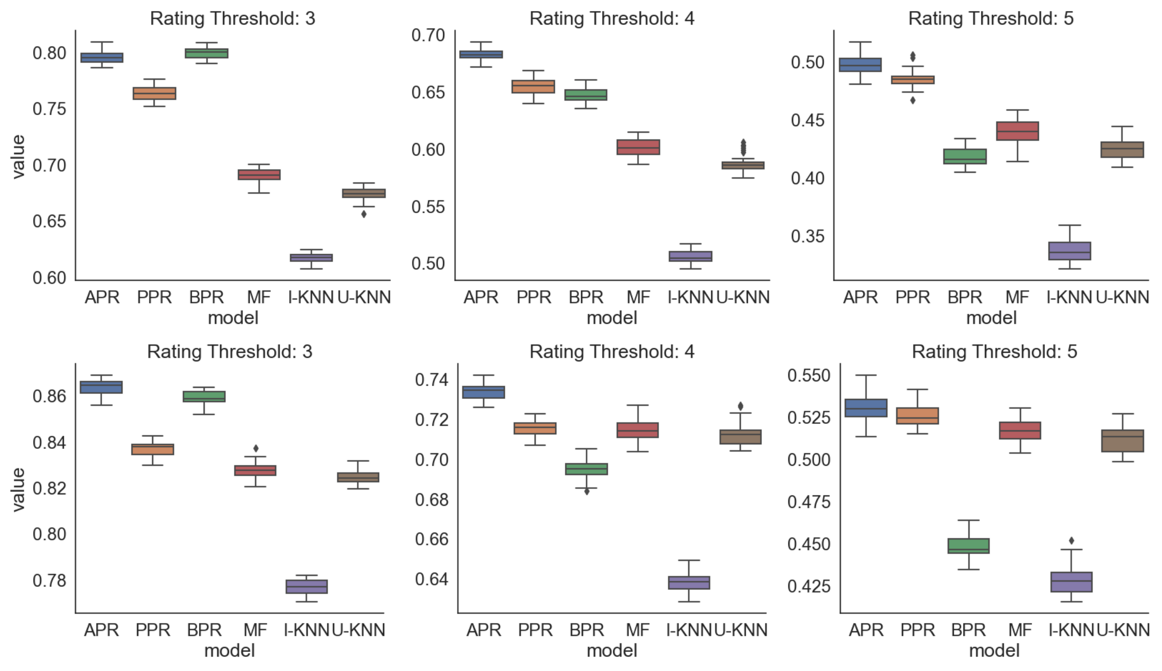


Figure 2.19: Results of NDCG at 5 for $N = 20$ (top) and for $N = 10$ (bottom).

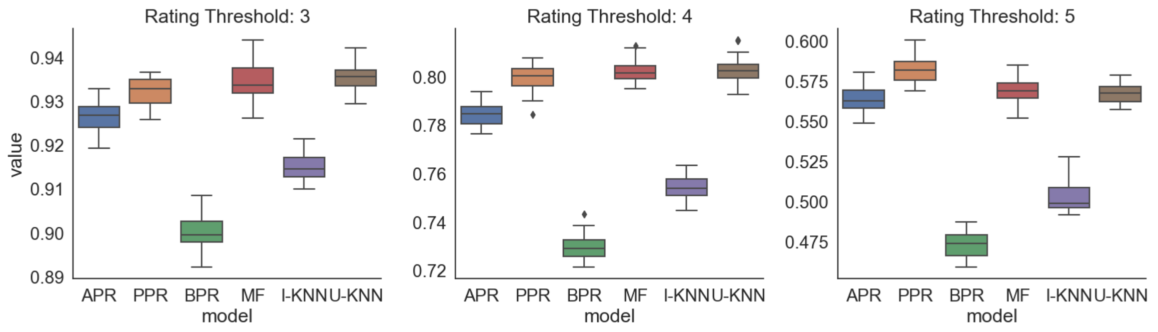


Figure 2.20: Results of NDCG at 5 for Minimal S_u.

as N decreases. On the other hand, the rating prediction methods like MF, I-KNN, and U-KNN start performing relatively better as those unobserved items get removed. This is due to the fact that those rating prediction models are able to accurately learn patterns in the observed data but cannot translate the information to the unobserved data effectively.

For the APR we continue to see very strong performance in the $N = 20$ and $N = 10$ cases but it falls off in the minimal S_u case. PPR performs just behind APR in the $N = 20$ and $N = 10$ cases but outperforms it when using the minimal S_u. Furthermore, PPR does not fall off in the minimal S_u case and performs just as well as the other methods specifically used for rating prediction. This shows the flexibility of the PPR model in using the explicit feedback to generalize patterns even in the presence of unobserved items.

Looking at the trends in Figure 2.21, we see that the pairwise methods differentiate themselves the most from the pointwise methods in the maximal S_u case. This implies that the 3 rating prediction models are not able to effectively rank the unobserved items relative to the positive items. As the size of S_u decreases, this gap between them decreases. And at the minimal S_u we see a small inversion of the original pattern. Where the rating prediction methods are shown to effectively order the observed

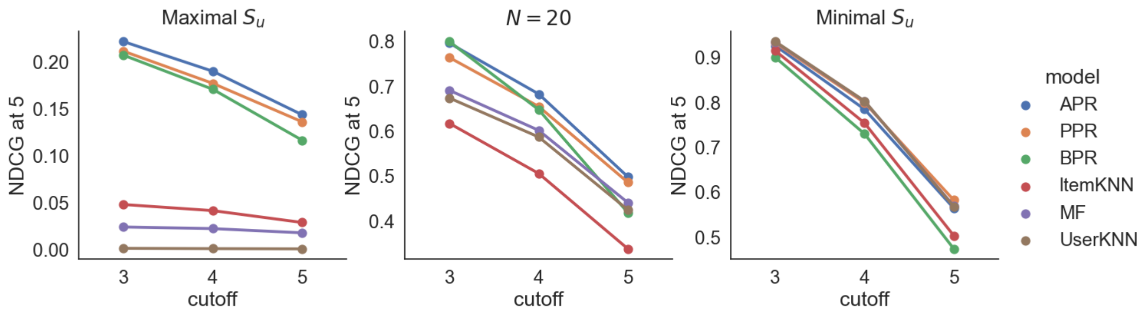


Figure 2.21: Trend of model performance across different S_u.

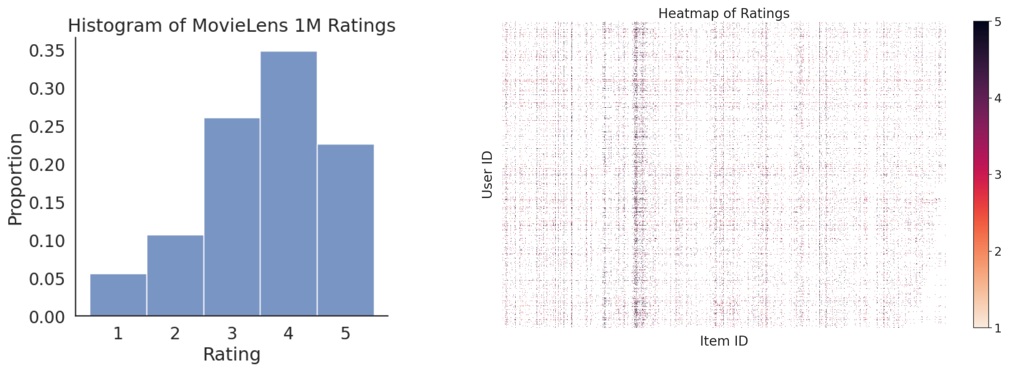


Figure 2.22: Histogram and heatmap of MovieLens 1M ratings.

positive and negative items when the unobserved items are not considered.

2.6.4 MovieLens 1M

The MovieLens 1M dataset is a larger version of the MovieLens 100K dataset. In it, each user also has at least 20 ratings. The result in is similar to that of the MovieLens 100K dataset. Both APR and PPR outperform the best on nearly all metrics with $c_u = 5$ and $c_u = Q_{85}$ with APR performing the best. PPR does fall off as c_u decreases and is generally outperformed by BPR when the $c_u = 4$ and $c_u = 3$. On the other hand, APR does not lose it's number one spot until the cutoff decreases to $c_u = 3$, where BPR then outperforms other methods.

Cutoff: 5	NDCG at 5	NDCG at 10	Recall at 5	Recall at 10
APR	0.137	0.144	0.086	0.142
PPR	0.133	0.140	0.085	0.141
BPR	0.129	0.134	0.078	0.127
ItemKNN	0.048	0.049	0.021	0.039
MF	0.016	0.022	0.010	0.025
UserKNN	0.000	0.003	0.000	0.005
Cutoff: .85 q	NDCG at 5	NDCG at 10	Recall at 5	Recall at 10
APR	0.154	0.153	0.071	0.119
PPR	0.148	0.148	0.069	0.117
OriginalBPR	0.144	0.143	0.065	0.107
MF	0.041	0.046	0.018	0.037
UserKNN	0.002	0.008	0.001	0.011
ItemKNN	0.000	0.000	0.000	0.000
Cutoff: 4	NDCG at 5	NDCG at 10	Recall at 5	Recall at 10
APR	0.182	0.172	0.051	0.088
BPR	0.185	0.172	0.051	0.085
PPR	0.171	0.163	0.049	0.085
ItemKNN	0.062	0.062	0.010	0.021
MF	0.014	0.019	0.004	0.010
UserKNN	0.000	0.003	0.000	0.002
Cutoff: 3				
BPR	0.216	0.199	0.044	0.074
APR	0.205	0.192	0.042	0.073
PPR	0.187	0.177	0.036	0.065
ItemKNN	0.059	0.062	0.006	0.014
MF	0.015	0.019	0.003	0.007
UserKNN	0.001	0.004	0.001	0.003

Table 2.3: Results on MovieLens 1M dataset for various cutoff using the Maximal S_u.

2.7 Discussions

One important finding is that PPR performs well even on in the Maximal S_u case, which includes a large number of missing items. This implies that although PPR is only trained on the observed rating data, it is able to correctly order positive items amongst missing items that it had never seen - even to performance better than than BPR which does use missing items in training. This emphasizes that fact that leveraging explicit feedback can improve model performance in the top-k item recommendation space, which typically ignores such explicit feedback and only considers implicit feedback. Furthermore, we see that using both explicit and implicit feedback provides a further boost in performance. Although the Triplet loss function we used did not lead to better results, we believe that this may be a promising area of exploration.

Chapter 3

Decentralized Recommender System

Recommender systems have witnessed significant advancements in the past decade, impacting billions of people worldwide. However, these systems often collect a vast amounts of personal data, raising concerns about privacy. To address these issues, federated methods have emerged, allowing models to be trained without sharing users' personal data with a central server. Despite these advancements, existing federated methods encounter challenges related to centralized bottlenecks and model aggregation between users.

In this study, we present a fully decentralized federated learning approach, wherein each user's model is optimized using their own data and gradients transferred from their neighboring models. This ensures that personal data remains distributed and eliminates the necessity for central server-side aggregation or model merging steps. Empirical experiments demonstrate that our approach achieves a 6.6% improvement in RMSE compared to other decentralized methods like FedAvg and Gossip Learning, across various network structures.

3.1 Introduction

Recommender systems have become indispensable for enhancing user experiences with personalized content suggestions. As recommendation systems improve through the adoption of advanced model architectures, their ability to provide accurate recommendations to users continues to grow. With the integration of deep learning into the recommendation system (Guo et al. 2017) and the utilization of increasing amounts of data, recommendation strategies have become adept at capturing intricate user behaviors, integrating rich item information, accommodating environmental or contextual dynamics (Adomavicius 2011), and ultimately culminating in a highly personalized user experience.

However, with recommender systems becoming increasingly tailored to each individual, there is a growing concern for user privacy and data security. Some concerns can be associated with the traditional approach of collecting and analyzing user data on a central server. For example, pooling ostensibly non-confidential data, such as preferences on movie watching, may reveal sensitive personal information such as one's political ideology (Narayanan and Shmatikov 2008). This privacy issue stemmed from the Netflix Prize dataset, where the users were supposed to be anonymized. However, by pooling the anonymized with public facing dataset on IMDB, the entire rating history of users could be revealed (Schneier 2007). This led to a class action lawsuit against Netflix and highlights the difficulty of maintaining proper data security, even with the right intentions.

Furthermore, aggregating large amounts of personal data can be misused by those collecting the data, such as by Cambride Analytica, or through secondary means as a data breach. In the Cambridge Analytica scandal, the British political consulting firm collected the data of millions of Facebook users with their explicit consent and used the data to create targeted political advertisements (Confessore 2018). The 2014 data breach of Yahoo, one of the largest is history, resulted in over 500 million user accounts being compromised (Perlroth 2016). The need to protect sensitive user information and comply with stringent privacy regulations (Radley-Gardner, Beale, and Zimmermann 2016) has accelerated the search and investigation of decentralized solutions. Federated Learning (FL), which aims to train a global model while keeping data local to participating devices (Konečný et al. 2017) has been one significant advancement in this regard and has gained momentum in recent years(C. Hu, Jiang, and Z. Wang 2019; Jiang and L. Hu 2020).

The conventional federated learning approach has a global model in a centralized server broadcasted to local clients, who then use their local data to update their copy of the model. Only model updates sent back to the central server for aggregation and merged into the global model. By transferring intermediate statistics (such as parameters or gradients) instead of raw user data, federated learning reduces the risk of data leakage. However, this method's reliance on the central server has raised concerns about its scalability, reliability, and potential single points of failure. This central server requires very high bandwidth to not be a bottleneck for the system (Konečný et al. 2017). And if the server experiences issues or goes offline, the entire system's performance and functionality may be compromised. Furthermore, existing literature has shown that sufficient gradient sharing may reveal the original training data (e.g., Zhu, Z. Liu, and Han 2019; Zhao, Mopuri, and Bilen 2020), which is salient in federated learning due to the central server receiving the gradients of all of the models.

Even in the federated learning space, which avoids sharing private information, there are concerns on how to effectively utilize the distributed data while mitigating associated risks. For instance, malicious adversaries may attempt to learn other users' private states and deviate from the transmission protocol by corrupting, replaying, or removing information (Bouacida and Mohapatra 2021).

To overcome the limitations of centralized structures and further improve the privacy protection of recommender systems, researchers have explored Decentralized Federated Learning (DFL) methods. Decentralized methods are characterized by the absence of a central server, enabling direct communication between clients. This approach mitigates the inherent problems of centralized servers, such as data privacy, singular point of failure, and high communication costs on the server.

In general, DFL methods can be broadly categorized into two primary approaches. The first approach draws inspiration from federated learning, where either a surrogate individual assumes a central role or each individual takes turns playing the surrogate role (Warnat-Herresthal et al. 2021). The second approach employs an aggregation and update scheme, involving the collection of information from neighboring individuals and the local update of model parameters (Hegedűs, Berta, et al. 2016). Various aggregation methods are employed in this context, including metric-based aggregation (Belal et al. 2022), as well as information extraction measured by KL divergence (C. Li, G. Li, and Varshney 2022) or through mutual information maximization (Long et al. 2022). Additionally, segmented model schemes have been introduced, involving the random exchange of these segments (C. Hu, Jiang, and Z. Wang 2019) or even the exchange of partial gradients (Jiang and L. Hu 2020).

Despite the great advancement of decentralization methods, several critical challenges are still present. First, many existing methods include an aggregation or merge step to reconcile the local model parameters into a global one (Konečný et al. 2017; C. Hu, Jiang, and Z. Wang 2019; Belal et al. 2022; Jiang and L. Hu 2020). This heuristic approach may enhance model performance and predictability for their algorithms, but it adds complexity and alters the models in an arbitrary way. Our proposed method avoids this aggregation and merging step completely allowing for a more elegant training solution.

Second, to the best of our knowledge, many existing methods require a high level of participant connectivity (Konečný et al. 2017; C. Hu, Jiang, and Z. Wang 2019), the ability to send data to any other node in the network (Jiang and L. Hu 2020), or do not provide many details about the network structure (Belal et al. 2022). In practice, however, communications among participants can be limited due to reasons such as geographical distance, internet connectivity, and user privacy preferences. In other words, model aggregation can be unrealistic, and some participants may not be able to obtain information from all other participants. In fact, it is likely that some participants may not want to share to a large number of participants.

Third, the use of model aggregation in federated learning may lead to surges in communications and data transfer. Nodes that receive a large number of participants data may need a ultra-high bandwidth in order to functions without running into any issues. Our fully decentralized approach distributes the computational requirements more evenly throughout all the nodes in the network.

In this article, we introduce the PushGrad learning (PUG) method, a novel decentralized learning method designed to tackle the previously mentioned challenges. Our proposed method trains a fully decentralized recommender system even on a limited connectivity network structure while bypassing the model merging step often seen in similar approaches. This is achieved by each node training on it's local data, and at each iteration of stochastic gradient descent, it "pushes" its current gradient with its neighbors who then also utilize that gradient step. The gradient may be re-distributed to the neighbor's neighbors as well. The decentralized approach allows our method to accommodate a wide spectrum of network structures, including both centralized and disconnected structures. Furthermore, by not requiring an aggregation step, our method removes complexity from the training which reduces the computational burden on individual nodes during the training process. Lastly, since each node is only responsible to sending model information to its neighbors (or a subset), it avoids the potential bottleneck of overloading one single central node. For example, in practice, each user may directly select which other users they want to send model information to (i.e. their trusted parties) and the target user may choose to receive or decline the model sharing. This can allow for a sparse network where users have a stronger control over their private data.

Secondly, we introduced a Generalized Singular Value Decomposition (SVD) architecture that allows for personalized design by customizing the dimensions and the number of layers to meet our expectations. Furthermore, it demonstrated its adaptability within our decentralized recommender system scheme.

In our experimental studies, we conducted two sets of investigations: one utilizing the well known MovieLens 100K dataset and the other employing the H&M Personalized Fashion Recommendations dataset, which comprises 116,733 transactions from 1,760 customers involving 8,618 fashion products. With the MovieLens 100K dataset, our proposed method is able to achieve a Root Mean Squared Error (RMSE) within 5 to 10% of the centralized model depending on the network structure. Furthermore, it outperformed both FedAvg and Gossip learning models. In the H&M experiments, the PUG model performed within 8 and 12% of the centralized model depending on the network structure and significantly outperformed the FedAvg and Gossip Learning models. These results show our model's ability to learn user-item preferences in a decentralized manner even under sparse network conditions.

The subsequent sections of this article are organized as follows. Section 3.2 provides an introduction to related papers in this field, while Section 3.3 outlines the Push-Gradmethod and network structures adopted in our method. Section 3.4 offers a description of the utilized datasets, followed by a detailed account of our experiments in Section 3.5. In Section 3.6, we discuss the conclusion and possible future directions of our work.

3.2 Literature Review

In this section, we present an overview of existing literature regarding recommender systems, and decentralized federated learning. Research in recommender systems has exploded over the past two decades. Catalyzed by the Netflix Prize in 2006 (Bennett and Lanning 2007) where Netflix released a 100 million rating dataset and offered a one million dollar grand prize to teams who could beat Netflix's current recommendation engine by a certain amount, researchers have flocked to find ways to incrementally outperform previous state-of-the-art models through a myriad of different models (Koren, Bell, and Volinsky 2009; Steffen Rendle et al. 2009), evaluation metrics (H. Li 2011), and performance optimizations (Krichene, Mayoraz, et al. 2018). These advances work together to power the highly tailored recommendation systems that we interact with every day.

The canonical recommendation model is Matrix Factorization, where latent user and item factors are embedded in the same embedding space and the similarity (such as inner product) between a user and item factor represents how much the user will

”like” the item (Koren, Bell, and Volinsky 2009). This method adeptly captures user-item preferences while only utilizing the interaction and data between users and items. Despite encountering challenges such as data sparsity and cold-start issues (Adomavicius and Tuzhilin n.d.), matrix factorization and its many varients continue to perform effectively in various scenarios (Steffen Rendle, Krichene, et al. 2020).

Neural network based recommender systems have proliferated as well. Restricted Boltzmann machines were one of the first tested on the recommender space (Salakhutdinov, Mnih, and Hinton 2007) and have been followed models incorporating Multi-Layer Perceptrons (He et al. 2017), autoencoders (Liang et al. 2018), as well as deep learning varieties (S. Zhang et al. 2019).

Outside of new models, a variety of new evaluation metrics have been developed to measure the quality of a predicted user-item interaction. Root Mean Squared Error (RMSE) is a widely used metric to evaluate the performance of recommender systems and has been used since the Netflix Prize (Bennett and Lanning 2007). It is especially common in the rating prediction space, where the goal of a model is typically to predict the the exact rating a given user would give to an item. On the other hand, for Top-k recommendation, another popular recommendation system approach, models do not necessarily try to predict the exact score a user would give an item. Instead, the k highest scoring items are assumed to be recommended to the user, and the class of those recommended items (e.g. liked, disliked) determine the quality of the recommendations. Common evaluation metrics in the top-k recommendation literature include Recall at k and Normalized Discounted Cumulative Gain at k which are truncated versions of their classification metrics. Lastly, metrics for properties like serendipity (McNee, Riedl, and Konstan 2006) or fairness (Beutel et al. 2019) have been developed.

Federated learning, a decentralized learning paradigm popularized by Google (Konečný et al. 2017; Shokri and Shmatikov 2015), facilitates collaborative model training by sharing model parameters rather than raw data, thereby encouraging privacy. Federated learning frameworks demonstrate adaptability across various model architectures and exhibit their superiority across a range of applications. In recent developments, J. Zhang et al. 2019 introduced a novel Federated Learning approach based on generative adversarial networks (GANs), enabling secure and privacy-preserving model training while enhancing model robustness against adversarial attacks. Additionally, Miao et al. 2021 proposed a reinforcement learning-based Federated Learning framework that incorporates personalized reward mechanisms. The FedFast model improves FedAvg with faster convergence through active sampling for clients each round through clustering and hierarchical model merging in those clusters Muhammad et al. 2020.

Nonetheless, federated learning approaches, constrained by the presence of a central server, may exhibit sub-optimal performance in scenarios where the central server's high communication cost acts as a bottleneck and leads to various associated issues. Consequently, an increasing amount of research is concentrated on the development of fully decentralized machine learning frameworks, facilitating direct peer-to-peer communication.

The realm of decentralized federated learning presents diverse approaches that strive to enhance collaboration while without the existence of a central server to keep better privacy and release the high communication cost over server part. One of the first fully decentralized models was a decentralized low rank matrix decomposition Hegedűs, Berta, et al. 2016. In this method, each local model takes a random walk through client nodes, and at each client node the traveling model would be updated using the local data. At the start of the next round, the model is passed to the next random client. An improvement to this was the gossip learning framework (Hegedűs, Danner, and Jelasity 2020). The gossip learning approach has nodes exchange models, but instead of simply training on the received model, the node first aggregates the models in some way before training on local data. It also utilizes an age vector to track update frequency, and this vector is used to take the weighted average of the local and peer model during merging.

One method segments deep models for updates using client data, merging parts to form a new model (C. Hu, Jiang, and Z. Wang 2019). Although efficient, it lacks a direct link to real-world networks. Another strategy involves exchanging partial gradients instead of full parameters, merging local and received gradients based on dataset sizes. These methods efficiently combine models but lack aggregation and real-world network analysis (Jiang and L. Hu 2020).

The PEPPER framework scores and groups user data, determining the weight of the merged model (Belal et al. 2022). Another avenue aligns model layers using KL divergence, leveraging insights from participant data to enhance generalization (C. Li, G. Li, and Varshney 2022). A distinct path involves decentralized collaborative learning for POI recommendation. Another framework refers to knowledge exchange among neighbors enriched model information driven by mutual information maximization (Long et al. 2022).

In the landscape of decentralized methods, aggregation has been a cornerstone for achieving model generality. In contrast, our proposed approach adopts a simpler alternative that does not utilize a model merging procedure, which could add a computational burden to client systems. Instead, we rely solely on the gradients of neighbors and use them directly to update local model parameters. Our experiments show that even this naive approach can result in competitive performance.

3.3 Methodology

3.3.1 Notation and Background

Next, we introduce the two recommender models we use to highlight our decentralized learning method - the Matrix Factorization model and our novel Generalized SVD model. Suppose we have a recommender system with m individuals and n items. Let $\boldsymbol{Y}^{m \times n}$ be the utility matrix, where y_{ij} denotes the interaction (e.g., rating or purchase quantity) of user i with item j.

The Matrix Factorization (MF) recommendation model learns to decompose the matrix $\boldsymbol{Y}$ into two latent embedding matrices with rank k: $\boldsymbol{P}^{m \times k}$, the user embeddings, and $\boldsymbol{Q}^{n \times k}$, the item embeddings. $\boldsymbol{P}$ and $\boldsymbol{Q}$ are learned so that $\boldsymbol{Y}$ can be approximated by the inner product $\boldsymbol{P}\boldsymbol{Q}^T$ (Koren, Bell, and Volinsky 2009). Given $\boldsymbol{p}_u = (p_{i1}, \ldots, p_{ik})$ and $\mathbf{q}_i = (q_{j1}, \ldots, q_{jk})$, the inner product $\boldsymbol{p}_u^T \mathbf{q}_j$ is the approximation of y_{ij}, and the model is optimized to minimize the squared distance between the predicted and actual values of y_{ij}.

The utility matrix $\boldsymbol{Y}$ is typically sparse since most users tend to interact with a small proportion of the total item set. Let $\Omega = \{(u, i) : y_{ui} \text{ is observed}\}$ be the set of observed interactions and let $|\Omega| = N$. To address overfitting when training a MF model on data that is sparse, the L_2 penalty is often used. This term is penalizes the magnitude of parameter weights by α so that no parameter will grow too large. The MF optimization function is then as follows,

$$L(\boldsymbol{P}, \boldsymbol{Q}) = \sum_{(u,i) \in \Omega} (y_{ui} - \boldsymbol{p}_u^\top \boldsymbol{q}_i)^2 + \frac{\lambda}{2}(||\boldsymbol{p}_u||_2^2 + ||\boldsymbol{q}_i||_2^2), \tag{3.1}$$

One approach to minimize the loss function is through the Alternating Least Squares algorithm (ALS) (Koren, Bell, and Volinsky 2009). By updating $\boldsymbol{P}$and $\boldsymbol{Q}$ in an alternating manner, ALS simplifies the originally non-convex problem into a sequence of quadratic problems with known optimal solutions. Another approach is stochastic gradient descent. Given the loss function, gradients for matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ are approximated using a small sample of the data and a learning rate η is applied as a step size to iteratively reduce the loss by following the stochastic gradient direction. Given appropriate hyper-parameters like the learning rate, η, and λ, the model converges noisily after a number of iterations, ultimately yielding locally optimal representations of users and items in the form of matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$.

After obtaining the optimal matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$, we can predict the value of an interaction between user i and item j by computing the inner dot product between their respective embeddings

$$\hat{y}_{ij} = \mathbf{p}_i \cdot \mathbf{q}_j. \tag{3.2}$$

Matrix Factorization has proven to be a simple yet powerful model for rating prediction (Steffen Rendle, Krichene, et al. 2020).

3.3.2 Decentralized System with Network Communication Structure

The PushGrad is a fully decentralized learning method. This means that each node maintains its local user and item embedding matrices, which serves as its dedicated local model. The local user embeddings are never shared with other nodes, but information on the item embeddings are shared through the gradients. While the node is training on its local data, it will push its gradient of the item embeddings,

$\nabla\boldsymbol{Q}$ to its neighbors. When the neighbors receives the gradient they will immediately apply it to their own item embeddings. Users can train on their data and share the gradients in rounds or in an online manner when new observational data is generated. In the first case, each user will go through Algorithm 5 once and wait till all other users have gone before going again. In the second case, Algorithm 5 will be executed on a specific node as new interaction data is generated for that node. Within each round, each node takes turns training on their local data and sharing gradients so race conditions are avoided.

Algorithm 5 PushGrad Learning

```
Require: Local data $\mathcal{Y}_u = \{(c_0, y_0), (c_1, y_1), \ldots, (c_{n_u}, y_{n_u})\}$, $c_i$ item, $y_i$ rating
  $(\boldsymbol{p}, \boldsymbol{Q}) \leftarrow$ initModel()
  loop
      wait($\Delta_g$)
      $\nabla\boldsymbol{p}, \nabla\boldsymbol{Q} \leftarrow$ gradients($\boldsymbol{p}, \boldsymbol{Q}, \mathcal{Y}_u$)
      $\boldsymbol{p} \leftarrow \boldsymbol{p} - \alpha\nabla\boldsymbol{p}$
      $\boldsymbol{Q} \leftarrow \boldsymbol{Q} - \alpha\nabla\boldsymbol{Q}$
      send $\nabla\boldsymbol{Q}$ to neighbors
  end loop

  procedure OnReceiveGradient($\nabla\boldsymbol{Q}$)
      $\boldsymbol{Q} \leftarrow \boldsymbol{Q} - \alpha\nabla\boldsymbol{Q}$
  end procedure
```

Algorithm 6 Calculate the model gradient

```
  procedure gradients($\boldsymbol{p}, \boldsymbol{Q}, \mathcal{Y}$)
      Sample batch, $\tilde{\mathcal{Y}}$, from $\mathcal{Y}$
      $\nabla\boldsymbol{p} \leftarrow \sum_{(c,y)\in\tilde{\mathcal{Y}}} -\boldsymbol{q}_c(y - \boldsymbol{p}^T\boldsymbol{q}_c) + \lambda\boldsymbol{p}$
      for $(c, y) \in \mathcal{Y}$ do
          $\nabla\boldsymbol{Q}_{c,:} \leftarrow -\boldsymbol{p}(y - \boldsymbol{p}^T\boldsymbol{Q}_{c,:}) + \lambda\boldsymbol{Q}_{c,:}$
      end for
      return $\nabla\boldsymbol{p}, \nabla\boldsymbol{Q}$
  end procedure
```

Algorithm 6 shows how the model gradients are calculated given the node's local data, $\mathcal{Y}_u$. Note that the gradient can be compressed prior to transferring to reduce the bandwidth required such as by only sharing the specific rows of $\nabla \boldsymbol{Q}$ that were updated by the local data $\mathcal{Y}$. Information about the user embeddings, $\mathbf{p}$ is never shared directly with neighbors and $\mathbf{p}$ is kept private to the node.

Several other algorithms were developed to try to speed up the decentralized training process. They primarily differed in how the users were ordered in each epoch as well as how much local data would be trained on before sharing the gradient with neighbors. We denote the different round types as Training and describe them in the algorithms below.

User Stratified Training

The User Stratified Train Type in Algorithm 7 loops through each user once per round and randomly samples a fixed size N for the gradients calculation. From our experiments, we found that this method achieved the best results.

Algorithm 7 User Stratified Train Type

```
Require: Set of all users $U = \{0, 1, \dots, m\}$, batch size $N$.
  repeat
      for all $u \in U$ do
          Execute Algorithm 5 on node $u$ with batch size, $N$
      end for
  until until convergence
```

Online Training

The Online Train Type loops through each single observation chronologically. Each individual (u, i) pair is then used for Algorithm 5. This method allows for online training, where new data points can be trained on as they arrive. However, we found that this method works best when the models are able to cycle through the data multiple times.

Algorithm 8 Online Train Type

```
Require: Set of observed data ordered chronologically, $\Omega^t = \{(u,i)_0, (u,i)_1, \ldots, (u,i)_T\}$.
  repeat
      for $j \in \{1, \ldots, T\}$ do
          Execute Algorithm 5 on node $u$ with observation $(u,i)_j$
      end for
  until until convergence
```

Proportional Training

Since each user u can have a varying amount of local data, we hypothesized whether using proportion, p, of their local data instead of a fixed batch size N would lead to better performance. This would allow users with more data to share more information to their neighbors in a single step as opposed to waiting multiple rounds. Algorithm 9 demonstrates this method of training the decentralized recommender system.

Algorithm 9 Proportional Train Type

```
Require: Set of all users $U = \{0, 1, \ldots, m\}$, proportion $p$.
  repeat
      for all $u \in U$ do
          Execute Algorithm 5 on node $u$ with batch size $N_u = p|\mathcal{Y}|$
      end for
  until until convergence
```

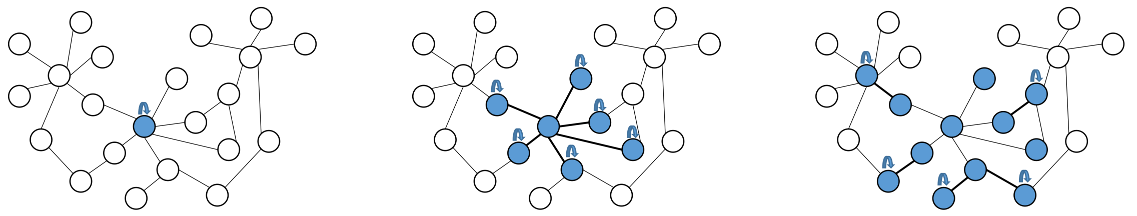
Lastly, the Random Sampling Training Type (RS) is similar to the Online/One-at-a-time Training type. However, instead of taking 1 data point at a time, it randomly samples n data points for that user. We found that the different train types had different evaluation results, particularly with respect to convergence times. We present those results in detail in Section 3.5.2.

The neighbors are determined by the underlying typically sparse networks structure denoted as $\mathbf{G}$. Each node within the network represents a user in our experiments, denoted as ν. The edges, symbolized as ε, establish direct signal connections, and serve as paths for information transfer between nodes. In our scenario, these edge connections can facilitate the transfer of gradients between users.

Additionally, we introduce the concept of order-level for user connections within the network. When two users communicate directly without any intermediary, we classify this relationship as a *first-order* neighbor. Conversely, if their connection necessitates transfer through an intermediary user, we classify it as a *second-order* neighbor, and so on.

In the decentralized framework, each user independently manages their item embeddings. For user i, their j-th item embedding is represented as $\mathbf{q}_{ij}$. When employing our order selection method to select a user i, the algorithm identifies and establishes connections with neighboring users within the network, following the network structure. Initially, our method identifies the immediate neighbors, referred to as 'first-order' users, surrounding user i. Then, by selecting 'second-order' users, we extend our search iteratively to higher-order users until reaching a predefined order limit. Users selected within a given order are denoted as ξ, facilitating the transfer of item embedding gradients, represented as $\nabla \mathbf{q} ij$. Neighboring user κ utilizes the received gradients to perform gradient descent, updating their respective item embeddings $\mathbf{q}_{\kappa j}$. This process iterates until the model converges, processing all users or observations, depending on whether learning occurs in rounds or online.

The proposed algorithm is applicable to any arbitrary network structure due to the



(a) Local update (b) First-order transfer (c) Second-order transfer

Figure 3.1: Decentralized scheme for recommender systems.

information transfer scheme as we described above. In other words, we do not need all individuals to be adjacent to each other by the direct edge of the network, nor do we need certain "hub" individuals to play a surrogate role. Information exclusively circulates among our chosen neighbors of the specific user. This helps overcome technical difficulties when certain individuals do not have good network connectivity as other individuals do.

Our algorithm removes the necessity for a central server and model aggregation. In a federated learning setup, users typically compute gradients $\nabla \mathbf{q}_{ij}$ or parameters $\mathbf{q}_{ij}$ locally. These are then uploaded to a central server, which then aggregates gradients $\frac{1}{|\Omega|}\sum_{(i,j)\in|\Omega|}\nabla \mathbf{q}_{ij}$ to update the global model or directly synchronizes the model parameters $\frac{1}{|\Omega|}\sum_{(i,j)\in|\Omega|}\mathbf{q}_{ij}$. In the Gossip Learning framework, nodes request and merge model parameters from other nodes and may require two copies of the model - one local and one global.

In contrast, our proposed method establishes a fully peer-to-peer framework, eliminating the requirement for a central server and potential bottleneck issues associated with it as well as the overhead of aggregating neighbor model parameters directly.

3.3.3 Generalized SVD

Our proposed Generalized SVD method is an extended architectural model based on Matrix Factorization where each embedding now has their own sequence of layers. This means that now $\boldsymbol{p}_u$ and $\boldsymbol{q}_i$ will be denoted as $\boldsymbol{p}_u^0$ and $\boldsymbol{q}_i^0$ and we employ two distinct linear layers with potential activation function f to transform $\mathbf{p}_i^0$ and $\mathbf{q}_j^0$ into $\mathbf{p}_i^1$ and $\mathbf{q}_j^1$, respectively. The weight parameters for a given user factor at a given layer k is denoted as W_u^k, and the corresponding item weight is V_i^k with activation function g.

$$\boldsymbol{p}_u^{k+1} = f(\boldsymbol{W}_u^k \boldsymbol{p}_u^k) \tag{3.3}$$

$$\boldsymbol{q}_i^{k+1} = g(\boldsymbol{V}_i^k \boldsymbol{q}_i^k). \tag{3.4}$$

This process is repeated for n layers with an optional activation function f and g in-between the layers until we obtain the final user representation, $\mathbf{p}_i^n$, and item representation, $\mathbf{q}_j^n$. Finally, we apply the dot product to these two final representations to make an estimated prediction

$$y_{ij} = \mathbf{p}_i^n \cdot \mathbf{q}_j^n. \tag{3.5}$$

This Generalized SVD still has the property that each user and item parameters are not mixed with other user and item parameters.

The Decentralized Generalized SVD architecture adheres to the same principles as the Matrix Factorization, which itself is based off of the SVD method. The key distinction being that the gradients are propagated not only from the embedding layer but also through subsequent representational linear layers.

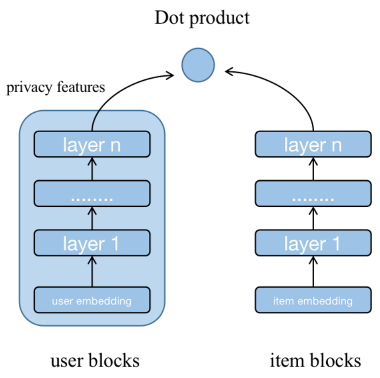


Figure 3.2: Generalized SVD architecture.

When new observations are generated, the model architecture obtains the final user representation, $\mathbf{p}_i^n$, and the specific item representation for this user, $\mathbf{q}_{ij}^n$. Backpropagation is then performed after generating the final prediction. Mathematically, the gradients are calculated as follows

$$\begin{aligned}\nabla\mathbf{p}_i^n &= -\mathbf{q}_{ij}^n(Y_{ij} - \mathbf{p}_i^n\mathbf{q}_ij^n) + 2\lambda\mathbf{p}_i^n, \qquad (3.6)\\ \nabla\mathbf{q}_{ij}^n &= -\mathbf{p}_i^n(Y_{ij} - \mathbf{p}_i^n\mathbf{q}_{ij}^n) + 2\lambda\mathbf{q}_{ij}^n.\end{aligned}$$

Following backpropagation, gradients for each representation layer are obtained, including $\nabla\mathbf{p}_i^0$,$\nabla\mathbf{p}_i^1$,...,$\nabla\mathbf{p}_i^{n-1}$ and $\nabla\mathbf{q}_{ij}^0$,$\nabla\mathbf{q}_{ij}^1$,...,$\nabla\mathbf{q}_{ij}^{n-1}$. Subsequently, a gradient descent method is applied to update the local model parameters. To ensure user privacy, only the gradients $\nabla\mathbf{q}_{ij}^0$, $\nabla\mathbf{q}_{ij}^1$,...,$\nabla\mathbf{q}_{ij}^n$ are shared with neighboring users, while $\nabla\mathbf{p}_i^0$,$\nabla\mathbf{p}_i^1$,...,$\nabla\mathbf{p}_i^n$ are kept confidential for a specific user. Ultimately, neighbors receive the gradients from one another to update their local model parameters layer by layer

$$\begin{aligned}\mathbf{q}_{\kappa j}^{n} &= \mathbf{q}_{\kappa j}^{n} - \mathrm{lr} \cdot \nabla \mathbf{q}_{ij}^{n} \\ \mathbf{q}_{\kappa j}^{n-1} &= \mathbf{q}_{\kappa j}^{n-1} - \mathrm{lr} \cdot \nabla \mathbf{q}_{ij}^{n-1} \\ &\cdots \\ \mathbf{q}_{\kappa j}^{0} &= \mathbf{q}_{\kappa j}^{0} - \mathrm{lr} \cdot \nabla \mathbf{q}_{ij}^{0}.\end{aligned} \tag{3.7}$$

3.3.4 Order selection

In our investigation, we examine not only the direct exchange of information among users but also the influence of their connected neighbors, encompassing various orders of connectivity within the network. Selection of a lower order expedites information propagation across the network, thereby accelerating model training. However, such choices may introduce challenges such as non-convergence and decreased accuracy, despite the advantage of reduced communication overhead among users. Conversely, in a scenario where all users are extensively interconnected and the order of connectivity is sufficiently high, our proposed methodology demonstrates performance more akin to centralized learning, effectively emulating data aggregation at a central server. Our primary aim is to leverage network information to improve model performance while minimizing communication costs within the selected order. This necessitates active engagement with a diverse array of users to gather information from neighboring nodes via gradient-based mechanisms.

To determine the optimal number of selected orders and the corresponding quantity of edges that can be incorporated into the network, we introduce an order selection method aimed at identifying the optimal number of orders, denoted as α_{opt} for efficient gradient information transfer. Unlike traditional cross-validation techniques, our proposed order selection method offers an efficient approach to circumvent computationally intensive computation. In our research, we have observed that as the number of orders α increases , the number of connected users also increases, but this growth occurs at varying rates. Furthermore, there is a limit on how many connected users are possible. In the undirected case, the limit is equal to the number of edges in a fully connected graph, but in the directed case, the limit depends on the graph structure and may stop increasing well before the fully connected value is reached. The derivative (forward difference) can tell us the rate of change of the number of connected users.

We identify a specific inflection point where the second derivative of the number of edges reaches its maximum value. This observation suggests that at this critical point, the network's performance can surpass that of other competitive methods, while still managing to strike a balance between model performance and communication overhead. This pivotal point serves as the foundation for selecting the number of orders α_{opt} in our method for subsequent research. Through this approach, we effectively harness data transferring between networks while mitigating issues stemming from excessive communication costs.

For example, when selecting order in the undirected random 5-out graph as shown in Figure 3.3 below, we refer to the maximum value of the second derivative and choose order = 3 as the applied value. Our subsequent experiments demonstrated that, although the number of selected orders may be only half of the total number of orders, the performance in the dataset is superior to other competing methods.

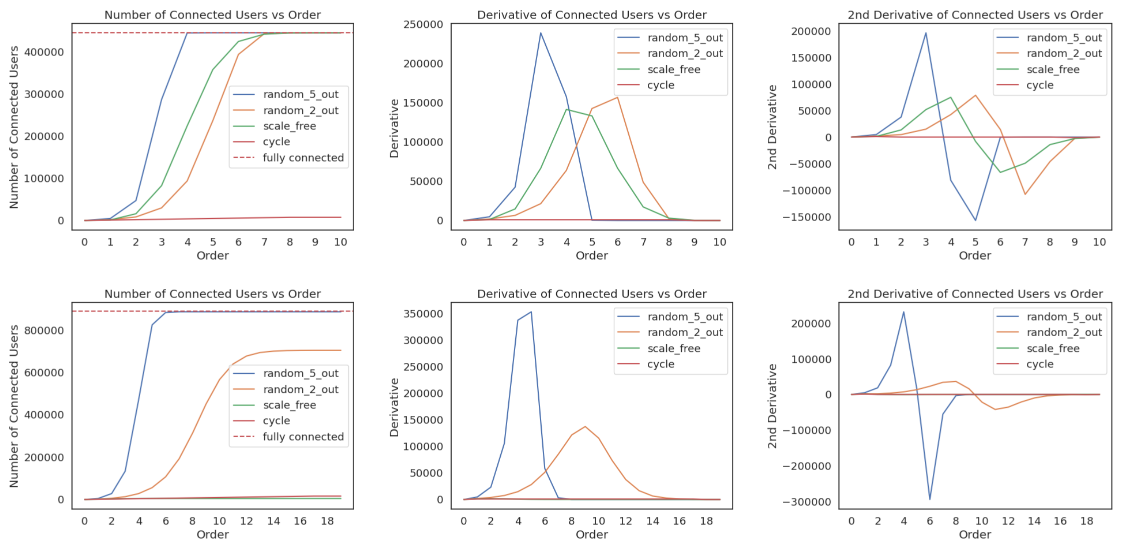


Figure 3.3: Order selection for undirected graph types (top) and directed graph types (bottom).

3.4 Data Description

In our experiments, we use two datasets: the MovieLens 100K dataset and the H&M Personalized Fashion Recommendations dataset. The MovieLens dataset contains movie ratings and is a widely used benchmark in recommender system studies. The H&M dataset, on the other hand, is a recent addition, consisting of user interactions representing purchases of fashion products.

3.4.1 MovieLens Dataset

The MovieLens 100K dataset contains a comprehensive collection of user ratings covering a diverse range of movies. Specifically, it captures users' explicit feedback on specific movies. Each rating entry features a timestamp, pinpointing the moment of user-movie interaction and the corresponding rating. This dataset enables researchers to quantitatively evaluate recommendation algorithm performance by comparing pre-

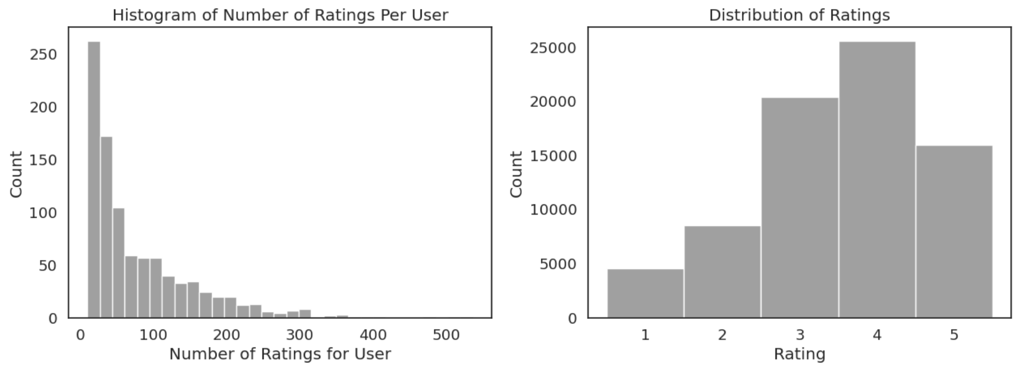


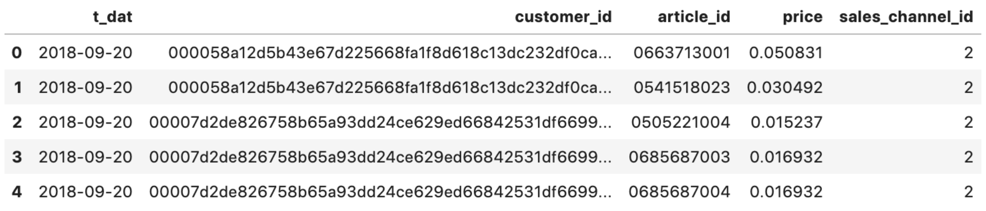
Figure 3.4: Distribution of MovieLens 100K data.

dicted ratings with actual user feedback. This dataset includes a total of 100,000 ratings ranging from 1 to 5, contributed by 943 users across 1,682 movies. Notably, each user has rated a minimum of 20 movies. The distribution of these ratings is visualized below.

The data is split 75-25 into a train and test set. We excluded users and movies from the test dataset if they were not present in the training dataset to address cold start issues. Consequently, the training dataset comprised a total of 75,000 entries, while the testing dataset included 24,929 entries. In terms of user and movie counts, there were 943 unique users and 1,628 distinct movies in the dataset. The training set was further split 80-20 into a training and validation set, which is used for early stopping during training.

3.4.2 H&M Personalized Fashion Recommendations Dataset

This dataset is provided by the H&M Group for a Kaggle competition available on the website. It contains article and customer information, transaction history, and corresponding item images.



	t_dat	customer_id	article_id	price	sales_channel_id
0	2018-09-20	000058a12d5b43e67d225668fa1f8d618c13dc232df0ca...	0663713001	0.050831	2
1	2018-09-20	000058a12d5b43e67d225668fa1f8d618c13dc232df0ca...	0541518023	0.030492	2
2	2018-09-20	00007d2de826758b65a93dd24ce629ed66842531df6699...	0505221004	0.015237	2
3	2018-09-20	00007d2de826758b65a93dd24ce629ed66842531df6699...	0685687003	0.016932	2
4	2018-09-20	00007d2de826758b65a93dd24ce629ed66842531df6699...	0685687004	0.016932	2

Figure 3.5: Initial five rows of Transaction data.

The article data consists of 25 columns, each representing various attributes of the articles. The `article_id` column contains a total of 105,542 unique values, each corresponding to an article with available images. Additionally, the `product_code` serves as a unique 9-digit identifier, acting as a high-level product code for each article. Other columns, such as department, type, and more, provide comprehensive information about each article. with corresponding images available for each product.

The customer dataframe contains a total of 1,371,980 unique customers and includes attributes such as `postal_code`, `age`, and more.

The transaction data spans from September 20th, 2018, to September 22nd, 2020, and comprises 5 columns documenting the purchase history. This dataframe contains 31,788,324 rows, with each row representing a customer's purchase of a specific article at a specified price through a particular sales channel. Below are the first five rows of this dataframe.

Our analysis focused on the top 5,000 locations where customers were situated, specifically targeting customers whose purchases of certain product types exceeded 120. We selected the six most popular categories for our analysis: 'Blouse', 'Top', 'T-shirt', 'Sweater', 'Dress', and 'Trousers'. The target variable was the frequency of a user's purchase of a specific product.

To create training and testing datasets, we divided the data based on the number of weeks leading up to the most recent date. The data from the most recent 50 weeks were designated as the testing dataset, while the remaining weeks formed the training dataset. Similar to our approach with the MovieLens 100K data, we addressed cold start issues by excluding items not present in the training dataset from the testing dataset. Consequently, the training dataset consisted of 86,480 transactions, while the testing dataset comprised 30,253 transactions, representing approximately 26% of the entire dataset. Overall, the dataset included 1,760 unique customers and 8,618 distinct products.

3.5 Experiments

In our experiments we compare a variety of federated learning models as well as the centralized model to our decentralized method on a variety of network structures. Since we are primarily concerned with the evaluation performance of our method, we assume a churn-free network scenario, i.e. all nodes are available and without drop-out. The decentralized training occurs in rounds, where each user executes Algorithm 5 one time per round. Lastly, we do not explicitly account to bandwidth connectivity or networking delays.

3.5.1 Network Structure

We conduct experiments on a variety of network structures including sparse networks and compare the performance of our decentralized method to related methods. We utilize the Python library `networkx` (Hagberg, Schult, and Swart 2008) to work with the network structures in this paper. Although we primarily focus our work on

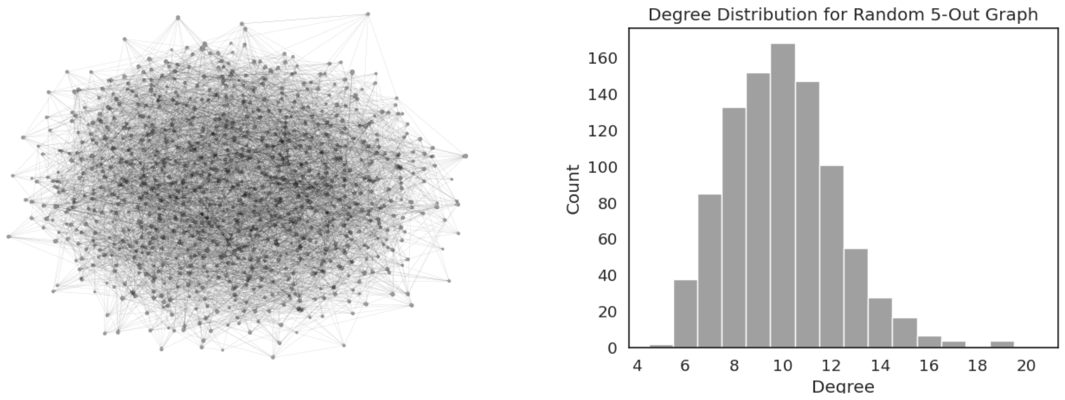


Figure 3.6: Random 5-out graph and degree distribution.

undirected graphs, we include experimental results on directed graphs as well.

Random K-Out Graph

The random k-out graph is a type of directed graph where each node has a k out degree and an unconstrained in-degree. Given a source node n_i, the simplest implementation has the target nodes n_j selected uniformly at random. We utilize a weighted target node selection that includes a slight preferential attachment. This means that nodes with more in-degrees will have a higher probability of becoming a target node. In our experiments we use both a random 5-out graph and a random 2-out graph. To create the undirected version of the random k-out graph, we first create the directed version, and then convert all the directed edges to become undirected edges.

Scale-Free Graph

Scale-free graphs exhibit a characteristic where a small subset of nodes has significantly higher connectivity compared to the remaining nodes, which exhibit lower connectivity. In this network, there are three controlled parameters. The first pa-

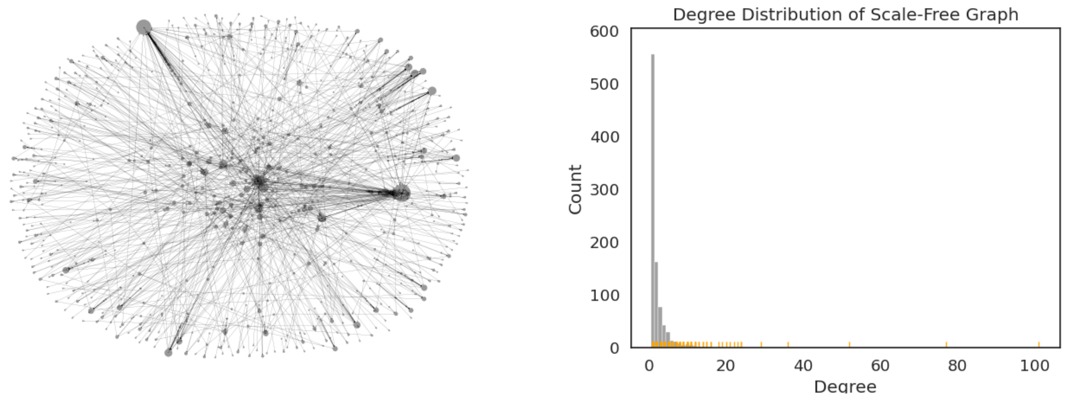


Figure 3.7: Scale-free graph and degree distribution

rameter, alpha, represents the probability of adding a new node that is connected to an existing node chosen randomly based on the in-degree distribution. The second parameter, beta, controls the probability of adding an edge between two existing nodes. The last parameter, gamma, determines the probability of adding a new node connected to an existing node chosen randomly based on the out-degree distribution. We use values of $\alpha = 0.5$, $\beta = 0.25$ and $\gamma = 0.25$. Similar to the random k-out graph, the undirected version of the scale-free graph is created by converting the all the edges to be undirected.

Cycle graph

The cycle graph depicts a configuration of cyclically connected users, facilitating sequential connections among them. The corresponding figures below show the cycle graph used in experiments as well as a simple cycle graph for reference. We do not use the optimal order selection for the cycle graph as it's specialized structure forces the number of increased user connections to grow linearly as the order increases. The undirected cycle graph has connections to both neighbors.

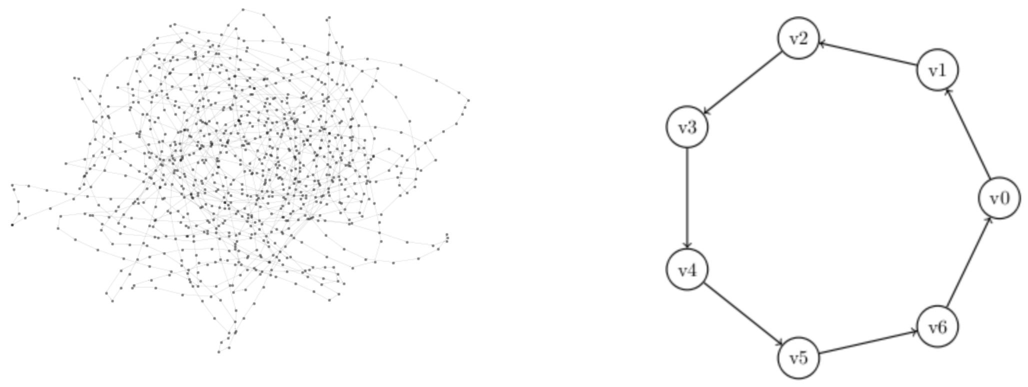


Figure 3.8: Cycle graph and simple example

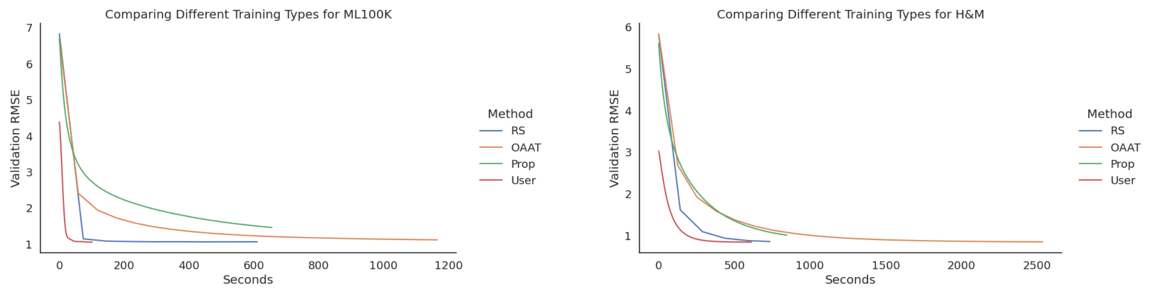


Figure 3.9: Comparison of Training Types for ML100K (left) and H&M (right).

3.5.2 Comparing Train Types

Next, we show the results comparing the various train types in Algorithm 7, 8, and 9. To deduce which training methodology was the most efficient we conducted various experiments involving hyper-parameter search on the train types using the random 5-out network structure. The RS method is similar to the OAAT method but instead of taking a single data point, it randomly samples N data points to use for training.

The results of these experiments in Figure 3.9 show that although most of the methods eventually achieve a comparable performance, the User Stratified (User) method achieves it at the fastest rate, closely followed by the RS method. For the experiments in the next section, we use the User Stratified training method with a batch size of

10.

3.5.3 Performance Evaluation

In this section, we compare the proposed method with four competing decentralization schemes. For the proposed method and each of the four schemes, we use matrix factorization as the underlying recommender system. Below, we provide detailed descriptions of each scheme. For our decentralized method, we use 3 factors for each user's model.

Central Learning (CL)

Central learning assumes that the central server has all individuals' data. The implementation of the central learning is therefore independent of the network structure. Whenever a new interaction occurs, the updated item embedding is shared with all users, ensuring that all users have access to the latest information. The CL matrix factorization model uses 10 factors for the user and item embeddings. It serves as the best possible performance a decentralized model could reach.

Federated Averaging (FL)

Federated Avgeraging is a canonical federated learning scheme (McMahan et al. 2017), where the central server collects and aggregates model information from clients, updates the global model parameters, and shares the updated parameters back to clients. When combining the client models into the global model, a simple average is used such that

$$\theta_{t+1} = \frac{1}{S_t} \sum_{k \in S_t} \theta^k_{t+1}, \tag{3.8}$$

where θ_{t+1} is the updated global model, θ^k are the local models, and S_t is the set of randomly chosen clients for that round.

Using the notation of Federated Averaging, C is the fraction of clients per round, E is the number of local epochs trained on by the clients selected each round, and B is the local minibatch size used during the local training. We use $C = \frac{500}{\text{n\_users}}$, $E = 1$, and $B = 10$. The model used is matrix factorization model with 3 factors. In each round the average of model parameters from these users and utilize this average to update the model parameters across all users. To imitate the real-world scenario, we assume that only a portion of the individuals respond to the central server's quest at one time. In our experiments we set this number to be 500, which gave the best results. We found that using a small value for C resulted in extremely slow convergence.

Gossip Learning (GL)

In the gossip learning approach (Hegedűs, Danner, and Jelasity 2020), a node uses its local data to update the model. Then it sends a potentially compressed copy of its model to a random peer in the network. Once a node receives a neighbor's model, it merges it with its local model. Various merging approaches are possible. The simplest approach is to use completely replace the local model with the neighbor's model. This is the the random walk approach seen in (Hegedűs, Berta, et al. 2016). Another merging approach is to take the average of both sets of model parameters, where an age vector is used to weight parameters that have been updated more frequently. We use the weighted averaging approach, but restrict the peer sampling

Table 3.1: PushGrad results on the MovieLens dataset (top) and the H&M dataset (bottom) over various undirected networks compared to 4 other methods.

ML100K | Graph	Max Order	Opt Order	1st Order	CL	FL	GL	LL
Random 2-Out	1.026	1.036	1.069	0.930	1.110	1.120	1.141
Random 5-Out	1.026	1.037	1.054	0.930	1.110	1.113	1.141
Scale-Free	1.026	1.039	1.104	0.930	1.110	1.114	1.141
Cycle	1.026	—	1.073	0.930	1.110	1.230	1.141
H&M | Graph	**Max Order**	**Opt Order**	**1st Order**	**CL**	**FL**	**GL**	**LL**
Random 2-Out	1.030	1.034	1.036	0.999	1.087	1.158	1.129
Random 5-Out	1.030	1.044	1.039	0.999	1.087	1.178	1.129
Scale-Free	1.030	1.039	1.054	0.999	1.087	1.194	1.129
Cycle	1.030	—	1.039	0.999	1.087	1.353	1.129

mechanism to the same network as our method.

Local Learning (LL)

In this approach, we assume that communication between nodes is not allowed. Each user's model is trained on their local data only.

We compare the performance of each method by evaluating the root mean square error (RMSE), on the test set created from before. The RMSE provides a quantitative measure of the average squared difference between the predicted and actual values. Given Ω' is the set of user item pairs (i,j) to be evaluated over, the RMSE is

$$\text{RMSE} = \sqrt{\frac{1}{|\Omega'|}\sum_{(i,j)\in\Omega'}(y_{ij}-\hat{y}_{ij})^2}. \tag{3.9}$$

Among the competing methods assessed using the MovieLens 100K and H&M personalization datasets, Central learning exhibited the highest performance as expected due to global data access. Central learning achieves this by assuming that all individuals' data are stored at the central server, thereby ignoring the network structure.

Table 3.2: PushGrad results on the MovieLens dataset over various directed networks compared to 3 other methods.

Graph	Max Order	2nd Order	1st Order	CL	FL	GL
Random 2-Out	1.161	1.088	1.089	0.930	1.154	1.196
Random 5-Out	1.037	1.060	1.064	0.930	1.154	1.123
Scale-Free	1.110	1.138	1.159	0.930	1.154	1.502
Cycle	-	-	1.101	0.930	1.154	1.201

In contrast, gossip learning demonstrated the weakest performance among all the methods. In both MovieLens and H&M, the FL method outperformed GL by a small margin.

The poor performance of GL may be due to the fact that the network structure is quite sparse. The original GL algorithm has each node sampling peer from a larger number of neighbors but in these experiments we restricted it to be just the direct neighbors of the nodes which was often in the single digits. On the other hand, for the FL method, we used a larger client sample size each epoch as using a smaller size led to poor performance. The larger sample size may also contribute to the longer performance time required by FL.

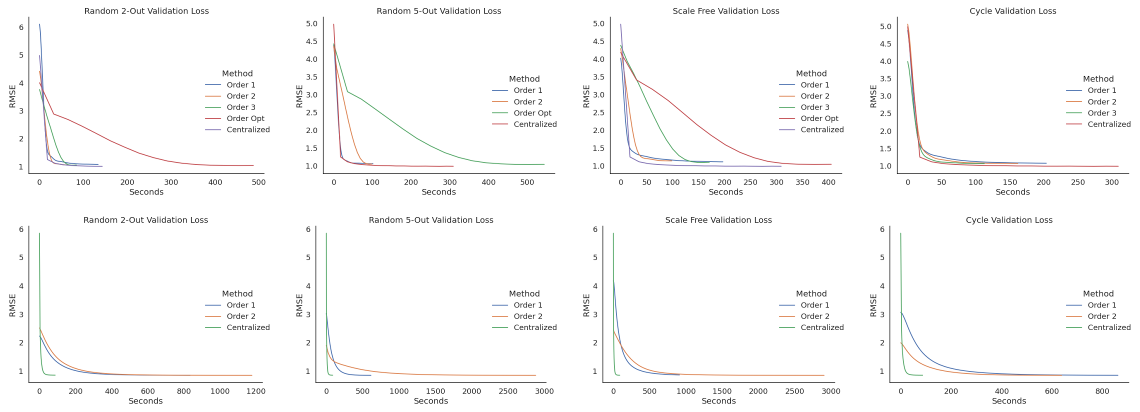


Figure 3.10: Comparing performance of various orders of the decentralized method to the centralized method over networks in MovieLens dataset(top) and H&M personalization dataset(bottom).

Figure 3.10 compares the validation loss curves of different orders of our method. We see that as the the order increases, the performance tends to improve as well. However, this often comes at the cost of training time. One interesting thing to note is that the loss curve particularly for the optimal order sharing method decreases inconsistently.

Our proposed decentralized method achieved significant performance gains even at lower orders compared to the other federated methods, and performed up to 6.6% better than FedAvg and 9.2% better than Local Learning on the MovieLens dataset. Our method also outperformed FedAvg by around 5% and Local Learning by 8.4% in the H&M dataset. Our decentralized method performed worst on the scale-free graph, which is more sparse than the Random K-Out graphs and has an extremely skewed node degree distribution. Particularly in the scale-free order 1 scenario, many nodes are only connect to one other hub-like node which may negatively impact their training. Once the order increases, the performance on scale-free improves significantly as now all nodes are much more connected than before. The scale-free directed graphs show unusual behavior as many nodes that do no have any in-degrees, which likely results in the poor performance. Furthermore, as seen in Figure 3.3, we see that increasing the order of the directed scale-free network does not result in a significant increase in connected users.

Finally, Figure 3.11 and Figure 3.12 compare the validation loss curves for all the models using order 1 and the optimal order respectively. The plots use a log scale for time. This allows to see the large runtime differences between FedAvg and the other models. However, in the optimal order plot we also see that our method can also take a significantly longer time than if the order was just 1.

In our proposed Generalized SVD architecture, we utilize a single representation layer

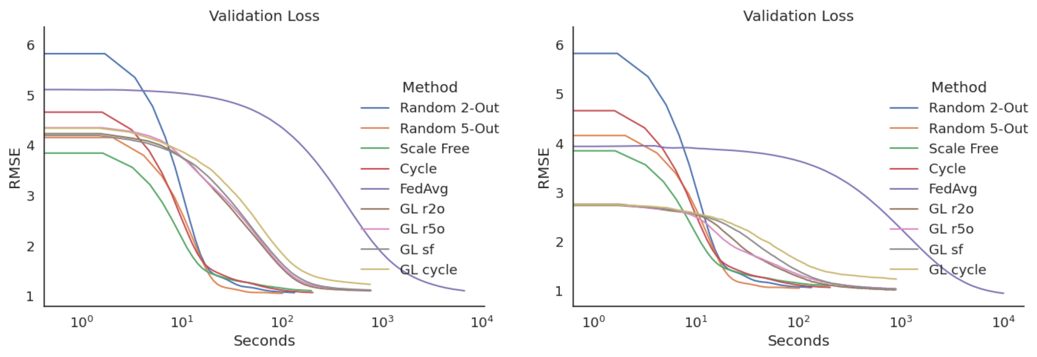


Figure 3.11: Validation losses with first order sharing for ML100K (left) and H&M (right). Gossip takes longer than 1st order and FedAvg takes an order of magnitude more time to achieve similar performance.

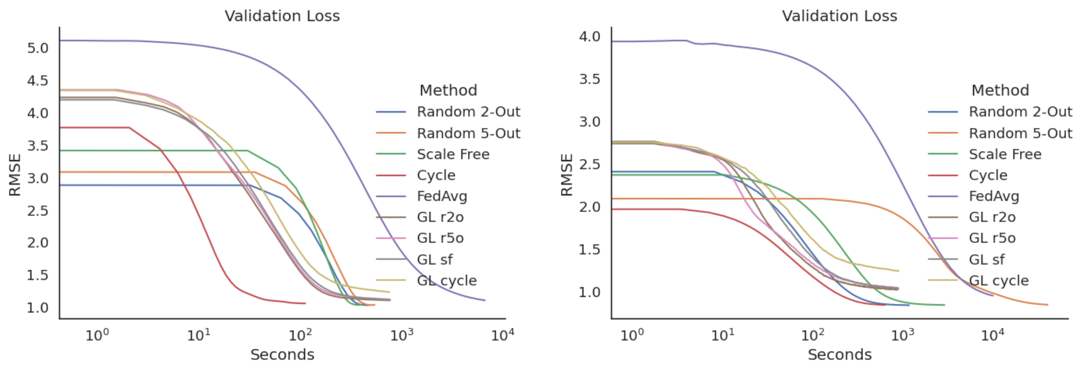


Figure 3.12: Validation losses with optimal sharing for ML100K (left) and H&M (right). Gossip takes a similar amount of time but FedAvg takes an order of magnitude more time to achieve similar performance.

Graph	1st Order	CL	FL	GL
Random 2-Out	1.133	0.930	1.110	1.120
Random 5-Out	1.129	0.930	1.110	1.113
Scale-Free	1.302	0.930	1.110	1.114
Cycle	1.237	0.930	1.110	1.230

Table 3.3: Generalized SVD performance using the decentralized scheme on the MovieLens dataset.

which still results in a much larger model than that of simple Matrix Factorization. This model architecture still exhibits a substantial improvement over other federated methods. However, due to it's size, it does take considerably longer than MF and is therefore more difficult to perform hyper-parameter turning as well as challenging to run on larger datasets. The results on the MovieLens data show that the method can work with even larger models that contain an order of magnitude more parameters than MF. However, the performance is not quite on par as the previous simpler models.

One issue with the GSVD model is that it's large size combined with the large number of users results in a large amount of memory required to train. Even with memory optimizations to reduce the required memory we reached limits on a typical consumer device. Table 3.4 show the estimated memory required to train the GSVD on various dataset on a single device.

3.6 Discussion

Our PushGrad describes a fully decentralized learning scheme that requires minimal information sharing in a network structure. Nodes share local model gradients with neighbor nodes to facilitate diffusion of local data effects. This means that no model

Table 3.4: Memory Used for Experiments

Model	Dataset	Memory Peak (GB)	k	Layer Size
GSVD	HM	60.00[1]	7	5
GSVD	HM Subset	19.00	7	5
GSVD	ML100K	8.93	30	10
GSVD	ML100K	6.74	7	5

Table 3.5: Predicted memory usage for model initialization.

merging step is taken within the nodes during training. We also provide a efficient method to select the optimal order of neighbors to share data to.

Our proposed decentralized scheme significantly improves the accuracy of content recommendations through the incorporation of neighbors' gradient information compared to other federated methods. Importantly, this scheme exhibits remarkable versatility, as it can adapt to a wide range of networks. Due to the limited nature of information transfer, our approach places a strong emphasis on safeguarding user privacy and data security without the need to share user information with neighbors, which is particularly important in today's digital landscape. Furthermore, the elimination of central server dependency streamlines the operational infrastructure, reducing complexities and potential points of failure. This optimization not only enhances system reliability but also simplifies the management of recommender systems. Lastly, our specially designed Generalized SVD method can integrate into this decentralized scheme while maintaining privacy safeguards, and demonstrates our methods ability to scale to larger models. This method offers the flexibility to be tailored according to specific requirements, adding customized layers to model architecture.

The PushGrad shows strong results in our experiments. We conjecture that this is due to a combination of a amenable model structure as well as the sparseness of data and network structure. The model structure for the matrix factorization model neatly isolates the large number item parameters by separating them into individual rows. This means that the shared gradient from a neighbor will only affect a small number of rows. Combined with the fact that each node typically does not have a lot of data, this means that the gradient data shared can update the factors of items that the user would otherwise never have updated themselves. This results in a information gained through neighbors since that data was not present on the node itself.

The relative sparseness of the network structure may also work in our favor. The worst case scenario for our PushGrad would be if two neighboring nodes had the same set of items but with different preferences (ratings). This could result in poor convergence as the gradients shared would take the neighbor model parameter in the opposite direction they would want them to go. However, the sparseness of the network means that such a situation is relatively rare as the number of neighbors is small. Therefore, we hypothesize the the strong performance of our model is due to the combination of model structure, low amount of data per user, and the sparsity of the network. The ideal case of our PushGrad would be when two neighbors have similar preferences, but completely different data points. This would allow non-overlapping gradient sharing to maximize the number of items an single node could learn about while avoiding the non-convergence issue from opposing preferences. Finally, our decentralized gradient sharing may be similar in effect to a noisy SGD where neighbors eventually converge to local minima that are potentially distinct but close together.

Potential avenues for future research lies in the exploration of a more efficient gradient transfer mechanism to further enhance user privacy or by conditioning the use of neighbor gradients during training. There is a trade-off between efficient yet small gradient transfers. Currently, neighbor models utilize the gradient shared with them immediately. It may be beneficial to condition the use of the gradient by ensuring it always improve the loss, or at least does not increase the loss by a certain threshold to reduce variance in the SGD process. Additionally, assumptions in Section 3.5 may be relaxed to demonstrate performance in more challenging scenarios. For example, a fully asynchronous set up would be ideal as then we could include the inherent parallelism in the federated and decentralized approaches. Furthermore, realistic network activity could be incorporated

Another idea is to have a probabilistic approach to chained gradient sharing rather than a fixed order selection. For example, each node could have a fixed probability of re-sharing a gradient it just received to its own neighbors. This would allow for further decentralization as now the nodes do not need to communicate how information on how many more steps a gradient needs to be shared. Further the number of chained gradient shares could be modeled using a geometric distribution. We hope to also apply our Pairwise Personalized Ranking model from Chapter 2 to the decentralized case. One advantage a pairwise model like PPR can have is that it can continue to create new data points by sampling different negative times for each positive items. This could allow the model to overcome problem of having a small amount of data on each node.

Bibliography

Goldberg, David et al. (Dec. 1992). “Using collaborative filtering to weave an information tapestry”. In: *Communications of the ACM* 35.12, pp. 61–70. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/138859.138867. URL: https://dl.acm.org/doi/10.1145/138859.138867 (visited on 02/18/2021).

Herlocker, Jonathan L. et al. (Jan. 2004). “Evaluating collaborative filtering recommender systems”. In: *ACM Transactions on Information Systems* 22.1, pp. 5–53. ISSN: 1046-8188, 1558-2868. DOI: 10.1145/963770.963772. URL: https://dl.acm.org/doi/10.1145/963770.963772 (visited on 02/10/2021).

McNee, Sean M., John Riedl, and Joseph A. Konstan (Apr. 21, 2006). “Being accurate is not enough: how accuracy metrics have hurt recommender systems”. In: *CHI ’06 Extended Abstracts on Human Factors in Computing Systems*. CHI06: CHI 2006 Conference on Human Factors in Computing Systems. Montréal Québec Canada: ACM, pp. 1097–1101. ISBN: 978-1-59593-298-3. DOI: 10.1145/1125451.1125659. URL: https://dl.acm.org/doi/10.1145/1125451.1125659 (visited on 08/24/2021).

Bennett, James and Stan Lanning (2007). “The Netflix Prize”. In: URL: https://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf (visited on 02/17/2021).

Marlin, Benjamin M et al. (2007). “Collaborative Filtering and the Missing at Random Assumption”. In: p. 9.

Salakhutdinov, Ruslan, Andriy Mnih, and Geoffrey Hinton (2007). “Restricted Boltzmann machines for collaborative filtering”. In: *Proceedings of the 24th interna-*

tional conference on Machine learning - ICML '07. the 24th international conference. Corvalis, Oregon: ACM Press, pp. 791–798. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273596. URL: `http://portal.acm.org/citation.cfm?doid=1273496.1273596` (visited on 02/21/2021).

Schneier, Bruce (Dec. 13, 2007). "Why 'Anonymous' Data Sometimes Isn't". In: *Wired*. Section: tags. ISSN: 1059-1028. URL: `https://www.wired.com/2007/12/why-anonymous-data-sometimes-isnt/` (visited on 03/26/2024).

Hagberg, Aric A, Daniel A Schult, and Pieter J Swart (2008). "Exploring Network Structure, Dynamics, and Function using NetworkX". In.

Koren, Yehuda (2008). "Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model". In: p. 9.

Narayanan, Arvind and Vitaly Shmatikov (May 2008). "Robust De-anonymization of Large Sparse Datasets". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 2008 IEEE Symposium on Security and Privacy (sp 2008). ISSN: 1081-6011. Oakland, CA, USA: IEEE, pp. 111–125. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.33. URL: `http://ieeexplore.ieee.org/document/4531148/` (visited on 03/04/2024).

Salakhutdinov, Ruslan and Andriy Mnih (2008). "Bayesian probabilistic matrix factorization using Markov chain Monte Carlo". In: *Proceedings of the 25th international conference on Machine learning - ICML '08*. the 25th international conference. Helsinki, Finland: ACM Press, pp. 880–887. ISBN: 978-1-60558-205-4. DOI: 10.1145/1390156.1390267. URL: `http://portal.acm.org/citation.cfm?doid=1390156.1390267` (visited on 02/17/2021).

Zhou, Yunhong et al. (2008). "Large-Scale Parallel Collaborative Filtering for the Netflix Prize". In: *Algorithmic Aspects in Information and Management*. Ed. by Rudolf Fleischer and Jinhui Xu. Vol. 5034. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 337–348. ISBN: 978-3-540-68865-5 978-3-540-68880-8. DOI: `10.1007/978-3-540-68880-8_32`. URL: `http://link.springer.com/10.1007/978-3-540-68880-8_32` (visited on 03/27/2024).

Cremonesi, Paolo, Yehuda Koren, and Roberto Turrin (2009). "Performance of recommender algorithms on top-N recommendation tasks". In.

Koren, Yehuda, Robert Bell, and Chris Volinsky (Aug. 2009). "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42.8, pp. 30–37. ISSN: 0018-9162. DOI: `10.1109/MC.2009.263`. URL: `http://ieeexplore.ieee.org/document/5197422/` (visited on 02/17/2021).

Rendle, Steffen et al. (2009). "BPR: Bayesian Personalized Ranking from Implicit Feedback". In: p. 10.

Adomavicius, Gediminas (2011). "Context-Aware Recommender Systems". In: p. 14.

Last.FM (2011). URL: `https://grouplens.org/datasets/hetrec-2011/`.

Li, Hang (2011). "A Short Introduction to Learning to Rank". In: *IEICE Transactions on Information and Systems* E94-D.10, pp. 1854–1862. ISSN: 0916-8532, 1745-1361. DOI: `10.1587/transinf.E94.D.1854`. URL: `http://www.jstage.jst.go.jp/article/transinf/E94.D/10/E94.D_10_1854/_article` (visited on 08/17/2021).

Ning, Xia and George Karypis (Dec. 2011). "SLIM: Sparse Linear Methods for Top-N Recommender Systems". In: *2011 IEEE 11th International Conference on Data Mining.* 2011 IEEE 11th International Conference on Data Mining. ISSN: 2374-8486, pp. 497–506. DOI: `10.1109/ICDM.2011.134`. URL: `https://ieeexplore.ieee.org/document/6137254` (visited on 03/30/2024).

Davenport, Mark A. et al. (July 1, 2014). “1-Bit Matrix Completion”. In: *Information and Inference*. arXiv: 1209.3672. URL: http://arxiv.org/abs/1209.3672 (visited on 03/11/2021).

Sedhain, Suvash et al. (May 18, 2015). “AutoRec: Autoencoders Meet Collaborative Filtering”. In: *Proceedings of the 24th International Conference on World Wide Web*. WWW ’15: 24th International World Wide Web Conference. Florence Italy: ACM, pp. 111–112. ISBN: 978-1-4503-3473-0. DOI: 10.1145/2740908.2742726. URL: https://dl.acm.org/doi/10.1145/2740908.2742726 (visited on 02/17/2021).

Shokri, Reza and Vitaly Shmatikov (Oct. 12, 2015). “Privacy-Preserving Deep Learning”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS’15: The 22nd ACM Conference on Computer and Communications Security. Denver Colorado USA: ACM, pp. 1310–1321. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813687. URL: https://dl.acm.org/doi/10.1145/2810103.2813687 (visited on 03/06/2024).

Wang, Hao, Naiyan Wang, and Dit-Yan Yeung (Aug. 10, 2015). “Collaborative Deep Learning for Recommender Systems”. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’15: The 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Sydney NSW Australia: ACM, pp. 1235–1244. ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783273. URL: https://dl.acm.org/doi/10.1145/2783258.2783273 (visited on 02/25/2021).

Bonawitz, Keith et al. (Nov. 14, 2016). *Practical Secure Aggregation for Federated Learning on User-Held Data*. DOI: 10.48550/arXiv.1611.04482. arXiv: 1611.04482[cs,stat]. URL: http://arxiv.org/abs/1611.04482 (visited on 03/05/2024).

Harper, F. Maxwell and Joseph A. Konstan (Jan. 7, 2016). “The MovieLens Datasets: History and Context”. In: *ACM Transactions on Interactive Intelligent Systems* 5.4, pp. 1–19. ISSN: 2160-6455, 2160-6463. DOI: 10.1145/2827872. URL: https://dl.acm.org/doi/10.1145/2827872 (visited on 02/17/2021).

Hegedűs, István, Árpád Berta, et al. (July 14, 2016). “Robust Decentralized Low-Rank Matrix Decomposition”. In: *ACM Transactions on Intelligent Systems and Technology* 7.4, pp. 1–24. ISSN: 2157-6904, 2157-6912. DOI: 10.1145/2854157. URL: https://dl.acm.org/doi/10.1145/2854157 (visited on 04/10/2024).

Perlroth, Nicole (Sept. 22, 2016). “Yahoo Says Hackers Stole Data on 500 Million Users in 2014”. In: *The New York Times*. ISSN: 0362-4331. URL: https://www.nytimes.com/2016/09/23/technology/yahoo-hackers.html (visited on 03/06/2024).

Radley-Gardner, Oliver, Hugh Beale, and Reinhard Zimmermann, eds. (2016). *Fundamental Texts On European Private Law*. Hart Publishing. ISBN: 978-1-78225-864-3 978-1-78225-865-0 978-1-78225-866-7 978-1-78225-867-4. DOI: 10.5040/9781782258674. URL: http://www.bloomsburycollections.com/book/fundamental-texts-on-european-private-law-1 (visited on 03/06/2024).

Bi, Xuan (2017). “A Group Specific Recommendor System”. In: *Journal of the American Statistical Association*. DOI: https://doi.org/10.1080/01621459.2016.1219261. URL: https://doi.org/10.1080/01621459.2016.1219261.

Guo, Huifeng et al. (Mar. 13, 2017). *DeepFM: A Factorization-Machine based Neural Network for CTR Prediction*. DOI: 10.48550/arXiv.1703.04247. arXiv: 1703.04247[cs]. URL: http://arxiv.org/abs/1703.04247 (visited on 03/04/2024).

He, Xiangnan et al. (Apr. 3, 2017). “Neural Collaborative Filtering”. In: *Proceedings of the 26th International Conference on World Wide Web*. WWW ’17: 26th International World Wide Web Conference. Perth Australia: International World Wide

Web Conferences Steering Committee, pp. 173–182. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052569. URL: https://dl.acm.org/doi/10.1145/3038912.3052569 (visited on 02/10/2021).

Konečný, Jakub et al. (Oct. 30, 2017). *Federated Learning: Strategies for Improving Communication Efficiency*. arXiv: 1610.05492[cs]. URL: http://arxiv.org/abs/1610.05492 (visited on 03/06/2024).

Li, Dongsheng, Chao Chen, Wei Liu, et al. (2017). “Mixture-Rank Matrix Approximation for Collaborative Filtering”. In: *Conference on Neural Information Processing Systems*.

McMahan, H. Brendan et al. (2017). *Communication-Efficient Learning of Deep Networks from Decentralized Data*. DOI: 10.48550/arXiv.1602.05629. arXiv: 1602.05629[cs]. URL: http://arxiv.org/abs/1602.05629 (visited on 03/04/2024).

Confessore, Nicholas (Apr. 4, 2018). “Cambridge Analytica and Facebook: The Scandal and the Fallout So Far”. In: *The New York Times*. ISSN: 0362-4331. URL: https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html (visited on 03/06/2024).

Krichene, Walid, Nicolas Mayoraz, et al. (July 18, 2018). “Efficient Training on Very Large Corpora via Gramian Estimation”. In: *arXiv:1807.07187 [cs, stat]*. arXiv: 1807.07187. URL: http://arxiv.org/abs/1807.07187 (visited on 02/25/2021).

Li, Dongsheng, Chao Chen, Qin Lv, et al. (2018). “AdaError: An Adaptive Learning Rate Method for Matrix Approximation-based Collaborative Filtering”. In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW ’18*. the 2018 World Wide Web Conference. Lyon, France: ACM Press, pp. 741–751. ISBN: 978-1-4503-5639-8. DOI: 10.1145/3178876.3186155. URL: http://dl.acm.org/citation.cfm?doid=3178876.3186155 (visited on 03/28/2024).

Liang, Dawen et al. (Feb. 15, 2018). *Variational Autoencoders for Collaborative Filtering*. arXiv: 1802.05814[cs,stat]. URL: http://arxiv.org/abs/1802.05814 (visited on 03/29/2024).

Beutel, Alex et al. (Mar. 2, 2019). "Fairness in Recommendation Ranking through Pairwise Comparisons". In: *arXiv:1903.00780 [cs, stat]*. arXiv: 1903.00780. URL: http://arxiv.org/abs/1903.00780 (visited on 02/23/2021).

Dacrema, Maurizio Ferrari, Paolo Cremonesi, and Dietmar Jannach (Sept. 10, 2019). "Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches". In: *Proceedings of the 13th ACM Conference on Recommender Systems*, pp. 101–109. DOI: 10.1145/3298689.3347058. arXiv: 1907.06902. URL: http://arxiv.org/abs/1907.06902 (visited on 02/05/2021).

Hu, Chenghao, Jingyan Jiang, and Zhi Wang (Aug. 21, 2019). *Decentralized Federated Learning: A Segmented Gossip Approach*. DOI: 10.48550/arXiv.1908.07782. arXiv: 1908.07782[cs,stat]. URL: http://arxiv.org/abs/1908.07782 (visited on 03/04/2024).

Ma, Wei and George H Chen (2019). "Missing Not at Random in Matrix Completion: The Effectiveness of Estimating Missingness Probabilities Under a Low Nuclear Norm Assumption". In: p. 10.

Rendle, Steffen, Li Zhang, and Yehuda Koren (May 3, 2019). "On the Difficulty of Evaluating Baselines: A Study on Recommender Systems". In: *arXiv:1905.01395 [cs]*. arXiv: 1905.01395. URL: http://arxiv.org/abs/1905.01395 (visited on 02/10/2021).

Steck, Harald (May 13, 2019). "Embarrassingly Shallow Autoencoders for Sparse Data". In: *The World Wide Web Conference*, pp. 3251–3257. DOI: 10.1145/3308558.3313710. arXiv: 1905.03375[cs,stat]. URL: http://arxiv.org/abs/1905.03375 (visited on 03/30/2024).

Zhang, Jiale et al. (Aug. 2019). "Poisoning Attack in Federated Learning using Generative Adversarial Nets". In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). ISSN: 2324-9013, pp. 374–380. DOI: `10.1109/TrustCom/BigDataSE.2019.00057`. URL: `https://ieeexplore.ieee.org/document/8887357` (visited on 03/04/2024).

Zhang, Shuai et al. (Feb. 28, 2019). "Deep Learning based Recommender System: A Survey and New Perspectives". In: *ACM Computing Surveys* 52.1, pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: `10.1145/3285029`. arXiv: `1707.07435`. URL: `http://arxiv.org/abs/1707.07435` (visited on 02/21/2021).

Zhu, Ligeng, Zhijian Liu, and Song Han (Dec. 19, 2019). *Deep Leakage from Gradients*. DOI: `10.48550/arXiv.1906.08935`. arXiv: `1906.08935[cs,stat]`. URL: `http://arxiv.org/abs/1906.08935` (visited on 03/04/2024).

Harris, Charles R. et al. (Sept. 17, 2020). "Array programming with NumPy". In: *Nature* 585.7825, pp. 357–362. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/s41586-020-2649-2`. URL: `https://www.nature.com/articles/s41586-020-2649-2` (visited on 04/01/2024).

Hegedűs, István, Gábor Danner, and Márk Jelasity (2020). "Decentralized Recommendation Based on Matrix Factorization: A Comparison of Gossip and Federated Learning". In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Peggy Cellier and Kurt Driessens. Vol. 1167. Series Title: Communications in Computer and Information Science. Cham: Springer International Publishing, pp. 317–332. ISBN: 978-3-030-43822-7 978-3-030-43823-4. DOI: `10.1007/978-3-`

030-43823-4_27. URL: https://link.springer.com/10.1007/978-3-030-43823-4_27 (visited on 03/04/2024).

Jiang, Jingyan and Liang Hu (2020). "Decentralised federated learning with adaptive partial gradient aggregation". In: *CAAI Transactions on Intelligence Technology* 5.3. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/trit.2020.0082, pp. 230–236. ISSN: 2468-2322. DOI: 10.1049/trit.2020.0082. URL: https://onlinelibrary.wiley.com/doi/abs/10.1049/trit.2020.0082 (visited on 03/04/2024).

Krichene, Walid and Steffen Rendle (Aug. 23, 2020). "On Sampled Metrics for Item Recommendation". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Virtual Event CA USA: ACM, pp. 1748–1757. ISBN: 978-1-4503-7998-4. DOI: 10.1145/3394486.3403226. URL: https://dl.acm.org/doi/10.1145/3394486.3403226 (visited on 02/25/2021).

Lian, Defu, Qi Liu, and Enhong Chen (Apr. 20, 2020). "Personalized Ranking with Importance Sampling". In: *Proceedings of The Web Conference 2020.* WWW '20: The Web Conference 2020. Taipei Taiwan: ACM, pp. 1093–1103. ISBN: 978-1-4503-7023-3. DOI: 10.1145/3366423.3380187. URL: https://dl.acm.org/doi/10.1145/3366423.3380187 (visited on 04/21/2022).

Muhammad, Khalil et al. (Aug. 23, 2020). "FedFast: Going Beyond Average for Faster Training of Federated Recommender Systems". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Virtual Event CA USA: ACM, pp. 1234–1242. ISBN: 978-1-4503-7998-4. DOI:

`10.1145/3394486.3403176`. URL: `https://dl.acm.org/doi/10.1145/3394486.3403176` (visited on 04/16/2024).

Rendle, Steffen, Walid Krichene, et al. (Sept. 22, 2020). "Neural Collaborative Filtering vs. Matrix Factorization Revisited". In: *Fourteenth ACM Conference on Recommender Systems*. RecSys '20: Fourteenth ACM Conference on Recommender Systems. Virtual Event Brazil: ACM, pp. 240–248. ISBN: 978-1-4503-7583-2. DOI: `10.1145/3383313.3412488`. URL: `https://dl.acm.org/doi/10.1145/3383313.3412488` (visited on 02/05/2021).

Zhao, Bo, Konda Reddy Mopuri, and Hakan Bilen (Jan. 8, 2020). *iDLG: Improved Deep Leakage from Gradients*. DOI: `10.48550/arXiv.2001.02610`. arXiv: `2001.02610[cs,stat]`. URL: `http://arxiv.org/abs/2001.02610` (visited on 03/04/2024).

Bouacida, Nader and Prasant Mohapatra (2021). "Vulnerabilities in Federated Learning". In: *IEEE Access* 9, pp. 63229–63249. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2021.3075203`. URL: `https://ieeexplore.ieee.org/document/9411833/` (visited on 03/04/2024).

Chen, Jiawei et al. (Dec. 29, 2021). "Bias and Debias in Recommender System: A Survey and Future Directions". In: *arXiv:2010.03240 [cs]*. arXiv: `2010.03240`. URL: `http://arxiv.org/abs/2010.03240` (visited on 04/21/2022).

Dacrema, Maurizio Ferrari, Simone Boglio, et al. (Feb. 3, 2021). "A Troubling Analysis of Reproducibility and Progress in Recommender Systems Research". In: *ACM Transactions on Information Systems* 39.2. version: 3, pp. 1–49. ISSN: 1046-8188, 1558-2868. DOI: `10.1145/3434185`. arXiv: `1911.07698`. URL: `http://arxiv.org/abs/1911.07698` (visited on 02/05/2021).

Miao, Qinyang et al. (Oct. 9, 2021). "Federated deep reinforcement learning based secure data sharing for Internet of Things". In: *Computer Networks* 197, p. 108327. ISSN: 1389-1286. DOI: `10.1016/j.comnet.2021.108327`. URL: `https://www.`

sciencedirect.com/science/article/pii/S1389128621003285 (visited on 03/04/2024).

Rendle, Steffen (Jan. 21, 2021). “Item Recommendation from Implicit Feedback”. In: *arXiv:2101.08769 [cs]*. arXiv: 2101.08769. URL: http://arxiv.org/abs/2101.08769 (visited on 02/25/2021).

Tamm, Yan-Martin, Rinchin Damdinov, and Alexey Vasilev (Sept. 13, 2021). “Quality Metrics in Recommender Systems: Do We Calculate Metrics Consistently?” In: *Fifteenth ACM Conference on Recommender Systems*, pp. 708–713. DOI: 10.1145/3460231.3478848. arXiv: 2206.12858[cs]. URL: http://arxiv.org/abs/2206.12858 (visited on 03/30/2024).

Warnat-Herresthal, Stefanie et al. (June 2021). “Swarm Learning for decentralized and confidential clinical machine learning”. In: *Nature* 594.7862. Number: 7862 Publisher: Nature Publishing Group, pp. 265–270. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03583-3. URL: https://www.nature.com/articles/s41586-021-03583-3 (visited on 03/04/2024).

Belal, Yacine et al. (Sept. 6, 2022). “PEPPER: Empowering User-Centric Recommender Systems over Gossip Learning”. In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6.3, pp. 1–27. ISSN: 2474-9567. DOI: 10.1145/3550302. arXiv: 2208.05320[cs]. URL: http://arxiv.org/abs/2208.05320 (visited on 03/04/2024).

Li, Chengxi, Gang Li, and Pramod K. Varshney (Jan. 15, 2022). “Decentralized Federated Learning via Mutual Knowledge Transfer”. In: *IEEE Internet of Things Journal* 9.2, pp. 1136–1147. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2021.3078543. URL: https://ieeexplore.ieee.org/document/9426904/ (visited on 03/08/2024).

Long, Jing et al. (July 31, 2022). *Decentralized Collaborative Learning Framework for Next POI Recommendation*. DOI: 10.48550/arXiv.2204.06516. arXiv: 2204.06516[cs]. URL: http://arxiv.org/abs/2204.06516 (visited on 03/04/2024).

Adomavicius, Gediminas and Alexander Tuzhilin (n.d.). "Towards the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions". In: ().

sgd-vs-gd (n.d.). URL: https://i.stack.imgur.com/G7BBG.png.

Appendices

Appendix A

Pairwise Personalized Learning

A.1 Computational Complexity

In this section, we use Big O notation to briefly discuss the computational costs of the matrix factorization model, which is at the core of our work. Training the matrix factorization model using stochastic gradient descent is not a deterministic process and depends on the data size, number of users, number of items, number of factors, and the number of epochs among other things. With this in mind we first describe the complexity of a single forward pass on the matrix factorization model. A single forward pass of the simple MF model can be described as the following:

$$\hat{y}_{ui} = \boldsymbol{p}_u^\top \boldsymbol{q}_i \tag{A.1}$$

Using this equation we can see that the complexity of a single forward pass is $\mathcal{O}(k)$ where k is the number of factors. Then the complexity of predicting the ratings for the set of observed data Ω is $\mathcal{O}(|\Omega| \cdot k)$.

To train the MF model using SGD we refer the update steps within each iteration. Given that the error term is

$$e_{ui} \equiv y_{ui} - \boldsymbol{p}_u^\top \boldsymbol{q}_i, \tag{A.2}$$

the parameter update step equations are

$$\begin{aligned} \boldsymbol{p}_u &\leftarrow \boldsymbol{p}_u + \gamma(e_{ui}\boldsymbol{q}_i - \lambda \boldsymbol{p}_u) \\ \boldsymbol{q}_i &\leftarrow \boldsymbol{q}_i + \gamma(e_{ui}\boldsymbol{p}_u - \lambda \boldsymbol{q}_i) \ . \end{aligned} \tag{A.3}$$

Then, for a single data point the number of operations is $(1 + k) + 2(5k)$. If we consider one epoch, the complexity will be $\mathcal{O}(|\Omega| \cdot (1 + k))$. Finally, the algorithm may take several epochs to converge. Given N epochs, the total complexity to train the matrix factorization model becomes $\mathcal{O}(N \cdot |\Omega| \cdot (1 + k))$.

Next, we discuss the complexity of the Pairwise Personalized Ranking model. In these discussions we consider the PPR using the simple matrix factorization model seen in A.1. Given a set of triples T, we see in Algorithm 1 that the SGD loop is very similar to that of MF. Calculating $\hat{y}_{uij}$ takes $(2k + 1)$ operations which is $\mathcal{O}(k + 1)$ and further calculating z_{uij} will also be $\mathcal{O}(k + 1)$. Finally update steps only add $\mathcal{O}(k)$ operations so the final complexity of the PPR model for a single training point is $\mathcal{O}(k + 1)$. Again, assuming N epochs where the entire T is used per epoch we see that it will be $\mathcal{O}(N \cdot |T|(k + 1))$. One nice quality of the matrix factorization model is that although there are a total of $(k \times (n + m))$ parameters, only a small fixed number of parameters will be updated for each training point.

Finally, we discuss how T might grow depending the size of the dataset. In the implicit case, T is generated by pairing observed items for each user with their unobserved items. If we assume that the observed data scales proportionally with the total size of the dataset for each user - i.e. that a fixed proportion p_u of the total dataset is observed for each user, then $|T|$, is $\mathcal{O}(|\mathcal{U}| \times |\mathcal{I}|^2)$. For the PPR model, we split the set of observed data into true positive and true negative triplets per user. With the same proportionality assumption as before, we see the T is also $\mathcal{O}(|\mathcal{U}| \times |\mathcal{I}|^2)$, albeit with a much smaller constant factor. However, as we note in Section 2.3, we do not opt to use the entire T for training for each epoch and use a sampling method instead.

A.2 ML100K Results

Plots for the results on the ML100K were included in Chapter 2. Table A.1 shows the full results including the various S_u sizes.

A.2.1 Results Including EASE and VAECF

Results including the EASE and VAECF models are included in Figure A.1. We see that EASE and VAECF are the strongest performers with our proposed methods coming close when the rating threshold is equal to 5.

A.3 Models

A.3.1 Rating Weighted PPR

We hypothesized that in the PPR model, a weighting scheme could be used to that depended directly on the ratings of the true positive item and the true negative item. A pair of items that were far apart, like 1 and 5 would be easier for the model to differentiate than a pair of items that were closer together like 3 and 4. Therefore, we tested a weighting scheme that gave more weight to the loss when the ratings were

Table A.1: MovieLens 100K Results of Maximal S_u for rating cutoffs 3, 4, 5, and the 85th user-quantiles.

Cutoff: 5	NDCG at 5	NDCG at 10	Recall at 5	Recall at 10
APR	0.143	0.154	0.107	0.168
PPR	0.136	0.146	0.104	0.161
BPR	0.116	0.136	0.097	0.146
ItemKNN	0.029	0.036	0.021	0.043
MF	0.018	0.025	0.015	0.034
UserKNN	0.001	0.004	0.001	0.008
Cutoff: 85 quantile				
APR	0.157	0.159	0.095	0.149
PPR	0.149	0.146	0.087	0.133
BPR	0.129	0.143	0.084	0.144
MF	0.053	0.053	0.028	0.045
UserKNN	0.001	0.003	0.000	0.004
ItemKNN	0.000	0.001	0.000	0.001
Cutoff: 4	NDCG at 5	NDCG at 10	Recall at 5	Recall at 10
APR	0.190	0.175	0.071	0.112
PPR	0.177	0.159	0.064	0.096
BPR	0.170	0.169	0.074	0.130
ItemKNN	0.042	0.043	0.012	0.025
MF	0.022	0.027	0.009	0.019
UserKNN	0.001	0.004	0.001	0.005
Cutoff: 3				
APR	0.221	0.197	0.059	0.092
PPR	0.211	0.186	0.050	0.079
BPR	0.207	0.204	0.073	0.122
ItemKNN	0.048	0.049	0.009	0.020
MF	0.024	0.029	0.006	0.015
UserKNN	0.001	0.005	0.000	0.004

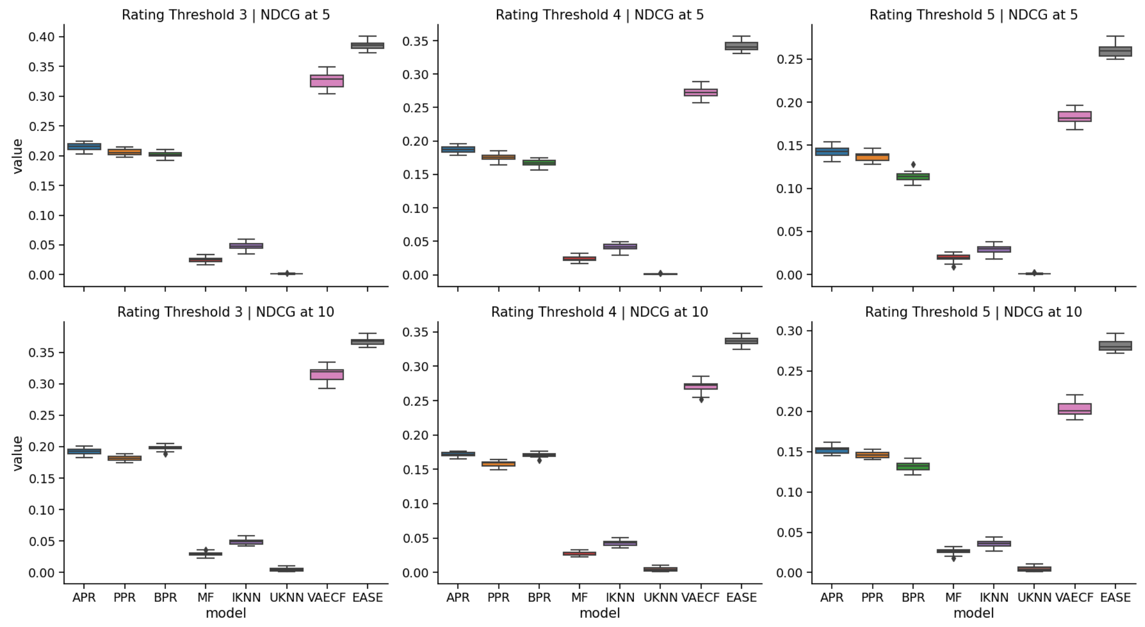


Figure A.1: Results on ML100K dataset including EASE and VAECF for cutoffs 3, 4, and 5.

closer together to help the model learn the difficult pairs more quickly.

$$\min_{\theta} \sum_{(u,i,j)\in T} -w(y_{ui}, y_{uj}) \log(\sigma(\hat{y}_{u,i,\theta} - \hat{y}_{u,j,\theta})) + \frac{\lambda}{2}(||\theta||) \tag{A.4}$$

However, preliminary results showed no performance gain while training was slowed.

Appendix B

Code

The codebases for the Pairwise Personalized Ranking model and the Decentralized Recommender Method was a large part of our work as we spent significant effort to make the model high performance and easy to update. The entire codebase for the Pairwise Personalized Ranking model can be found on github in the rec-sys repository and the code for the decentralized learning method can be found here.