



Computer Science Department

Optimizing Product Recommendations in E-commerce: A Machine Learning Approach with RBM (Restricted Boltzmann Machines) and MLP Using the H&M Dataset

Nikolaos Theodorou

31 May 2024

Acknowledgements

I would like to express my deepest gratitude to my professor, Chris Christodoulou, for his exceptional guidance and support throughout this journey. His course on Machine Learning ignited my passion for this fascinating field of computer science. I am particularly thankful for his supervision and the insightful knowledge he imparted, which significantly contributed to the development and completion of this thesis.

I would also like to extend my sincere thanks to my other professor, Michalis Agathokleous, for his invaluable contributions. His teaching during the labs of the Machine Learning course and his constructive feedback and tips on this thesis were instrumental in shaping the quality of my work.

Lastly, I would like to express my appreciation to my employer, Parallax Works LTD, for their generous sponsorship of this thesis. Their support and provision of the required hardware were crucial for the successful execution of my research. Their commitment to fostering an environment that encourages academic and professional growth is truly commendable.

Abstract

This thesis investigates the optimization of product recommendation systems in the e-commerce sector, focusing specifically on the application of Restricted Boltzmann Machines (RBM) and Multi-Layer Perceptrons (MLP) using the H&M Personalized Fashion Recommendations dataset. The study aims to enhance the accuracy and efficiency of recommender systems by leveraging implicit feedback from customer transaction data.

The primary objectives of this research are to develop an RBM-based recommender system capable of efficiently utilizing minimal transactional data and to extend an MLP-based model optimized for richer datasets, including user demographics and item characteristics. Additionally, the study benchmarks these models against traditional and contemporary machine learning models such as Matrix Factorization (MF), Generalized Matrix Factorization (GMF), and Neural Collaborative Filtering (NCF).

A key component of this research is the advanced data preprocessing designed to utilize datasets that contain only implicit data, such as transaction records. This preprocessing extracts insightful information, culminating in the proposed affinity score formula. The affinity score integrates the z-score distance from the expected purchasing quantity when a user buys a product and a recency score, providing a comprehensive measure of user-item interaction strength.

The research employs advanced feature engineering, strategic data preprocessing, and extensive model tuning. Results demonstrate that the RBM model consistently outperforms other models across all metrics, proving to be the most efficient and effective in predicting customer preferences. However, the performance of RBM begins to plateau after a certain number of epochs, suggesting the model's capacity limit rather than overfitting. The study confirms the generalizability of the proposed affinity score formula, indicating its potential application beyond the fashion e-commerce industry to other retail sectors.

The findings of this thesis contribute to the broader understanding of machine learning applications in recommender systems, offering valuable insights for enhancing recommendation accuracy and system efficiency. Future work will focus on further refining these models and exploring their applicability in various e-commerce contexts.

Table of Contents

Chapter 1: Introduction	8
1.1 The Personal Project, a starting point	8
1.2 Value of Recommender systems in marketing	8
1.3 Problem Statement	9
1.4 Objectives	9
1.5 Significance of the Study	10
1.6 Limitations	11
Chapter 2: Current Scientific Landscape of Recommender Systems	13
2.1 Fundamental Approaches	13
2.1.1 Collaborative Filtering	13
2.1.2 Content-Based Filtering	14
2.1.3 Hybrid Systems	15
2.2 Machine Learning in Recommender Systems	15
2.2.1 Matrix Factorization Techniques	15
2.2.2 Deep Learning Approaches	16
2.3 Restricted Boltzmann Machines (RBM) in Recommender Systems	17
2.3.1 The RBM Landscape	17
2.3.2 Diving into RBMs functioning	18
2.3.3 Multinomial RBMs	21
2.4 Multi-Layer Perceptrons (MLP) in Recommender Systems	22
2.4.1 Neural Collaborative Filtering	23
2.5 Challenges	25
2.6 Summary	25
Chapter 3: Dataset Description	28
3.1 Overview of H&M Personalized Fashion Recommendations Dataset	28
3.1.1 Dataset Composition	28
3.1.2 Characteristics and Considerations	29
3.1.3 Temporal Dynamics	32
3.1.4 Missing Values	32
3.2 Competition Winners	32
Chapter 4: Methodology	35

4.1 Data Preprocessing	35
4.1.1 Shrinking the Dataset	35
4.1.2 Calculate the User-Item Affinity Score	38
4.1.3 Preparing the Input Vectors for the MF, GMF and NCF	45
4.1.4 Preparing the Input Vectors for the RBM	46
4.1.5 Preparing the Input Vectors for the feature accommodated NCF	46
4.2 Overview of decisions across the models' implementation	47
4.3 Implementation of the Traditional Matrix Factorization Model Class	48
4.4 Implementation of the Generalized Matrix Factorization Model Class	49
4.5 Implementation of the Neural Collaborative Filtering Model Class	51
4.6 Implementation details of MF, GMF and NCF	52
4.6.1 Data Preparation	52
4.6.2 Loading Datasets and Encoding User and Item IDs	53
4.6.3 Model Initialization and Configuration	54
4.6.4 Training Procedure	55
4.6.5 Model Evaluation and Metrics Calculation	56
4.6.6 Evaluation Metrics at K	58
4.6.7 Exporting Results in DataFrame	59
4.7 Demographic and Feature Enhanced Neural Collaborative Filtering (DFNCF)	60
4.7.1 How DFNCF Works	61
4.7.2 Implementation of DFNCF Class	62
4.8 Multinomial RBM	64
4.8.1 Implementation of Multinomial RBM class	65
4.8.2 Loading the Data	66
4.8.3 Training the RBM	67
4.9 Tuning the Models	68
4.10 Tuning Results of Model Hyperparameters	70
4.11 Tuning results of Coefficients for Affinity Score Formula Across Models	72
Chapter 5: Results & Discussion	74
5.1 Evaluation of the Proposed Affinity Score Formula on the Matrix Factorization Model	75
5.1.1 Results	76
5.1.2 Discussion	78

5.2 Evaluation of the Proposed Affinity Score Formula on the Generalized Matrix Factorization (GMF) Model.....	79
5.2.1 Results.....	79
5.2.2 Discussion	83
5.3 Evaluation of the Proposed Affinity Score Formula on the Neural Collaborative Filtering (NCF) Model.....	84
5.3.1 Results.....	84
5.3.2 Discussion	88
5.4 Evaluation of the Proposed Affinity Score Formula on the DFNCF Model.....	89
5.4.1 Results.....	89
5.4.2 Discussion	93
5.5 Evaluation of the Proposed Affinity Score Formula on the Multinomial RBM Model	94
5.5.1 Results.....	94
5.5.2 Discussion	98
5.6 Comparing the performance of the models	98
5.6.1 Comparing the models on the top 1% subset	99
5.6.2 Comparing the models on the top 10% subset	104
Chapter 6: Conclusion	110
Chapter 7: Future Works	113
References:	115
APPENDIX.....	118
1. Large code snippets and code documentation.....	118

Chapter 1: Introduction

1.1 The Personar Project, a starting point

In the rapidly evolving e-commerce landscape, personalized shopping experiences have become a cornerstone of consumer satisfaction and business success. This drive towards personalization has brought the widespread adoption of marketing systems to generate product recommendations tailored to individual user preferences. Among these initiatives is the Personar project by Parallax Works LTD, which utilizes data-driven strategies for marketing campaigns, offering a foundational understanding of customer engagement through data. Currently, its core functionality is limited to the traditional marketing technique of generating RFM (Recency, Frequency, Monetary) metrics labels, that evaluates customers based on their recent purchase behavior, purchase frequency, and spending amount. This analysis helps identify customers' purchasing behavior and segment high-value customers for targeted marketing strategies.

1.2 Value of Recommender systems in marketing

Recommender systems are critical tools in the digital marketplace, analyzing vast amounts of data to predict and present items users are likely to find appealing. They analyze user behaviors, like past purchases and views, along with item characteristics, to suggest relevant products. This not only personalizes the user experience but also enhances engagement and loyalty by accurately reflecting individual preferences. The adaptability of these systems ensures they continuously refine their recommendations, making them indispensable for businesses looking to optimize marketing strategies and improve customer satisfaction. Their broad applicability across industries highlights their role in tailoring user interactions, making them a key component in modern e-commerce ecosystems.

1.3 Problem Statement

Recommender systems are essential tools in modern customer-centric marketing but face substantial challenges, particularly when utilizing implicit feedback such as purchase history to predict user preferences accurately. These challenges are further compounded by the variability in dataset composition; while most datasets predominantly consist of transactional interactions between users and items, only a select few encompass richer data elements like user demographics (age, gender) and detailed item attributes (color, type, category). The Personar project strives to lead in customer-centric marketing by demanding a recommender system that is not just effective but also versatile, capable of functioning optimally with both minimal and detailed data levels. To bridge this divide, this thesis proposes the development and evaluation of two distinct machine learning model-based recommender systems: one optimized for datasets with mere transaction history and another designed to capitalize on richer datasets that include user demographics and item features, that could also mitigate the cold start problem recommender systems usually face. Additionally, we incorporate data analysis techniques in our preprocessing steps to develop a formula aimed at enhancing recommendation accuracy, ensuring our models adapt and perform robustly within a dynamic and flexible framework.

1.4 Objectives

The aim of this thesis is to support the Personar project's vision by creating a flexible, data-agnostic recommender system capable of providing accurate product recommendations based on implicit feedback.

To address the challenges faced by recommender systems in utilizing implicit feedback effectively, this thesis outlines the following objectives:

1. **Develop an RBM-based recommender system:** This system will be designed to efficiently leverage implicit feedback to predict user preferences across datasets of poor detail levels with minimal transactional data.
2. **Extend the MLP-based model:** Create an MLP-based recommender system that is specifically optimized to harness extended data details, such as demographics and item characteristics and is cold start proof.
3. **Benchmarking against traditional and modern models:** Evaluate and compare the performance of both the RBM and the extended MLP models against traditional and contemporary benchmarks. This includes the Matrix Factorization (MF) model, Generalized Matrix Factorization (GMF), and Neural Collaborative Filtering (NCF) models. This comparative analysis will help ascertain their relative effectiveness in real world problems.
4. **Investigate advanced modeling techniques:** Explore the impact of a proposed preference scoring system and negative sampling on model performance, during the data preprocessing step.

These objectives are designed to culminate in the development of innovative recommender systems that not only meet the diverse needs of modern datasets but also set new standards in recommendation accuracy and adaptability.

1.5 Significance of the Study

This research endeavors to make a substantial contribution across several fronts. By delving into the comparative analysis of Restricted Boltzmann Machines (RBM) and Multi-Layer Perceptrons (MLP) within the context of recommender systems, this study stands at the intersection of machine learning, data science, and e-commerce, reflecting the multidisciplinary nature of contemporary computer science education.

From an academic perspective, this thesis aims to enrich the body of knowledge in machine learning applications for recommender systems, a domain that has seen significant interest but also poses considerable challenges. The exploration of RBM models

in handling implicit feedback and dataset variability provides a detailed case study on the practical application of these theories. This aligns closely with the core principles of computer science and data analytics, making it a pertinent example of how theoretical knowledge is applied to solve real-world problems. Furthermore, by examining the efficacy of these models in a dynamic and commercially relevant context, the study offers insights that could guide future research, curriculum development, and the evolution of machine learning methodologies.

In the broader context of e-commerce, the findings of this research promise to transcend the specific case of the Personar project, offering actionable insights that could be leveraged by marketing experts seeking to improve their customer loyalty and sales.

Lastly, the relevance of this study to my *course and career cannot be overstated. It embodies the application of advanced computer science and machine learning concepts to address a real challenge in the e-commerce sector, demonstrating the critical thinking, analytical, and technical skills that are the hallmark of my education. This thesis not only serves as a final project, synthesizing my learning and research but also positions me at the forefront of emerging technologies and their application in digital commerce.*

1.6 Limitations

The scope of this research is defined by several practical constraints, notably limitations in hardware resources and time schedule constraints, which naturally delimit the scope of the project. While the objectives outlined aim to contribute significantly to the understanding and development of machine learning-based recommender systems, it's essential to acknowledge these constraints as they influence the methodology, scale, and depth of analysis possible within the timeframe of this thesis.

The computational demands of training and testing sophisticated machine learning models like RBM and MLP are substantial. Given the extensive datasets involved in e-

commerce recommendations, such as the H&M dataset utilized in this study, the hardware limitations present a significant constraint. Therefore, the scope of model training, parameter tuning, and simulations will be tailored to fit within the available computational resources. This may involve sampling smaller subsets of the dataset for training and validation, which, while still providing valuable insights, might limit the generalizability of the findings to larger or more complex datasets.

The timeline for this thesis project is another critical factor shaping its scope. With a finite period, available to move from conceptualization through to experimentation and analysis, certain limitations become necessary. For example, the comparative analysis of RBM and MLP models will focus on key performance metrics and configurations rather than an exhaustive exploration of all potential model variations and parameter settings. Additionally, the development and testing phases will be designed to fit within this timeframe, prioritizing depth over breadth in the investigation of these models' applicability to recommender systems.

Chapter 2: Current Scientific Landscape of Recommender Systems

Recommender systems have become an integral component in various industries, significantly enhancing user experience and business strategies. The seminal work by [1] Goldberg et al. on using collaborative filtering in the Tapestry system laid the foundation for modern recommender systems. Since then, the field has evolved, integrating advanced algorithms and machine learning techniques. As we explore the transition from traditional techniques to advanced models like RBM and MLP, this background serves as a critical guide, illuminating the path that has led to the current landscape of recommender systems and framing the context for the project at hand.

2.1 Fundamental Approaches

2.1.1 Collaborative Filtering

Collaborative filtering, a basis in the domain of recommender systems, harnesses the power of collective user behavior to generate personalized recommendations. This method started gaining attention after the important GroupLens project in 1994 [2], which operates on the principle that users with similar preferences in the past are likely to have similar tastes in the future. The technique is divided into two distinct categories: user-based and item-based collaborative filtering. The user-based approach focuses on finding users with similar preferences and recommending items they have liked, while the item-based method identifies items that are similar to those the user has previously favored, as elucidated in Sarwar et al.'s influential 2001 paper [3]. According to Ben Schafer et al. (2007) [4] Collaborative filtering can be useful in datasets where there are many items, many ratings per item, more users than items and users rate multiple items. The underlying meaning of the data should be that for each user of the community, there are other users with common needs or tastes, the item evaluation requires personal taste

and that items are homogenous. A challenge highlighted from Ben Schafer et al. is that in collaborative filtering is that the taste of the users should persist. This is a big challenge for domains like clothing, where fashion changes and user tastes from a few years ago might not be relevant. Another challenge is that they can suffer from the cold-start problem, where the system struggles to make a recommendation to a user that has no transaction data yet.

Overall, collaborative filtering is still being considered as the main approach when building recommender systems where many user-item interaction data are available.

2.1.1.2 Content-Based Filtering

Following the collaborative filtering paradigm, content-based filtering represents another foundational technique in the realm of recommender systems, prioritizing the attributes of items over user interactions. Pazzani and Billsus [5] describe that this method starts by creating user profiles according to given user preferences or history of interactions with items. Then, it utilizes the inherent properties of items (such as movie genres or authors of books) to align recommendations with a user's established preferences. By drawing on the similarities between items, content-based filtering proposes that users will favor items similar to those they previously enjoyed. One of the method's significant strengths lies in its capacity to provide personalized recommendations with limited data records, effectively navigating the cold-start problem that new users present by relying solely on item attributes. However, the cold-start problem persists in the user profiling procedure.

The efficacy of content-based filtering is inherently tied to the richness and comprehensiveness of the dataset describing item features. The approach presupposes a dataset filled with detailed, descriptive attributes of each item, enabling the algorithm to leverage similarities effectively. In scenarios where item metadata is sparse, content-based filtering may encounter limitations, struggling to expand beyond a user's explicit preferences and risking a constrained scope of recommendations. This underscores the crucial role of detailed item descriptions in the dataset, as the depth and accuracy of these

attributes directly influence the quality and diversity of the recommendations generated. Despite potential challenges associated with dataset limitations, content-based filtering remains an essential strategy in the recommender systems arsenal, particularly valuable in settings where extensive item metadata can be matched accurately to user profiles.

2.1.3 Hybrid Systems

Hybrid recommender systems merge collaborative and content-based filtering to enhance recommendation accuracy and diversity. As outlined by Burke [6], these systems employ various integration strategies, such as adding content-based features to collaborative models or combining separate recommendations. This approach leverages the depth of content-based item attributes with the broad appeal of user interactions from collaborative filtering, offering a nuanced solution to challenges like the cold-start problem and data sparsity. By blending these methodologies, hybrid systems achieve greater flexibility and adaptability, making them ideal for dynamic and evolving datasets commonly found in e-commerce. They represent a strategic evolution in recommender systems, capable of addressing a wider array of user preferences and scenarios.

2.2 Machine Learning in Recommender Systems

The advent of machine learning has significantly transformed the landscape of recommender systems, introducing sophisticated algorithms that learn from data to make predictions or decisions. This shift towards machine learning approaches has enabled recommender systems to achieve unprecedented levels of personalization and accuracy.

2.2.1 Matrix Factorization Techniques

One pivotal advancement is the application of matrix factorization techniques, which have become a staple in collaborative filtering. Koren et al. [7] highlighted the effectiveness of these techniques, particularly Singular Value Decomposition (SVD) and its adaptations, in capturing latent factors underlying user-item interactions. Matrix factorization models

decompose the user-item interaction matrix into lower-dimensional matrices, revealing latent features that represent underlying user preferences and item characteristics. This method enhances the system's ability to predict unseen interactions, providing a solid foundation for personalized recommendations.

2.2.2 Deep Learning Approaches

The integration of deep learning into recommender systems has marked a significant evolution in the field, enabling the modeling of complex, non-linear relationships between users and items. Deep learning approaches leverage layered architectures of neural networks to learn from vast amounts of data, capturing hidden yet important patterns and interactions that traditional algorithms might overlook.

Zhang et al. [10] provide a comprehensive survey on deep learning-based recommender systems, shedding light on the diverse architectures and strategies that have been explored to date. Their work underscores the versatility of deep learning in addressing various challenges inherent to recommendation tasks, including the cold start problem, scalability, and the need for personalization.

A notable application of deep learning in this domain is with Restricted Boltzmann Machines (RBM), which Salakhutdinov, Mnih, and Hinton [8] initially explored for collaborative filtering. RBMs are a type of neural network that can learn a probability distribution over its set of inputs, making them well-suited for uncovering latent structures within user-item interaction data, covered in detail later in this document.

Among the more innovative deep learning applications in recommendation systems is CoupledCF, developed by Zhang Q. et al [17]. By using a convolutional neural network (CNN) and integrating user and item features, this model can significantly enhance the recommendation accuracy compared to traditional recommendation systems.

2.3 Restricted Boltzmann Machines (RBM) in Recommender Systems

2.3.1 The RBM Landscape

Restricted Boltzmann Machines (RBM) have carved a niche in the domain of recommender systems, displaying their effectiveness in unraveling the complicated latent structures within user-item interaction data. The seminal contribution by Salakhutdinov, Mnih, and Hinton [8] set a standard in employing RBMs for collaborative filtering, illustrating their potential to refine the accuracy and relevance of recommendations significantly.

An RBM is a stochastic neural network comprising a layer of visible units, which correspond to observed data points like user ratings, and a layer of hidden units, which aim to capture latent factors influencing these interactions. The architecture facilitates the modeling of complex data distributions through a 2-layer network, enabling efficient training even without inter-layer connections.

The integration of RBMs in recommender systems is primarily valued for their ability to discern latent relationships that define user preferences and item attributes, thus predicting potential user-item interactions with a high degree of precision. This feature is crucial for personalizing recommendations in a manner that resonates with individual user tastes.

Additionally, RBMs are very good at managing sparse datasets (a common challenge in recommender systems) by probabilistically inferring missing entries, thereby offering plausible recommendations for new users or items with minimal interaction history. This capability is instrumental in addressing the cold-start problem, enhancing the system's utility from the outset.

The implementation of RBMs, while offering considerable benefits, also poses challenges, notably in parameter tuning and computational efficiency. Nevertheless, subsequent studies have built upon the foundational work of Salakhutdinov et al. [8], exploring the nuances of RBM applications. Xie et al. [13] expanded on this by incorporating user demographic information through a Conditional Restricted Boltzmann Machine (CRBM) approach, further personalizing recommendation systems. Yamashita et al. [14] delved into the debate on utilizing Bernoulli versus Gaussian units for RBMs in different data contexts, enhancing model flexibility. Ben Yedder et al. [15] focused on optimizing RBM for predictive accuracy in recommender systems, demonstrating the evolving landscape of deep learning techniques in this field.

Through these advancements, RBMs have underscored the transformative impact of deep learning on recommender systems, pushing the boundaries of what is achievable in terms of personalized content curation and user satisfaction.

2.3.2 Diving into RBMs functioning

Diving deeper into Restricted Boltzmann Machines (RBMs) and their functioning reveals a complex interplay of mathematical and algorithmic principles that enable these models to perform remarkably well in the context of recommender systems.

An RBM is constituted by a bipartite graph with two layers: a visible layer and a hidden layer. These layers are fully connected to each other but not within themselves, a restriction that simplifies the learning algorithm and makes it computationally feasible. The visible units correspond to input data (e.g., user ratings in recommender systems), while the hidden units aim to capture latent variables or factors that might not be explicitly observed but influence the input data's structure.

RBMs are energy-based models, where the configuration of the network (i.e., the states of

its visible and hidden units) defines the energy of the system. The energy function for a binary-state RBM is given by:

$$E(v, h) = - \sum_{i \in \text{visible}} a_i v_i - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

Here, a_i and b_j are the bias terms for the visible and hidden units, respectively, and w_{ij} represents the weight between visible unit i and hidden unit j . Lower energy states are more probable, and the network learns by adjusting its parameters (weights and biases) to minimize the overall energy.

Training a traditional Restricted Boltzmann Machine (RBM) involves learning the model parameters—the weights and biases—that best approximate the distribution of the input data. The core objective is to maximize the likelihood of the training data under the model, which is often done through gradient-based optimization methods.

The energy function is used to compute the joint distribution of the visible and hidden units, as well as the conditional probabilities of hidden units given visible units (and vice versa), which are essential for the learning algorithm:

- Joint distribution of v and h : $P(v, h) = \frac{1}{Z} e^{-E(v, h)}$
- Marginal distribution of v and h : $P(v, h) = \frac{1}{Z} \sum_{v, h} e^{-E(v, h)}$
- where Z is the partition function, a normalization factor computed as $Z = \sum_{v, h} e^{-E(v, h)}$

The learning algorithm commonly used for RBMs is Contrastive Divergence (CD), proposed by Hinton. It approximates the gradient of the log-likelihood with respect to the weights and biases. The update rules for the parameters are derived from the gradients:

- Update for weights w_{ij} : $\Delta w_{ij} = \epsilon (\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{reconstruction}})$
- Update for visible biases a_i : $\Delta a_i = \epsilon (\langle v_i \rangle_{\text{data}} - \langle v_i \rangle_{\text{reconstruction}})$

- Update for hidden biases b_j : $\Delta b_j = \epsilon(\langle h_j \rangle_{data} - \langle h_j \rangle_{reconstruction})$

where ϵ is the learning rate, $\langle \cdot \rangle_{data}$ denotes expectations under the distribution defined by the training data, and $\langle \cdot \rangle_{reconstruction}$ denotes expectations under the distribution defined by the model after a Gibbs sampling step starting from the training data.

In practice, the CD algorithm performs the following steps for each update:

1. Compute the positive phase by calculating $\langle v_i h_j \rangle_{data}$, which involves computing the activations of the hidden units given the visible units from the data.
2. Perform a Gibbs sampling step to obtain a sample of the visible units from the model distribution, starting from the data. This involves sampling the hidden units given the visible units and then resampling the visible units given these hidden units.
3. Compute the negative phase by calculating $\langle v_i h_j \rangle_{reconstruction}$ using the sampled visible units from step 2.
4. Update the weights and biases according to the computed gradients.

In recommender systems, the visible units of an RBM can be designed to represent user ratings for items, with each unit corresponding to a specific item's rating. The hidden units capture latent factors that influence user preferences, such as genres or themes in movies. By learning these latent factors, RBMs can predict unseen ratings, filling in the gaps in the user-item matrix and thus providing personalized recommendations.

2.3.3 Multinomial RBMs

While traditional RBMs and their binary units are well-suited for datasets where variables are binary or have been binarized, Multinomial RBMs extend this framework to handle multinomial data directly. This is particularly useful in recommender systems where the ratings can take on multiple discrete values.

Multinomial RBMs are designed with visible units that can take on multiple states to represent multinomial data. For instance, in a movie rating system where ratings range from 1 to 5 stars, each visible unit can represent the rating given by a user to a movie. The energy function for Multinomial RBMs is adapted to account for the multiple states of the visible units. This adaptation involves summing over all possible states of the visible units to calculate the partition function, which is used to normalize the probabilities.

The energy function for a Multinomial RBM configuration is:

$$E(v, h) = \sum_i \sum_k a_{ik} v_{ik} - \sum_j b_j h_j - \sum_{i,j} \sum_k v_{ik} h_j w_{ijk}$$

where:

- v_{ik} represents the state k of visible unit i (for instance, the rating of a movie),
- h_j are the binary states of hidden unit j ,
- a_{ik} are the bias terms for each state k of visible unit i ,
- b_j are the bias terms for the hidden units,
- w_{ijk} are the weights connecting state k of visible unit i to hidden unit j .

Multinomial RBMs can directly model the distribution of ratings, capturing the nuances in user preferences more effectively than binary models. This leads to a richer understanding of user-item interactions.

By accurately modeling the actual distribution of ratings, Multinomial RBMs can potentially offer more precise recommendations, improving user satisfaction with the recommender system

2.4 Multi-Layer Perceptrons (MLP) in Recommender Systems

Multi-Layer Perceptrons (MLP), fundamental to the deep learning toolkit, have been widely adopted in recommender systems for their ability to model complex and non-linear relationships between users and items. By employing multiple layers of neurons, MLPs (Multi-Layer Perceptrons) can learn deep representations of data, enhancing the system's capacity to generate personalized recommendations.

Zhang et al. [10] highlight the versatility of MLPs in recommendation tasks, noting their efficacy in capturing intricate user-item interaction patterns. This capability makes MLPs particularly useful as a baseline model for benchmarking the performance of recommender systems. By establishing a performance threshold with MLPs, researchers and practitioners can evaluate the effectiveness of novel algorithms or hybrid models against this established benchmark, ensuring that new methodologies offer tangible improvements over traditional deep learning approaches.

MLPs serve not only as a powerful modeling tool but also as a critical benchmark in the continual quest to enhance recommender systems. Their straightforward architecture and the ability to process high-dimensional data make them an ideal starting point for comparing different machine learning techniques in recommendation contexts. However, the simplicity of MLPs may also limit their ability to handle certain recommendation-specific challenges, such as the cold-start problem or capturing time-dependent dynamics without additional modifications or hybridization with other models.

2.4.1 Neural Collaborative Filtering

The Neural Matrix Factorization (NeuMF) model, introduced by Xiangnan He et al. [16], represents a significant advance in the use of MLPs in recommender systems. This model merges the traditional matrix factorization techniques with multi-layer perceptrons to create a powerful hybrid approach that captures the nonlinear interactions between users and items more effectively than traditional methods alone.

NeuMF is built on the premise that the interaction between user and item features can be modeled more precisely through a neural network framework than by mere dot products, as in classical matrix factorization. The architecture of NeuMF consists of two main components: a Generalized Matrix Factorization (GMF) model and a Neural Collaborative Filtering (NCF) model. The NCF model is an MLP with embeddings as inputs. The GMF model handles the linear part of the user-item interaction, while the MLP captures the complex and non-linear relationships.

The integration of these two components is typically achieved via a concatenation layer that combines the last hidden layer of the MLP with the output of the GMF. This concatenated output is then passed through a final prediction layer, which is usually a single neuron with a sigmoid activation function to predict the interaction probability.

Training the NeuMF model involves optimizing a binary cross-entropy loss, which effectively measures the discrepancy between the predicted and actual interactions in the training data. This loss function is particularly suited for recommendation tasks where the goal is to predict a 'like' or 'dislike' (often encoded as 1 or 0).

One of the key strengths of NeuMF is its flexibility in learning from implicit feedback (e.g., clicks, views) rather than explicit ratings. This makes it well-suited for scenarios where explicit feedback is sparse or unavailable.

Since its introduction, the NeuMF framework has been adapted and extended in several ways. For example, researchers have incorporated attention mechanisms to weigh different parts of the interaction differently or have added recurrent layers to better capture the sequence of interactions over time, enhancing the model's ability to handle dynamic user preferences and temporal effects.

NeuMF has shown substantial improvements in recommendation quality across diverse domains, from e-commerce to multimedia services, proving its effectiveness over traditional collaborative filtering techniques. Its ability to integrate with other data sources and to be customized for specific contextual information further enhances its applicability and performance in real-world scenarios.

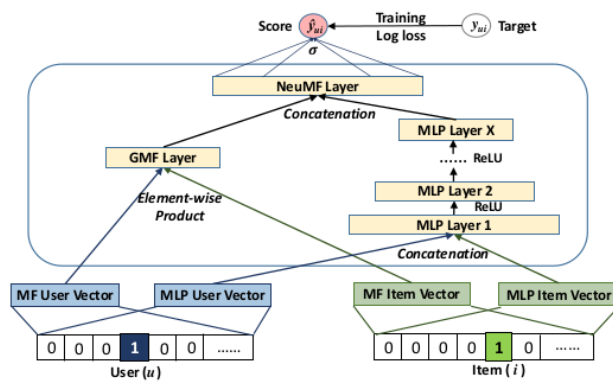


Figure 3: Neural matrix factorization model

Figure 1: Figure 3 from He et al. paper "Neural Collaborative Filtering" [16], showcasing the architecture of the Neural matrix factorization model

2.5 Challenges

The integration of machine learning into recommender systems, while transformative, presents a unique set of challenges that drive ongoing research and innovation in the field.

Scalability and Computational Resources: As recommender systems aim to handle increasingly large and complex datasets, the computational demands rise correspondingly. Training sophisticated models like MLPs or RBMs requires substantial computational power, often necessitating advanced hardware or distributed computing environments [7,8].

Cold-Start Problem: The issue of making accurate recommendations for new users or items with little to no interaction history remains a significant challenge. While methods like content-based filtering and hybrid models offer potential solutions, fully overcoming this hurdle continues to be an area of active research [6].

Data Sparsity and Quality: Many recommender systems suffer from sparse datasets, where most potential user-item interactions are not observed. Additionally, the quality of available data, including the richness of item descriptions and the reliability of user feedback, can vary greatly, affecting the system's performance [4,5].

Model Interpretability and User Trust: Ensuring that the recommendations provided by machine learning models are interpretable and transparent is crucial for building user trust. This is particularly challenging with complex models like deep neural networks, where the decision-making process is not always clear [10].

2.6 Summary

This chapter has charted the progression of recommender systems from their initial collaborative and content-based filtering roots to the advanced machine learning

frameworks that now define the field. Through the foundational insights of Goldberg et al. [1], Resnick et al. [2], Sarwar et al. [3], and the innovative applications of matrix factorization by Koren et al. [7], as well as deep learning explorations by Zhang et al. [10], we gain a comprehensive understanding of the mechanisms driving personalized recommendations.

The integration of hybrid systems, particularly highlighted by Burke [6], underlines the importance of combining different methodologies to surmount the limitations inherent to any single approach. This multifaceted landscape, enriched by the machine learning advancements of matrix factorization and deep learning models like MLPs and RBMs, displays the relentless quest for more refined, accurate, and user-centric recommendation systems.

However, the journey is punctuated by challenges (scalability, data sparsity, and the quest for interpretability) that remind us of the complexity involved in crafting effective recommender systems.

This thesis positions itself within this dynamic narrative, aiming to leverage the discussed methodologies and insights to implement a flexible, data-agnostic recommender system based on RBM and MLP. The literature underscores the importance of addressing data sparsity and ensuring model scalability, which are crucial considerations for my project.

Furthermore, the exploration of RBMs by Salakhutdinov, Mnih, and Hinton [8], along with the subsequent studies by Xie et al. [13], Yamashita et al. [14], and Ben Yedder et al. [15], provides a valuable framework for understanding the potential and challenges of employing such models in recommender systems. This background not only informs the technical implementation of the project but also highlights the need for careful consideration of model parameters, training strategies, and the integration of contextual information to enhance the recommendation process.

Similarly work by Xiangnan He [16], Zhang Q [17] reveal that MLPs can solve recommendation problems effectively. Zhang Q [17] and Rehman [20] also suggest that incorporating contextual information in the MLP model can enhance the recommendation quality.

Chapter 3: Dataset Description

3.1 Overview of H&M Personalized Fashion Recommendations Dataset

The dataset chosen for this study is the H&M Personalized Fashion Recommendations dataset, made available through a competition hosted on Kaggle. This dataset is particularly suited for building and evaluating recommender systems in the retail fashion domain. It includes diverse data types drawn from H&M Group's extensive online and retail store operations, which encompasses 53 online markets and about 4,850 stores globally. The dataset is primarily composed of transaction records, customer metadata, product metadata, and images, facilitating a comprehensive exploration of user preferences and product characteristics.

3.1.1 Dataset Composition

The dataset is structured into several key components:

Transaction Data: At the heart of the dataset are over 32 million transaction records, providing a deep well of implicit feedback on customer preferences. Each transaction record encapsulates the essence of consumer behavior through user IDs, product IDs, purchase timestamps, and normalized prices. This data layer is instrumental for analyzing purchasing patterns and inferring user preferences.

User Data: Enhancing the dataset's utility for personalized recommendation systems is the inclusion of demographic information about the users. Details such as age and postal code enable a nuanced segmentation of the customer base, allowing for tailored recommendations that resonate with specific demographic groups.

Product Catalog: The dataset's richness is further augmented by an exhaustive product catalog, detailing product IDs, names, descriptions, and a plethora of categorical attributes such as color, size, and style. This detailed product information is crucial for content-based filtering and understanding the attributes that may influence purchasing decisions.

3.1.2 Characteristics and Considerations

Descriptive Statistics of the Transactions Dataset:

- **Number of unique customers:** 1,362,281
- **Number of unique articles:** 104,547
- **Total number of transactions:** 27,306,439
- **Average transactions per customer:** 20.04
- **Average transactions per article:** 261.19

Distribution of transactions per customer:

- **Standard Deviation:** 31.98
- **Min:** 1
- **25th Percentile:** 3
- **50th Percentile (Median):** 8
- **75th Percentile:** 24
- **Max:** 1,346

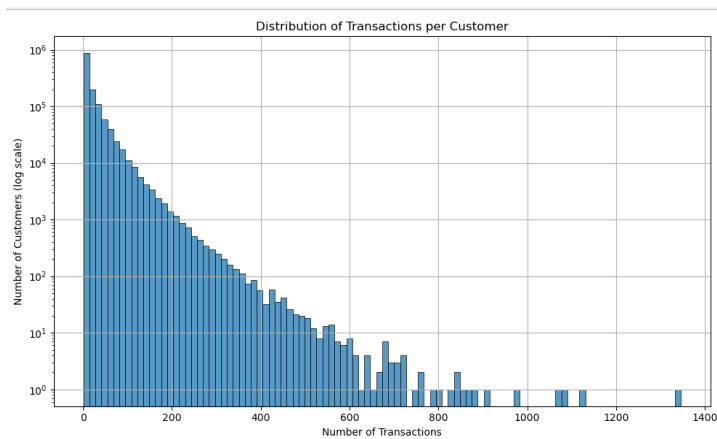


Figure 2: Distribution of Transactions per Customer. This histogram illustrates the distribution of the number of transactions per customer in the dataset. The x-axis represents the number of transactions made by individual customers, while the y-axis, displayed on a logarithmic scale, shows the number of customers who made that many transactions.

The chart reveals that most customers have a low number of 5-10 transactions, with a few outliers having significantly higher transaction counts.

Distribution of transactions per article:

- **Standard Deviation:** 637.91
- **Min:** 1
- **25th Percentile:** 12
- **50th Percentile (Median):** 58
- **75th Percentile:** 250
- **Max:** 32,251

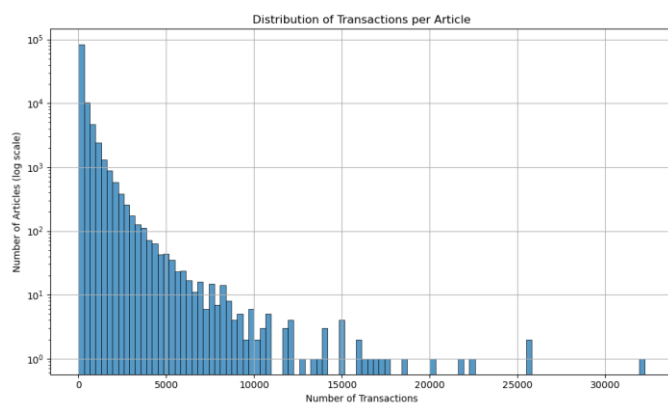


Figure 3: Distribution of Transactions per Article. This histogram shows the distribution of the number of transactions per article in the dataset. The x-axis indicates the number of times each article was purchased, and the y-axis, displayed on a logarithmic scale, represents the number of articles with that many transactions. The distribution highlights that most articles are purchased relatively few times 200-300, with some articles experiencing a very high number of purchases.

Volume and Variety: The large size of the dataset and its diverse range of data types (from transactional records to textual descriptions) presents an opportunity for data-driven insights. However, this also introduces challenges in managing and processing the data effectively.

Sparsity: A common hurdle in large datasets is the sparsity of user-item interactions. Despite the voluminous transaction records, the vast array of products means that any single user interacts with only a small fraction of the inventory, posing challenges for collaborative filtering approaches. The dataset has a sparsity of 0.9998, indicating that only 0.02% of the possible user-item interactions are observed.

Implicit Feedback: The dataset's reliance on purchase history as a form of implicit feedback, rather than explicit ratings or reviews, demands specialized modeling techniques. Unlike explicit feedback, implicit signals do not provide direct indications of dislike, requiring careful interpretation and handling in the recommendation algorithm.

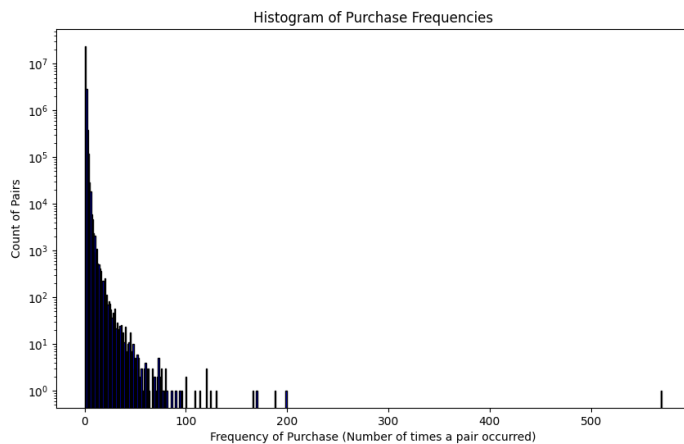


Figure 4: Histogram of Purchase Frequencies with Logarithmic Scale. This graph illustrates the number of occurrences for various purchase quantities within the transaction's dataset. The x-axis indicates the number of times a specific product was bought by a customer, while the y-axis, on a logarithmic scale, shows the total number of such purchase instances observed. Notably, the plot reveals several outliers where products were purchased an exceptionally high number of times, deviating from the common purchasing pattern.

Diverse types of customers: By reviewing Figure 4 we can see that the dataset consists of diverse types of customers. Although there are mostly transactions with 1 or 2 quantities,

there are many transactions with more than 10 quantities. This could mean that we have customers that buy in bulk, potentially drop shippers or resellers.

3.1.3 Temporal Dynamics

Seasonality and Trends: The dataset provides a historical record of purchases that can be analyzed for seasonality and trends over time. Understanding these temporal dynamics can be critical for making timely recommendations, adjusting inventory levels, and planning marketing campaigns.

Product Lifecycle: The inclusion of transaction timestamps allows for the study of product life cycles, from launch to clearance. This can inform strategies for recommending new items versus established products and managing the promotion of end-of-life products.

3.1.4 Missing Values

The dataset is reflective of real-world complexities, one of which is missing data. A notable example within the user demographic information is the age attribute. Missing age data can hinder the ability to fully personalize the shopping experience, as age is often a significant factor in fashion preferences.

To mitigate the issues arising from incomplete data, we fill the missing values with the average value found in the dataset.

3.2 Competition Winners

The winning methodology employed by SENKIN13 in the H&M Personalized Fashion Recommendations Kaggle competition involved a strategic approach to candidate generation, feature engineering, model selection, and optimization. The key components of their approach included detailed feature engineering, model selection, data augmentation, downsampling, optimization, and a robust machine setup.

In terms of feature engineering, SENKIN13 focused on user-item interaction by analyzing user purchase frequency, recency, and item characteristics. They included temporal features such as the first and last transaction days and aggregated features for different periods like weeks, months, and seasons. Additionally, they incorporated aggregation features, such as the mean, max, and min of customer demographics, as well as difference and ratio features that captured the variations between various attributes. To enhance similarity measures, they utilized collaborative filtering scores and cosine similarity for both item-to-item and user-to-item comparisons.

For model selection, the primary model used was LightGBM, which achieved a cv score of 0.0441 and an lb score of 0.0367. CatBoost was also employed for handling categorical features, although it was less effective than LightGBM. An ensemble approach was adopted, combining five LightGBM and seven CatBoost classifiers, which resulted in a final lb score of 0.0371.

Data augmentation and downsampling played a crucial role in their methodology. Various retrieval strategies were employed to increase the pool of relevant candidate items. To balance the dataset, they selected 1-2 million negative samples per week through negative downsampling.

Optimization techniques were pivotal to their success. They improved inference speed by using TreeLite for LightGBM, achieving a twofold speed increase, and by running CatBoost on a GPU, resulting in a 30-fold speed improvement. Memory optimization was achieved by transforming categorical features into label encoding and applying memory reduction techniques. They also utilized distributed processing by splitting users into 28 groups for simultaneous inference across multiple servers, employing high-memory Google Cloud Platform (GCP) instances in the final week.

The machine setup for their operations included a desktop with 128GB RAM, a 64-core CPU, a TITAN RTX GPU, and GCP instances with up to 300GB RAM. This combination of advanced feature engineering, model selection, data augmentation, and optimization

techniques enabled the team to effectively model complex interactions, leading to their first-place performance in the competition.

.

Chapter 4: Methodology

4.1 Data Preprocessing

The preprocessing pipeline is designed to refine and transform raw e-commerce transaction data into an optimized format for model input. This section details the purpose and methodology of each step, highlighting how they collectively contribute to preparing the data for effective GMF, NCF, feature accommodated NCF and RBM networks training.

4.1.1 Shrinking the Dataset

Since we have a dataset containing 32 million transactions, we need to shrink it to fit our limitations. Since we have limited computational resources a shrink of the dataset could speed up the data preprocessing and network training process, as well as minimizing the computer memory we will eventually need for the referred procedures. The substantial size of our data set permits a significant reduction in its volume without negatively impacting the accuracy of our predictions. By retaining only a part of the dataset, we still have access to thousands of transactions, which is more than sufficient for training a neural network effectively. However, it's crucial to carefully select which part of the dataset to retain. By focusing on the part with the most active customers and most frequently purchased products, we ensure the dataset remains high in quality and density, making it an ideal input for our neural network.

First, we will retain the top 1% of most active customers and best seller products of the dataset, which will be computationally lightweight and allow us to tune our models within the thesis's time constraints. This subset will also help us explore how our models perform in denser (less sparse) datasets.

Second, we will retain the top 10% of most active customers and best seller products of the dataset. This reduction is necessary due to our hardware limitations and will enable us

to examine how our models behave in sparser datasets and how they scale when applied to larger data problems.

Step 1: Transactions Count and Organization

The initial step involves counting the purchases for each customer-article pair, keeping only the most recent transactions. This action groups the user-item transaction pairs reducing redundancy and adds a “purchase_count” column for each pair.

Customer X's Transactions		
1	Product A	15/05/2024
2	Product A	27/05/2024
3	Product B	28/05/2024

Customer X's Transactions			
1	Product A	27/05/2024	2
2	Product B	28/05/2024	1

Figure 5: Compression of transaction data by retaining only the latest transaction date for each user-item pair and counting the total purchases

After this data compression we are left with 27.3 million transactions.

Step 2: Enriching Customers with Distinct Purchases

Customers are enriched with data indicating the number of distinct articles they have purchased. This step identifies the highly engaged customers, allowing us to pick a subset of dataset with the top customers.

Customers	
1	Customer X
2	Customer Y
3	Customer Z

Customers		
1	Customer Y	7
2	Customer X	2

Figure 6: Customers enriched with the count of distinct articles purchased, highlighting highly engaged customers for further analysis.

Step 3: Filtering Top K% Customers

The dataset is narrowed down to the top K% of customers based on their purchasing diversity. This focus on active customers shrinks the dataset, reducing the sparsity of the dataset and training complexity.

Step 4: Filtering Articles by Top K% Customers

Articles are then filtered to include only those purchased by the top K% customers. This ensures the model focuses on products that have a broad appeal to the most engaged segment of the customer base.

Step 5: Enriching Articles with Customer Purchase Count

Articles are further enriched by counting the distinct customers who purchased each product. This metric provides insight into the popularity and reach of products within the active customer base, allowing us to identify the best-selling products.

The diagram illustrates the process of enriching a products table. On the left, a table with two columns (Index and Product Name) is shown. An arrow points to a second table on the right, which has three columns (Index, Product Name, and Customer Count). The data in the second table is derived from the first table, with the count of distinct customers added for each product.

Products	
1	Product A
2	Product B
3	Product C

Products		
1	Product A	6
2	Product B	5

Figure 7: Products enriched with the count of distinct customers, highlighting the best-selling items.

Step 6: Filtering Top K% Articles

The articles dataset is refined to include only the top K% of products, based on customer purchase count. This selection prioritizes articles with wider reducing the size and sparsity, while increasing the quality of the dataset.

Step 7: Compact Transactions Filtering

Transactions are filtered to retain only those involving the top K% customers and articles. This step ensures the model is trained on the most relevant and impactful customer-product interactions.

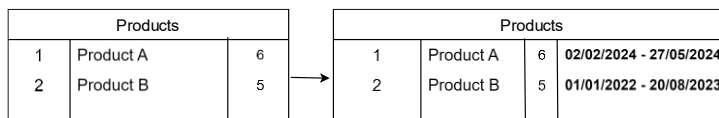
Steps 8: Recalculating Metrics in Filtered Dataset by repeating steps

With the dataset now focused on the top K% of customers and articles, distinct product purchases by customers and distinct customer purchases per product are recalculated to reflect the refined dataset.

4.1.2 Calculate the User-Item Affinity Score

Step 9: Append Article Availability Periods

Each product is supplemented with information on its start and end dates of availability. This temporal data is crucial for accurately modeling when products were accessible to customers, which directly influences subsequent steps, particularly in identifying suitable negative samples.



The diagram illustrates the process of adding availability dates to a product table. It shows two tables. The first table, titled 'Products', has three columns: an ID column with values 1 and 2, a name column with values 'Product A' and 'Product B', and a quantity column with values 6 and 5. An arrow points to the second table, which is also titled 'Products' and has four columns. The first three columns are identical to the first table, but a fourth column has been added containing the availability date ranges: '02/02/2024 - 27/05/2024' for Product A and '01/01/2022 - 20/08/2023' for Product B.

Products		
1	Product A	6
2	Product B	5

Products			
1	Product A	6	02/02/2024 - 27/05/2024
2	Product B	5	01/01/2022 - 20/08/2023

Figure 8: Products supplemented with start and end dates of availability, crucial for modeling product accessibility

For the following steps we will adopt a method that centers data around each user's or item's unique purchasing behavior, compensating for variations akin to differing rating scales highlighted by Schafer et al. [4]. This method adjusts for each user's and item's average purchase frequency, similar to how one might adjust for a user's average rating, ensuring a consistent baseline for comparison across varying purchasing behaviors.

Step 10: Determining Customer Quantity Purchasing Behavior

We calculate the median number of each product a customer buys. This step helps distinguish between bulk buyers like drop shippers and regular customers based on their purchase patterns.

Consider a scenario where a drop shipper typically purchases items in bulk, such as 10 units of a product per transaction, but buys only 3 units of a new product. This reduced quantity might suggest the product doesn't meet their usual criteria for bulk buying.

Conversely, a regular customer who usually buys one unit of a product but purchases 3 units of the same product indicates a strong preference. This differentiation is crucial for understanding customer satisfaction and product preference.

Step 11: Assessing Product Purchase Quantity Rates

We analyze the median purchase quantity for each product across all customers. This helps identify if a product is commonly bought in bulk or singly, which can vary significantly across product categories.

In evaluating product purchase rates, let's contrast two types of products: furniture and socks. Furniture, due to its higher price point and long-term use case, is typically purchased once. If a customer buys a piece of furniture multiple times, it could indicate an exceptional preference or a need for multiple units (e.g., outfitting a new home or office). However, the same quantity of purchases applied to socks (a product usually bought in multiples due to their everyday use and wear-and-tear) would have a different interpretation.

Step 12: Establishing the Test Set Split Date

For each customer, we identify a specific split date to divide transactions into training and test sets, following an 80-20 split. This division is based on temporal data, treating the test set date as the current reference point for model predictions. Transactions including and prior to this date are used for training, while those after are earmarked for prediction in the test set, ensuring a dynamic and personalized approach to data splitting.

Step 13: Tracking Weeks Since Last Purchase

We calculate the time elapsed, in weeks, between the most recent purchase date and the test set date for each customer-article combination. This metric is pivotal for refining the user-item affinity score, providing temporal context to each interaction.

Step 14: Aggregating Purchase Behavior Metrics

For each transaction, we compute the mean of the purchase quantities previously determined in Steps 10 and 11. This average encapsulates both the customer's buying habits and the product's typical purchase rate, laying the groundwork for the forthcoming calculation of the user-item affinity score.

Step 15: Computing Z-Scores for Transactions

Z-scores are calculated to measure the deviation of a specific purchase quantity from the established average. This statistical metric helps to quantify the degree of preference (or lack thereof) a customer has for an item, based on their purchase history.

The Z-score is a statistical measure that indicates the number of standard deviations a given value is from the mean. For our case, this translates into understanding how much a particular customer-article purchase quantity deviates from the expected purchasing pattern.

The Z-score is computed using the traditional formula, however with alternations in the meaning of μ and σ :

$$Z = \frac{(X - \mu)}{\sigma}$$

Where:

- **Z** is the Z-score,
- **X** is the purchase frequency of user-item relationship,
- **μ** is the mean of the means (calculated in step 14) between the purchasing quantity customer behavior for all articles (from Step 10) and the purchase quantity rate for that article across all customers (from Step 11).
- **σ** is the standard deviation of purchase quantities for the article across all customers and for all articles purchased by the customer.

In our case for instance, if a customer has a history of purchasing any article in quantities averaging to 3 units and the article is usually purchased once (1), It means that $\mu = 2$ (mean (3,1)). Let's assume a standard deviation of 0.5 units.

We notice that the customer purchased in his lifetime 4 units ($X=4$) of this product, the Z-score for this transaction would be:

$$Z = \frac{(4 - 2)}{0.5} = 4$$

This Z-score of 4 indicates that the purchase quantity is significantly higher than the average, suggesting a strong preference for the product at that time. Conversely, a Z-score close to 0 would indicate that the purchase quantity is very close to the average, suggesting a standard transaction without strong deviation from the norm.

Top of Form

Step 16: Affinity Score Formulation

In this critical step, lies the core of my thesis on converting the implicit transaction data into an affinity score. We calculate the affinity score for each user-item interaction. This score is a key metric that synthesizes both the frequency and recency of purchases into a unified preference indicator.

The formula for calculating the affinity score is as follows:

$$\text{Affinity Score} = 1 + \alpha \times Z + \beta \times \frac{1}{\log_2(\text{Weeks Since Last Purchase} + 2)}$$

Where:

- The $\alpha \times Z$ part can be assumed to be the **score** based on the **purchasing behavior**.
- The $\beta \times 1/\log_2(\text{Weeks Since Last Purchase}+2)$ part can be assumed to be the **recency score**.
- **Z** is the Z-score representing the normalized purchase frequency, highlighting how much a user's purchase quantity for an item deviates from their average behavior and the item's behavior.

- **Weeks Since Last Purchase** quantifies the recency of the purchase, with a "+2" in the denominator to ensure the formula remains defined even for recent purchases (where the value might be 0).
- α and β are tunable parameters that allow the model to be adjusted according to the specific characteristics of the dataset, ensuring that both frequency and recency contributions can be balanced appropriately.

Rationale Behind the Formula:

1. **Comprehensive User Preferences:** The formula integrates both frequency (through the Z-score) and recency (through the weeks since the last purchase), offering a holistic view of user preferences. This dual consideration ensures that both consistent purchasing patterns and the timeliness of those purchases are factored into the model's understanding of user-item affinity.
2. **Normalization Across Users:** By utilizing the Z-score for purchase frequency, the formula normalizes user behavior, allowing for equitable comparisons across users with varying purchasing habits. This normalization is crucial for identifying true preferences in datasets with diverse user profiles.
3. **Weighted Recency:** The inverse logarithm applied to the recency of purchases ensures that recent interactions are weighted more significantly than older ones, reflecting the evolving nature of user preferences. However, this weighting is carefully moderated to prevent recent purchases from disproportionately influencing the affinity score, maintaining a balanced perspective on user behavior over time.
4. **Model Fine-tuning:** The inclusion of α and β parameters offers flexibility in adjusting the influence of purchase frequency and recency, respectively. This adaptability is essential for tailoring the model to best reflect the nuances of the specific dataset being analyzed, optimizing performance across a variety of e-commerce contexts.

User – Item pairs found in the test set have no tangible “Weeks since last purchase” number, as the purchases were made in the future and they are to be predicted, by the recommender system. The affinity score in the test set is irrelevant and is not used to evaluate the model’s performance. Nonetheless, we can calculate the affinity score using a slightly modified formula, to get a sense of the direct test error of the prediction by any trained model. The difference between the modified formula that can be used in the test set and the above formula is that the recency score is replaced by the average result of all user – items recency score.

The adoption of this formula stems from the necessity to capture the dynamic and multifaceted nature of user-item interactions within e-commerce platforms. By quantitatively modelling both the frequency and recency of purchases, the affinity score becomes a robust indicator of user preferences, capable of informing personalized recommendation systems with a high degree of accuracy and relevance.

Step 17: Incorporating Negative Samples

In this step we integrate negative sampling, a crucial technique in enhancing the training data for our model. This technique, noted in the research by He et al. [16], involves incorporating examples of non-purchase interactions. For each user, we look at the array of products that, while available, were not purchased by them during their active shopping timeline. From this pool, we randomly select a proportionate number of non-purchased items relative to their actual transactions.

These randomly selected, non-purchased items are then assigned an affinity score using the formula $\text{Affinity Score} = 1 - \delta$, where δ is a parameter greater than 0. This scoring reflects the absence of a positive interaction between the user and the item, suggesting a lower likelihood or preference towards these products. The choice of δ allows for

adjusting the degree of negative bias introduced, offering a means to control the influence of these negative samples on the model's learning process.

This process is crucial for training the neural network to distinguish between items that users have chosen versus those they have ignored, thereby enhancing the model's predictive accuracy.

Step 18: Normalizing the Affinity Score to k Classes

To prepare affinity scores for the Bernoulli input Multinomial RBM, we get the distribution ranking by percentile of these continuous scores, to normalize the outliers. Normalizing the scores directly led to having most of the scores near the middle and a few outliers at the edges of the scores, resulting to a not useful distribution. Then we normalize the distribution ranked by percentile to an integer scale from 1 to k , using a linear scaling formula that maps the full range of scores into this discrete set. This normalization not only facilitates computational efficiency by reducing the complexity of data processing but also enhances the interpretability of the model's outputs. The choice of integer values is particularly suited for the next steps in feeding data into the RBM, ensuring compatibility with the Bernoulli input Multinomial model's requirements and simplifying the representation of user-item interactions.

After shrinking the dataset and adding the negative samples, we are left with the 2 subset datasets with the following characteristics:

Description of Top 1% Dataset:

- **Sparsity of the dataset:** 0.9710
- **Number of unique customers:** 13,710
- **Number of unique articles:** 1,055
- **Total number of transactions:** 419,222
- **Average transactions per customer:** 30.58
- **Median transactions per customer:** 29.00
- **Average transactions per article:** 397.37
- **Median transactions per article:** 356.00

On average each customer is left with a pool of 530 products to choose from after the test date. On average the customers purchase about 5.50 articles in their test set period. Our Recommender system should aim to predict the purchase of those 5.50 articles of each customer out of the pool of 530 products.

Description of Top 10% Dataset:

- **Sparsity of the dataset:** 0.9923
- **Number of unique customers:** 137,200
- **Number of unique articles:** 10,559
- **Total number of transactions:** 11,128,166
- **Average transactions per customer:** 81.11
- **Median transactions per customer:** 70.00
- **Average transactions per article:** 1,053.90
- **Median transactions per article:** 923.00

On average, each customer is left with a pool of 6500 products to choose from after their test date. During the test period, customers typically purchase about 14.6 articles. Therefore, our recommender system should aim to accurately predict the purchase of these 14.6 articles for each customer from a pool of 6500 products.

4.1.3 Preparing the Input Vectors for the MF, GMF and NCF

For the MF (Matrix Factorization), GMF (Generalized Matrix Factorization) and NCF (Neural Collaborative Filtering MLP) models, we prepare our input vectors using the divided train and test datasets. We employ one-hot encoding for both users and items to feed into the respective embedding layers of our models. The target variable for these inputs is the affinity score, which serves as an indicator of user preference.

To assess each model's performance accurately, we compile an additional dataset. This dataset comprises all items that were active beyond our designated test period and were

not present in the training set. This ensures that our evaluation reflects the model's ability to predict preferences for new, previously unseen items.

4.1.4 Preparing the Input Vectors for the RBM

To align with the Multinomial RBM's, Bernoulli input requirements, we structure a matrix with each input vector (row) to represent a customer, where elements (columns) correspond to one-hot encoded affinity ratings across k classes for each item. For example, a rating of 4 out of 5 translates into a vector $[0,0,0,1,0]$, with the position of '1' indicating the rating class. A vector with all zeros indicates no interaction. Due to the extensive size and sparsity of our dataset (characterized by many transactions but limited interactions per customer-item pair) we use a sparse matrix in the Compressed Sparse Row (CSR) format. This setup ensures that each customer's interactions are compactly represented, preserving the data's richness while accommodating the RBM's computational constraints.

4.1.5 Preparing the Input Vectors for the feature accommodated NCF

In developing the input vectors for an NCF model that accommodates additional features, we leverage the encoded user and item data prepared for the GMF and NCF model inputs. To this foundational dataset, we append key demographic information—specifically, the customers' age, which has proven insightful for personalization.

For the items, we enhance the input vectors with a carefully selected set of features that provide significant insight without overburdening our computational resources. These features include the product type, graphical appearance, color group, department, and index group. The selection of these features is strategic, chosen for their potential impact on user preference while considering the constraints of our hardware and the project timeline.

4.2 Overview of decisions across the models' implementation

In this section, we detail the strategic decisions that apply to all models used in this thesis, ensuring a consistent approach across different machine learning techniques in our recommender system. Each decision is substantiated with relevant literature, highlighting the rationale behind our choices.

To optimize data handling in our recommender systems, we've chosen the Parquet file format for its efficient columnar storage, which is ideal for handling large datasets. Parquet offers high compression and fast read/write operations, crucial for the voluminous data involved in training and evaluating machine learning models. This format is especially beneficial when used in conjunction with Pandas, a robust Python library for data analysis. Pandas provide powerful data manipulation capabilities that, when paired with Parquet, facilitate efficient data operations and minimize memory usage. Additionally, we leverage Dask for its ability to handle large datasets in parallel, complementing Pandas by enabling scalable data processing across multiple cores. This strategic choice ensures scalability and efficiency in our data processing tasks, crucial for the dynamic requirements of neural network training.

We have selected the PyTorch framework for its dynamic computation graph and efficient memory usage, which are critical for the rapid prototyping of deep learning models. This framework's flexibility and ease of use in defining complex models.

PyTorch allows us to utilize CUDA which enables acceleration of deep learning algorithms by leveraging GPU hardware. This approach is essential for handling the computationally intensive tasks of training neural networks.

We are using the Adam optimizer due to its effectiveness in handling sparse gradients on noisy problems, as Kingma and Ba describe: "Adam is computationally efficient, has little memory requirement, invariant to diagonal rescale of the gradients, and is well suited for

problems that are large in terms of data and/or parameters" [18]. This makes Adam particularly suitable for the large-scale data sets typical in recommender systems. In optimizing our neural network training processes, we have implemented a learning rate scheduler that reduces the learning rate upon detecting a plateau in model performance, as supported by A. Khodamoradi, et al. [19]. According to their paper, this dynamically manages the learning rate to stabilize training and prevent prolonged inefficiency in updates.

The selection of evaluation metrics at K (such as precision@k, recall@k, NDCG@k, MAP@k and F1 Score@k) is pivotal for assessing the performance of recommender systems. These metrics are widely used in the literature to measure the relevance of recommended items within the top K suggestions, reflecting user satisfaction and system effectiveness. Zhang et al. provide a comprehensive overview of these metrics in their survey on deep learning-based recommender systems, validating their importance in measuring the quality of recommendations [10].

4.3 Implementation of the Traditional Matrix Factorization Model Class

For the implementation of any Neural Network in this study we extend the following BaseNetwork abstract class (Figure 36 in Appendix), which extends the torch.nn.Module class and contains lists that hold the evaluation metrics after each epoch.

Matrix Factorization (MF) is a foundational machine learning model used in collaborative filtering for recommender systems. Matrix Factorization will be used as the robust baseline in my study due to its simplicity and effectiveness, allowing for straightforward interpretation and efficient computation. It serves as an ideal starting point for evaluating more complex models or hybrid approaches in collaborative filtering.

In Matrix Factorization, the user-item interaction matrix is decomposed into two lower-dimensional latent factor matrices, one for users and one for items. Each user and each

item are associated with vectors (embeddings) in a shared latent space of dimension `embed_dim`. The interaction between a user and an item is predicted by taking the dot product of their respective latent vectors, representing the user's overall interest in the item's characteristics and capturing the interaction strength or preference.

In my implementation, MF (Figure 37 in Appendix) is defined as a class extending `BaseNetwork` with the following extensions:

- **User and Item Embeddings:** `nn.Embedding` layers create embeddings for users and items. The dimension of these embeddings, `embed_dim`, is a critical hyperparameter that defines the size of the latent space.
- **Forward Pass:** In the forward method, embeddings for users and items are retrieved based on `user_indices` and `item_indices`. The interaction score is computed as the sum of element-wise products (dot product) of these embeddings, which then is returned as the output.

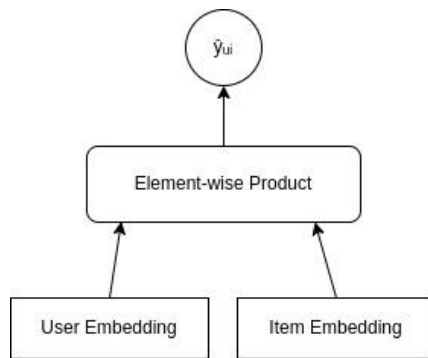


Figure 9: A diagram of the Matrix Factorization architecture

4.4 Implementation of the Generalized Matrix Factorization Model Class

In the realm of recommender systems, the Generalized Matrix Factorization (GMF) model, as described by He et al. in "Neural Collaborative Filtering" [16], builds upon the

traditional Matrix Factorization (MF) approach with an added layer for nonlinear transformations. This augmentation enhances the model's capacity to capture intricate user-item interactions, offering improved recommendation quality.

GMF decomposes the user-item interaction matrix into latent factor matrices, with each user and item represented by vectors in a shared latent space. The model's architecture includes embedding layers for users and items, followed by a layer for nonlinear transformations applied to the element-wise product of embeddings. This additional layer enables GMF to discern complex patterns in user preferences and item characteristics beyond linear relationships.

The implementation (Figure 38 in Appendix) and functions of the GMF model are almost identical to the implementation of the traditional MF model, with the only difference being the architecture definition, as it has an extra layer.

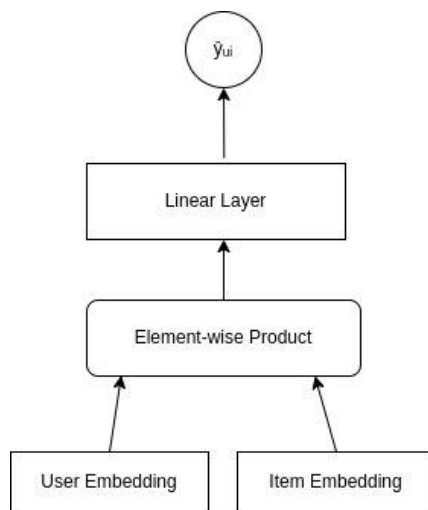


Figure 10: A diagram of the Generalized Matrix Factorization architecture

4.5 Implementation of the Neural Collaborative Filtering Model Class

Like the Generalized Matrix Factorization (GMF) model, NCF builds on Matrix Factorization (MF) foundations while introducing a neural network-based architecture. This architecture incorporates embedding layers for users and items and multiple hidden layers for nonlinear transformations. The dynamic addition of hidden layers allows NCF to discern complex patterns in user preferences and item characteristics, surpassing the linear limitations of traditional MF.

The implementation of the NCF model (Figure 39 in Appendix) closely mirrors that of the GMF model, with the primary distinction lying in the architecture definition. The NCF class in PyTorch consists of embedding layers for users and items, followed by a sequence of fully connected layers (`fc_layers`) for nonlinear transformations. This sequence dynamically adjusts its structure based on the specified number of hidden units and layers, allowing for flexibility in model complexity.

The forward method of the NCF class concatenates user and item embeddings and passes the resulting vector through the fully connected layers to generate the final output. The sigmoid activation function ensures output values are bounded between 0 and 1, representing the likelihood of user-item interactions.

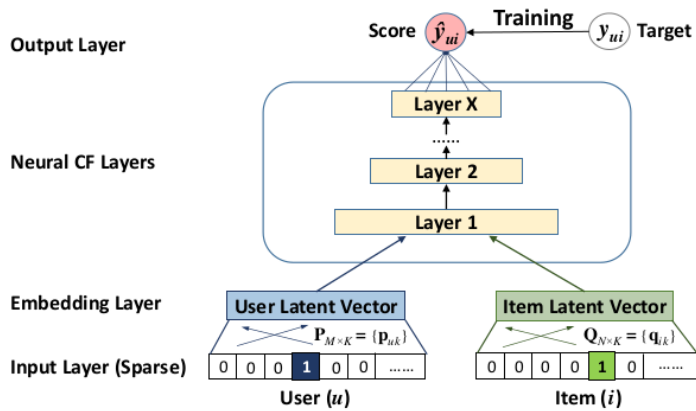


Figure 2: Neural collaborative filtering framework

Figure 11: Figure 2 from He et al. paper "Neural Collaborative Filtering" [16], showcasing the architecture of the NCF

4.6 Implementation details of MF, GMF and NCF

Despite the distinct architectures of the Matrix Factorization (MF), Generalized Matrix Factorization (GMF), and Neural Collaborative Filtering (NCF) models, their operational processes and application methods remain consistent across all three. This uniformity ensures that each model can be seamlessly integrated and tested within the same framework, allowing for direct comparison and efficient optimization of their respective functionalities. The consistent handling of data preprocessing, training routines, and evaluation metrics across these models simplifies the experimental setup and enables a straightforward assessment of each model's performance in a controlled environment. In this section, we will delve into the specifics of how these operational processes and application methods were implemented.

4.6.1 Data Preparation

The RecommendationDataset class (Figure 41 in Appendix) is crucial in managing the preprocessing and normalization of data for the Matrix Factorization model. This class is designed to handle three primary types of input data: user IDs, item IDs, and their

corresponding ratings. These inputs are fundamental to collaborative filtering models, where the goal is to predict user preferences based on historical interactions.

The structured approach to data handling ensures that the data fed into the neural network is clean, consistent, and correctly formatted. By standardizing the preprocessing steps within a dedicated dataset class, the scalability and maintainability of the model code are significantly enhanced, making it easier to adjust data inputs and preprocessing techniques as needed.

4.6.2 Loading Datasets and Encoding User and Item IDs

The training and testing datasets are loaded from parquet using pandas (Figure 42 in Appendix). Then we employ LabelEncoder from sklearn.preprocessing to convert each unique user and item identifier into a numerical format. This transformation is crucial because the MF model requires numerical input for indexing into user and item embedding matrices. Once encoded, these identifiers are used to create instances of the RecommendationDataset.

Then, we create the train and test data loaders. The train loader processes data in batches, significantly reducing training time by allowing the model to update more frequently with smaller subsets of data. For the test loader, a larger batch size is used to optimize evaluation speed. Shuffling the data in the training phase ensures diversity in the training examples presented to the model, preventing it from learning unintended biases and improving its generalization across different data scenarios.

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=256, shuffle=True, generator=torch.Generator(device=device))
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=6144, shuffle=False, generator=torch.Generator(device=device))
```

Figure 12: Initialization of training and testing data loaders in PyTorch, specifying batch sizes, shuffle settings, and generator devices.

Finally, we load the dataset with potential purchasable items for assessing the recommendation system's performance. This dataset organizes active items that each user may purchase post their last recorded interaction, excluding those previously

purchased as noted in the training set. By comparing the top 'k' recommended items from this dictionary for each user, against the actual purchases recorded in the test set, we can measure the effectiveness of the model in predicting user preferences and its potential real-world applicability.

4.6.3 Model Initialization and Configuration

Model initialization involves setting up essential components to ensure effective training. Key steps include determining the number of users and items from the encoded data, which informs the size of the embedding layers necessary for the Matrix Factorization model. An instance of the MF model is created with the calculated user and item counts. The mean squared error (MSE) loss function is employed for its suitability in regression-based tasks like rating prediction. Optimization is managed by the Adam optimizer, leveraging its advantages in handling sparse gradients, as discussed in section 4.2. A learning rate scheduler is also used to adjust the learning rate based on performance, enhancing the training process by reducing the rate when improvements plateau.

```
# Get the number of users and items
num_users = len(user_encoder.classes_)
num_items = len(item_encoder.classes_)

#Create model
model = MF(num_users, num_items, embed_dim=32768).to(device)

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-7)
scheduler = ReduceLR0nPlateau(optimizer, mode='max', factor=0.5, patience=0)
```

Figure 13: Setup of the matrix factorization model in PyTorch, detailing user and item counts, model initialization, and configuration of the loss function and optimizer.

4.6.4 Training Procedure

The training procedure is central to optimizing the Matrix Factorization model's performance. The process involves several key steps:

1. **Tracking Actual Purchases:** Before training, a dictionary is populated to track the actual items purchased by each user in the test set. This mapping is crucial for later validating the model's recommendation accuracy.
2. **Training Loop:** The model undergoes training over a specified number of epochs. Each epoch consists of:
 - a. Training the model with batches of user, item, and rating data from the training loader.
 - b. Using the optimizer to adjust model weights based on the loss computed from the output and actual ratings.
 - c. Periodically resetting gradients to prevent accumulation with `optimizer.zero_grad()`.
3. **Validation:** After each epoch, the model's performance is assessed on both training and test data to calculate loss and error rates. These metrics help monitor the model's learning progress and adjust strategies if necessary.
4. **Performance Metrics:** Key recommendation metrics such as hit rate, precision, recall, normalized discounted cumulative gain (NDCG), mean average precision (MAP), and F1 score are calculated to evaluate the model's effectiveness at predicting top K recommendations.
5. **Learning Rate Adjustment:** A learning rate scheduler adjusts the learning rate based on the precision metric from the validation phase, optimizing the model's response to training dynamics and enhancing convergence efficiency.

```

def train(model, train_loader, test_loader, criterion, optimizer, classes, active_items_per_user, num_epochs=10, k=100):
    # Create a dictionary to store the actual items from the test set for each user
    actual_items_dict = {}
    for batch_users, batch_items, _ in test_loader:
        for user, item in zip(batch_users, batch_items):
            user_id = user.item() # Convert single-element tensor to integer
            if user_id not in actual_items_dict.keys():
                actual_items_dict[user_id] = []
            actual_items_dict[user_id].append(item.item())

    # Start training loop
    for epoch in tqdm(range(num_epochs), desc="Epochs"):
        model.train() # Set model to training mod
        for users, items, ratings in tqdm(train_loader, desc="Training Batches", total=len(train_loader), leave=False):
            users, items, ratings = users.to(device), items.to(device), ratings.to(device)

            optimizer.zero_grad()
            outputs = model(users, items)
            loss = criterion(outputs, ratings)
            loss.backward()
            optimizer.step()

        # Validation phase
        # Calculate loss and error on the training and test sets
        training_loss, training_error = validate(model, train_loader, criterion, classes=classes)
        model.training_loss.append(training_loss)
        model.training_errors.append(training_error)
        test_loss, test_error = validate(model, test_loader, criterion, classes=classes)
        model.test_loss.append(test_loss)
        model.test_errors.append(test_error)

        # Calculate metrics on the test set
        hr, precision, recall, ndcg, map_k, f1 = evaluate_active_items(model, actual_items_dict, active_items_per_user, k=k)
        model.hr.append(hr)
        model.precision.append(precision)
        model.recall.append(recall)
        model.ndcg.append(ndcg)
        model.map_k.append(map_k)
        model.f1.append(f1)

        # Adjust learning rate based on the validation metrics
        scheduler.step(hr, precision, recall, ndcg, map_k, f1)

    tqdm.write(f'Epoch {epoch}: \nLearning Rate: {scheduler.get_last_lr()[0] if scheduler.get_last_lr()[0] else None} \nTrn
    return model

```

Figure 14: Comprehensive Python function for training the recommendation model, including detailed steps for training, validation, metric evaluation, and learning rate adjustment using PyTorch and TQDM for progress tracking.

4.6.5 Model Evaluation and Metrics Calculation

In the training cycle of our recommendation model, we employ the validation process as a critical checkpoint to ensure the model's generalization capabilities on unseen data. This step is vital for preventing overfitting and confirming consistent performance under varying conditions.

We initialize the validate function by setting the model to evaluation mode. This is crucial for disabling training-specific behaviors, ensuring the model behaves consistently when assessing its performance. During validation, we track loss and absolute error, starting both metrics at zero to aggregate these values accurately across the dataset. Throughout the validation phase, we iterate over batches from the provided DataLoader, which contains user IDs, item IDs, and ratings. For each batch, we calculate loss using the specified criterion, which evaluates the discrepancy between predicted and actual ratings. Additionally, to calculate absolute error, we adjust ratings back to their original scale because our model's output is normalized. This adjustment is crucial for an accurate evaluation of how closely the model's predictions match the actual ratings.

Both loss and absolute error are accumulated across all batches, offering a comprehensive view of the model's performance over the entire validation dataset. By dividing the total loss by the number of batches, we obtain the average loss, reflecting the typical level of error the model produces against the validation data. Similarly, the average absolute error provides insight into the average discrepancy between the model's predictions and the actual ratings, offering a direct measure of predictive accuracy.

We validate the model on both the training and test sets. Validating on the training set allows us to monitor the model's learning progress and convergence towards the training data. Although the ratings in the test set may not be entirely accurate due to the simplistic handling of recency scores during data preprocessing, we still calculate test error and loss. This provides a preliminary idea of how the model behaves on unseen data and helps us detect any potential overfitting.

```

def validate(model, loader, criterion, classes):
    model.eval() # Set the model to evaluation mode
    total_loss = 0
    total_absolute_error = 0
    total_samples = 0

    with torch.no_grad():
        for users, items, ratings in tqdm(loader, desc="Validation", total=len(loader), leave=False):
            users, items, ratings = users.to(device), items.to(device), ratings.to(device)

            outputs = model(users, items)
            loss = criterion(outputs, ratings)
            total_loss += loss.item()

            # Convert outputs and actual ratings back to the original scale
            actual_ratings_scaled = ratings * (classes - 1) + 1
            predicted_ratings_scaled = outputs * (classes - 1) + 1

            # Calculate absolute error on the original scale
            absolute_errors = torch.abs(predicted_ratings_scaled - actual_ratings_scaled)
            total_absolute_error += absolute_errors.sum().item()

            total_samples += ratings.size(0) # Update total number of samples

    average_loss = total_loss / len(loader)
    average_absolute_error = total_absolute_error / total_samples
    return average_loss, average_absolute_error

```

Figure 15: Python function `validate` for evaluating a recommendation model, demonstrating the calculation of loss and absolute error during model validation using PyTorch, with outputs and ratings scaled back to their original range.

4.6.6 Evaluation Metrics at K

In the realm of recommender systems, evaluating the model's performance extends beyond mere accuracy. It's crucial to assess how well the model predicts the top items that users are most likely to engage with. This is where metrics evaluated at a specified number, K , play a vital role. These metrics, focusing on the top K recommendations, mirror real-world scenarios where users interact with a limited number of suggested items, making them fundamental for measuring user satisfaction and model effectiveness.

Metrics Employed:

1. **Hit Rate at K (HR@K):** This metric checks if at least one of the top- K recommended items matches the items actually interacted with by the user, offering a quick indication of whether the recommendations are broadly relevant.

2. **Precision at K:** This measures the proportion of relevant recommendations within the top K items, indicating the accuracy of the model in identifying items that are truly of interest to the user.
3. **Recall at K:** It assesses the model's ability to capture all relevant items within its top K recommendations, crucial for understanding the model's completeness in retrieving relevant items.
4. **Normalized Discounted Cumulative Gain at K (NDCG@K):** This metric evaluates the ranking quality of the recommendations, giving more importance to hits at higher ranks, thus reflecting the practical utility of the ranking order in the recommendations.
5. **Mean Average Precision at K (MAP@K):** MAP@K calculates the average precision across all users, considering the order of recommendations, and emphasizes the placement of truly relevant items within the top K slots.
6. **F1 Score at K:** This combines precision and recall into a single metric, balancing the trade-offs between them and providing a holistic view of the model's performance at the specified K.

These metrics are computed on the test dataset to ensure that our model not only aligns with user preferences accurately but also meets the operational standards required for real-world applications. The evaluation implementation can be found at figure 44 and 45 in the appendix.

4.6.7 Exporting Results in DataFrame

To efficiently manage and analyze the results from our model evaluations, we utilize the Dask library to export the predicted scores into a DataFrame format. This approach is essential for handling large datasets that result from the model's predictions across both training and testing phases.

4.7 Demographic and Feature Enhanced Neural Collaborative Filtering (DFNCF)

Considering that in some e-commerce cases additional demographic and item feature data may be available, we can develop a model which can leverage this extended information. The incorporation of such comprehensive data has been shown to significantly improve recommendation quality, as evidenced by research from Zhang et al. [17] and Israr ur Rehman, et al. [20] highlighting the effectiveness of integrating explicit and implicit user-item couplings and contextual features in deep collaborative filtering. We introduce the Demographic and Feature Enhanced Neural Collaborative Filtering (DFNCF) model. This model aims to enhance recommendation quality by extracting more nuanced latent features between users and items and across their interactions.

In this section, we will explore the implementation of DFNCF, which builds upon the Neural Collaborative Filtering (NCF) framework by introducing additional input layers that incorporate item features and demographic data such as customer age. While maintaining operational consistency with the NCF model, DFNCF modifies the architectural framework and input handling in the forward function to accommodate the expanded data set.

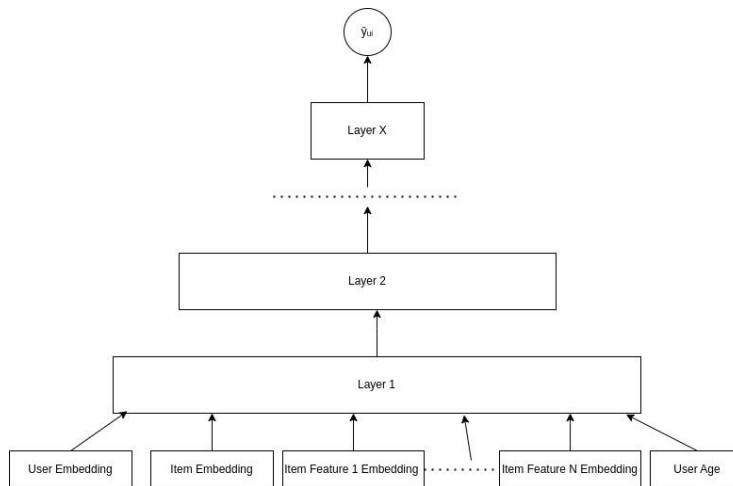


Figure 16: A diagram of the proposed Demographic and Feature Enhanced Neural Collaborative Filtering architecture

4.7.1 How DFNCF Works

The Demographic and Feature Enhanced Neural Collaborative Filtering (DFNCF) model significantly enhances the traditional Neural Collaborative Filtering (NCF) by integrating both demographic information and detailed item features into the recommendation process. This approach allows for a deeper and more nuanced understanding of user preferences and item characteristics.

Like in NCF, DFNCF uses user and item embeddings to capture latent factors, but it expands further by introducing additional embeddings for categorical features of items. Each item's feature is represented by its own embedding layer. These layers help the model capture specific attributes of items that might influence user preference:

1. **Product Type:** Embeddings for product type allow the model to differentiate between categories of items, understanding which types are preferred by which types of users.
2. **Graphical Appearance:** This feature captures the visual design or aesthetic of items, which can be crucial in domains like fashion or decor where visual aspects significantly impact user choice.
3. **Color Group:** By embedding color groups, DFNCF can cater to user preferences related to color, which are often strong personal preferences.
4. **Department:** This captures the broader classification of items, such as electronics, clothing, or home goods, aiding in segmenting recommendations according to departmental lines.
5. **Index Group:** Typically used in retail to group items into logistical categories, these embeddings can help in modeling operational aspects like supply chain preferences or seasonal trends.

In addition to these item features, DFNCF incorporates the age of users as a direct numerical input, acknowledging that age can significantly affect purchasing patterns and preferences.

The concatenated input layer, which combines user embeddings, item embeddings, categorical feature embeddings, and age, feeds into a multi-layer perceptron (MLP). The MLP similarly as in the NCF, consists of several hidden layers that process these inputs through non-linear transformations, facilitated by ReLU activation functions, culminating in a sigmoid output layer that predicts the probability of user-item interactions.

By leveraging a rich dataset that includes demographic and categorical item features, DFNCF not only adheres to the operational processes typical of NCF but also enhances its architecture to accommodate a broader spectrum of inputs. This comprehensive approach ensures that DFNCF is well-suited for complex recommendation scenarios where additional data layers provide critical insights into user behavior and item desirability, significantly improving recommendation quality and relevance.

4.7.2 Implementation of DFNCF Class

The DFNCF class extends the BaseNetwork to create a model that incorporates both user and item embeddings with additional demographic and feature embeddings into a unified recommendation system. The constructor of DFNCF initializes several embedding layers that are specific to both user-item interactions and item-specific attributes, which enrich the model's capability to discern and learn from complex patterns in the data.

- **User and Item Embeddings:** Like the NCF model, embeddings for users and items convert sparse indices into dense vectors, each of size `user_item_embed_dim`. These embeddings capture the latent factors from user-item interactions.

- **Feature Embeddings:** Each item feature such as product type, graphical appearance, color group, department, and index group is embedded separately. Each type of item feature is transformed into a dense vector of size `features_embed_dim`, allowing the model to capture specific properties of items that may influence user preferences.

The architecture also incorporates age as a continuous feature directly into the network, reflecting its impact on user preferences without the need for embedding.

The network's forward pass concatenates these embeddings along with the age input into a single feature vector. This combined vector is then processed through a sequence of fully connected layers (multilayer perceptron or MLP) defined in `self.fc_layers`. The MLP consists of several hidden layers that reduce dimensionality progressively, each followed by a ReLU activation to introduce non-linearity, culminating in a sigmoid output layer. This final layer outputs the probability that a user would prefer a given item, enabling binary classification of interactions.

```

class DFNCF(BaseNetwork):
    def __init__(self, num_users, num_items, num_product_types, num_graphical_appearance, num_colour_groups, num_departments, num_index_groups, user_item_embed_dim):
        super(DFNCF, self).__init__()
        self.user_embedding = nn.Embedding(num_users, user_item_embed_dim)
        self.item_embedding = nn.Embedding(num_items, user_item_embed_dim)
        # Embedding for each categorical feature
        self.product_type_no_embedding = nn.Embedding(num_product_types, features_embed_dim)
        self.graphical_appearance_no_embedding = nn.Embedding(num_graphical_appearance, features_embed_dim)
        self.colour_group_code_embedding = nn.Embedding(num_colour_groups, features_embed_dim)
        self.department_no_embedding = nn.Embedding(num_departments, features_embed_dim)
        self.index_group_no_embedding = nn.Embedding(num_index_groups, features_embed_dim)

        # Age is a continuous feature, we directly use it in concatenation
        total_feature_dim = user_item_embed_dim * 2 + features_embed_dim * 5 + 1 # 5 categorical features and age

        self.fc_layers = nn.Sequential(
            nn.Linear(total_feature_dim, number_of_hidden_units), nn.ReLU(),
            *[
                layer for i in range(1, number_of_hidden_layers)
                for layer in (nn.Linear(number_of_hidden_units // (2 ** (i - 1)), number_of_hidden_units // (2 ** i)), nn.ReLU())
            ],
            nn.Linear(number_of_hidden_units // (2 ** (number_of_hidden_layers - 1)), 1),
            nn.Sigmoid()
        )

    def forward(self, user_indices, item_indices, ages, product_type_no, graphical_appearance_no, colour_group_code, department_no, index_group_no):
        user_embedding = self.user_embedding(user_indices)
        item_embedding = self.item_embedding(item_indices)
        product_embedding = self.product_type_no_embedding(product_type_no)
        graphical_embedding = self.graphical_appearance_no_embedding(graphical_appearance_no)
        colour_embedding = self.colour_group_code_embedding(colour_group_code)
        department_embedding = self.department_no_embedding(department_no)
        index_embedding = self.index_group_no_embedding(index_group_no)

        x = torch.cat([user_embedding, item_embedding, ages.unsqueeze(1), product_embedding, graphical_embedding, colour_embedding, department_embedding, index_embedding])
        x = self.fc_layers(x)
        return x.squeeze()

```

Figure 17: Implementation of the Demographics and Feature Enhanced Neural Collaborative Filtering class in PyTorch, defining a neural network module for collaborative filtering with methods for embedding users and items.

This expanded approach maintains operational consistency while enhancing the model's input data for a more detailed recommendation system.

4.8 Multinomial RBM

In this section, we explore my implementation of the Multinomial Restricted Boltzmann Machine (RBM), a sophisticated model for collaborative filtering based on the seminal work by Salakhutdinov, Mnih, and Hinton [8].

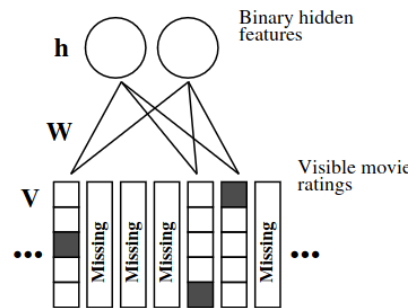


Figure 1. A restricted Boltzmann machine with binary hidden units and softmax visible units. For each user, the RBM only includes softmax units for the movies that user has rated. In addition to the symmetric weights between each hidden unit and each of the $K = 5$ values of a softmax unit, there are 5 biases for each softmax unit and one for each hidden unit. When modeling user ratings with an RBM that has Gaussian hidden units, the top layer is composed of linear units with Gaussian noise.

Figure 18: Figure 1 from Salakhutdinov et al. paper "Neural Collaborative Filtering" [16], showcasing the architecture of the Multinomial RBM [8]

4.8.1 Implementation of Multinomial RBM class

The MultinomialRBM (Figure 47 in Appendix) is initialized with the number of visible units, hidden units, and rating classes (k). The visible units represent the product of items and their possible rating classes, and the hidden units capture latent features within the data. Weights and biases are initialized using a scaled normal distribution to start the training from a neutral stance, which helps in stabilizing the learning process early on.

The implementation class (Figure 47 in Appendix) has the following functionalities:

- **Forward and Reverse Mappings:** The model includes methods for mapping visible units to hidden probabilities (`to_hidden`) and reconverting hidden states back to visible probabilities (`to_visible`), leveraging the sigmoid and softmax functions respectively. These transformations are crucial for the Gibbs sampling process, allowing the model to approximate the underlying probability distribution of the data.

- **Free Energy Calculation:** The free energy function calculates the energy of a given visible state, providing a measure used during training to evaluate the likelihood of observing a given state. This function helps in optimizing the model parameters by minimizing the energy states of visible units corresponding to higher probabilities.
- **Gibbs Sampling:** This method performs Gibbs sampling to approximate the likelihood of the data under the current model. It alternates between sampling hidden states given visible states and then resampling visible states, which is essential for training RBMs. The ability to compute the weighted average or exact sampling at the end of Gibbs steps provides flexibility in how the model's outputs are used for evaluation or further training.

It is important to understand the purpose and process of Gibbs Sampling in RBMs, as detailed by Salakhutdinov, Mnih, and Hinton [8]. Gibbs sampling is employed to approximate the hidden states given visible data, crucial for learning the model's distribution. Starting with an initial visible state, the method iteratively samples hidden states based on current visible states and then resamples visible states from these hidden states. This process, repeated for a defined number of Gibbs steps per training iteration, helps the model to better approximate the underlying data distribution. Adjusting the number of Gibbs steps dynamically during training can optimize convergence and enhance the model's ability to reconstruct input data accurately, thereby improving training efficiency and the quality of the generated recommendations.

4.8.2 Loading the Data

The `RBMDataset` class (Figure 48 in Appendix) is specifically designed to manage the loading of user-item interaction data from a sparse matrix format into a PyTorch-friendly format for training. It supports either direct loading from a file path where the CSR matrix is stored or handling an already loaded CSR matrix. Depending on the memory constraints and requirements of the training process, this class offers the flexibility to work with data in its native sparse format or convert it to a dense array.

Once the dataset is prepared, it is loaded into a DataLoader object, which facilitates batch processing during model training. In this context, we do not shuffle the records as the data represents specific user-item interactions where the order of users does not impact the training dynamics.

4.8.3 Training the RBM

We manage RBM training (Figure 49 in Appendix) using established deep learning procedures, selecting Adam as the optimizer due to its efficacy [18]. We set an initial learning rate and apply decay to optimize weight adjustments during training, including weight decay to curb overfitting. Throughout training, we dynamically adjust Gibbs sampling steps to refine the model's convergence and data reconstruction accuracy, a process critical for learning the probability distribution effectively as detailed by Salakhutdinov, Mnih, and Hinton [8].

During each epoch, we process batches of data where Gibbs sampling—vital for training RBMs—is performed. This Monte Carlo method alternates sampling between hidden states from visible layers and resampling visible states, crucial for understanding the model's stability via free energy calculations. Our approach adjusts learning rates upon performance plateaus, enhancing efficiency using a strategy similar to the adaptive scheduler described by Khodamoradi et al. [19].

To evaluate our model's efficacy, we compute metrics such as HR@K, Precision@K, and others, reflecting the model's predictive accuracy on user preferences. These metrics not only guide iterative improvements during training but also validate our methodological choices, aligning with studies emphasizing the importance of precision metrics in collaborative filtering systems [3, 10].

4.9 Tuning the Models

In this section, we describe the process of tuning hyperparameters for the traditional Matrix Factorization (MF) model, illustrating the broader strategy applicable to all models discussed in this thesis. The MF model is chosen for its simplicity, which facilitates a clear understanding of the hyperparameter tuning strategy.

We aim to determine the optimal configuration of the MF model's hyperparameters using a dataset comprised of the top 1% of best-buying customers and best-selling products. This dataset has been condensed to enhance processing speeds. The affinity score within this dataset is binary, where '1' indicates interaction and '0' for non-interactions generated through negative sampling. We are not using the proposed formula yet, as we need to find the optimal values of the formula's coefficients α , β and δ .

The hyperparameters we focus on include embedding dimension size of users and items, learning rate, learning rate decay on plateau, and batch size. The optimal settings are identified based on their performance, measured by the Normalized Discounted Cumulative Gain (NDCG) and F1 score at the top K recommended items. NDCG is a standard research benchmark, while the F1 score is particularly suited to our e-commerce context, emphasizing both precision and recall. We choose K to be 12, as it is a small yet effective number that can be used for a marketing campaign. The choice of K = 12 was also the choice made by the Kaggle competition where we sourced our dataset. We can in a later stage, compare our MAP@12 score with other submissions in the competition to determine how well our recommender system behaves.

To establish the optimal values for the hyperparameters, we set up the following arrays containing potential values for each parameter.

```
embed_dims = [512, 1024, 2048, 4096]
lr = [0.01, 0.005, 0.001]
lr_decay = [0.25, 0.5, 0.75]
batch_sizes = [128, 256]
```

We systematically explore every possible combination by training the model for 10 epochs with each set. The performance metrics, NDCG@12 and F1@12, are recorded for each combination and stored in a CSV file. This data will be analyzed post-training to identify the combination that maximizes these metrics, thereby determining the most effective hyperparameter settings.

After choosing the most effective hyperparameters, we do the same iterative procedure to find the optimal values for the proposed Affinity Score formula's coefficients α , β and δ . For the coefficients we check α and β starting from 0 and incrementing it by 0.25 until it reaches 3. Similarly, for δ we start from 1 and we increase it by 1 until we reach 3. We choose as the optimal values the combination of which the NDCG@12 and F1@12 are at a maximum.

The tuning procedure adapts when optimizing other models to accommodate their specific architectures and additional parameters. For the Generalized Matrix Factorization (GMF) model, we employ a strategy similar to the MF model, focusing on the same core hyperparameters.

We trialed the optimal parameters of the GMF by iterating over the following hyperparameter arrays:

```
embed_dims = [512, 1024, 2048, 4096]
lr = [0.00001, 0.000001, 0.0000001]
lr_decay = [0.25, 0.5, 0.75]
batch_sizes = [128, 256]
```

However, the Neural Collaborative Filtering (NCF) model introduces more complexity due to its multi-layer perceptron (MLP) architecture. Here, not only do we optimize the typical hyperparameters like embedding dimensions and learning rates, but we also determine the ideal number of hidden layers and the size of each layer, crucial for capturing complex user-item interactions.

We trialed the hyperparameters of the NCF over these arrays:

```
embed_dims = [1024, 2048, 4096]
hidden_units = [512, 1024]
hidden_layers = [3, 4, 5]
lr = [0.00001, 0.000005, 0.000001]
```

```
lr_decay = [0.25, 0.5, 0.75]
batch_sizes = [128, 256]
```

The Demographic and Feature Enhanced Neural Collaborative Filtering (DFNCF) model was trialed on the following arrays:

```
embed_dims = [512, 1024, 2048]
num_hidden_units = [512, 1024, 2048]
num_hidden_layers = [3, 4, 5]
lr = [1e-5, 5e-6, 1e-6]
lr_decay = [0.25, 0.5, 0.75]
batch_sizes = [128, 256]
```

Similarly, for the Multinomial Restricted Boltzmann Machine (RBM), the focus extends to determining the optimal size of the hidden layer, which affects the model's capacity to learn and represent the data effectively. The RBM was tuned on the following hyperparameter choices:

```
n_hidden_units_fractions = [0.5, 0.15, 0.25, 0.35]
lr_list = [5e-6, 1e-6]
lr_decay = [0.25, 0.5, 0.75]
weight_decay_list = [1e-5, 1e-6]
batch_sizes = [128, 256]
```

Each model's unique characteristics necessitate adjustments in the tuning strategy to ensure optimal performance across diverse recommender system architectures.

4.10 Tuning Results of Model Hyperparameters

After thorough training and optimization of various models with different hyperparameter combinations, we have successfully identified the optimal settings for each of our recommendation models. The results can be found in the appendix tables. Below are the details of the optimal hyperparameters for each model, as derived from extensive experiments catalogued in the Appendix.

Matrix Factorization (MF) Model

Commented [NT1]: To do

- **Embedding Dimension:** 512
- **Learning Rate:** 0.01
- **Learning Rate Decay:** 0.75
- **Batch Size:** 128

Generalized Matrix Factorization (GMF) Model

- **Embedding Dimension:** 2048
- **Learning Rate:** 0.005
- **Learning Rate Decay:** 0.75
- **Batch Size:** 256

Neural Collaborative Filtering (NCF) Model

- **Embedding Dimension:** 2048
- **Hidden Units:** 256
- **Hidden Layers:** 4
- **Learning Rate:** 0.000005
- **Learning Rate Decay:** 0.75
- **Batch Size:** 256

Demographic and Feature-Enhanced Neural Collaborative Filtering (DFNCF) Model

- **User-Item Embed Dimension:** 1024
- **Feature Embed Dimension:** 1024
- **Hidden Units:** 2048
- **Hidden Layers:** 5
- **Learning Rate:** 0.00001
- **Learning Rate Decay:** 0.25

Multinomial Restricted Boltzmann Machine (RBM) Model

- **Hidden Units Fraction:** 0.35
- **Learning Rate:** 0.000005
- **Learning Rate Decay:** 0.5
- **Weight Decay:** 0.000001
- **Batch Size:** 256

These results will be applied to adjust the coefficients of the Affinity Score formula for each model.

4.11 Tuning results of Coefficients for Affinity Score Formula Across Models

Upon employing the determined optimal hyperparameters, we experimented with various combinations of coefficients for the Affinity Score formula across different models. These coefficients play a crucial role in accurately reflecting user preferences and improving recommendation quality. The results can be found in the appendix tables. Here are the optimal values found for each model:

Commented [NT2]: To do

Matrix Factorization (MF):

- $a = 1$
- $b = 2.75$
- $d = 1$

Generalized Matrix Factorization (GMF):

- $a = 1$
- $b = 3$
- $d = 2$

Neural Collaborative Filtering (NCF):

- $a = 0.5$
- $b = 2.5$
- $d = 1$

Restricted Boltzmann Machine (RBM):

- $a = 2$
- $b = 0.5$
- $d = 2$

Demographic and Feature-Enhanced Neural Collaborative Filtering (DFNCF):

- $a = 0$
- $b = 1$
- $d = 2$

Given this variance, adopting a one-size-fits-all approach for the coefficients might not be ideal. Each model behaves differently based on its architecture and the specific nature of the data it handles, which suggests that tailor-fitting the coefficients to each model could leverage these unique characteristics for better performance. Therefore, it might be more effective for us to continue using distinct combinations for each model.

Chapter 5: Results & Discussion

In this chapter, we evaluate whether the proposed affinity scoring formula, which classifies user-item interactions into five distinct classes rather than the traditional binary classification, enhances the quality of recommendations and to compare the performance of advanced models, Multinomial RBM and DFNCF, against traditional models such as MF, GMF, and NCF.

We have trained these models using the optimal hyperparameters identified in previous sections and the data preprocessing incorporates the optimal coefficients of the affinity score formula for each model as discovered from our analyses.

For assessing whether the proposed affinity scoring formula, which classifies user-item interactions into five distinct classes rather than the traditional binary classification, enhances the quality of recommendations, we do a comparative analysis of the performance metrics for each model under two different scenarios:

- The user-item affinity score is binary (0 for no interaction and 1 for interaction),
- The user-item affinity score is spread across 5 classes as defined by our optimized formula.

In the top 1% dense subset, we aim to predict whether the 5.5 articles purchased by customers, from an average pool of 530 available products, are found in the top 12 items ranked by the model (HR@12). Precision at 12 (Precision@12) measures the proportion of relevant items among the top 12 recommendations. Recall at 12 (Recall@12) indicates the proportion of the 5.5 purchased items that are successfully predicted within the top 12. Normalized Discounted Cumulative Gain at 12 (NDCG@12) assesses the ranking quality of the recommendations, giving higher scores for correctly predicting the more relevant items at higher ranks. Mean Average Precision at 12 (MAP@12) computes the average

precision across all relevant items, rewarding models that predict more relevant items earlier in the list. F1 Score at 12 (F1@12) combines Precision and Recall into a single metric to provide a balance between them.

For the top 10% subset, the customers purchase on average 14.6 products, with an available pool of 6,500 products per customer. Here, HR@12 evaluates whether the model's top 12 ranked items include the 14.6 purchased products. Precision@12 measures the proportion of relevant items among the top 12 recommendations, while Recall@12 indicates the proportion of the 14.6 purchased items found within the top 12. NDCG@12 assesses the ranking quality, rewarding higher ranks for more relevant items. MAP@12 computes the average precision across all relevant items, and F1@12 provides a balanced measure of Precision and Recall.

5.1 Evaluation of the Proposed Affinity Score Formula on the Matrix Factorization Model

In this section, we analyze the effectiveness of the proposed affinity score formula on the performance of the Matrix Factorization (MF) model. The MF model was evaluated over 30 epochs on the top 1% subset and over 15 epochs on the top 10% subset for time efficiency. Performance metrics were recorded for each epoch.

5.1.1 Results

Results on the top 1% dense subset is as follows:

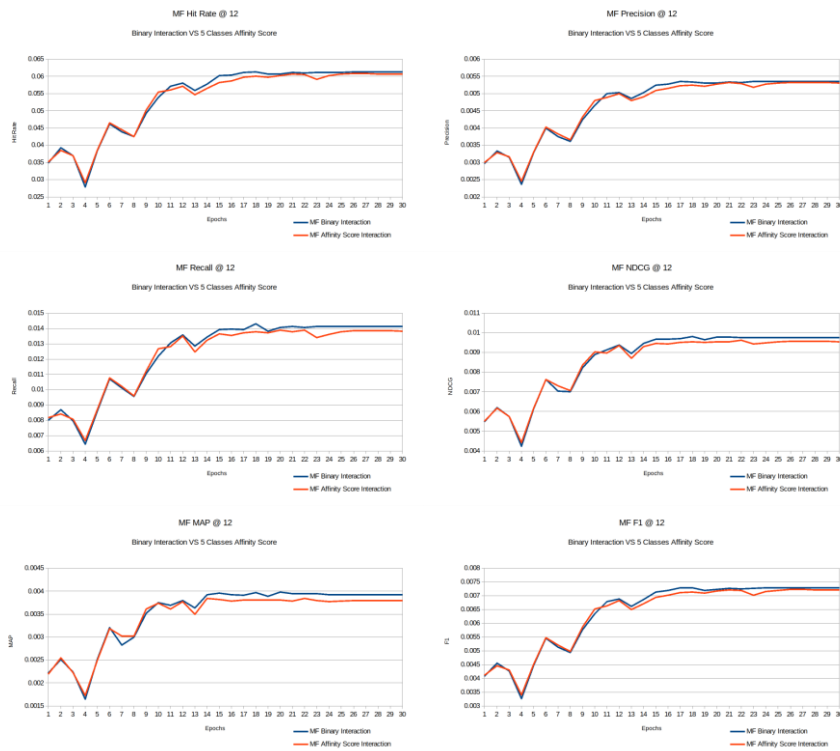


Figure 19: Matrix Factorization metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 1% subset of the initial dataset.

From the chart, we can see a similar pattern for both approaches where we have increasing recommendation quality and plateaus at epoch 15.

For binary interaction, the hit rate maxed at 0.0613, while for the affinity score calculated interaction, it maxed at 0.0608.

The precision for binary interaction peaked at 0.0054, while for the affinity score interaction, it reached a high of 0.0053.

The recall trajectories are closely aligned between the two methods. For binary

interaction, recall at 12 reached a maximum of 0.0141, whereas for the affinity score interaction, the peak was slightly lower at 0.0138.

The NDCG for binary interaction topped at 0.0098, compared to 0.0096 for the affinity score interaction.

The behavior of the MAP at 12 metric shows closely matching trends across epochs. The maximum MAP at 12 for binary interaction was 0.0039, and for affinity score interaction, it was slightly less at 0.0038.

The F1 score at 12 for binary interaction reached its highest at 0.0073, and for affinity score interaction, it was very close at 0.0072.

The results on the more challenging and expansive top 10% subset are as follows:

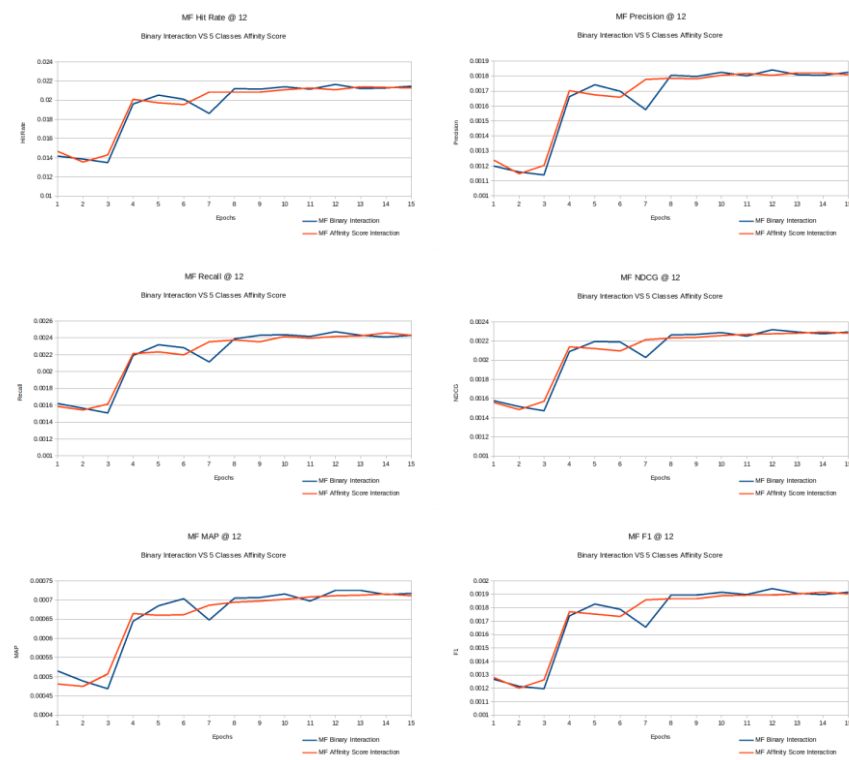


Figure 20: Matrix Factorization metrics at 12 items comparison chart with two data preprocessing approaches: binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 10% subset of the initial dataset.

The binary interaction method achieves a maximum hit rate of 0.0217, while the affinity score interaction method sees a peak of 0.0214. This represents a slight decrease of 1.36% when using the affinity score, indicating that in sparser datasets, the more granular rating system may not significantly enhance the model's ability to retrieve relevant items.

The peak precision is 0.0018 for both approaches.

The recall measures the model's ability to identify all relevant items, with both approaches reaching a maximum of 0.0025.

The NDCG, which accounts for the hit rate's position, again, both reached a maximum of 0.0023.

The MAP shows a maximum of 0.0007 for both approaches.

The F1 Score, balancing precision and recall, was highest at 0.0019 for both approaches.

5.1.2 Discussion

Across both the denser top 1% subset and the more expansive and sparser top 10% subset, the multi-class scoring system demonstrated performance very similar to the traditional binary interaction approach across all metrics. This consistency suggests that the multi-class scoring system can be used as an alternative approach without any performance loss compared to the binary system, regardless of data density. However, it is evident that the simple Matrix Factorization (MF) model fails to fully leverage the more diverse and rich information that the multi-class approach offers. This limitation indicates that while the multi-class system is robust and adaptable, more advanced or tailored models may be required to effectively exploit the richer information provided by the multi-class scoring in real-world recommender system applications.

It is also evident that the simple Matrix Factorization (MF) model fails to fully leverage the more diverse and rich information that the multi-class approach offers. The expected benefits of capturing nuanced user-item interactions were not realized, indicating a

potential limitation in the MF model's capacity to utilize the additional granularity provided by the multi-class scoring system. This highlights the need for more advanced or tailored models to effectively exploit the richer information encapsulated by the multi-class approach in real-world recommender system applications.

5.2 Evaluation of the Proposed Affinity Score Formula on the Generalized Matrix Factorization (GMF) Model

In this section, we evaluate the influence of the proposed affinity score formula on the performance of the Generalized Matrix Factorization (GMF) model. Over 30 epochs on the top 1% subset and 15 epochs on the top 10% subset, we measured the same metrics as on the MF model. This evaluation aims to discern how different scoring methods affect the model's efficacy, particularly in how well it adapts to variations in data density.

5.2.1 Results

The data from the top 1% dense subset offers a critical insight into how the GMF model handles a scenario with frequent and diverse user-item interactions. By implementing a nuanced multi-class scoring system, we expect to observe a notable improvement in the model's ability to deliver precise and relevant recommendations, leveraging the rich interaction data available in this subset.



Figure 21: Generalized Matrix Factorization metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 1% subset of the initial dataset.

The binary interaction method plateaus just after 8 training epochs and achieves a maximum hit rate of 0.0283, while the affinity score interaction method sees a peak of 0.0391 after 22 epochs. This represents a **38% increase** when using the affinity score, suggesting that a more granular rating system significantly enhances the model's ability to retrieve relevant items.

The peak precision for the binary interaction is 0.0024 compared to 0.0034 for the affinity score interaction, indicating a **42% improvement**. This higher precision demonstrates the affinity method's effectiveness in increasing the proportion of true positive

recommendations.

The recall measures the model's ability to identify all relevant items, with binary interaction reaching a maximum of 0.0069 and affinity score interaction peaking at 0.0091, a **32% improvement**. This increase affirms that the multi-class scoring method is better at capturing relevant items across the dataset.

The NDCG, which accounts for the position of the hit rate, saw a maximum of 0.0044 for binary interaction and 0.0063 for affinity score interaction, a **43% increase**. This suggests that the affinity score method not only identifies more relevant items but also ranks them more effectively at the top of the recommendation list.

The MAP shows a maximum of 0.0018 for binary interaction and 0.0026 for affinity score interaction, a **47% improvement**. This increase reflects a more accurate overall prediction across all queries.

The F1 Score, balancing precision and recall, was highest at 0.0033 for binary interaction and 0.0046 for affinity score interaction, demonstrating a **40% improvement**. This confirms that the affinity scoring method yields a more balanced performance between precision and recall.

The top 10% sparse subset of the dataset offers a contrast to the denser subset, exploring how the GMF model performs under less optimal conditions with fewer interactions per item. As with the denser subset, the nuanced multi-class scoring is anticipated to bring improvements, albeit possibly with more modest gains due to the inherent limitations of sparse data.

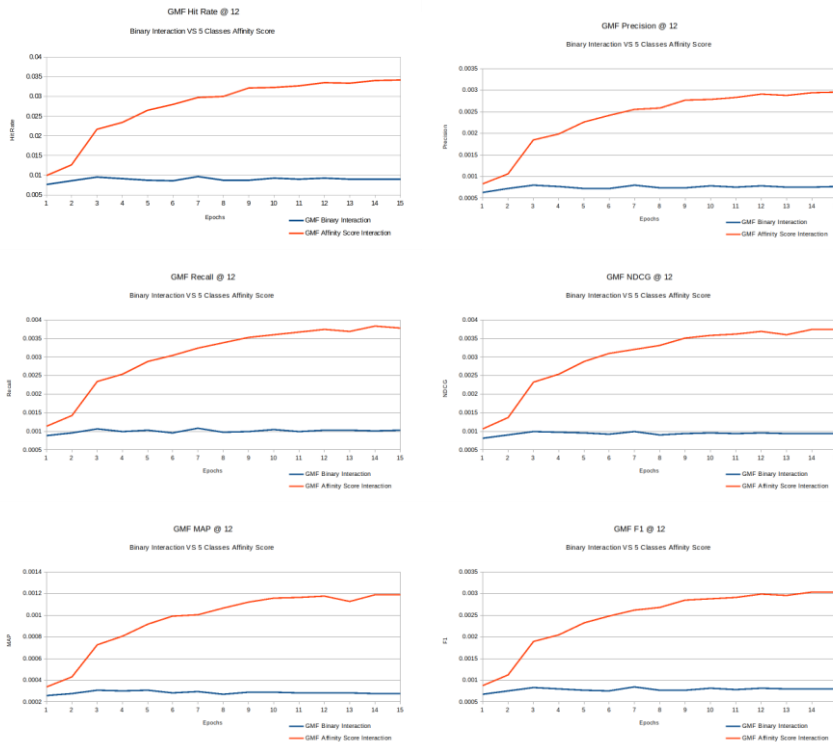


Figure 22: Generalized Matrix Factorization metrics at 12 items comparison chart with two data preprocessing approaches: binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 10% subset of the initial dataset.

The binary interaction method achieves a maximum hit rate of 0.0096 after 7 epochs, while the affinity score interaction method sees a peak of 0.0343. This represents a **256% increase** when using the affinity score, suggesting that a more granular rating system significantly enhances the model's ability to retrieve relevant items even in sparser scenarios.

The peak precision for the binary interaction is 0.0008 compared to 0.0030 for the affinity score interaction, indicating a **266% improvement**. This higher precision demonstrates the affinity method's effectiveness in increasing the proportion of true positive recommendations in a sparse data environment.

The recall measures the model's ability to identify all relevant items, with binary

interaction reaching a maximum of 0.0011 and affinity score interaction peaking at 0.0038, a **255% improvement**. This increase affirms that the multi-class scoring method is better at capturing relevant items across the sparse dataset.

The NDCG, which accounts for the position of the hit rate, saw a maximum of 0.0010 for binary interaction and 0.0037 for affinity score interaction, a **274% increase**. This suggests that the affinity score method not only identifies more relevant items but also ranks them more effectively at the top of the recommendation list.

The MAP shows a maximum of 0.0003 for binary interaction and 0.0012 for affinity score interaction, a **284% improvement**. This increase reflects a more accurate overall prediction across all queries.

The F1 Score, balancing precision, and recall, was highest at 0.0008 for binary interaction and 0.0030 for affinity score interaction, demonstrating a **259% improvement**. This confirms that the affinity scoring method yields a more balanced performance between precision and recall.

5.2.2 Discussion

Across both the denser top 1% subset and the more expansive and sparser top 10% subset, the multi-class scoring system demonstrated significant improvements over the traditional binary interaction approach across all metrics. This consistency suggests that the multi-class scoring system can be effectively used as an alternative approach with substantial performance gains compared to the binary system, regardless of data density. Notably, the improvements were more pronounced in the sparse dataset, indicating the robustness and adaptability of the multi-class scoring method in less optimal conditions. Furthermore, the slightly more advanced Generalized Matrix Factorization (GMF) model, unlike the simpler Matrix Factorization (MF) model, successfully leverages the additional information provided by the multi-class approach, highlighting its potential to enhance recommendation quality by capturing more nuanced user-item interactions.

5.3 Evaluation of the Proposed Affinity Score Formula on the Neural Collaborative Filtering (NCF) Model

In this section, we apply a similar approach to that used in the evaluation of the MF and GMF models to assess the impact of the proposed affinity score formula on the performance of the Neural Collaborative Filtering (NCF) model.

The evaluation spans 30 epochs on the top 1% subset and 15 epochs on the top 10% subset, recording the same metrics as above.

5.3.1 Results

The results from the top 1% dense subset show significant insights into the NCF model's handling of scenarios with high interaction density. The implementation of a multi-class scoring system is anticipated to significantly enhance the model's capability in delivering precise and relevant recommendations by effectively utilizing the rich interaction data.

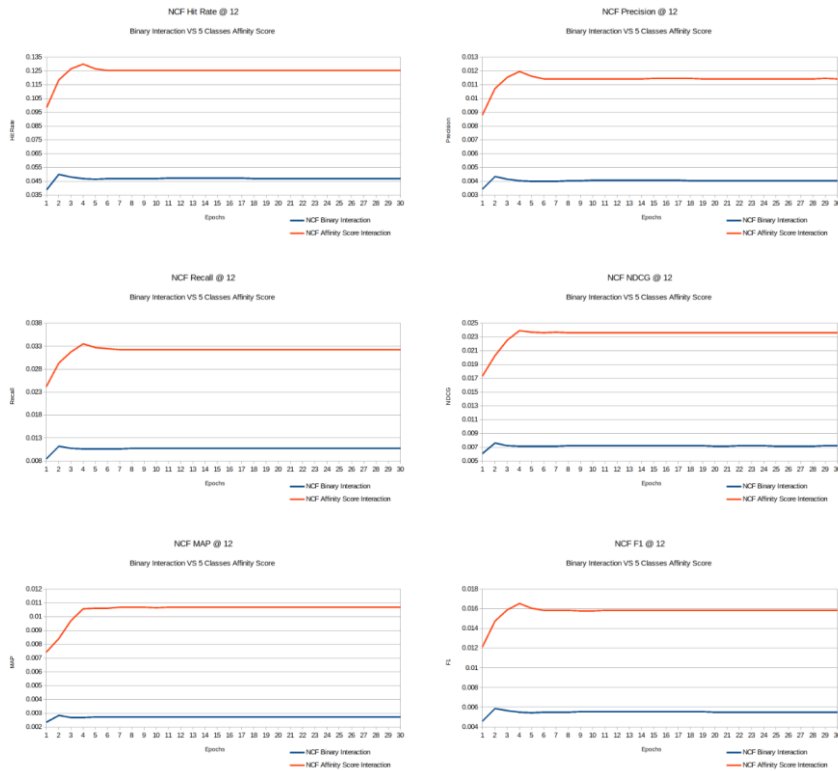


Figure 23: Neural Collaborative Filtering metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 1% subset of the initial dataset.

In the NCF model's evaluation all metrics are correlated. We observe that the recommendation quality for both the binary interaction and the affinity score interaction methods demonstrates a distinct pattern of early improvements followed by stabilization. The affinity score interaction method shows a steady increase in performance across all metrics until about the 4th epoch, where it begins to plateau, maintaining relatively stable values through subsequent epochs. This indicates that the model quickly adapts to the nuances of the multi-class scoring system but reaches its optimization limit early. In contrast, the binary interaction method shows minimal early gains and reaches a plateau

almost immediately from the 2nd epoch, indicating a more limited capacity to adapt and improve over time.

The binary interaction peaks at a Hit Rate of 0.04995, whereas the affinity score interaction reaches a significantly higher peak of 0.12982, marking an **increase** of approximately **160%**. This substantial improvement underlines the effectiveness of the affinity score in capturing more relevant user-item interactions.

Precision for the binary approach peaks at 0.00432, with the affinity score method rising to 0.01197, representing an impressive **177% improvement**. This reflects a markedly enhanced accuracy in the recommendations provided by the affinity score system.

The binary method achieves a maximum Recall of 0.01122, which is substantially enhanced to 0.03353 by the affinity score, **improving by about 199%**. This increase indicates that the multi-class system is significantly better at retrieving relevant items.

The NDCG for binary interaction tops at 0.00757, while the affinity score interaction climbs to a peak of 0.02395, an **enhancement of approximately 217%**. This improvement suggests a better ranking of recommended items that are more aligned with user preferences.

The maximum MAP for the binary interaction reaches 0.00284, compared to a much higher 0.01070 for the affinity score interaction, which is an **increase of around 277%**. This indicates a more precise prediction of user preferences across all recommended item sets.

The F1 Score for binary interaction reaches a peak of 0.00588, while for the affinity score interaction, it is significantly enhanced to 0.01654, marking a **181% improvement**.

The NCF model on the top 10% subset performs as follows:

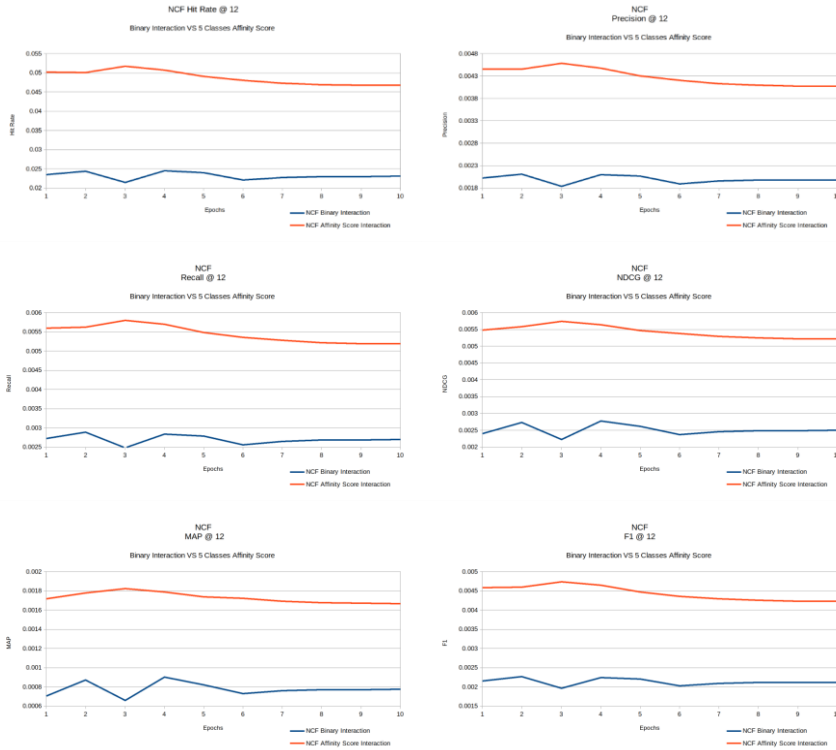


Figure 24: Neural Collaborative Filtering metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 10% subset of the initial dataset.

In the evaluation of the NCF model on the top 10% subset, we observed the following trends across all metrics. Both the binary interaction and affinity score interaction methods showed a pattern of early improvements followed by stabilization until about the 3rd epoch. This behavior indicates that the model quickly adapts to the nuances of the multi-class scoring system but reaches its optimization limit early. The binary interaction peaks at a Hit Rate of 0.024448, while the affinity score interaction reaches a significantly higher peak of 0.051718, marking an **112% improvement**. This substantial enhancement underscores the effectiveness of the affinity score in capturing more relevant user-item interactions.

Precision for the binary approach peaks at 0.002109, whereas the affinity score method rises to 0.004583, representing an **117% improvement**. This reflects a markedly enhanced accuracy in the recommendations provided by the affinity score system.

The binary method achieves a maximum Recall of 0.002886, which is substantially enhanced to 0.005796 by the affinity score, marking a **101% improvement**. This increase indicates that the multi-class system is significantly better at retrieving relevant items.

The NDCG for binary interaction tops at 0.002774, while the affinity score interaction climbs to a peak of 0.005741, showing a **107% improvement**. This suggests a better ranking of recommended items that are more aligned with user preferences.

The maximum MAP for the binary interaction reaches 0.000902, compared to a much higher 0.001823 for the affinity score interaction, indicating a **102% improvement**. This demonstrates a more precise prediction of user preferences across all recommended item sets.

Finally, the F1 Score for binary interaction peaks at 0.002269, while for the affinity score interaction, it is significantly enhanced to 0.004742, marking a **109% improvement**.

These results highlight the superior performance of the affinity score interaction method over the binary interaction method across all metrics, reaffirming its effectiveness in enhancing recommendation quality, especially in sparser datasets like the top 10% subset.

5.3.2 Discussion

In the denser top 1% of the dataset, the adoption of the multi-class affinity scoring method substantially improves all performance metrics measured. This enhancement highlights that in scenarios with frequent and varied interactions, a refined scoring system that can capture a broader spectrum of user-item interactions markedly boosts the accuracy and relevance of the model's recommendations. The results show significant gains across all metrics, with improvements ranging from 160% to 277% over the traditional binary interaction model. Such substantial increases underscore the capability of the affinity score formula to leverage dense interaction data effectively.

In the more expansive and sparser top 10% subset of the dataset, the adoption of the multi-class affinity scoring method continues to yield notable improvements across all performance metrics. We observe an increase in recommendation quality, with the metrics improving by **101% to 117%**. These results underscore the efficacy of the multi-class affinity scoring method in handling sparser datasets, where traditional binary interaction methods tend to struggle.

The consistency of these improvements across all metrics indicates that the affinity scoring method can significantly enhance the accuracy and relevance of recommendations in a variety of data scenarios, not just those with high interaction density.

5.4 Evaluation of the Proposed Affinity Score Formula on the DFNCF Model

Our methodology for training and obtaining results is identical to the evaluation procedure used for the above models. We chose optimal hyperparameters for the model and only altered the input data according to the two data preprocessing approaches: binary interaction and affinity score interaction.

5.4.1 Results

In the top 1% dense subset, the recommendation quality of the DFNCF model increases with epochs, with the affinity score interaction approach plateauing after epoch 4 and the binary interaction approach plateauing on epoch 13.



Figure 25: Demographic and Feature-enhanced Neural Collaborative Filtering metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 1% subset of the initial dataset.

The DFNCF model's performance on the dense subset using the affinity score interaction approach achieves a peak hit rate of 0.1339. In contrast, the binary interaction approach reaches a maximum hit rate of 0.0637. This represents a **110% improvement** in the hit rate using the affinity score interaction method.

The precision with the affinity score interaction approach reaches a peak of 0.0123. The binary interaction approach achieves a maximum precision of 0.0055. The precision improved by **122%** with the affinity score interaction method.

The recall using the affinity score interaction approach achieves a maximum of 0.0364.

The binary interaction approach reaches a maximum recall of 0.0155. We observed a **135% improvement** in recall using the affinity score interaction method.

The NDCG for the model with the affinity score interaction approach reaches a maximum of 0.0249. The binary interaction approach achieves a maximum NDCG of 0.0107. The NDCG improved by **134%** with the affinity score interaction method.

The MAP using the affinity score interaction approach reaches a peak of 0.0112. The binary interaction approach shows a maximum MAP of 0.0043. MAP improved by **160%** with the affinity score interaction method.

The F1 score for the DFNCF model using the affinity score interaction approach achieves a peak of 0.0172. The binary interaction approach reaches a maximum F1 score of 0.0077.

The F1 score improved by **125%** with the affinity score interaction method.

In the top 10% sparse subset, the recommendation quality of the DFNCF model shows improvement with epochs, with the affinity score interaction approach plateauing after epoch 3 and the binary interaction approach plateauing after epoch 2.



Figure 26: Demographic and Feature-enhanced Neural Collaborative Filtering metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 10% subset of the initial dataset.

The DFNCF model's performance on the sparse subset using the affinity score interaction seems to overfit after epoch 3. The multi-class approach achieves a peak hit rate of 0.0479. In contrast, the binary interaction approach reaches a maximum hit rate of 0.0328. This represents a **46% improvement** in the hit rate using the affinity score interaction method.

The precision with the affinity score interaction approach reaches a peak of 0.0042. The binary interaction approach achieves a maximum precision of 0.0029. The precision improved by **44%** with the affinity score interaction method.

The recall using the affinity score interaction approach achieves a maximum of 0.0053.

The binary interaction approach reaches a maximum recall of 0.0039. We observed a **36% improvement** in recall using the affinity score interaction method.

The NDCG with the affinity score interaction approach reaches a maximum of 0.0054. The binary interaction approach achieves a maximum NDCG of 0.0046. The NDCG improved by **18%** with the affinity score interaction method.

The MAP using the affinity score interaction approach reaches a peak of 0.0017. The binary interaction approach shows a maximum MAP of 0.0018. MAP decreased by **4%** with the affinity score interaction method.

The F1 score using the affinity score interaction approach achieves a peak of 0.0043. The binary interaction approach reaches a maximum F1 score of 0.0031. The F1 score improved by **40%** with the affinity score interaction method.

5.4.2 Discussion

The evaluation of the DFNCF model across different data densities—comparing the top 1% and top 10% subsets—highlights the effectiveness of the multi-class affinity score system. In the denser top 1% subset, the multi-class affinity scoring method significantly improved all performance metrics, showing that a nuanced scoring system enhances recommendation accuracy and relevance in datasets with frequent interactions. In the sparser top 10% subset, although the improvements were less pronounced, the affinity score interaction still outperformed the binary interaction in most metrics.

Overall, the multi-class affinity score formula proves to be versatile and consistently enhances recommendation quality across various data densities, making it a valuable tool for DFNCF-based recommender systems.

5.5 Evaluation of the Proposed Affinity Score Formula on the Multinomial RBM Model

In this section, we assess the impact of the proposed affinity score formula on the performance of the Multinomial RBM model across two different subsets of data: the top 1% dense subset and the top 10% more sparse subset. We compare two data preprocessing approaches: the traditional binary interaction, where the user-item affinity score is set to 0 for no interaction and 1 for interaction, and our proposed method that categorizes ratings into five classes according to our optimized formula coefficients. The Multinomial RBM model was trained using the same optimal hyperparameters but with these two distinct data preprocessing approaches to ascertain the impact of our affinity scoring method on various performance metrics. The model was evaluated over 30 epochs, and metrics recorded include Hit Rate at K (HR@K), Precision at K, Recall at K, Normalized Discounted Cumulative Gain at K (NDCG@K), Mean Average Precision at K (MAP@K), and F1 Score at K. This evaluation helps us understand how the affinity score formula influences the model's performance in different data density scenarios, providing insights into its applicability across varying scales of user-item interactions.

5.5.1 Results

The results shown in the following charts are derived from the top 1% dense subset of the dataset. This subset represents a scenario with higher interaction density, offering a rich context for testing the efficacy of our multi-class affinity score system. The enhancements in recommendation quality observed here may reflect the model's ability to leverage the increased informational content in denser interaction environments.

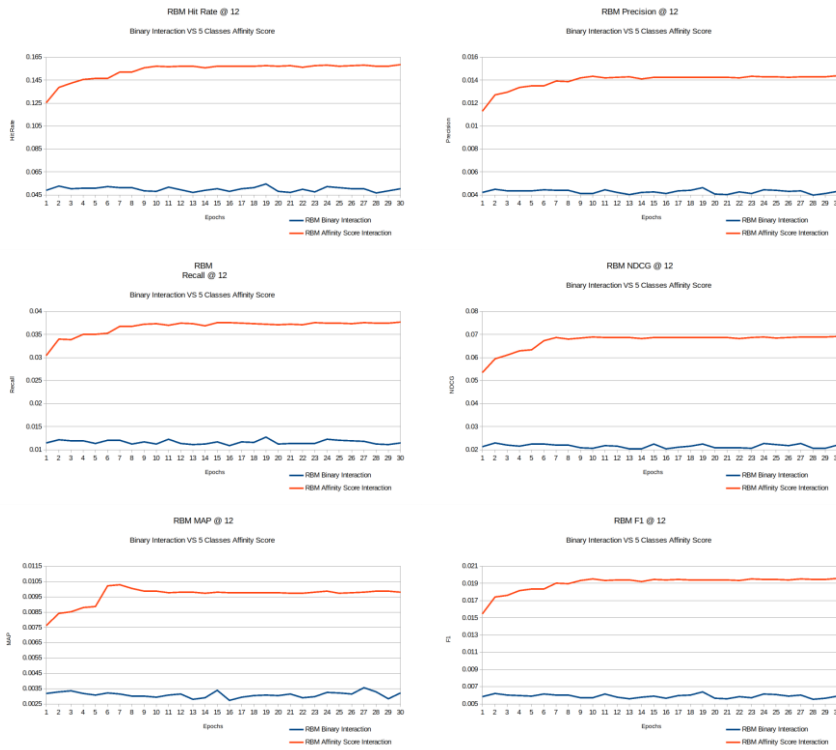


Figure 27: Multinomial Restricted Boltzmann Machine metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 1% subset of the initial dataset.

The Affinity Score interaction method shows an increasing trend through the epochs and plateaus after 10 epochs of training. On the other hand, the binary interaction method plateaus after just 2 epochs. For binary interaction, the hit rate peaked at 0.0544, whereas for the affinity score interaction, it reached a maximum of 0.1587, indicating a significant **improvement of 192%** with the multi-class system.

Since all metrics are correlated, we observe a similar behavior as before. The precision for binary interaction topped at 0.0046, while for the affinity score interaction, it climbed to a peak of 0.0144, meaning a **210% improvement**.

For binary interaction, recall at 12 reached a maximum of 0.0127, while for the affinity score interaction, the peak was substantially higher at 0.0377, showing a **196% improvement**.

The NDCG for binary interaction reached a high of 0.0230, whereas for the affinity score interaction, it significantly increased to 0.0690, indicating a **200% improvement**.

The maximum MAP at 12 for binary interaction was 0.0036, compared to a much higher 0.0103 for the affinity score interaction, indicating a **189% improvement**. The F1 score at 12 for binary interaction hit 0.0064, while for affinity score interaction, it was much improved at 0.0196, indicating an **improvement of 207%**.

Additionally, we analyze the performance of the model on the top 10% more sparse subset of the dataset. This part of the evaluation focuses on understanding how well the affinity score formula performs under conditions of sparser data availability, which is often a challenge in practical recommender system applications. The results from this subset help illustrate the robustness and adaptability of the proposed method across different levels of data sparsity.

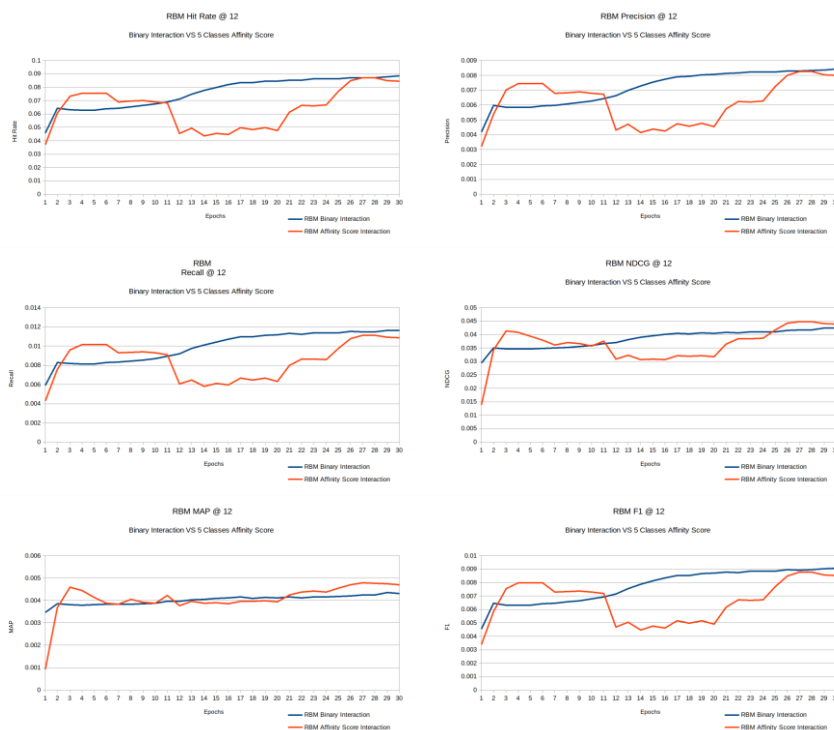


Figure 28: Multinomial Restricted Boltzmann Machine metrics at 12 items comparison chart with 2 data preprocessing of the target approaches, binary interaction and affinity score calculated with the proposed formula. The dataset used is the top 10% subset of the initial dataset.

The Affinity Score multi-class method indicates an increasing trend for the first 3 epochs, but plateaus at epoch 4, until epoch 10 when the Gibbs steps scheduler shifts from 1 Gibbs step to 3 Gibbs steps. where it results in a quality collapse, at epoch 20 when the Gibbs steps scheduler shifts to 5 Gibbs steps, the quality of the results increases and plateaus at epoch 27. This behavior could be because we may reach the capacity of the model by epoch 10 and we see a quality collapse. Then probably due to the reduce on plateau scheduler, with smaller learning rates we fine tune the weights to reach the maximum quality once again.

The binary interaction method demonstrates a more consistent improvement in recommendation quality through the epochs. It shows a steady increase in performance, reaching a plateau around the 15th epoch, indicating that the model has effectively learned the user-item interactions and optimized its recommendations.

The binary interaction peaks at a Hit Rate of 0.0884, whereas the affinity score interaction reaches a slightly lower peak of 0.0872, marking a **-1% decrease**. Precision for the binary approach peaks at 0.0084, with the affinity score method slightly decreasing to 0.0083, representing a **-2% decrease**. The binary method achieves a maximum Recall of 0.0116, which is slightly reduced to 0.0111 by the affinity score, resulting in a **-4% decrease**. The NDCG for binary interaction tops at 0.0425, while the affinity score interaction climbs to a peak of 0.0449, showing a **6% improvement**. The maximum MAP for the binary interaction reaches 0.0044, compared to a higher 0.0048 for the affinity score interaction, which is an **11% improvement**. The F1 Score for binary interaction reaches a peak of 0.0091, while for the affinity score interaction, it is slightly lower at 0.0088, marking a **-3% decrease**.

These results indicate that while the affinity score method shows improvements in certain metrics like NDCG and MAP, it experiences slight decreases in Hit Rate, Precision, Recall, and F1 Score compared to the binary interaction method.

5.5.2 Discussion

The evaluation of the Multinomial Restricted Boltzmann Machine (RBM) model across both the top 1% dense subset and the top 10% sparse subset reveals key insights into the effectiveness of the multi-class affinity score system. In the top 1% subset, characterized by higher interaction density, the affinity score interaction method significantly improves recommendation quality. The metrics score dramatic increases, ranging from 189% to 210%, demonstrating the method's ability to capture nuanced user-item interactions and enhance recommendation accuracy in dense datasets.

In the top 10% sparse subset, the binary interaction method shows a more consistent improvement, plateauing around the 15th epoch. While the affinity score method experiences some fluctuations, it still achieves comparable results. It shows slight decreases in Hit Rate (-1%), Precision (-2%), Recall (-4%), and F1 Score (-3%), but improves NDCG (6%) and MAP (11%). These results indicate that the affinity score method, while not always outperforming the binary method in every metric, enhances ranking quality and average precision, capturing more detailed user-item interactions even in sparse datasets.

Overall, the multi-class affinity score system proves robust and adaptable. It excels in dense datasets by significantly enhancing all performance metrics and offers advantages in ranking and precision in sparse datasets. This suggests that the affinity score system is a valuable tool for recommender systems, capable of improving performance across varying data densities.

5.6 Comparing the performance of the models

In this section, we will compare the performance of all models across various metrics and training time to assess their efficiency. The models were trained on a machine with an

Nvidia GPU RTX 4090, consisting of 24GB VRAM and 16,384 CUDA cores with clock speeds of 2595 Mhz. The machine also had 192GB of DDR5 RAM with clock speed of 5200 MHz. The CPU of the machine is Intel's i7 14700 with clock speed of 5.3 GHz, was not utilized for the training of the network as we leveraged PyTorch's ability to train models on the GPU, however it was used for transferring data between RAM and GPU VRAM and for minor calculations when using numpy arrays.

5.6.1 Comparing the models on the top 1% subset

We begin with a detailed examination of the binary interaction results on the top 1% dense subset, followed by comprehensive charts illustrating the performance of each model.



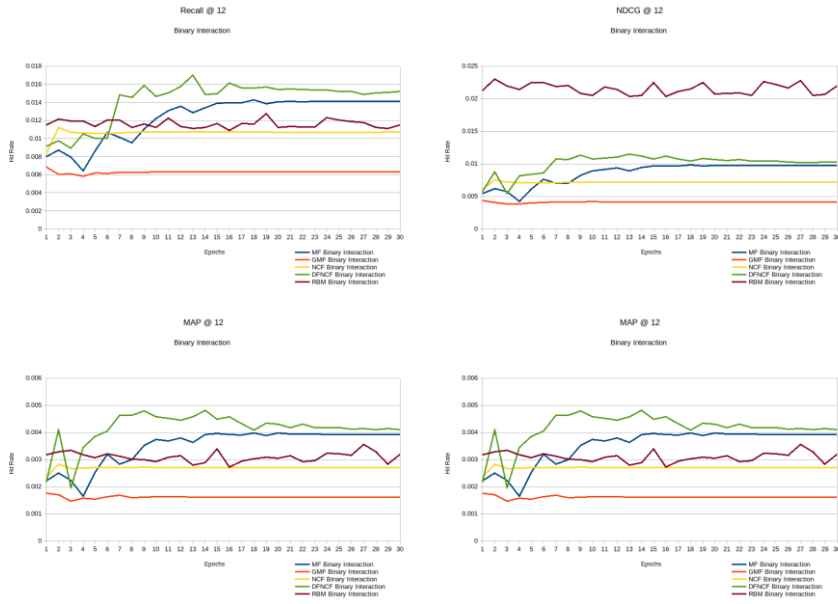


Figure 29: Comparative Performance of all 5 Models Across Different Metrics
 The charts illustrate the performance comparison of five models (MF, NMF, GMF, DFNCF, and RBM) on the top 1% subset of the dataset using the binary interaction method. Each chart represents one performance metric: Hit Rate @ 12, Precision @ 12, Recall @ 12, NDCG @ 12, MAP @ 12, and F1 Score @ 12. The x-axis shows the number of epochs, and the y-axis shows the corresponding metric values.

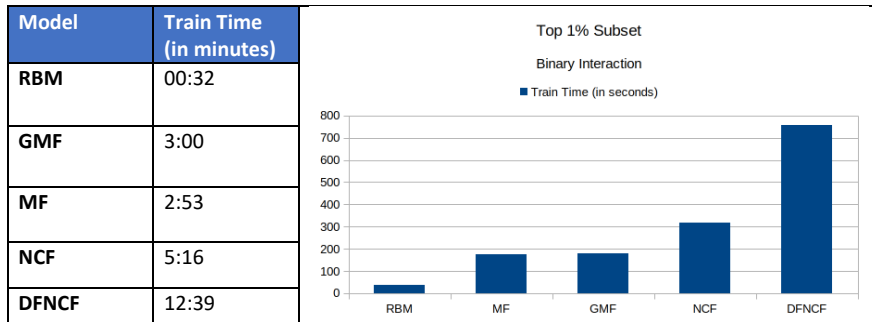


Figure 30: Training times for different models on the top 1% subset using Binary Interaction, with RBM being the fastest and DFNCF the slowest.

After studying the charts, we observed the following:
 DFNCF consistently achieves the highest metrics in almost all categories. These results indicate its

superior ability to capture user preferences and provide accurate recommendations. DFNCF requires the longest training time, clocking in at over 12 minutes. This extended time can be attributed to its complex modeling of user-item interactions and extra embedding inputs, which, while effective, demands significant computational resources.

RBM performs exceptionally well in terms of ranking quality, achieving the highest NDCG@12 of 0.0230 and competitive scores on the rest of the metrics. This indicates its robust capability in ranking relevant items higher. RBM stands out for its efficiency, with the shortest training time of just 32 seconds. This makes it an attractive option when quick model training is necessary, though it slightly trails DFNCF in some performance metrics.

NCF also demonstrates strong performance, with a training time of just over 5 minutes, NCF offers a balanced trade-off between performance and training duration. Its use of neural networks to model complex interactions helps achieve this balance.

MF shows good performance, showcasing the reason why it is considered as the established recommendation algorithm. These metrics indicate that MF, although simple, can effectively model user-item interactions but doesn't outperform DFNCF. MF's training time is relatively efficient compared to DFNCF, but less so than RBM.

GMF lags behind the other models in performance metrics, although it stabilizes quickly, its capability to capture and predict user preferences is limited. It has close training time to the MF model as it they have similar architecture.

Based on the comparative analysis, DFNCF emerges as the top-performing model in terms of recommendation quality, consistently achieving the highest metrics across most categories. However, this superior performance comes with the longest training time trade-off. MF follows closely in recommendation quality, striking a balance between effectiveness and training duration. RBM stands out for its exceptional efficiency, achieving competitive ranking quality with the shortest training time, making it a strong candidate when quick model training is crucial. NCF offers good performance metrics, but its efficiency falls behind as it takes moderate training time. GMF is the least effective in capturing user preferences and could be avoided for such a scenario.

We continue our exploration journey by comparing the models on the same top 1% dense subset but on the proposed multi-class interaction target, using the proposed formula during data preprocessing.

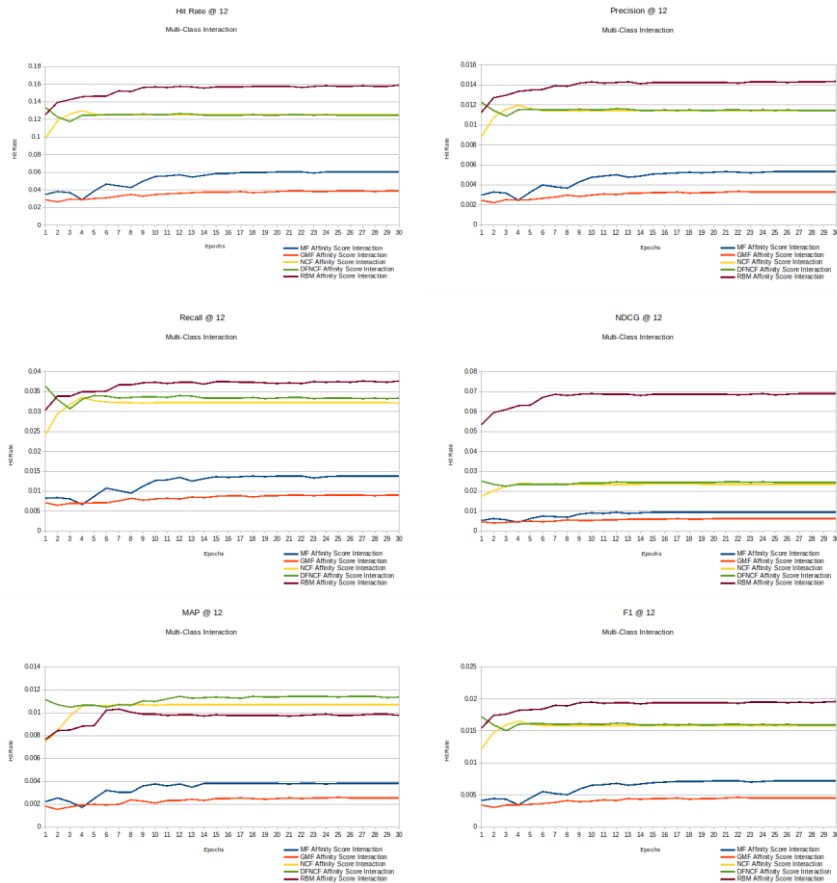


Figure 31: Comparative Performance of all 5 Models Across Different Metrics
 The charts illustrate the performance comparison of five models (MF, NMF, GMF, DFNCF, and RBM) on the top 1% subset of the dataset using the multi-class interaction method. Each chart represents one performance metric: Hit Rate @ 12, Precision @ 12, Recall @ 12, NDCG @ 12, MAP @ 12, and F1 Score @ 12. The x-axis shows the number of epochs, and the y-axis shows the corresponding metric values.

Model	Train Time (in minutes)
RBM	00:38
GMF	3:00
MF	2:54

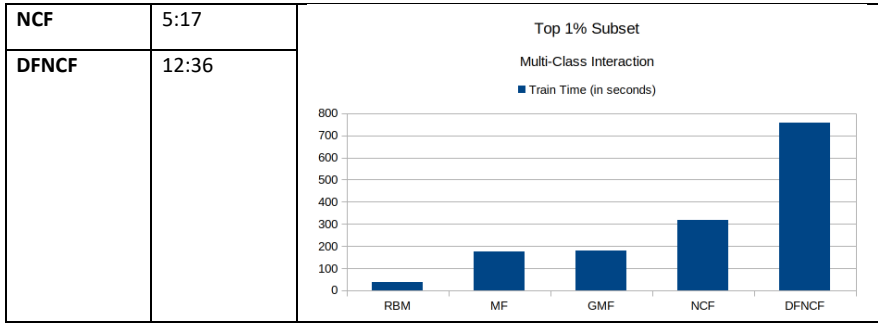


Figure 32: Training times for different models on the top 1% subset using Multi-Class Interaction, with RBM being the fastest and DFNCF the slowest.

After examining the results of the multi-class interaction target, we observed similar results regarding the training times of the models, compared to the binary interaction. The RBM is still the fastest, even with the increase in the number of nodes in the visible layer from 2 to 5, and more nodes in the hidden layer, as the hidden layer size is relative to the visible layer and, as found from our fine-tuning procedure, accounts for 35% of the size of the visible layer.

Regarding recommendation quality, the RBM model leverages the more nuanced information created from the proposed formula most effectively. RBM outperforms all models across all metrics except MAP@12. DFNCF is the second-best performer, surpassing all other models except RBM across all metrics. In the MAP metric, DFNCF outperforms RBM.

NCF ranks third in recommendation quality and is very close to DFNCF in performance. MF and GMF models perform similarly, with MF performing slightly better, but both are significantly worse than the top three models. This result suggests that the simplicity of the product dot element approach on the embeddings in MF and GMF fails to recognize user-item affinities as effectively as the deep learning models.

Based on the comparative analysis of the multi-class interaction target, RBM emerges as the top-performing model, achieving the highest metrics across almost all categories with the shortest training time.

5.6.2 Comparing the models on the top 10% subset

Here is a detailed examination of the binary interaction results on the top 10% expansive subset with charts illustrating the performance of each model.

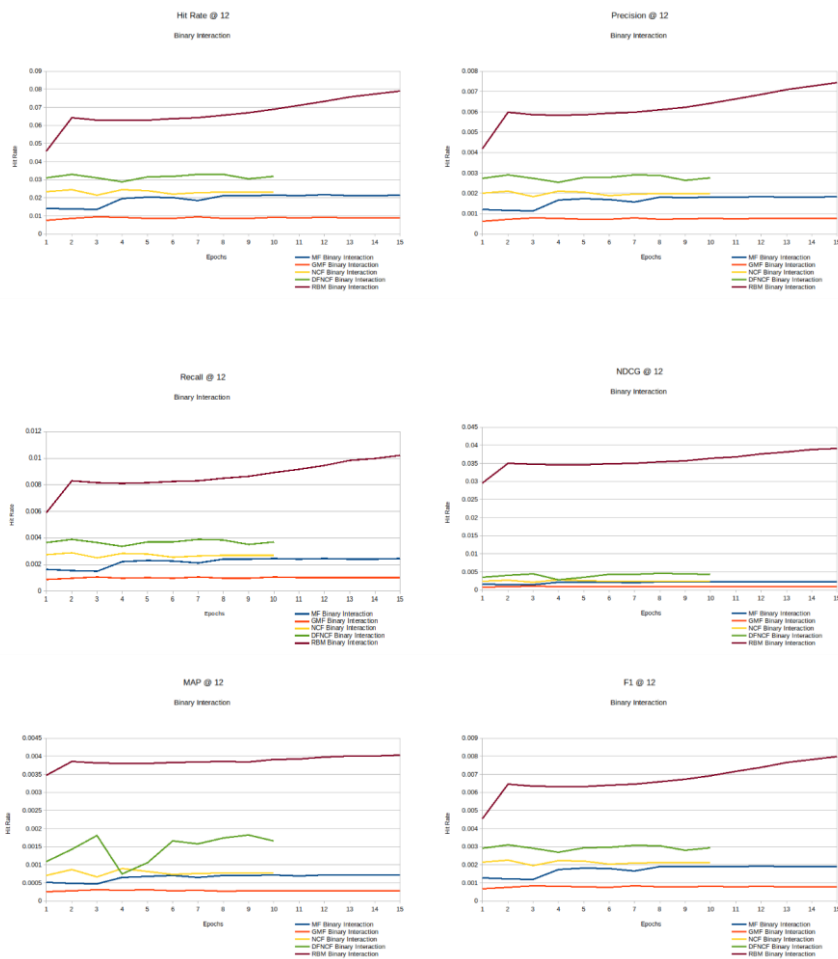


Figure 33: Comparative Performance of all 5 Models Across Different Metrics
 The charts illustrate the performance comparison of five models (MF, NMF, GMF, DFNCF, and RBM) on the top 10% subset of the dataset using the binary interaction method. Each chart represents one performance metric: Hit Rate @ 12, Precision @ 12, Recall @ 12, NDCG @ 12, MAP @ 12, and F1 Score @ 12. The x-axis shows the number of epochs, and the y-axis shows the corresponding metric values.

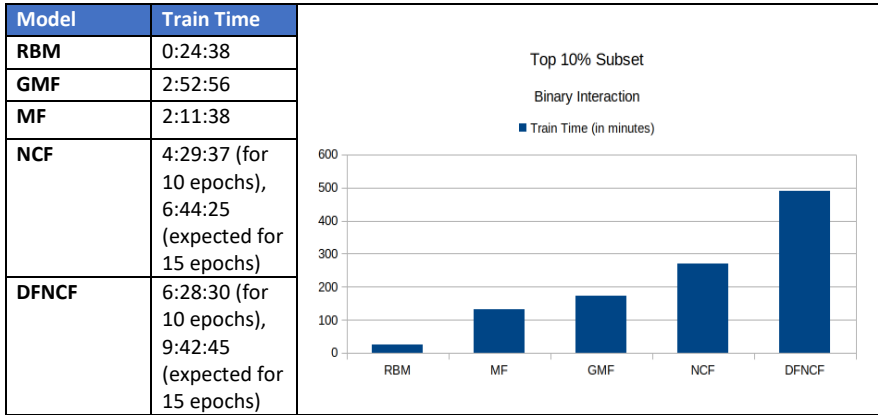


Figure 34: Training times for different models on the top 10% subset using Binary Interaction, with RBM being the fastest and DFNCF the slowest.

After examining the performance of the models on the top 10% subset using binary interactions, we observed the following results:

The RBM model outperforms all other models across all metrics, making it the best performing and most efficient model. It achieves the highest score across all metrics. Additionally, RBM has the shortest training time of just over 24 minutes, highlighting its robust capability in ranking relevant items higher and offering exceptional efficiency. The RBM model performs exceptionally well in terms of ranking quality, achieving the highest NDCG@12 of 0.0391 and competitive scores on the other metrics. Its efficiency is notable, with the shortest training time of just over 24 minutes, making it an attractive option when quick model training is necessary. RBM's ability to rank relevant items higher makes it a strong candidate despite slightly trailing DFNCF in some performance metrics. The DFNCF model ranks second in terms of performance. It achieves high metrics across most categories, demonstrating its superior ability to capture user preferences and provide accurate recommendations. However, this model requires the longest training time at over 6 hours. The complexity of modeling user-item interactions and additional embedding inputs contribute to its extended training duration.

The NCF model ranks third, providing good recommendation quality. It offers a balanced trade-off between performance and training duration, with a training time of just under 4.5 hours. NCF effectively leverages neural networks to model complex interactions, achieving strong precision and recall metrics, though it falls behind DFNCF in overall performance. Its performance metrics, particularly in recall, make it a dependable model, although it falls short of the top performers.

The MF model, although simple, ranks fourth. It shows good performance with a relatively efficient training time of approximately 4.5 hours. While it effectively models user-item interactions, MF does not outperform the top three models in recommendation quality. Its recall and precision metrics are reliable but not as high as those of RBM, DFNCF, and NCF.

The GMF model is the worst performing among all models. It takes over 2 hours to train, which is longer than the MF model. GMF's ability to capture and predict user preferences is limited, resulting in lower performance metrics across the board.

In the comparative analysis of the top 10% subset using binary interactions, RBM emerges as the top-performing model, achieving the highest metrics across all categories with the shortest training time. This makes RBM both the best performing and the most efficient model. DFNCF, while achieving high metrics, requires the longest training time and thus ranks second. Our proposed advanced models seem to outperform the traditional models in more scaled problems.

We also note that the NCF and DFNCF models were trained for 10 epochs instead of 15 due to their long training times and the performance plateau observed. This decision was made to manage the extensive training times while still achieving reliable performance results within the thesis time constraints.

We continue with the multi-class interaction comparison

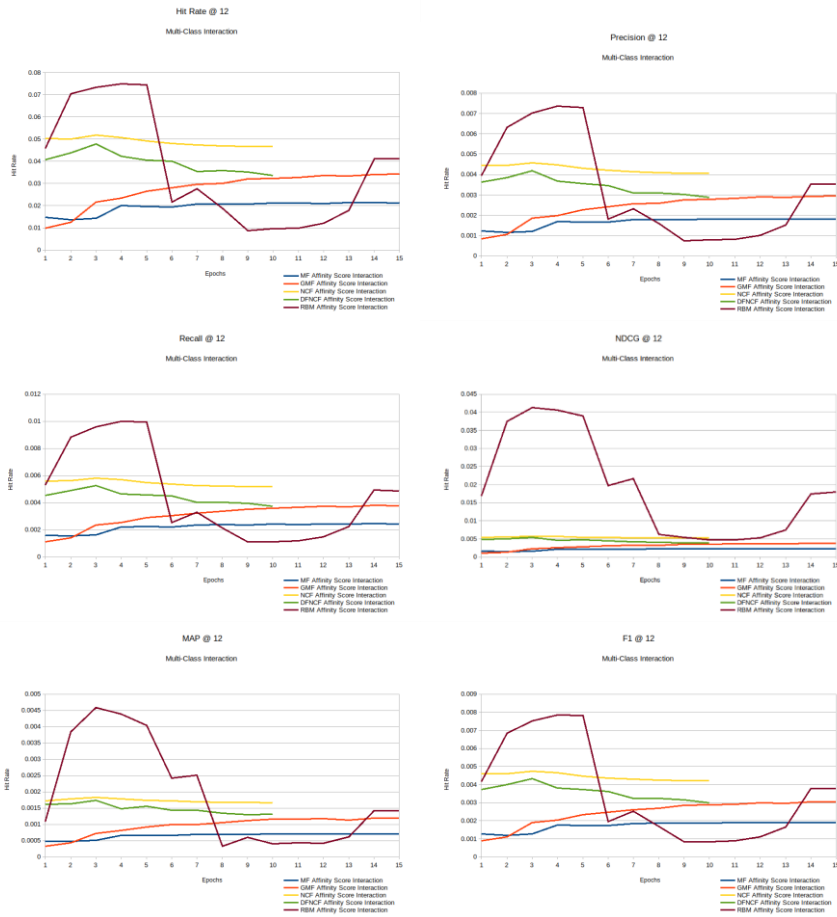


Figure 35: Comparative Performance of all 5 Models Across Different Metrics
 The charts illustrate the performance comparison of five models (MF, NMF, GMF, DFNCF, and RBM) on the top 10% subset of the dataset using the multi-class interaction method. Each chart represents one performance metric: Hit Rate @ 12, Precision @ 12, Recall @ 12, NDCG @ 12, MAP @ 12, and F1 Score @ 12. The x-axis shows the number of epochs, and the y-axis shows the corresponding metric values.

The training times for the models are identical with the ones measured on the binary interaction. The only model with different training time is the RBM as it has now more nodes in the visible layer, as we need 5 nodes per product to represent the rating, instead of 2 and the hidden layer size grows proportionally to the visible layer. The training time for the RBM doubles to 1 hour.

By analyzing the results, we conclude that:

The RBM model once again outperforms all other models across all metrics, solidifying its status as the best performing and most efficient model. However, after epoch 5, RBM's performance plateaus, indicating it has maybe reached its capacity. It is probable we assigned a limiting number of hidden nodes to solve the problem on the expansive dataset, as the proportion of hidden to visible nodes was set to 35% during tuning the model on the smaller dataset subset at section 4.9. This leads to a decline in recommendation quality. Despite this, RBM remains the top choice for both performance and efficiency.

Interestingly, the NCF model outperforms the more advanced DFNCF model across all metrics in the multi-class interaction setting. NCF demonstrates strong recommendation quality, leveraging neural networks to effectively model complex user-item interactions. Its training time is reasonable compared to DFNCF, making it a robust and efficient option.

The DFNCF model, while still performing well, is outperformed by NCF in this setting. It shows good metrics but does not reach the same level of precision, recall, or ranking quality as NCF. DFNCF's extended training time remains a significant factor, requiring careful consideration when choosing a model for deployment.

The GMF model demonstrates improved performance in the multi-class interaction scenario, outperforming the MF model. GMF effectively leverages the more insightful multi-class interaction data, achieving better metrics across the board compared to MF. This improvement highlights GMF's potential in scenarios where deeper insights from data can be harnessed.

The MF model, which performed relatively well in the binary interaction scenario, is now outperformed by GMF. The traditional simplicity of MF does not capture the nuanced information as effectively as GMF in the multi-class interaction setting. As a result, MF ranks lower in this comparison.

In the multi-class interaction scenario for the top 10% subset, RBM once again emerges as the best performing and most efficient model, with its performance peaking before potential overfitting. NCF surprisingly surpasses DFNCF, showcasing strong metrics and efficient training. DFNCF, while still effective, is edged out by NCF in this context. GMF outperforms MF, leveraging the multi-class interaction data more effectively. These observations suggest that RBM is consistently the top

model, with NCF and DFNCF offering robust alternatives depending on the specific requirements and constraints of the recommendation system.

Chapter 6: Conclusion

In this thesis, we have delved deeply into the development and evaluation of advanced machine learning-based recommender systems, aimed at optimizing product recommendations in the e-commerce sector using the H&M dataset. Our comprehensive approach has yielded several significant conclusions, highlighting the effectiveness of our methods and providing valuable insights for future research and practical applications.

Affinity Score Formula Impact

One of this thesis' pivotal contributions is introducing a novel multi-class affinity score formula. This formula classifies user-item interactions into five distinct classes, as opposed to the traditional binary interaction method. The multi-class approach significantly enhances the granularity and accuracy of the recommendations.

- **Enhanced Performance:** The multi-class affinity score formula consistently outperformed the traditional binary method across GFM, NCF and DFNCF models and datasets tested. This improvement was especially pronounced in the denser dataset, where the detailed classification of interactions helped the models better understand user preferences and make more accurate predictions. On the MF and RBM it achieved similar performance.
- **Robustness and Adaptability:** The success of the multi-class formula in both dense and sparse datasets underscore its robustness and adaptability. This formula can effectively capture the nuances of user behavior, making it a valuable tool for a wide range of e-commerce applications. Since the proposed formula is calculated in the data preprocessing step it can work with any machine learning model, without any tweaks in the architecture.

-

Model Performance

We evaluated the performance of five different models where:

- **The Multinomial RBM model** emerged as the top performer, achieving the highest metrics across almost all evaluation categories. Its ability to handle sparse datasets effectively and its relatively short training times make it an excellent choice for real-world applications where computational efficiency is crucial.
- **The DFNCF model** also demonstrated high performance, particularly excelling in the dense dataset when using the binary interaction approach. However, its complexity resulted in longer training times. Despite this, the inclusion of demographic and item-specific features, makes it cold-start-proof to new users or items, as they can be clustered to the known demographic and item-specific features embeddings.

The above more advanced models perform better than the following traditional ones.

- **The MF model** served as a robust baseline due to its traditional establishment, simplicity and effectiveness in collaborative filtering. It performed well in scenarios with dense data but showed limitations in handling sparse datasets. It could not leverage the more nuanced multi-class interaction data. The primary advantage of MF is its computational efficiency and ease of implementation, making it suitable for applications where quick deployment and scalability are essential. However, its linear nature limits its ability to capture complex interactions between users and items.
- **GMF** extends the traditional MF model by incorporating a layer for nonlinear transformations. This augmentation enhances its capacity to capture intricate user-item interactions, offering improved recommendation quality over MF in sparse datasets.
- **NCF** demonstrated high performance across various datasets, excelling in capturing non-linear relationships between users and items. However, it falls behind the DFNCF and RBM network.

While our results are derived from a dataset specific to the fashion e-commerce industry, the principles and methodologies we employed have broader applicability. The proposed affinity score formula relies on transaction dates and purchasing quantities, which are common data points in most e-commerce and retail datasets. This suggests that our approach can potentially be generalized to other sectors within the e-commerce and retail industries, provided that similar transactional data is available.

The comparative analysis of different collaborative filtering modules, as presented in our study, also offers valuable insights that extend beyond the specific use case of fashion recommendations. The methodologies for evaluating and contrasting collaborative filtering techniques can be applied to a wide range of recommendation systems, regardless of the industry.

This thesis has made significant strides in advancing the understanding and development of recommender systems using machine learning techniques. Our novel multi-class affinity score formula, along with the comprehensive evaluation of multiple models, has demonstrated how integrating detailed user and item data can substantially enhance recommendation accuracy. The findings highlight the importance of robust, adaptable, and scalable approaches in developing effective recommender systems for the e-commerce industry.

Chapter 7: Future Works

This thesis has explored various machine learning techniques to improve recommender systems' accuracy and robustness. While significant advancements have been made, several promising directions for future research could further enhance these systems' capabilities.

Fusion of DFNCF and GMF to Create DFNeuMF

The fusion of the Demographic and Feature-enhanced Neural Collaborative Filtering (DFNCF) with Generalized Matrix Factorization (GMF) could potentially enhance the recommendation quality. This approach is inspired by the Neural Collaborative Filtering (NeuMF) model [16], which combines MLP with GMF to capture both linear and non-linear user-item interactions.

Incorporating Raw Inputs into MLP

The current Affinity Score formula synthesizes various factors, such as the quantities of the same product bought by the user and the weeks since the last purchase. Future research could involve:

- **Using Raw Inputs Directly:** Feeding the raw quantities and weeks since the last purchase as inputs into the MLP, allowing the neural network to determine their relationship with the binary interaction target.
- **Model Training and Comparison:** Training the model with these raw inputs and comparing its performance to the current approach to assess improvements in prediction accuracy.

Adding a Conditional Demographic Layer to RBM

Incorporating demographic information, such as age, can further personalize recommendations. Inspired by the Occupation-Aware RBM [13], future work could involve:

- **Conditional Demographic Layer:** Adding a layer to the RBM network that introduces age information, enhancing the model's ability to make age-specific recommendations.
- **Model Evaluation:** Testing the modified RBM to evaluate the impact of demographic information on recommendation accuracy.

Testing on Diverse Datasets

To ensure the generalizability and robustness of the developed models, it is essential to test them on datasets of different natures and industries. Future research could focus on:

- **Cross-Domain Evaluation:** Applying the models to datasets from various domains, such as healthcare, finance, and entertainment, to assess their performance across different contexts.
- **Industry-Specific Adjustments:** Identifying and incorporating industry-specific features that may enhance recommendation accuracy.

Labeling Latent Features

Understanding and interpreting the latent features detected by the models can provide valuable insights into user preferences and item characteristics. Future work could involve:

- **Latent Feature Labeling:** Developing methods to label and interpret the latent features identified by the models, providing more transparency and interpretability.
- **Visualization and Analysis:** Creating visualization tools to explore these latent features, helping stakeholders understand the factors driving recommendations.

Continuous research and innovation in these areas will help address the remaining challenges and fully realize the potential of these systems in various applications.

References:

1. Goldberg, D., Nichols, D., Oki, B. M. & Terry, D. (1992). "Using Collaborative Filtering to Weave an Information Tapestry." *Communications of the ACM*, 35(12), pp. 61-70.
2. Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., & Riedl, J. (1994). "GroupLens: An Open Architecture for Collaborative Filtering of Netnews." *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, pp. 175-186.
3. Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). "Item-Based Collaborative Filtering Recommendation Algorithms." *Proceedings of the 10th International World Wide Web Conference (WWW10)*, pp. 285-295.
4. Schafer, J.B., Frankowski, D., Herlocker, J. & Sen, S. (2007). "Collaborative Filtering Recommender Systems". In: Brusilovsky, P., Kobsa, A., Nejdl, W. (eds) *The Adaptive Web. Lecture Notes in Computer Science*, vol 4321. Berlin: Springer, pp. 291-324.
5. Pazzani, M. J., & Billsus, D. (2007). "Content-Based Recommendation Systems." In *The Adaptive Web: Methods and Strategies of Web Personalization*, Brusilovsky, P., Kobsa, A., & Nejdl, W. (Eds.), pp. 325-341. Springer Berlin Heidelberg.
6. Burke, R. (2002). "Hybrid Recommender Systems: Survey and Experiments." *User Modeling and User-Adapted Interaction*, 12(4), 331-370.
7. Koren, Y., Bell, R., & Volinsky, C. (2009). "Matrix Factorization Techniques for Recommender Systems." *IEEE Computer*, 42(8), 30-37.
8. Salakhutdinov, R., Mnih, A., & Hinton, G. (2007). "Restricted Boltzmann Machines for Collaborative Filtering." *Proceedings of the 24th International Conference on Machine Learning*, pp. 791-798.

9. Hu, Y., Koren, Y., & Volinsky, C. (2008). "Collaborative Filtering for Implicit Feedback Datasets." *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pp. 263-272.
10. Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). "Deep Learning Based Recommender System: A Survey and New Perspectives." *ACM Computing Surveys*, 52(1), Article 5.
11. Krishnan, R., et al. (2021). "RFM-Based Customer Analysis and Product Recommendation System." *Advanced Computing and Intelligent Technologies Proceedings of ICACIT 2021*, pp. 623-632.
12. Biswas, P. K., et al. (2022). "A Hybrid Recommender System for Recommending Smartphones to Prospective Customers." *Expert Systems with Applications*, 185, Article 115675.
13. Xie, W., et al. (2016). "User Occupation Aware Conditional Restricted Boltzmann Machine Based Recommendation." *Proceedings of the 2016 IEEE Conference on Cyber, Physical and Social Computing*, pp. 518-521.
14. Yamashita, T., et al. (2014). "To Be Bernoulli or to Be Gaussian, for a Restricted Boltzmann Machine." *Proceedings of the 22nd International Conference on Pattern Recognition*, pp. 1526-1531.
15. Ben Yedder, H., et al. (2017). "Modeling Prediction in Recommender Systems Using Restricted Boltzmann Machine." *Proceedings of the 2017 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 2343-2348.
16. He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T.-S. (2017). "Neural Collaborative Filtering." In *Proceedings of the 26th International Conference on the World Wide Web*, pp. 173-182.
17. Zhang, Q., Cao, L., Zhu, C., Li, Z., & Sun, J. (2018). "CoupledCF: Learning Explicit and Implicit User-item Couplings in Recommendation for Deep Collaborative Filtering."

Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18), pp. 3662-3668.

18. Kingma, D. P., & Ba, J. (2014). "Adam: A Method for Stochastic Optimization." Proceedings of the International Conference on Learning Representations (ICLR).
19. Khodamoradi, A., Denolf, K., Vissers, K., et al. (2021). "ASLR: An Adaptive Scheduler for Learning Rate." Proceedings of the International Joint Conference on Neural Networks (IJCNN), pp. 1-8.
20. Rehman, I.u., Hanif, M.S., Ali, Z. et al. (2023) Empowering neural collaborative filtering with contextual features for multimedia recommendation. *Multimedia Systems* 29, pp. 2375–2388. <https://doi.org/10.1007/s00530-023-01107-9>

APPENDIX

1. Large code snippets and code documentation

```
class BaseNetwork(nn.Module):
    """
    Abstract base network class that defines the
    common metrics lists and functions for all models.
    """
    def __init__(self):
        super(BaseNetwork, self).__init__()
        self.training_loss = []
        self.test_loss = []
        self.training_errors = []
        self.test_errors = []
        self.hr = []
        self.precision = []
        self.recall = []
        self.ndcg = []
        self.map_k = []
        self.f1 = []

    def forward(self):
        raise NotImplementedError

    def save_model(self, path):
        with open(path, 'wb') as f:
            pickle.dump(self, f)

    def load_model(self, path):
        with open(path, 'rb') as f:
            return pickle.load(f)

    class Meta:
        abstract = True
```

Figure 36: Code snippet of the BaseNetwork class in PyTorch, designed as an abstract base class for neural network models, including metric tracking and model saving/loading functionalities.

```
class MF(BaseNetwork):
    def __init__(self, num_users, num_items, embed_dim=8):
        super(MF, self).__init__()
        self.user_embedding = nn.Embedding(num_users, embed_dim)
        self.item_embedding = nn.Embedding(num_items, embed_dim)

    def forward(self, user_indices, item_indices):
        user_embedding = self.user_embedding(user_indices)
        item_embedding = self.item_embedding(item_indices)
        output = (user_embedding * item_embedding).sum(1)
        return output.squeeze()
```

Figure 37: Implementation of the Matrix Factorization class in PyTorch, defining a neural network module for collaborative filtering with methods for embedding users and items.

```

class GMF(BaseNetwork):
    def __init__(self, num_users, num_items, embed_dim=8):
        super(GMF, self).__init__()
        self.user_embedding = nn.Embedding(num_users, embed_dim)
        self.item_embedding = nn.Embedding(num_items, embed_dim)
        self.affine_output = nn.Linear(embed_dim, 1)
        self.activation = nn.Sigmoid()

    def forward(self, user_indices, item_indices):
        user_embedding = self.user_embedding(user_indices)
        item_embedding = self.item_embedding(item_indices)
        interaction = torch.mul(user_embedding, item_embedding) # Element-wise product
        output = self.activation(self.affine_output(interaction)) # Linear transformation followed by a sigmoid activation
        return output.squeeze()

```

Figure 38: Implementation of the Generalized Matrix Factorization class in PyTorch, defining a neural network module for collaborative filtering with methods for embedding users and items.

```

class NCF(BaseNetwork):
    def __init__(self, num_users, num_items, embed_dim=8, number_of_hidden_units=64, number_of_hidden_layers=3):
        super(NCF, self).__init__()
        self.user_embedding = nn.Embedding(num_users, embed_dim)
        self.item_embedding = nn.Embedding(num_items, embed_dim)
        layers = [ nn.Linear(embed_dim * 2, number_of_hidden_units),
                  nn.ReLU(), ]
        # Dynamically add hidden layers
        for i in range(1, number_of_hidden_layers):
            input_size = number_of_hidden_units // (2 ** (i-1))
            output_size = number_of_hidden_units // (2 ** i)
            layers.append(nn.Linear(input_size, output_size))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(number_of_hidden_units // (2 ** (number_of_hidden_layers - 1)), 1))
        layers.append(nn.Sigmoid())
        self.fc_layers = nn.Sequential(*layers)

    def forward(self, user_indices, item_indices):
        user_embedding = self.user_embedding(user_indices)
        item_embedding = self.item_embedding(item_indices)
        x = torch.cat([user_embedding, item_embedding], dim=-1)
        x = self.fc_layers(x)
        return x.squeeze()

```

Figure 40: Implementation of the Neural Collaborative Filtering class in PyTorch, defining a neural network module for collaborative filtering with methods for embedding users and items.

```

class RecommendationDataset(Dataset):
    def __init__(self, users, items, ratings, classes = 5, needs_normalization = True):
        self.users = torch.tensor(users, dtype=torch.int64)
        self.items = torch.tensor(items, dtype=torch.int64)
        if needs_normalization:
            self.ratings = torch.tensor((ratings - 1) / (classes - 1), dtype=torch.float32)
        else:
            self.ratings = torch.tensor(ratings, dtype=torch.float32)

    def __len__(self):
        return len(self.ratings)

    def __getitem__(self, idx):
        return self.users[idx], self.items[idx], self.ratings[idx]

```

Figure 41: Definition of the RecommendationDataset class in PyTorch, constructing a dataset suitable for a recommendation system with methods for initialization, length retrieval, and item access, including optional normalization of ratings.

The class, derived from PyTorch's Dataset, performs several essential functions:

- **Initialization (`__init__` method):** This method accepts arrays of users, items, and ratings, along with two optional parameters: `classes`, which defaults to 5, and `needs_normalization`, a boolean that indicates whether the rating data requires normalization. The parameters `users` and `items` are converted into PyTorch tensors with data type `torch.int64`, suitable for indexing operations within embedding layers.
- **Normalization:** If `needs_normalization` is set to `True`, the ratings are normalized to a scale from 0 to 1. This is done by subtracting 1 from each rating (assuming ratings start from 1), dividing by `classes - 1`. This is needed as the MLP network with sigmoid function returns a value between 0 and 1 and we need to align the target output to this constraint.
- **Data Retrieval (`__getitem__` method):** This method retrieves an individual sample from the dataset at a specified index, returning the user ID, item ID, and the normalized or raw rating. This functionality is critical for training the model in batches and individually processing each interaction during the training and evaluation phases.
- **Dataset Length (`__len__` method):** It provides the total number of ratings available in the dataset, which is used by data loaders in PyTorch to manage batch processing effectively.

```
def load_datasets(train_filename, test_filename, active_items_filename):
    # Load training and test data
    print("Loading training and test data")
    train_df = pd.read_parquet(train_filename)
    test_df = pd.read_parquet(test_filename)

    print("Loading active items per customer")
    active_items_df = pd.read_parquet(active_items_filename)
```

Figure 42: Python function `load_datasets` for loading training, test, and active items data from Parquet files using pandas, including print statements to track the progress of data loading.


```

def encode_data(train_df, test_df, active_items_df):
    # Concatenate train and test dataframes
    full_df = pd.concat([train_df, test_df])

    # Initialize encoders
    user_encoder = LabelEncoder()
    item_encoder = LabelEncoder()

    # Fit encoders on the concatenated data and transform customer and article IDs
    full_df['customer_id'] = user_encoder.fit_transform(full_df['customer_id'])
    full_df['article_id'] = item_encoder.fit_transform(full_df['article_id'])

    # Transform customer and article IDs in training data using the fitted encoders
    train_df['customer_id'] = user_encoder.transform(train_df['customer_id'])
    train_df['article_id'] = item_encoder.transform(train_df['article_id'])

    # Transform customer and article IDs in test data using the fitted encoders
    test_df['customer_id'] = user_encoder.transform(test_df['customer_id'])
    test_df['article_id'] = item_encoder.transform(test_df['article_id'])

    # Transform customer and article IDs in active items data using the fitted encoders
    active_items_df['customer_id'] = user_encoder.transform(active_items_df['customer_id'])
    active_items_df['article_id'] = item_encoder.transform(active_items_df['article_id'])

    # Return datasets, counts of users and items, and potentially the active items DataFrame
    return train_df, test_df, active_items_df, user_encoder, item_encoder

```

Figure 43: Python function `encode_data` for preprocessing customer and item IDs in training, test, and active items datasets using `LabelEncoder`. The function concatenates datasets for unified encoding, applies transformation, and returns encoded data, allowing for consistent reference across datasets in a recommendation system.

- **Initialization:** Upon initialization, the `LabelEncoder` is fitted on the combined dataset of user and item identifiers from both training and test datasets. This ensures a consistent encoding scheme across the entire data, preventing any discrepancies during model training or predictions.
- **Transformation:** After fitting, the encoder transforms the user and item identifiers in both training and test datasets into normalized indices. This step is vital to maintain data consistency and integrity, ensuring that every unique identifier is assigned a unique index.

```

def evaluate_active_items(model, actual_items_dict=dict, active_items_per_user, k):
    """Evaluate the model on user-specific active items using test data for actual interactions."""
    hr_list, p_list, r_list, ndcg_list, ap_list, f1_list = [], [], [], [], [], []

    # Predict scores and sort active items per user
    user_specific_predictions = predict_scores_of_active_items_only(model, list(active_items_per_user.keys()), active_items_per_user)

    # Calculate metrics for each user
    for user, items_and_scores in user_specific_predictions.items():
        actual_items = actual_items_dict.get(user)
        if actual_items is None:
            logging.debug(f"No actual items found for user {user}")
            # logging.debug(f"Predicted items: {items_and_scores}")
            continue # Skip if no actual items found for the user in the test set

        top_k_predictions = [item for item, score in items_and_scores[:k]]

        hr = hit_rate_at_k(actual_items, top_k_predictions, k)
        p = precision_at_k(actual_items, top_k_predictions, k)
        r = recall_at_k(actual_items, top_k_predictions, k)
        ndcg = ndcg_at_k(actual_items, top_k_predictions, k)
        ap = average_precision_at_k(actual_items, top_k_predictions, k)
        f1 = f1_score_at_k(p, r)

        hr_list.append(hr)
        p_list.append(p)
        r_list.append(r)
        ndcg_list.append(ndcg)
        ap_list.append(ap)
        f1_list.append(f1)

    return np.mean(hr_list), np.mean(p_list), np.mean(r_list), np.mean(ndcg_list), np.mean(ap_list), np.mean(f1_list)

```

Figure 44: Python function for evaluating the relevance of the top k recommendations against the actual purchased items per user.

```

def predict_scores_of_active_items_only(model, users, active_items_per_user):
    """Predict scores for user-specific active items."""
    predictions = {}
    model.eval()
    with torch.no_grad():
        for user in tqdm(users, desc="Predicting Active Items", leave=False, total=len(users)):
            active_items = active_items_per_user[user]
            user_tensor = torch.tensor([user] * len(active_items), dtype=torch.long).squeeze().to(device)
            item_tensor = torch.tensor(active_items, dtype=torch.long).to(device)
            scores = model(user_tensor, item_tensor).cpu().numpy()
            # Pair each score with the corresponding item ID and store them
            predictions[user] = sorted(zip(active_items, scores), key=lambda x: x[1], reverse=True)
    return predictions

```

Figure 45: The above function forward passes all the candidate items for each user and predicts their rating. Then for each user it sorts the items in descending order based on their predicted rating and saves it in a dictionary.

```

def predict_scores_to_dask_dataframe(model, train_loader, test_loader, classes, user_encoder, item_encoder, save_paquet=False, filename='predict')
'''
Predicts scores using a model and returns a Dask DataFrame with User ID, Article ID, and predicted score using label encoders, with specific
Args:
- model: The trained MF model.
- train_loader: DataLoader containing the train data.
- train_pairs: Set of tuples containing the user-article pairs in the train set.
- test_pairs: Set of tuples containing the user-article pairs in the test set.
- user_encoder: LabelEncoder for user IDs.
- item_encoder: LabelEncoder for article IDs.
- save_paquet: If True, saves the DataFrame to a parquet file.
- filename: The filename for the parquet file if saved.

Returns:
- Dask DataFrame with the columns User ID, Article ID, and predicted score.
'''
model.eval() # Ensure the model is in evaluation mode
delayed_data = []

with torch.no_grad():
    for users, items, _ in tqdm(train_loader, desc="Predicting Train", leave=False):
        users, items = users.to(device), items.to(device)
        outputs = model(users, items)
        # Scale outputs back to the original scale
        outputs = outputs * (classes - 1) + 1
        outputs = outputs.squeeze().cpu().numpy()

        # Map indices back to original IDs using label encoders
        user_ids = user_encoder.inverse_transform(users.cpu().numpy()) # Ensure user IDs are strings
        article_ids = item_encoder.inverse_transform(items.cpu().numpy()) # Ensure article IDs are integers

        # Create a Pandas DataFrame for this batch
        temp_df = pd.DataFrame({
            "customer_id": user_ids,
            "article_id": article_ids,
            "predicted score": outputs,
            "set_type": "train",
        })

        delayed_data.append(delayed(temp_df))

    for users, items, _ in tqdm(test_loader, desc="Predicting Test", leave=False):
        users, items = users.to(device), items.to(device)
        # Scale outputs back to the original scale
        outputs = outputs * (classes - 1) + 1
        outputs = outputs.squeeze().cpu().numpy()

        # Map indices back to original IDs using label encoders
        user_ids = user_encoder.inverse_transform(users.cpu().numpy()) # Ensure user IDs are strings
        article_ids = item_encoder.inverse_transform(items.cpu().numpy()) # Ensure article IDs are integers

        # Create a Pandas DataFrame for this batch
        temp_df = pd.DataFrame({
            "customer_id": user_ids,
            "article_id": article_ids,
            "predicted score": outputs,
            "set_type": "test",
        })

        delayed_data.append(delayed(temp_df))

# Use Dask to concatenate all the delayed objects into a Dask DataFrame
final_ddf = dd.from_delayed(delayed_data)

if save_paquet:
    final_ddf.to_parquet(filename, overwrite=True)
    print(f"Saved predicted data to {filename}")

return final_ddf

```

Figure 46: Python function `predict_scores_to_dask_dataframe` that uses a machine learning model to predict scores for user-article pairs from training and testing datasets, converts the predictions into a Dask DataFrame with user IDs, article IDs, and scores, and optionally saves the results to a Parquet file.

Within this function, the model is set to evaluation mode, and predictions are generated for both the training and testing datasets using the respective loaders. For each batch, the model's output scores are scaled back to their original range and then converted into a human-readable format by mapping user and item indices back to their original IDs using LabelEncoders. These mapped scores are then compiled into Pandas DataFrames, capturing essential information such as user IDs, article IDs, and the corresponding predicted scores.

Each DataFrame generated per batch is temporarily stored as a Dask delayed object, allowing for efficient management of memory and computational overhead. These delayed objects are then aggregated into a single Dask DataFrame. If specified by the user, this DataFrame can be saved to a Parquet file, a columnar storage file format that offers optimized disk storage and fast data retrieval.

```

class MultinomialRBM(BaseNetwork):
    tabnine: test | explain | document | ask
    def __init__(self, n_visible, n_hidden, k):
        super(MultinomialRBM, self).__init__()
        self.n_visible = n_visible # Number of visible units (items * rating classes)
        self.n_hidden = n_hidden # Number of hidden units (features)
        self.k = k # Number of rating classes

        # Initialize weights and biases
        # Use Xavier initialization for weights
        self.weights = nn.Parameter(torch.randn(n_hidden, n_visible, k) * 0.001)
        self.visible_bias = nn.Parameter(torch.randn(n_visible, k) * 0.001)
        self.hidden_bias = nn.Parameter(torch.randn(n_hidden) * 0.001)

    tabnine: test | explain | document | ask
    def to_hidden(self, v):
        # Reshape v to [batch_size, n_visible*k] to multiply with weights reshaped to [n_visible*k, n_hidden]
        v_reshaped = v.view(v.size(0), -1)
        weights_reshaped = self.weights.view(self.n_visible*self.k, self.n_hidden)
        # Matrix multiplication
        activation = torch.matmul(v_reshaped, weights_reshaped) + self.hidden_bias
        return torch.sigmoid(activation)

    tabnine: test | explain | document | ask
    def to_visible(self, h):
        # Reshape weights and perform matrix multiplication
        weights_reshaped = self.weights.view(self.n_hidden, self.n_visible*self.k)
        activation = torch.matmul(h, weights_reshaped) + self.visible_bias.view(-1)
        activation = activation.view(-1, self.n_visible, self.k) # Reshape to [batch_size, n_visible, k]

        # Apply softmax across the rating dimension (k) to get probabilities
        return torch.nn.functional.softmax(activation, dim=2)

    tabnine: test | explain | document | ask
    def free_energy(self, v):
        """
        Compute the free energy of visible vectors v.

        Parameters:
        - v: Visible vectors, reshaped back to [batch_size, n_visible, k]

        Returns:
        - Free energy of the visible vectors.
        """
        # Reshape visible biases to match v for direct multiplication
        vbias_term = torch.sum(v * self.visible_bias, dim=[1, 2])

        # Adjust v and weights for correct matrix multiplication
        # Flatten v for interaction with weights
        v_flat = v.view(-1, self.n_visible * self.k) # Reshape to [batch_size, n_visible*k]
        # Flatten weights for the interaction
        weights_flat = self.weights.view(self.n_hidden, -1).t() # Reshape to [n_visible*k, n_hidden]

        # Calculate the weighted sum of hidden units plus their bias
        wx_b = torch.matmul(v_flat, weights_flat) + self.hidden_bias # Result is [batch_size, n_hidden]
        hidden_term = torch.sum(F.softplus(wx_b), dim=1)

        return (-vbias_term - hidden_term).mean()

```

```

def gibbs_sampling(self, v0, gibbs_steps, return_average=True):
    vk = v0.clone()
    ratings = torch.arange(1, self.k + 1, dtype=torch.float32, device=device).view(1, 1, self.k)
    for step in range(gibbs_steps):
        h_prob = self.to_hidden(vk)
        h_sample = torch.bernoulli(h_prob)
        v_prob = self.to_visible(h_sample)

        if step < gibbs_steps - 1 or not return_average:
            v_avg = torch.sum(v_prob * ratings, dim=2, keepdim=True)
            rounded_avg = torch.round(v_avg).long().squeeze(-1)
            vk = torch.nn.functional.one_hot(rounded_avg - 1, num_classes=self.k).float().to(device)
        else:
            # For the last iteration, calculate the final average scores without rounding
            v_avg = torch.sum(v_prob * ratings, dim=2) # Calculate the weighted sum

    if return_average:
        return v_avg
    return vk

```

Figure 47: MultinomialRBM class implementation in Python

```

class RBMDataset(Dataset):
    tabnine: test | explain | document | ask
    def __init__(self, input_data, dense=True):
        """
        Initializes the dataset.

        Args:
            input_data (str or csr_matrix): Either the file path to load the csr_matrix from, or the csr_matrix itself.
            dense (bool): Whether to convert the csr_matrix to a dense array. Default is True.
        """
        self.dense = dense

        if isinstance(input_data, str):
            # Attempt to load from file path
            try:
                if os.path.exists(input_data):
                    data = load_npz(input_data)
                else:
                    raise FileNotFoundError(f"No file found at {input_data}")
            except Exception as e:
                # Handle exceptions raised by load_npz or if the file does not exist
                raise IOError(f"Failed to load data from {input_data}: {e}")
        elif isinstance(input_data, csr_matrix):
            # Direct csr_matrix
            data = input_data
        else:
            raise ValueError("input_data must be either a file path (str) or a csr_matrix")

        if not self.dense:
            self.data = data
        else:
            # Ensure data is in the correct format
            self.data = data.toarray() if isinstance(data, csr_matrix) else data

    tabnine: test | explain | document | ask
    def __len__(self):
        return self.data.shape[0]

    tabnine: test | explain | document | ask
    def __getitem__(self, idx):
        if not self.dense:
            # If data is sparse, convert the row to dense on the fly to save memory
            return torch.tensor(self.data[idx].toarray().squeeze(), dtype=torch.float32)
        else:
            # If data is already dense, simply return the corresponding row
            return torch.tensor(self.data[idx], dtype=torch.float32)

```

Figure 48: RBMDataset class implementation in Python extending Pytorch's Dataset class

The initialization of the dataset can be configured to either keep the matrix in its original sparse format (`dense=False`), which is more memory efficient, or convert it to a dense format (`dense=True`), which can simplify subsequent operations but at the cost of increased memory usage. This flexibility allows for efficient memory management during training, particularly important when dealing with large datasets typical in collaborative filtering scenarios.

```

def train_rbm(model, train_data_loader, test_data_loader, initial_lr=0.01, lr_decay=0.1, epochs=10, gibbs_steps_multiplier=1, weight_decay=3, optimizer='Adam', eval_k=10, hide_progress_bar=False):
    """
    Trains an RBM model using the specified data loaders and hyperparameters.

    Args:
        model (nn.Module): The RBM model to train.
        train_data_loader (DataLoader): DataLoader for the training set.
        test_data_loader (DataLoader): DataLoader for the test set.
        initial_lr (float, optional): Initial learning rate. Defaults to 0.01.
        lr_decay (float, optional): Learning rate decay factor. Defaults to 0.1.
        epochs (int, optional): Number of training epochs. Defaults to 10.
        gibbs_steps_multiplier (int, optional): Multiplier for the number of Gibbs sampling steps. Defaults to 1.
        weight_decay (float, optional): Weight decay for the optimizer. Defaults to 3e-3.
        optimizer (str, optional): Optimizer to use for training ('Adam' or 'SGD'). Defaults to 'Adam'.
        eval_k (int, optional): Number of recommendations to use for evaluation metrics (RMSE, Precision@k, etc.). Defaults to 10.
        plot (bool, optional): If True, plots the training and test reconstruction errors. Defaults to True.
        hide_progress_bar (bool, optional): If True, hides the tqdm progress bar. Defaults to False.
        save_model (bool, optional): If True, saves the trained model. Defaults to False.
        save_model_name (str, optional): The filename to use for saving the model. Defaults to None (uses the default filename).

    Returns:
        nn.Module: The trained RBM model.
    """
    if optimizer.lower() == 'adam':
        optimizer = optim.Adam(model.parameters(), lr=initial_lr, weight_decay=weight_decay)
    elif optimizer.lower() == 'sgd':
        optimizer = optim.SGD(model.parameters(), lr=initial_lr, weight_decay=weight_decay) # Use Adam optimizer
    else:
        raise ValueError(f"Invalid optimizer: {optimizer}. Use 'Adam' or 'SGD'.")

    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, lr_decay=lr_decay, patience=1) # Change LR on plateau
    gibbs_steps = ceil(gibbs_steps_multiplier * GIBBS_SAMPLING_K) # Initial number of Gibbs sampling steps

    for epoch in tqdm(range(epochs), desc="Epochs", disable=hide_progress_bar):
        epoch_loss = 0.0
        gibbs_steps = ceil(gibbs_steps_multiplier * (GIBBS_SAMPLING_K)) if epoch < epochs // 3 else GIBBS_SAMPLING_K // 3 if epoch < 2 * epochs // 3 else GIBBS_SAMPLING_K // 2 # Increase Gibbs steps over time
        print(f"Epoch {epoch} | Loss: {loss} | Test Error: {test_error}")
        for batch in tqdm(train_data_loader, leave=False, desc=f"Epoch {epoch} | Batch", disable=hide_progress_bar):
            vb = batch.view(batch.size(0), -1, model.k).to(device) # Reshape and send to device
            # Perform Gibbs sampling
            vb = model.gibbs_sampling(vb, gibbs_steps, return_averages=False)

            # Zero gradients, compute the loss, and perform a backward pass
            optimizer.zero_grad()

            loss = model.free_energy(vb) - model.free_energy(vk)

            loss.backward() # Update model parameters
            optimizer.step() # Accumulate the loss

        epoch_loss += loss.item() # Evaluate the reconstruction error before scheduler.step(metrics.values())

        metrics = evaluate_model_metrics(model, train_data_loader, test_data_loader, gibbs_steps, eval_k, hide_progress_bar) # Evaluate the reconstruction error before scheduler.step(metrics.values())

    # Log training progress
    print(f"Final Loss: {epoch_loss} | Final Test Error: {test_error}")
    # Evaluate the reconstruction error
    print(f"Final Test Error: {test_error} | Train Reconstruction Error: {train_reconstruction_error}")
    model.train_errors.append(metrics['train_reconstruction_error'])
    print(f"Model Metrics: {metrics}")
    model.br.append(metrics['RMSE'])
    model.precision.append(metrics['Precision@k'])
    model.recall.append(metrics['Recall@k'])
    model.nDCG.append(metrics['nDCG@k'])
    model.MAP.append(metrics['MAP@k'])
    model.F1.append(metrics['F1@k'])
    model.F2.append(metrics['F2@k'])
    model.F3.append(metrics['F3@k'])

    return model

```

Figure 49: Python code snippet for training the Multinomial RBM