Assignment of master's thesis

Title:	Analysis of Multi-Stage Recommendation Systems
Student:	Bc. Bruno Kraus
Supervisor:	Ing. Petr Kasalický
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Explore the topic of Recommendation Systems with a focus on a multi-stage recommendation. Describe the role of each stage of the multi-stage recommendation system. Examine commonly used methods as well as state-of-the-art methods used for each stage.
Propose an experiment to compare different retrieval and ranking models and study how they influence each other and contribute to the final recommendation. Design a multi-stage recommender system with implemented prototypes of described algorithms.
Select suitable non-trivial input data (e.g., Dressipi Recsys Challenge 2022) and perform the described experiment. Evaluate results and discuss possible future improvements of the suggested approach.

Literature:
Industrial Solution in Fashion-domain Recommendation by an Efficient Pipeline using GNN and Lightgbm: https://dl.acm.org/doi/abs/10.1145/3556702.3556850
Deep Neural Networks for YouTube Recommendations: https://dl.acm.org/doi/10.1145/2959100.2959190
Fashion Recommendation with a real Recommender System Flow: https://dl.acm.org/doi/abs/10.1145/3556702.3556792
Building and Deploying a Multi-Stage Recommender System with Merlin: https://dl.acm.org/doi/pdf/10.1145/3523227.3551468
Related Pins at Pinterest: The Evolution of a Real-World Recommender System: https://arxiv.org/abs/1702.07969

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 15 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Analysis of Multi-Stage Recommendation Systems

Bc. Bruno Kraus

Department of Applied Mathematics
Supervisor: Ing. Petr Kasalický

May 4, 2023

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ing. Petr Kasalický, for his invaluable expertise and guidance throughout the work on this thesis. Additionally, I would like to thank Recombee, s.r.o. for providing me with the hardware, which I utilized while working on this thesis, and for introducing me to the exciting area of real-world recommender systems. Last but not least, I am deeply grateful to my girlfriend and my parents for their unwavering support, love, and encouragement throughout my academic journey.

Declaration

In Prague on May 4, 2023 …………………

Citation of this thesis

Kraus, Bruno. *Analysis of Multi-Stage Recommendation Systems.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato práce provádí studii doporučovacích systémů se speciálním zaměřením na multi-stage doporučovací systémy. Práce zkoumá komplexní problematiku těchto systémů a zkoumá motivace, které stojí za použitím čtyř-fázového přístupu. Práce zkoumá různé retrieval algoritmy a přístupy pro následné řazení. V práci je navržen a implementován multi-stage doporučovací systém pro datovou sadu Dressipi. Práce vyhodnocuje řadu implementovaných retrieval algoritmů s cílem vytvořit robustní retrieval fázi. Pro řazení je pak využita implementace gradient-boosted rozhodovacích stromů, přičemž nejlepších výsledků se dosáhlo ensemblem listwise a pointwise přistupů. Vyhodnocení doporučovacích systémů ukazuje, že využití multi-stage přístupu pro doporučování překonává použití jednotlivých doporučovacích algoritmů nebo jejich ensemblů. Na závěr jsou diskutována možná budoucí vylepšení a směry dalšího výzkumu.

Klíčová slova Doporučovací systémy, multi-stage doporučovací systémy, retrieval algoritmy, řadící algorithmy, LightGBM, Dressipi dataset, Recsys Challenge

Abstract

This thesis provides a comprehensive study of recommender systems, with a specific focus on multi-stage recommender systems. The work investigates the complex nature of such systems and examines the motivations behind the use of a four-stage pipeline. The study covers various retrieval algorithms and approaches employed in the ranking stage. A tailored multi-stage recommender system for the Dressipi dataset is designed and implemented, with a range of retrieval models evaluated to create a robust retrieval stage. The state-of-the-art implementation of gradient-boosted decision trees is employed for ranking, with the best results obtained by ensembling multiple approaches. The importance of utilizing various features is examined, and the pointwise and listwise ranking approaches are compared. The performance evaluation demonstrates that utilizing a multi-stage pipeline for recommendation outperforms using single recommendation algorithms or their ensembles. Finally, future improvements and work are discussed.

Keywords Recommender systems, multi-stage recommender systems, retrieval algorithms, ranking algorithms, LightGBM, Dressipi dataset, Recsys Challenge

Contents

Introduction **1**

1 Analysis of Multi-Stage Recommender Systems **3**
- 1.1 Introduction to Recommender Systems . . . 3
- 1.2 Stages of Multi-Stage Recommenders . . . 6
  - 1.2.1 Retrieval stage (candidate generation) . . . 7
  - 1.2.2 Filtering stage . . . 7
  - 1.2.3 Ranking stage . . . 8
  - 1.2.4 Reordering stage . . . 8
- 1.3 Analysis of Retrieval Models . . . 9
  - 1.3.1 Rating Matrix . . . 9
  - 1.3.2 Measuring user and item similarity . . . 11
  - 1.3.3 User-based k-Nearest Neighbour . . . 12
  - 1.3.4 Item-based k-Nearest Neighbors . . . 13
  - 1.3.5 Items to item recommendation with ItemCF . . . 14
  - 1.3.6 Swing algorithm . . . 14
  - 1.3.7 Two towers . . . 15
- 1.4 Analysis of Ranking Models . . . 16
  - 1.4.1 Introduction to Learning to Rank . . . 17
  - 1.4.2 Learning to Rank methods . . . 18
  - 1.4.3 Introduction to Gradient Boosting . . . 19
    - 1.4.3.1 Boosting . . . 19
    - 1.4.3.2 Gradient Boosting . . . 20

2 Problem Statement **23**
- 2.1 Challenge task . . . 23
- 2.2 Dataset . . . 23
- 2.3 Evaluation methodology . . . 25
- 2.4 Related work . . . 26

3 Proposed Approach **29**
3.1 Design of our solution 29
3.2 Implementation architecture 31
3.3 Training, validation, and testing data split 32
3.4 Retrieval stage 33
3.4.1 Metrics for retrieval models 34
3.4.2 Popularity-based model 35
3.4.3 Content-based model 35
3.4.4 ItemKNN 36
3.4.5 UserKNN 37
3.4.6 Items-to-item models 37
3.4.6.1 Swing 38
3.4.6.2 ItemCF 38
3.4.6.3 Rating similarity 39
3.4.6.4 Word2vec 39
3.4.6.5 Factorization 39
3.4.6.6 Similarity-based ensemble model 40
3.5 Ranking Stage 40
3.5.1 Ranker Training 40
3.5.2 Features for ranking algorithms 41
3.5.3 Hyperparamer tuning 42

4 Experiments **43**
4.1 Retrieval Models 43
4.2 Selecting retrieval models 48
4.3 Ranking algorithms 53
4.4 Evaluation on the testing datasets 54
4.5 Discussion 54

Conclusion **57**

Bibliography **59**

A Acronyms **67**

B Contents of enclosed CD **69**

List of Figures

1.1 An illustration of multi-stage pipeline in different domains [13] . . 9
1.2 Two towers architecture [24] 17

2.1 Modification of sessions with multiple purchased items [45] 24

3.1 High-level inference flow from a session to recommended items . . 30
3.2 Heatmap of Jaccard coefficients between sets of purchased items in different months 33
3.3 Dataset organization for validation and testing 34

4.1 HitRate@N and MRR@N for retrieval models based on similarity to the last viewed item 45
4.2 HitRate@N and MRR@N for different retrieval models on validation data 46
4.3 Bump graph for HitRate@N in % for retrieval models on validation data 47
4.4 Bump graph for MRR@N for retrieval models on validation data . 47
4.5 HitRate@200 in % heatmap when combining 100 candidates from two retrieval models 49
4.6 HitRate@400 in % heatmap when combining 200 candidates from two retrieval models 50
4.7 HitRate@300 in % when ItemKNN, Content-based model, and another retrieval model each retrieve top-100 items 50
4.8 Average size of the candidate set when ItemKNN, Content-based model, and another retrieval model each retrieve top-100 items . . 50
4.9 HitRate@600 in % when ItemKNN, Content-based model, and another retrieval model each retrieve top-200 items 51
4.10 Average size of the candidate set when ItemKNN, Content-based model, and another retrieval model each retrieve top-200 items . . 51
4.11 HitRate@400 in % when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-100 items . . 51

4.12 Average size of the candidate set when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-100 items 51
4.13 HitRate@800 in % when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-200 items . . 52
4.14 Average size of the candidate set when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-200 items 52
4.15 MRR@100 of the ranking models on validation sessions 52
4.16 Gain-based feature importance on a logarithmic scale of a single pointwise LightGBM model 55
4.17 Gain-based feature importance on a logarithmic scale of a single listwise LightGBM model 55
4.18 Final MRR@100 for different recommendation algorithms on testing datasets 56
4.19 Vizualization of MRR@100 for different recommendation algorithms on leaderboard and final sessions 56

Introduction

In today's world, recommender systems play a vital role on online sites by providing users with relevant content tailored to their preferences and taste in real time. These systems can enhance the user experience by providing more personalized and engaging content, help users find searched items by navigating them through large item catalogs, and ultimately increase conversions for online businesses. The objectives of real-world recommender systems go beyond predicting suitable items for users. It is often essential to incorporate various business rules, optimize the entire pipeline for low latency, and maintain the flexibility to adapt the system as needed. Multi-stage recommender systems utilize a paradigm in which the system is divided into subsequent modules, each serving a distinct purpose. These modules work together to fulfill the requirements of a real-world recommendation system, ensuring its effectiveness and adaptability.

The primary objective of the chassis part of this thesis is to present the challenges associated with recommender systems and explore how multi-stage recommenders address these issues. We aim to provide an analysis of each stage within multi-stage recommenders, delving into the motivations and techniques employed in recommendation systems, particularly focusing on item retrieval and ranking stages that determine recommendation quality. In the practical segment, we aim to design and implement a multi-stage recommender for the Dressipi dataset. This will involve discussing the decision-making process during the development of the recommendation system, experimenting with retrieval methods, comparing the two-stage recommendation paradigm against traditional recommendation algorithms, and evaluating the overall performance of the implemented recommendation system.

The structure of this work is organized into four chapters. The first chapter provides an in-depth analysis of the multi-stage recommendation pipeline, while the second chapter introduces the task of the RecSys2022 challenge, presents the dataset, and outlines the multi-stage approaches employed by top-performing solutions. The third chapter proposes the design and implementation of our multi-stage recommender for the introduced task and dataset, detailing the retrieval and ranking algorithms utilized. The fourth chapter showcases the experiments conducted to evaluate the performance of our recommender system.

Chapter **1**

Analysis of Multi-Stage Recommender Systems

1.1 Introduction to Recommender Systems

In today's digital age, the average internet user spends hours on a daily basis consuming online content [1]. Websites that provide us with content range from media sites featuring news articles and e-Commerce stores presenting us their products to apps and platforms serving us media such as videos, movies, or music. The content we encounter online is usually not random, and although sometimes it can be based on its general popularity, most frequently, it is tailored to our specific interests. We refer to the systems responsible for using the collected user data and delivering personalized content as recommender systems.

A recommender system can help us find relevant products more efficiently on e-Commerce websites by guiding us through large catalogs of items, increasing the likelihood that we will discover products that fit our needs and tastes. Without this assistance, we might become discouraged while searching for such a product or get distracted and start doing another activity instead. Furthermore, a recommender system on a media website can also provide us with more interesting and engaging articles, keeping us on the website longer and exposing us to more ads and commercials. Better, more personalized content could lead to higher client satisfaction and to higher website revenues at the same time.

Every content-providing platform has a collection of items that can be shown to its users. These items could be articles on news sites and blogs, products on e-commerce websites, or videos, movies, and songs on streaming platforms. Such a complete set of items that a platform has available is called an item catalog.

A recommender system selects a subset of items from the item catalog that it identifies as the most relevant to a user. Items in this subset are referred to as recommendations and are typically displayed on the user's feed page or some dedicated part of the website or app. the placement of recommendations may vary depending on whether the site or app is entirely driven by the recommendation system or if it has specific scenarios and locations for recommendations to pop up. Often personalized recommendations are presented in a row or a column of items labeled ”Recommended for you” or with other context-based labels such as ”Because you liked X” or ”People also liked the following”.

The exact metrics and KPIs (key performance indicators) used to evaluate the quality of a recommender system and its recommendations often depend on the domain, business objectives, and even website implementation. Common examples of such metrics include the click-through rate (CTR) and the cart conversion rate (CCR). CTR measures the number of clicks on recommendations out of all recommendations displayed, which is essential when we want to optimize the number of page clicks and views for ad revenue. For e-Commerce sites, CCR is an interesting metric since it shows how many of the displayed recommended products are being added to the cart, with higher values of CCR indicating that the recommender system is able to recommend searched products faster or even more relevant products per recommendation in general.

One significant advantage of online stores over physical stores is that they do not have to be limited to a certain number of products that fit into the physical space of the store. For example, the number of books available in an online bookstore is much larger than in a physical bookstore. Physical stores typically deal with their capacity limitations by offering mainly items frequently purchased or popular items. Online stores with large item catalogs and help from recommender systems enable people to find more niche items that better fit their tastes and preferences. Interestingly, items that do not fit in the popular category, given their large amount, can ultimately form a significant portion of sales. This phenomenon is known as the long-tailed economy and was described in [2]. To ensure that the recommender system generally recommends a diverse set of items and does not limit itself to a particular cluster of items, we can measure a metric called catalog coverage. Catalog coverage is the ratio between the number of different items recommended by a recommender system and the size of the item catalog.

The evaluation of the effects of applying a recommender system in a production environment is known as online evaluation. It is the best way to assess the influence of the recommendation system on business performance. To compare the performance differences between the current and testing versions of the recommender system in live traffic, A/B testing is used [3].

However, before we expose live users to recommendations from a new recommender system, estimating or comparing its performance to other versions via offline experiments is often crucial. Off-line evaluation involves evaluating recommendation algorithms used in a recommender system on collected historical interaction data, which simulates the behavior of users that interact with the recommender system [4]. Although offline metrics do not always necessarily correlate with online results [5], they are essential to identify obvious flaws, give a first estimate of performance, and provide a sanity check before a deployment of the new system. Next to evaluating recommendation quality on historical data, it is also important to consider the system's computational efficiency and performance, which can be measured with metrics such as queries per seconder (QPS) or return time (RT). These metrics can also indicate the difference in costs of computational power between different recommendation systems [6].

We mentioned that recommender systems utilize collected user data. In the theory and literature of recommender systems, we distinguish between two types of user data that a recommender system might rely on: explicit and implicit feedback [7, 8]. Explicit feedback, also known as explicit ratings, is a logged action in which a user rates an item with some score. In contrast, implicit feedback is based on logged interactions between users and items, such as item views or item purchases indicating the item's relevance for a given user.

Examples of explicit feedback in recommender systems could include giving an item a score out of 10, a number of stars out of five, or simply liking or disliking an item. However, there are several issues associated with explicit ratings, which limit their usefulness in practical recommender systems. For instance, it can be time-consuming for users to rate each item they encounter, and they may not always remember to do so. In addition, users may have different rating habits. While some users could take their time to leave only negative ratings, other users could be biased towards giving only the highest ratings [9]. These challenges can make it more difficult to find similarities in the preferences of such different users.

Implicit ratings are typically logged user actions such as clicking on an item thumbnail to view its details, adding an item to the cart, and purchasing an item. For certain types of content, such as songs or videos, it might be useful to track how much of the given content the user went through [10]. For example, if a user skips to another song after 15 seconds, it might suggest that they did not enjoy the initial song at that moment. Similarly, if a user only watches a few minutes of a two-hour movie, it could indicate to the recommender system that the movie was not a good fit for their interests. On the contrary, listening to a whole song or seeing a video from start to finish could suggest that the user enjoyed that item.

As already hinted, implicit feedback is the standard in real-world recommender systems since the user does not have to go through the extra effort of creating an explicit rating for an item. Implicit feedback allows the recommender systems to improve their recommendations from a user's first item click. Still, if the site allows users to give explicit ratings to items, this information is likely to be used by the recommender system since a rating matrix, one of the primary data structures used by the recommendation algorithms, can incorporate many different interaction types. Its exact definition, notation, and usage by recommendation algorithms will be explained later in the chapter.

Recommender systems gain insight into users' preferences from their interactions, and generally, the more interactions a user has with different items, the better the recommendations they receive. However, what happens when a new user has not yet interacted with the items? This situation is known as the cold-start problem [7]. A cold-start user might encounter a diverse set of non-personalized recommendations, meaning that the recommended items are not yet based on their tastes and preferences. Often, the recommendation might be based on the popularity of those items in the last few days or hours. Such non-personalized recommendations will serve as a starting point for the recommender system to explore the user's likes and dislikes. On such a fresh feed, users may notice that recommended items change drastically after just a few first interactions or even one click of an item.

The cold-start problem can also apply to new items that have not yet been interacted with by users, making it difficult to recommend such items for the recommender system since it does not yet know which users might like those items. In recommender systems that work with constantly changing item catalogs, such as news sites, it is common to use bandit methods [11] [12], which address the uncertainty around the new items and allow exploration of the cold-start items.

In this thesis, we will focus on multi-stage recommender systems. This design pattern is commonly seen in practical recommender systems [13]. These recommender systems operate in multiple stages, the most notable being candidate generation (also known as the retrieval stage) and ranking stage. This order of stages is useful for recommenders that handle large item catalogs that might contain hundreds of millions of items. The retrieval stage narrows down the items from the catalog to only hundreds of possible candidates based on user interactions. Then, the ranking stage expands the candidate items by various features, based on which the recommender system ranks the generated candidates by their relevance. Using such features on all items in the catalog instead of the retrieved candidates may not be feasible since the time needed for recommendation computation should be in the low hundreds of milliseconds. A longer computation time would result in delay and waiting for the recommendations to load on the user's website, which could negatively impact user experience and ultimately result in lower conversions [14].

1.2 Stages of Multi-Stage Recommenders

At the end of the last section, we introduced the readers to the two-stage recommender paradigm consisting of candidate generation and ranking stages. In real-world recommender systems, it is often necessary to incorporate some business rules and constraints into the recommendations. This can be accomplished through two additional stages of filtering and reordering, resulting in a four-stage pipeline. Thus, a general multi-stage recommender system could, according to [13], have the four following stages:

1. retrieval (candidate generation) stage,
2. filtering stage,
3. ranking stage, and
4. reordering stage.

In the following sections, we will discuss the motivation and purpose behind each stage in more detail. Figure 1.1 illustrates the differences in the usage of the pipeline stages in different domains.

1.2.1 Retrieval stage (candidate generation)

At the retrieval stage, candidates, typically in the order of hundreds, are selected from the item catalog. Commonly used recommender algorithms in this stage take advantage of approaches such as collaborative filtering or even neural networks for candidate retrieval, as described in [15]. Specific retrieval algorithms, with their details, will be covered in the following section. It is worth noting that these algorithms can generate recommendations without the need for additional ranking, as they typically calculate scores for items they deem relevant. Thus, when using a single recommendation algorithm, the recommender system can simply recommend the top-N items with the highest scores. However, this approach would not fully leverage the benefits of the retrieval and ranking pattern. In the ranking stage, the algorithm can use the output scores from the retrieval algorithms and improve the ranking with many more features, as it is more computationally convenient to do so on a limited subset of items rather than on the entire item catalog for every real-time recommendation.

1.2.2 Filtering stage

The filtering stage is responsible for the removal of any unwanted items from the candidate set. E-Commerce or media sites may have blacklists of items they do not wish to present as a recommendation on their front pages. For example, e-commerce would not want to recommend out-of-stock products, a grocery delivery site might not want to recommend alcoholic beverages to users, a media platform might choose not to recommend any borderline content, or a movie streaming platform might choose not to recommend R-rated films to children's accounts. In some domains, it makes sense not to recommend items that the user has already interacted with. For instance, if the user has a product already in their cart or if the user has already read an article, recommending the same item again would likely be pointless and could lead to lower exploration of the item catalog.

1.2.3 Ranking stage

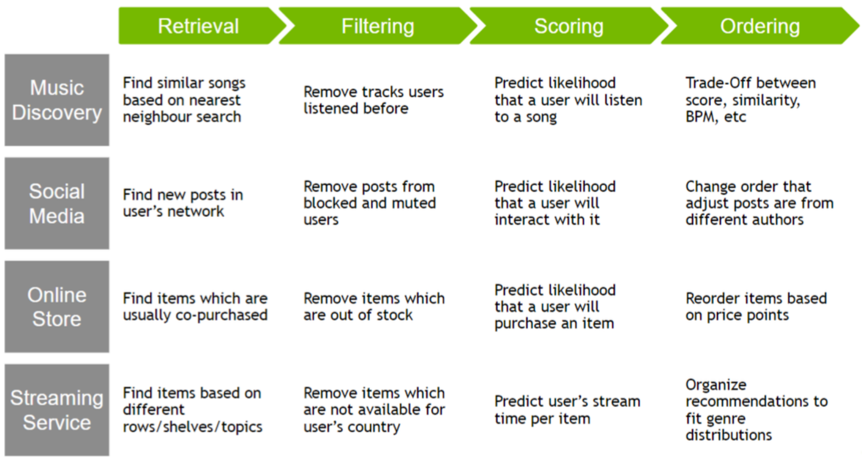
The ranking stage sorts the retrieved item candidates according to their relevance. The process of ranking candidates might be modeled as a classification task, where each item is assigned a score representing the probability that the user will interact with it and ranked accordingly. Other ranking approaches might model the relative ordering of the retrieved candidate items or optimize ranking metrics, such as normalized discounted cumulative gain (NDCG) or expected reciprocal rank (ERR) [16]. The machine learning task, where items are ordered by their relevancy for a given context, is known as Learning to Rank and will be covered further in this chapter.

In this stage, ranking algorithms can utilize a variety of data types to rank items based on their relevance. The features used for ranking candidate items may be based on the items themselves, the target user receiving the recommendations, or the context in which the recommendations are being served. User-based data could include demographic information such as age and region, details about the device and platform used to access the content, or features based on the user's interaction history. Item-based features can include attributes or statistics related to the items. Attributes might involve names, descriptions, categorizations, tags, or features extracted from images, videos, or audio associated with the candidate items. Examples of item statistics include the number of views or purchases within the last few hours or days. Additionally, the context, such as the time of the day, can play a role. Generally, features must be converted from their raw form into numerical values or vector representations that can be used as input for the ranking algorithm.

1.2.4 Reordering stage

The last stage is the reordering stage. Once we know how the candidate items are rated, we are ready to present recommendations to the user, typically as a row, column, or grid of items. The most obvious order of the items would be to sort them by relevance. However, this stage allows for the reordering of items according to specific domains, business, or UI needs. For some e-Commerce sites, it might make sense to first show products with higher margins. Alternatively, a diversity of items can be enforced at this stage so that the recommended items next to each other are not too similar, possibly leading to a greater exploration for the user [17, 18]. For example, fashion e-Commerce might ensure that recommended articles are not all from the same brand. Furthermore, some platforms might need to reorder relevant items and split them into rows of items with the same category or genre.

Figure 1.1: An illustration of multi-stage pipeline in different domains [13]



	Retrieval	Filtering	Scoring	Ordering
Music Discovery	Find similar songs based on nearest neighbour search	Remove tracks users listened before	Predict likelihood that a user will listen to a song	Trade-Off between score, similarity, BPM, etc
Social Media	Find new posts in user's network	Remove posts from blocked and muted users	Predict likelihood that a user will interact with it	Change order that adjust posts are from different authors
Online Store	Find items which are usually co-purchased	Remove items which are out of stock	Predict likelihood that a user will purchase an item	Reorder items based on price points
Streaming Service	Find items based on different rows/shelves/topics	Remove items which are not available for user's country	Predict user's stream time per item	Organize recommendations to fit genre distributions

1.3 Analysis of Retrieval Models

In this section, we will cover recommendation algorithms that can be used in the retrieval stage of the multi-stage recommender system. The algorithms we will use for candidate generation in this thesis are known recommendation algorithms, meaning that they can be used on their own in order to recommend items to users. These algorithms mainly fall into the collaborative filtering (CF) recommendation algorithms category. Collaborative filtering is a method based on evaluating items based on other people's opinions [7]. Collaborative filtering resembles real life, where we might listen to suggestions from people with similar tastes. Next to the cold-start problem, an issue of collaborative filtering methods, the grey sheep problem is a situation where a user has such a unique taste that recommending items using CF methods may be inaccurate.

In contrast to collaborative filtering, content-based filtering evaluates items based on their item attributes rather than interaction data. Collaborative filtering methods face the challenges with the cold start problem, which in contrast content-based filtering effectively handles. On the other hand, content-based filtering does not leverage interaction patterns between users and items. Hybrid collaborative filtering [19] approaches aim to overcome the shortcomings of collaborative and content-based filtering by combining both approaches.

1.3.1 Rating Matrix

Before we get to User-based and Item-based k-Nearest Neighbors, which are examples of collaborative filtering algorithms, we need to define a rating matrix, a data structure used by these algorithms. Having collected implicit interactions between users and items,

- let $U = \{u_1, u_2, \dots u_m\}$ be a set of all users,
- let $I = \{i_1, i_2, \dots i_n\}$ be a set of all items (item catalog),
- let U_i denote a set of all users that interacted with item i,
- let I_u denote a set of all items with which user u interacted, and
- let X be a set of all user-item interactions.

In the experimental part of the thesis, we will be using two types of interactions: item views and item purchases. Therefore,

- let $X^v \subseteq X$ denote a set of all view interactions and
- let $X^p \subseteq X$ denote a set of all purchases.

As we use only these two types of interactions and each interaction can be of only one type, it follows that $X = X^v \cup X^p$ and $X^v \cap X^p = \emptyset$.

Furthermore, since every interaction involves a user and an item with which the user interacted,

- let X_u denote a set of all interactions made by the given user u, and
- let X_i denote a set of all interactions involving the given item i.

In cases where this notation is used, it will always be clear from the context whether the value in the subscript refers to a user or an item. Finally, we need a function that assigns a positive weight to each interaction type. Let $w : X \to \mathbb{R}^+$ be the function that is defined as follows:

$$w(x) = \begin{cases} c_v & x \in X^v \\ c_p & i \in X^p. \end{cases}$$

The exact constants used in our work will be covered in a later chapter. For now, note that we want to give greater weight to the purchase of an item compared to a view since a purchase might provide more certainty that the item is relevant for the user. Therefore, note that $c_p > c_v > 0$.

With these notations in place, we are ready to formally define a rating matrix as a matrix $\mathbf{R} \in \mathbb{R}^{m,n}$ with values defined as:

$$\mathbf{R}_{u_i,i_j} = \min \left\{ c_{max}, \sum_{a \in X_{u_i} \cap X_{i_j}} w(a) \right\},$$

where c_{max} is the higher bound for the values in the rating matrix. The reason for such a bound is to prevent high outlier values that may occur due to repeated views or purchases by some users.

1.3.2 Measuring user and item similarity

When assessing the similarity between users, we can consider using the user attributes, such as demographics or the device used, if they are available. Similarly, when determining the similarity between items, using item attributes as descriptions and categorizations is a valid approach. However, it should be noted that two users of the same demographic group do not necessarily share the same tastes and preferences. Instead of comparing user attributes, we can compare users' tastes and preferences by looking at the items with which they interact. Then, the similarity between two users can be characterized by interacting and engaging with many of the same items. Similarly, items can be described by the groups of people who interact with them. To measure similarity in this manner, we can utilize the rating matrix defined in the previous section.

Given a rating matrix, we can look at individual users or items as their corresponding rows or columns in the rating matrix. Users would be vectors of size n denoted as $R_{u,:}$, and items would be vectors of size m denoted as $R_{:,i}$. Then, to compute the similarity between users or items, we can compute the cosine similarity between their corresponding vectors. The formula for cosine similarity between users u and v follows:

$$\mathrm{sim}_{cos}(u,v) = \frac{R_{u,:} \cdot R_{v,:}^T}{\|R_{u,:}\|\|R_{v,:}\|} = \frac{\sum_{i=1}^{n} R_{u,i} \cdot R_{v,i}}{\sqrt{\sum_{i=1}^{n} R_{u,i}^2} \cdot \sqrt{\sum_{i=1}^{n} R_{v,i}^2}}.$$

The cosine similarity generally gives a result in the $[-1,1]$ range. However, since we might consider only positive values in the rating matrix, given implicit feedback, we can expect a value between zero and one. Zero, in case the users did not interact with the same items, and one when their vectors in the rating matrix are the same or proportional.

Another function that we can use to compute the similarity of items or users is the Pearson correlation coefficient. Let $\overline{R}_{u,:}$ and $\overline{R}_{v,:}$ denote an average rating given by users u and v. The formula for computing Pearson similarity on users follows:

$$\mathrm{sim}_{pearson}(u,v) = \frac{\sum_{i=1}^{n} (R_{u,i} - \overline{R}_{u,:}) \cdot (R_{v,i} - \overline{R}_{v,:})}{\sqrt{\sum_{i=1}^{n} (R_{u,i} - \overline{R}_{u,:})^2} \cdot \sqrt{\sum_{i=1}^{n} (R_{v,i} - \overline{R}_{v,:})^2}}.$$

Similarly to cosine similarity, the Pearson coefficient can be in the $[-1,1]$ range. However, this measure makes sense mainly on explicit ratings with a sufficient variance of the rating values [20]. In the thesis, since we use implicit feedback for rating matrix creation, we will use cosine similarity for the computation of similarity based on interactions between users or items.

The following sections will cover two algorithms that recommend items to users using the rating matrix and similarities between users or items. These algorithms are called in short UserKNN and ItemKNN. In addition to recommending items to users, it is also possible to recommend items relevant to a given item. This kind of recommendation is called an items-to-item recommendation. One scenario where such recommendations might be helpful is when a user is looking at an out-of-stock product, and we might want to recommend similar items to the user. How items-to-item recommendations are accomplished by algorithms like ItemCF and Swing will be covered further in this chapter.

1.3.3 User-based k-Nearest Neighbour

User-based k-Nearest Neighbours, or UserKNN shortly, is a straightforward algorithm with great interpretability that utilizes ideas of collaborative filtering. Applying a user similarity function, we can identify the k most similar users to a target user. Since users with similar interactions are likely to share similar tastes, we can recommend items for a target user that similar users enjoyed and that the target user has not yet encountered. Given a similarity function $sim : U \times U \to \mathbb{R}$, let $NN_k(u)$ denote a set of the k most similar users to the user u. We will refer to the users in $NN_k(u)$ as the nearest neighbors of a user u. A set of recommendable items $RI(u) \subset I$ produced by the UserKNN algorithm for a user $u \in U$ can be expressed using the following formula:

$$RI_{\text{UserKNN}}(u) = \left(\bigcup_{v \in NN_k(u)} I_v \right) \setminus I_u.$$

Note that in this thesis, we are dealing with implicit interactions, and higher values in the rating matrix correspond to higher levels of positive engagement with the corresponding items. To rank the recommendable items according to their relevancy to a user u, each of the items is assigned a score the following way:

$$\text{score}_{\text{UserKNN}}(u, i) = \sum_{v \in NN_k(u) \cap U_i} R_{v,i}.$$

A high score by UserKNN of an item is achieved when the item has been interacted with by many users similar to the target user u or when some of the similar users have shown a great deal of engagement with the item, resulting in high values in the rating matrix $\mathbf{R}$ for that item.

Some users in the nearest neighbors are more similar to the target user than others. Those users' preferences might better resemble the tastes of the target user and thus should influence the final score more than the preferences of less similar users. To take this into account, we can weigh contributions to the score from individual users by their similarity to the target user. The following is the scoring formula of a weighted version of UserKNN:

$$\text{score}_{\text{UserKNN}}(u,i) = \sum_{v \in NN_k(u) \cap U_i} \text{sim}(u,v) \cdot R_{v,i}.$$

Finally, for completeness, let us point out that when dealing with explicit ratings in the rating matrix, as opposed to implicit ones, normalization of the score needs to be added. Without normalization, numerous low ratings could result in a high UserKNN score. For explicit ratings, the formula should be adjusted in the following way:

$$\text{score}_{\text{UserKNN}}(u,i) = \begin{cases} \dfrac{\sum_{v \in NN_k(u) \cap U_i} \text{sim}(u,v) \cdot R_{v,i}}{\sum_{v \in NN_k(u) \cap U_i} \text{sim}(u,v)} & NN_k(u) \cap U_i \neq \emptyset \\ ? & NN_k(u) \cap U_i = \emptyset. \end{cases}$$

1.3.4 Item-based k-Nearest Neighbors

The item-based k-nearest neighbors, known as ItemKNN, is another items-to-user recommendation algorithm. The ItemKNN assumes that users might be interested in items similar to those with which they have already interacted with. Instead of the user similarity function used in UserKNN, we will use the item similarity function $\text{sim} : I \times I \to \mathbb{R}$. Let $NN_k(i)$ denote the k most similar items to the item i, and we will refer to these items as the nearest neighbors of item i. Similarly to Section 1.3.3, let us first describe the set of items that can be recommended by ItemKNN to a user u in a more formal way:

$$\text{RI}(u) = \left(\bigcup_{i \in I_u} NN_k(i) \right) \setminus I_u.$$

The items that ItemKNN recommends the most are often very similar to the items with which the user was highly engaged. From the formula mentioned above, we can see that each recommendable item belongs to the nearest neighborhood of at least one item that is interacted with by the target user. Each such occurrence contributes to the score of the recommendable item. Specifically, let $j \in I_u$ be an item, whose nearest neighborhood contains item i, or more formally $i \in NN_k(j)$. Then, a contribution to the score of the item i for the target user u occurs, and it holds that the size of this contribution is significant when the rating of the item j by the user u is high and when the similarity between the items i and j is high. The formula for calculating the score is as follows:

$$\text{score}_{\text{ItemKNN}}(u,i) = \sum_{j \in \{a\ \in I_u | i \in NN_k(a)\}} R_{u,j} \cdot \text{sim}(i,j).$$

1.3.5 Items to item recommendation with ItemCF

ItemCF [21], a slightly modified version of which can also be found in the article [22], is a simple example of an items-to-item collaborative filtering recommendation algorithm. Unlike UserKNN and ItemKnn, which are user-to-item collaborative filtering algorithms that recommend relevant items to a given user, ItemCF recommends items related to a specific item. Some of the key ideas behind this algorithm include the idea that the co-occurrence of items in user interactions indicates their similarity. Moreover, when a user has many interactions, the indication of similarity between the interacted items should be smaller since that user might be more open to many different kinds of items. Furthermore, many users interact with popular items, and thus, they have to be penalized since they could end up being very similar to many niche items. The relevance score for an item j given an item i is computed using the following formulas:

$$w_i = \sum_{u \in U_i} \frac{1}{\mid I_u \mid},$$

$$\text{score}_{\text{ItemCF}}(i,j) = \frac{\sum_{u \in U_i \cap U_j} \frac{1}{\mid I_u \mid}}{\sqrt{w_i \cdot w_j}}.$$

In this case, the scoring function for item-to-item relevance is symmetrical, but note that, generally, this is not the rule. In the practical part of this thesis, we will use a modified version of ItemCF, which further adds time and some positional information to the equation.

1.3.6 Swing algorithm

Swing is an items-to-item recommendation algorithm introduced in [21], created with the purpose of capturing substitution relationships between items. The equation used to calculate the Swing score for an item j given an item i is

$$\text{score}_{\text{Swing}}(i,j) = \sum_{a \in U_i \cap U_j} \sum_{b \in U_i \cap U_j} \frac{1}{\alpha + \mid I_a \cap I_b \mid},$$

where α is a smoothing coefficient. The idea behind it is that when two different users interact with the same pair of items, it might indicate that those two items are somehow relevant to each other. Furthermore, the less similar the preferences of two users who interacted with the same pair of items are, the stronger the indication of actual relevance between the two items.

1.3.7 Two towers

With advancements in deep learning, neural approaches are now being applied in item retrieval for recommendation [15, 23] and retrieval in online advertising as the ad click prediction also leverages a similar cascading paradigm, which includes retrieval and ranking stages [6, 24].

A two-towers neural network is an approach for information retrieval originating in the area of web search [25]. In the context of recommender systems, this retrieval approach was applied for video recommendation at Youtube, where it showed improvement in user engagement during A/B testing [23]. Moreover, many new neural retrieval approaches improve on the two-tower architecture to achieve state-of-the-art (SOTA) performance in the recommendation and ad prediction tasks.[6, 24]

Let $\{u_i\}_{i=1}^{M}$ be a set of users and $\{i_j\}_{j=1}^{N}$ be a set of items defined by their corresponding feature vectors. User and item feature vectors are composed of various encoded user and item features, respectively. Note that u_i and i_j can both have very high dimensions.

The neural model consists of two parallel DNNs (deep neural networks) called towers. The input for the neural network is a pair of user and item feature vectors. Each of the towers takes an assigned type of the feature vector (item or user) as input and maps it through L fully connected layers into an embedding space of lower dimension $\mathbb{R}^d$. Both embeddings then go through the L2 normalization layer, and the network's final output is their inner product. In other words, the model's output is the cosine similarity between the final user and item embeddings. Figure1.2 depicts the two towers' architecture.

More formally, by choosing, e.g., the user part of the network, we can define the fully connected hidden layers of the user tower and the final user embedding using the following formulas:

$$\begin{aligned}\mathbf{h}_1 &= \text{ReLU}(\mathbf{W}_1 \cdot u + \mathbf{b}_1)\\ \mathbf{h}_i &= \text{ReLU}(\mathbf{W}_i \cdot \mathbf{h}_{i-1} + \mathbf{b}_i), \quad \forall i \in \{2, \dots, L\}\\ \mathbf{e}_u &= \frac{\mathbf{h}_L}{\|\mathbf{h}_L\|_2}.\end{aligned}$$

Let ℓ_i denote the number of neurons in the user tower's i^{th} hidden layer, the $\mathbf{W}_i \in \mathbb{R}^{\ell_i, \ell_{i-1}}$ from the above formulas is a weight matrix of i^{th} hidden layer and $\mathbf{b}_i \in \mathbb{R}^{\ell_i}$ is the bias vector of i^{th} hidden layer in the user tower. The item tower has the same structure and can be defined similarly, with the difference being the item feature vector as input into the first hidden layer. Further, let $\mathbf{e_i}$ denote the output item embedding from the item tower. Finally, the output of the model is $\hat{y} = \mathbf{e}_u^T \cdot \mathbf{e}_i$.

Let $(u_j, i_j, y_j)_{j=1}^{n}$ be the training data based on implicit feedback, where y_j is a rating or a positive value indicating a level of interaction of the user u_j with the item i_j. Furthermore, for each user-item pair u_j and i_j from the training data, multiple negative samples are retrieved, e.g., items with which the user u_j chose not to interact. Let D^- denote a set of feature vectors of the sampled non-interacted items and $D = D^- \cup \{i_j\}$. The probability that the item i_j is the one interacted item from items in D by the user u_j can be based on the following softmax function:

$$P(i_j \mid u_j; \theta) = \sum_{i_k \in D} \frac{\mathbf{e}_{u_j}^T \cdot \mathbf{e}_{i_j}}{\mathbf{e}_{u_j}^T \cdot \mathbf{e}_{i_k}}.$$

Then the model parameters can be optimized by using gradient descent on negative log-likelihood weighted by the ratings:

$$L(\theta) = -\sum_{j=0}^{n} y_j \cdot \log(P(i_j \mid u_j; \theta))$$

Once the model is trained, the item embeddings can be precomputed offline and therefore do not need to be computed for each recommendation query. During inference, only the user embeddings are computed and then compared to the precomputed item embeddings to find the most similar items. To find the most similar item embeddings given a user embedding, approximate nearest neighbors (ANN) algorithms are employed [26].

1.4 Analysis of Ranking Models

Once candidate items are generated by retrieval models and unwanted items are removed from the candidates in the filtering stage, what follows in multi-stage recommenders is to employ a model that orders the candidate items by their relevance. The task of training machine learning models using supervised learning to sort candidate objects according to their degrees of relevance, preference, or importance is known as Learning to Rank [27].

Ranking algorithms often found in real-world multi-stage recommender systems can be implemented as gradient-boosted decision trees (GBDT) as in Related Pins from Pinterest [28], or even as deep neural networks, which were used for app recommendation in Google [29] or are used for video recommendation in YouTube [15].

Even though DNNs are capable of learning complex interactions between features and reducing the need for task-specific feature engineering, whether they are the most efficient approach for practical tasks remains a question [30]. In recent recommendation challenges, approaches that used the multi-stage paradigm with ranking via GBDT applied to manually crafted, practical, and

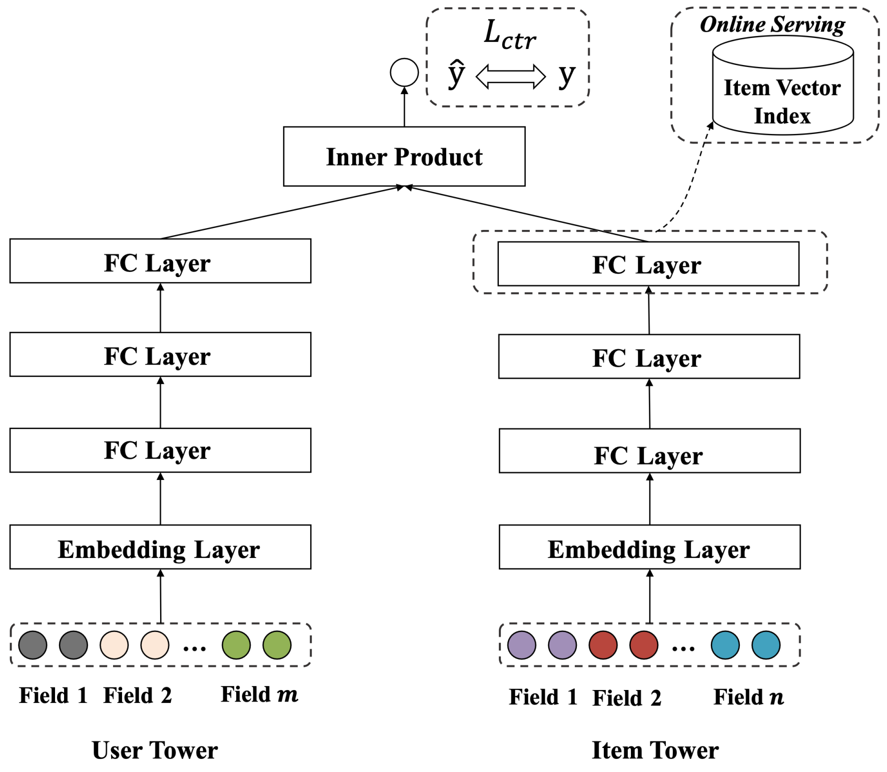


Figure 1.2: Two towers architecture [24]

low-level features, achieved first place in Recsys Challenge 2022 [31], the H&M Kaggle Competition[1] [32], and the WSDM Cup 2022 [33].

The following sections will provide a more detailed theoretical background for Learning to Rank and Gradient Boosting, which will be utilized for ranking in the practical part of this thesis.

1.4.1 Introduction to Learning to Rank

Learning to Rank is a widely studied area of machine learning within the field of information retrieval, with important applications such as search engines [34]. The ranking problem, in its original context, involves a query and a set of documents, with the goal of ranking the documents based on their relevance to the given query. Similarly, in recommendation systems, we can approach the ranking problem as the task of ranking items according to their relevance for a given user. However, while covering this subsection will adhere to

[1]An overview of the first place solution can be found here: https://www.kaggle.com/competitions/h-and-m-personalized-fashion-recommendations/discussion/324070

the traditional document-query context. More formally, the Learning to Rank problem consists of the following:

- a set of **queries** $Q = \{q_1, q_2, \ldots, q_n\}$ of size n,
- a set of **documents** $D_i = \{d_1^i, d_2^i, \ldots, d_{m_{q_i}}^i\}$ for each query $q_i \in Q$, and
- a set of **labels** $L_i = \{l_1^i, l_2^i, \ldots, l_{m_{q_i}}^i\}$ for each set of documents D_i, where a label $l_i^k \in L_i$ corresponds to a document $d_i^k \in D_i$.

The labels represent the level of relevance of documents for a given query. The values assigned to labels could capture more degrees of relevancy by being elements of an ordinal set such as {highly relevant, relevant, marginally relevant, irrelevant}, or they can even be simply binary, stating whether a document is relevant or irrelevant.

Furthermore, let us note that Learning to Rank models work with document-query pairs, which are represented by their feature vectors. Let Φ be a feature extractor function. Given a query $q_i \in Q$ and a document $d_k^i \in D_i$, their corresponding feature vector can be defined as $\mathbf{x}_i^k = \Phi(q_i, d_k^i)$. Such a feature vector can usually be divided into the following three parts:

- a part that depends only on the given query,
- a part dependent only on the document, and
- a part dependent on a combination of the query and the document [34].

With notation already defined, a training set for Learning to Rank tasks can be represented as $\{(\mathbf{x}_i^k, l_i^k) : 1 \leq i \leq n, 1 \leq k \leq m_{q_i}\}$, a set of feature vectors and their corresponding labels.

The following section will introduce the categorization of Learning to rank methods.

1.4.2 Learning to Rank methods

According to [27], Learning to Rank methods can be categorized into the following three approaches: point-wise approach, pairwise approach, and listwise approach.

Pointwise methods work with one document-query feature vector at a time and aim to predict the degree of relevance of the document for the given query. Unsurprisingly, pointwise methods can be modeled as regression or multi-class or binary classification depending on the form of given relevance judgment. For a trained pointwise model, a computation of a relevance score for a document is independent of other documents in the list. It is also good to note that computing relevance scores as pointwise methods do might not be necessary to end up with a ranked list.

Pairwise methods solve ranking by evaluating pairs of documents and predicting the relative order of the documents according to relevancy for a given query. The goal is to do binary classification for document pairs and minimize the number of classifications that do not match the ground truth. This approach models ranking in a slightly more natural way than pointwise methods due to the focus on the relative order in pairwise methods. However, as [16] points out, minimizing the number of pairwise errors does not always correlate with Learning to Rank metrics such as NDCG or ERR. An example algorithm of this approach is RankNet [35].

Listwise methods in comparison with the methods mentioned above, do not train only on documents one at a time or by pair, but add consideration for the list of documents as a whole. This approach permits the final order to result from more complex interactions between documents in the list. An algorithm that introduced this approach is Listnet [36]. Other examples of this approach are LambdaRank and LambdaMart, which improve the pairwise method RankNet by considering listwise metrics [16]. An ensemble of LambdaMart algorithms won the Yahoo! Learning To Rank Challenge [34].

Such categorization might be interesting further in the thesis when we try out and evaluate ranking algorithms utilizing different Learning to Rank approaches.

1.4.3 Introduction to Gradient Boosting

Gradient Boosting, especially Gradient Tree Boosting, also known as Gradient Boosting Decision Tree (GBDT), is a powerful machine learning method used for various machine learning tasks, including Learning to Rank [16]. State-of-the-art implementations of GBDT include XGBoost [37], LighGBM [38], and CatBoost [39]. These GBDT implementations have achieved success in many recommender systems competitions, where they typically outperform other deep learning methods [40]. To explain Gradient Tree Boosting, we will start with a short introduction to boosting ensemble technique and continue with a review of Gradient Boosting and Gradient Tree Boosting.

1.4.3.1 Boosting

Instead of focusing on training a single strong predictive model, ensemble methods employ a combination of relatively weaker models, also known as weak learners, to achieve better and more robust predictions.

Well-known ensemble algorithms include Random Forests, an example of the bagging ensemble approach. In the bagging approach, weak learners are trained independently on their respective datasets, which consist of bootstrapped samples from the original training dataset. The final result is then obtained by averaging (for regression tasks) or using majority voting (for classification tasks).

Boosting is another category of ensemble algorithms that uses a slightly different approach to bagging. Unlike bagging, boosting methods rely on a more dependent relationship between weak learners. In boosting, new weak learners are trained sequentially based on the errors of the collective prediction of previously trained weak learners. This approach aims to iteratively improve its predictions on the training dataset.

1.4.3.2 Gradient Boosting

Gradient Boosting, introduced in [41], given a training dataset $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$, where $\mathbf{x}_i \in \mathbb{R}^m$ stands for an input variable and $y_i \in \mathbb{R}$ is its corresponding output variable or label, tries to estimate the dependencies between the input variables and the output variable with an estimated function $\widehat{f}(\mathbf{x}) = \hat{y}$ in a way that minimizes arbitrary differentiable loss function $\Psi(y, \mathbf{f})$. To address the optimization in the function space, Gradient Boosting constrains $\hat{f}$ to an additive form:

$$\hat{f}(x) = \sum_{m=0}^{M} \hat{f}_m(x).$$

The functions f_m are defined in an iterative manner using basis functions $h(x;\theta)$ from some family of parameterized functions and a step size parameter ρ as in the following equation:

$$\hat{f}_m = \begin{cases} c & i = 1, \\ \hat{f}_{m-1} + \rho_m \cdot h(x, \theta_m) & \text{otherwise.} \end{cases}$$

The minimization of the cost function Ψ is approached by taking a negative gradient step approximated by the $h(x, \theta)$, parameterized in a way that for the given training samples, it is the most correlated to $-g_m(x)$, where $g_m(x)$ is defined as

$$g_m(x_i) = \frac{\partial \Psi(y, \hat{f}_{m-1}(x_i))}{\partial \hat{f}_{m-1}(x_i)}.$$

Therefore, each base learner $h(x, \theta_m)$ is fitted to the $g_m(x)$ values referred to as "pseudo"-responses in [41] or "pseudo"-residuals as in [42]. The parameterization of the base learner function can be attained from the following equation:

$$\theta_m = \arg\min_{\theta,\rho} \sum_{i=1}^{n} (-g_m(x_i) - \rho \cdot h_m(x_i, \theta))^2$$

And finally, having the parameterization the ρ_m parameter is chosen to optimize the loss function over the training data:

$$\rho_m = \arg\min_{\rho} \sum_{i=1}^{n} \Psi(y_i, \hat{f}_{m-1} + \rho \cdot h_m(x_i, \theta_m)). \tag{1.1}$$

Gradient Tree Boosting is a specification of Gradient Boosting where the base learner functions are regression trees. In m^{th} boosting iteration, a regression tree with J terminal nodes (also known as leaves) splits the input space into J disjoint regions $\{R_{m,i}\}_{i=1}^{J}$. The regression tree predicts the same constant value for all the region's input data points. The predicted value can be set to the average of "pseudo"-residuals $\overline{g}_{m,j}$ of all training data points in the given region

$$\overline{g}_{m,j} = \sum_{i=1}^{N} \frac{\mathbb{1}_{x \in R_{m,j}} \cdot g_m(x_i)}{\mathbb{1}_{x \in R_{m,j}}},$$

and the formula for such a base learner can be expressed as

$$h(x, \{R_{m,j}\}_1^J) = \sum_{j=1}^{J} \mathbb{1}_{x \in R_{m,j}} \cdot \overline{g}_{m,j},$$

where $\mathbb{1}_{x \in R_{m_j}}$ is an indicator function equal to 1, when x belongs to R_{m_j} and 0 otherwise. The parameters of the regression tree are the splitting rules, defined by a splitting variable and the corresponding splitting value, that are evaluated in the non-terminal nodes in the tree. The structure of the tree and its rules in the non-terminal nodes ultimately determines to which region an input data point belongs and is built in a greedy, best-first manner [43]. For each currently non-terminal node, which has not reached the specified maximal depth, an optimal splitting rule is computed. And from those nodes and rules combinations, one with the most significant value of a tree build criterion is then chosen for splitting. This is repeated until the number of terminal nodes is reached.

For better experimental results, additional regularization is usually applied to help to prevent the model from overfitting on the training data. A well-established method is to use a shrinkage parameter in the range $(0, 1]$, also known as learning rate, which limits the effects of new base learners by multiplying the multiplication of output from the base learner and step size [41]. Another method of regularization, which also leads to faster computation, is using a random subsample of the training data at each iteration for training for fitting a base learner [42] or using a column subsample, where for each tree consider only a random subsample of features for splitting [44].

Chapter 2

Problem Statement

This chapter provides an overview of the Recsys 2022 Challenge[2] task and its associated dataset. Moreover, we will discuss the evaluation methodology used in the challenge and examine the strategies employed by the top-performing teams. In the following chapters, we will develop and evaluate a solution using a multi-stage recommender system specifically tailored to the presented task and dataset.

2.1 Challenge task

The Recsys 2022 Challenge was organized in partnership with Dressipi, a platform specializing in recommendations for the fashion domain. The challenge focuses on the recommendations based on session-based interactions, and its task is to predict ultimately purchased items. The decision behind using solely implicit feedback from anonymous sessions for the recommendations is rooted in the fact that, on average, 51% of the Dressipi visitors are new with no available historical data. In addition, the fashion domain is known for fast shifts in trends, causing historical interactions to quickly lose relevance in representing current user preferences, further highlighting the importance of session-based recommendations.

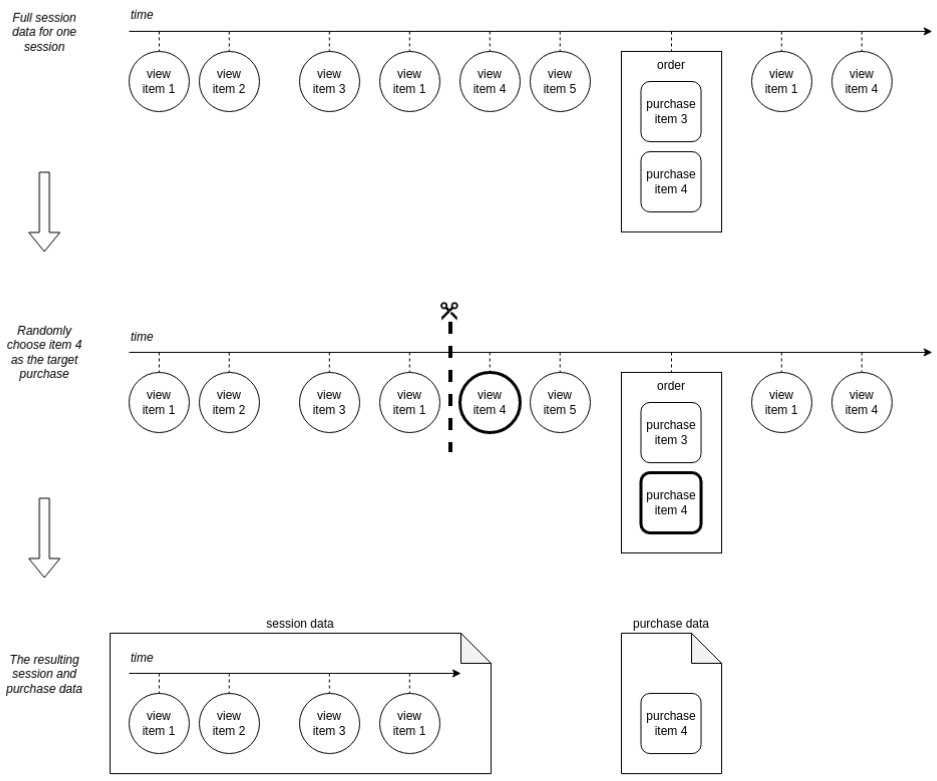
2.2 Dataset

The challenge dataset consists of over a million shopping sessions sampled over an 18-month period. Each session is labeled with a unique ID, and although multiple sessions could be made by the same user, this information is not present in the data. Most importantly, each session consists of a sequence of viewed items and is assigned exactly one purchased item. The sequence of item views in a session ends just before the first view of the purchased item.

[2]http://www.recsyschallenge.com/2022/

If multiple purchases were made in a session, only one purchase was chosen randomly, and the sequence of item views was truncated accordingly. This method, which changes a session of multiple purchases to a session with one purchase is illustrated in Figure 2.1

Figure 2.1: Modification of sessions with multiple purchased items [45]



Furthermore, the dataset contains 23,000 items and their corresponding item attributes. The item attributes are provided as pairs of values, where the first value encodes a feature category, and the second value encodes a feature value. The feature categories are not provided with descriptions or descriptive names and are labeled as numbers. A feature category can take a value from some corresponding range of feature values, which are also labeled with numbers and do not include further descriptions. To illustrate with an example: an item might be assigned item properties (5,1) and (7,1), where the first attribute pair might correspond, e.g., to the color category and value blue, and the second attribute pair might correspond to the size category and value medium.

The sessions from the dataset were divided by the authors of the challenge into training and testing sessions. The training data contains 1 million sessions from the first 17 months of the dataset. The testing data consist of

100,000 sessions from the last month of the dataset, with the finally purchased items excluded. Moreover, the sessions from the testing data were randomly divided into two sets of equal size: leaderboard sessions and final sessions. The leaderboard data was used during the challenge to evaluate solutions and rank them on the public leaderboard, while the final sessions were used to evaluate the final solutions and determine the final standings. Additionally, the sequences of item views in the testing sessions were randomly truncated to include only 50-100% of the first views. The purpose of the random truncation of the session was to motivate the recommender systems to predict the final purchase as soon as possible. Finally, a subset of 5,000 items eligible for the purchase prediction in the leaderboard and final sessions was provided.

2.3 Evaluation methodology

For every session in the leaderboard and the final session sets, the task was to submit an ordered list of 100 items most likely to be purchased. The evaluation metric chosen to rate the quality of the recommendations was MRR (mean reciprocal rank). The value of the reciprocal rank of an item recommended at position p is $\frac{1}{p}$. Therefore, if a purchased item is recommended at the first, second, or third place, the reciprocal rank values would be 1, 1/2, and 1/3.

Let us define the MRR score for our use case formally. Let $S = \{s_1, \ldots, s_N\}$ be a set of testing sessions and let I be a set of candidate items. For each $i \in \{1, 2, \ldots, N\}$,

- let C_i be a set of recommended items for a session s_i such that $\mid C_i \mid \leq 100$, and
- let $\text{pos}_{C_i} : C_i \to \{1, 2, \ldots |C_i|\}$ be a bijection assigning items from C_i their positions.

Let $\text{reciprocal\_rank}(x) : I \to \mathbb{R}$ be a function defined as:

$$\text{reciprocal\_rank}_i(x) = \begin{cases} \dfrac{1}{\mathrm{pos_{C_i}(x)}} & x \in C_i \\ 0 & x \notin C_i. \end{cases}$$

We can now define the MRR as:

$$\text{MRR} = \frac{1}{N} \cdot \sum_{i=1}^{N} \text{reciprocal\_rank}_i.$$

To provide some intuition behind MRR values, if the purchased item is recommended at the first position 50%, 20%, or 10% of the time, we will receive MRR greater than or equal to 0.5, 0.2, or 0.1. In a scenario where the recommender system would recommend the purchased item uniformly at the first 10, 15, or 20 positions, the MRR score would equal approximately 0.29, 0.22, and 0.18.

Considering situations where a multi-stage recommender fails to generate the purchased candidate in the retrieval stage. To obtain better approximations of the final MRR score, we could multiply the example scores above with the ratio α with which the recommender is able to retrieve the purchased items, for example, $\alpha = 0.8$.

2.4 Related work

Many of the best solutions [31, 46–49], including the winning solution, in the RecSys2022 challenge used the multi-stage recommender approach consisting of candidate retrieval and ranking paradigm, supported by feature extraction and feature engineering. Other teams that did not use the multi-stage recommender employed a variety of neural approaches. Namely, the second-place solution [50] was based on the transformer architecture, the fourth-place team [51] utilized heterogeneous graph neural networks (GNN) with side information, or another successful solution [52] employed an ensemble of LSTM neural networks with other probabilistic methods. Since the topic of interest in this thesis are multi-stage recommender systems, we will mainly focus on reviewing techniques and observations from the solutions that employ this approach.

The strategy behind the first-place solution is discussed in the paper called Industrial Solution in Fashion-domain Recommendation by an Efficient Pipeline using GNN and LightGBM [31]. The team behind this solution created a fast multi-stage recommender pipeline, trained on the last three months of the dataset. The retrieval stage used a combination of collaborative filtering and popularity-based models, and the ranking stage utilized LightGBM [38] trained with lambda rank objective. The best-performing features were based on collaborative filtering similarities such as ItemCF enriched with time and position information, Swing, and similarity between GNN (graph neural network) based embeddings, which the team designed specifically for the challenge dataset.

The third-place solution [46] chose an approach with a three-stage recommender system, where the first stage is the retrieval stage, and the second and third stages both deal with ranking. In contrast to the winning solution, which applied data clipping to use only recent training data, the team behind this solution used augmentation of the training dataset to increase the number of training sessions by 5-10 times. The candidate generation stage employed various diverse algorithms based on conditional popularity, Session k-Nearest Neighbors, LightGBM, RNNs, MLP, and transformer networks. The second stage of the pipeline employed ranking using an optimized weighted average of item retrieval scores, LightGBM trained with lambda rank objective, and XGBoost trained with binary classification logistic regression objective.

The ranking algorithms were then ensembled together in the last stage of the pipeline using a weighted average of item rankings from the second stage. Similarly, the tenth-place solution [48] leveraged an ensemble of ranking algorithms, including LightGBM, CatBoost, and XGBoost pointwise rankers, LambdaMart, and deep interest network (DIN) [53].

The fifth-place solution [47] used a two-stage pipeline quite similar to the first-place solution. The training data for the final prediction contained sessions from two months before the final testing month. Similarly to the first-place solution, LightGBM with the lambda rank objective trained with 500 features for each item was used for ranking. Candidates were retrieved based on ItemCF, the similarity between item embeddings, and item popularity.

Interestingly, variants of ItemCF, which utilized positional and time differences between co-occurring items, were employed in [31, 46, 47]. A strategy re-occurring in the retrieval stage was to develop a diverse set of algorithms to generate a diverse set of candidate items for the ranking stage. One of the main motivations behind such a goal was to be able to cover purchased items. Since missing a purchased item in the list of candidates for a test sample would negatively impact the final MRR score. Furthermore, since, in some cases, collaborative filtering algorithms might not even be able to come up with personalized candidates (sessions containing cold items), popularity-based algorithms often complemented collaborative filtering as in [31, 47, 48] or also content-based algorithms were employed as in [47–49].

Chapter 3

Proposed Approach

In this chapter, we will cover our approach regarding the design and implementation of a multi-stage recommender system specifically developed for the Dressipi dataset described in the last chapter. Sections in this chapter will cover the systems architecture and a discussion of implementation decisions. Further, we will propose the data splitting methodology to perform validation of the used retrieval and ranking algorithms. Finally, we will describe the implementation of the individual retrieval and ranking algorithms.

3.1 Design of our solution

Our approach will use a multi-stage recommender flow as described in Section 1.2, similar to the top solutions of the Recsys Challenge 2022, in order to achieve better recommendations compared to using individual recommendation algorithms or their ensembles. The recommender system inference flow, the process of generating predictions for given sessions, is depicted in Figure 3.1 and consists of the following phases:

1. retrieval stage for generating item candidates for a user (defined by its session),
2. filtering stage for removing items already viewed by the user and items that are not in the candidate whitelist,
3. ranking stage that ranks the candidates based on constructed features, and
4. reordering stage, which in our case, keeps the order from the previous stage and only crops the top-100 candidates.

The retrieval stage will be realized by employing various retrieval algorithms introduced in Section 1.3 to generate the item candidates set. Candidate generation should be sufficiently robust since the testing interval is one

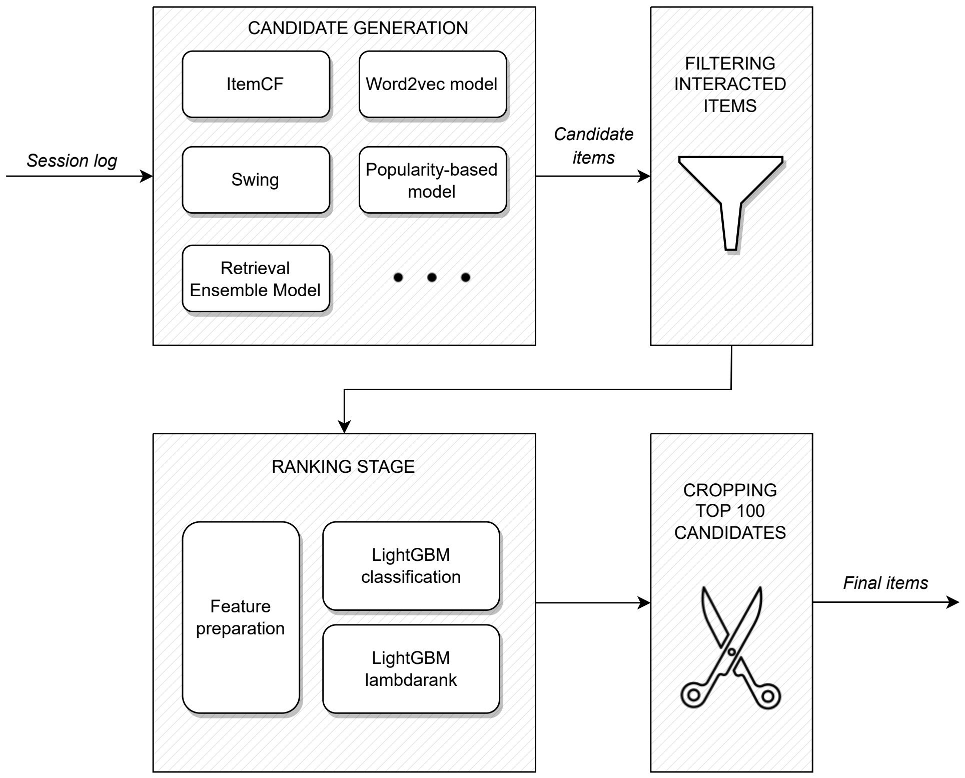


Figure 3.1: High-level inference flow from a session to recommended items

month long and trends in the fashion domain change quickly, as illustrated in Figure 3.2, and therefore relying only on collaborative filtering methods might not be effective as new items do not necessarily need to have interactions in the training data.

The filtering will be enforced by passing a whitelist containing possible candidates described in Section 2.2 and a blacklist for already-seen items in the session to the retrieval models.

In the ranking stage, we will employ pointwise and listwise ranking approaches implemented in the gradient-boosting library LightGBM. LightGBM was the library of choice in multiple top solutions for the Recsys Challenge 2022. Based on the results of the two ranking approaches, we may further consider creating an ensemble ranking model that combines their outputs for the final ranking.

An explanation of how the training dataset will be used for training and validation of the retrieval and ranking algorithms will be covered in dedicated sections. The following section will describe our realization of the recommender system.

3.2 Implementation architecture

Our recommender system implementation comprises the following classes:

- `RetrievalModel`,
- `RetrievalModelFactoy`,
- `ItemSimilarityAlgorithm`,
- `ItemSimilarityPrecomputer`,
- `ItemSimilarityStore`,
- `ItemFeatureGenerator`,
- `FeatureRetrievalModule`,
- `RankingModel`, and
- `Evaluator`.

Initialized with a set of testing sessions and a candidate set of items, `Evaluator` can evaluate the performance of instances of `RetrievalModel` and `RankingModel`. One instance of `Evaluator` component will evaluate the retrieval and ranking algorithms on the validation dataset, and two further instances will perform final testing on the final and leaderboard datasets.

The `ItemSimilarityAlgorithm` is an interface for algorithms that calculate the similarity scores between items, such as cosine similarity between item vectors in the rating matrix or relevance scores between items, such as Swing or ItemCF score. The `ItemSimilarityPrecomputer` employs the item similarity algorithms to precompute the item similarities for later usage. The resulting precomputed item similarities are accessible for other components through the `ItemSimilarityStore`. A parameter `count` of similarity `ItemSimilarityPrecomputer`, by default set to 1000, defines how many of the most similar items for each item are stored together with the corresponding similarity scores in `ItemSimilarityStore`.

`RetrievalModel` defines a common interface for candidate item retrieval based on a given session. Thus, the implementations of the `RetrievalModel` are the retrieval algorithms whose implementations will be further covered in a separate section. The initialization of the retrieval models is handled by the `RetrievalModelFactoy`. This component is helpful since different instances of the `RetrievalModelFactoy` handle the creation of retrieval models trained for validation and final testing.

`RankingModel` is a class encompassing the whole retrieval and ranking stage. The ranking model is initialized with a set of retrieval models and corresponding counts of how many items should each retrieval model retrieve.

Different subclasses of the `RankingModel` employ different ranking methods. During inference and training, the ranking model receives the candidate items for a session from its retrieval models. These candidate items are then enriched with features from `FeatureRetrievalModule` to form corresponding item feature vectors. The item feature vectors of the candidate items are then used for training or are ranked by the already trained ranking algorithm. `FeatureRetrievalModule` contains features for fast access that we created by `ItemFeatureGenerator` and also includes item-to-item similarities that were stored in `ItemSimilarityStore`.

As a language of our choice for implementing the multi-stage recommender system, we chose Python as it has well-established libraries for many machine learning tasks and data manipulation and analysis libraries such as pandas [54]. The environment that enabled us fast iteration during the development of our components and reproducible testing of the recommender system was Jupyter Notebook [55].

3.3 Training, validation, and testing data split

The testing sessions for the Recsys Challenge 2022 are split into final and leaderboard datasets as outlined in Chapter 2. During the competition, 100 items were predicted for each testing session, and after submission to the Recsys server, the scores were determined based on actual items purchased. With the purchased items now publicly accessible, we can independently compute the final and leaderboard scores. With the testing set already defined in Section 2.2, a validation set still has to be created to compare various algorithm variants before obtaining the final performance estimates on the final and leaderboard datasets.

As all available sessions from June 2021 are already in the testing datasets, we had to validate our recommender system using data prior to June 2021 in order to identify the best-performing algorithms and their hyperparameters. We opted to assess our recommender system and algorithms using data from May 2021, the closest month to June, and to use the entire month to evaluate the algorithms over a comparably lengthy future period. The retrieval and ranking algorithms in the validation phase will be trained on the months preceding May 2021. Furthermore, based on the best-performing algorithms on validation data, the recommender system would subsequently be trained on data that includes May 2021 and finally tested on the testing dataset.

Following the approach of the winning solution in the competition, we trained our models using data from the three months immediately before the validation and testing months. This strategy reduces memory requirements and training time while ideally not compromising performance, as domain trends tend to shift over time—an aspect illustrated in Figure 3.2. Figure 3.3 depicts the organization of datasets by month and usage.

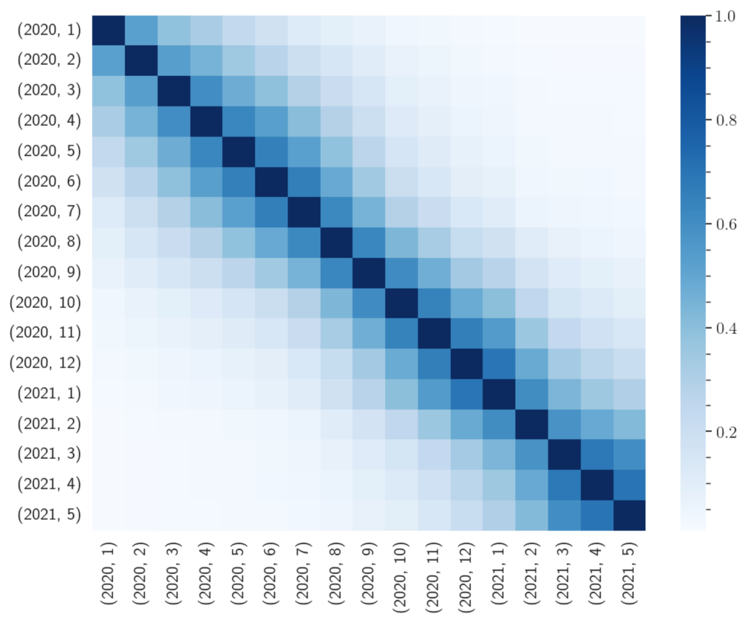


Figure 3.2: Heatmap of Jaccard coefficients between sets of purchased items in different months

Since a subset of items could be predicted for the testing sessions was provided by the challenge authors as described in Chapter 2.2. We created a whitelist containing items that were purchased in the validation month. This whitelist was then passed to the retrieval models to perform filtering during their evaluation on validation sessions or when generating candidate items for the ranking models during validation.

3.4 Retrieval stage

This section will discuss the retrieval models implemented in this thesis. The implemented retrieval models will serve as algorithms in the retrieval stage for candidate generation, and their candidate scores can be utilized as features during the ranking stage. The performance metrics assessed for each retrieval model are the mean reciprocal rank and hit rate [56]. The mean reciprocal rank indicates the quality of the model's performance when used as a single recommendation algorithm solution or as a feature for the ranking algorithm. The hit rate measures the model's ability to retrieve the items that are eventually purchased. In the following chapter, a subset of retrieval algorithms will be experimentally chosen to generate candidates in the retrieval stage, as running all implemented retrieval models might be inefficient and computationally expensive.

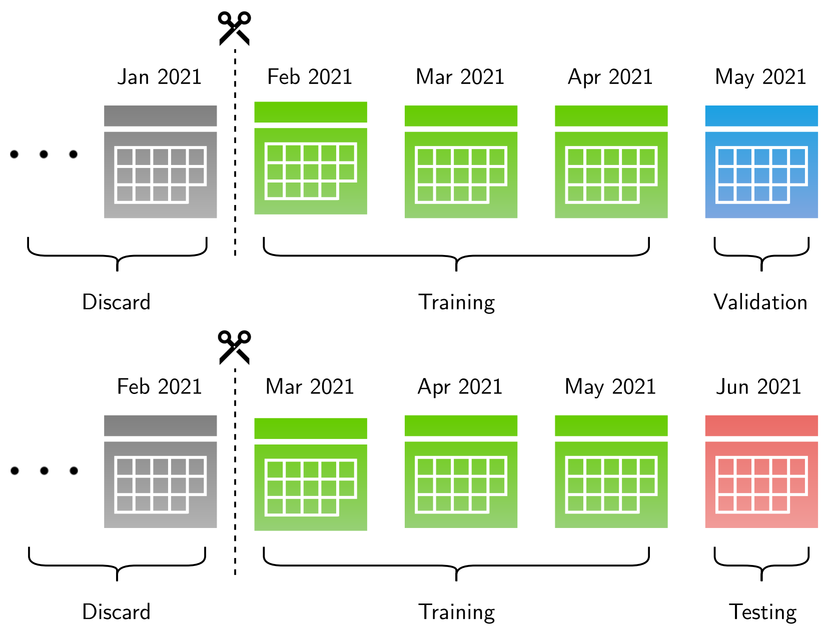


Figure 3.3: Dataset organization for validation and testing

3.4.1 Metrics for retrieval models

To define the metrics we will employ for retrieval algorithms more formally, let m denote a model and let S be a set of evaluating sessions. For each $s \in S$, let p_s denote the purchased item for the session s, and let $C^n_{m,s}$ denote a set of n retrieved candidates by model m for the session s. The formula for HitRate@N follows:

$$\text{hit}^n_m(s) = \begin{cases} 1, & \text{if } p_s \in C^n_{m,s} \\ 0, & \text{otherwise} \end{cases},$$

$$\text{HitRate@N}(S, m) = \frac{\sum_{s \in S} \text{hit}^N_m(s)}{|S|}.$$

Similarly, given a bijection $\text{pos}^n_{m,s} : C^n_{m,s} \to \{1, 2, \ldots, n\}$, assigning candidate items their rank, we can formulate MRR@N in the following way:

$$\text{reciprocal\_rank}^n_{m,s}(i) = \begin{cases} \dfrac{1}{\text{pos}^n_{m,s}(i)} & i \in C^n_{m,s} \\ 0 & \text{otherwise} \end{cases},$$

$$\text{MRR@N}(S, m) = \frac{\sum_{s \in S} \text{reciprocal\_rank}^N_{m,s}(p_s)}{|S|}.$$

Note that given a set of session S and a model m, both metrics MRR@N and HitRate@N are non-decreasing for increasing values of N.

3.4.2 Popularity-based model

Popularity-based models are non-personalized recommendation algorithms, which recommend items based on their overall popularity. One straightforward approach to measuring an item's popularity is to count the number of purchases or views of that item in the training data. However, since our goal is to predict purchased items, it is more sensible to primarily consider purchases, as the distribution of item purchases may not necessarily align with the distribution of item views. For instance, an expensive item may be frequently viewed but rarely purchased.

In our study, we attempted to incorporate both the number of purchases and views by giving purchases more weight compared to views. However, the highest HitRate and MRR scores during validation were achieved by focusing solely on purchases. The model was further improved by incorporating time decay to give more weight to more recent trends. To implement the time decay, we would multiply each purchase x by $\frac{1}{\log(1 + \text{day\_delta}(x))}$, where $\text{day\_delta}(x)$ is the number of days between the purchase x and the last date in the training data. Thus, the popularity score could be calculated as:

$$\text{score}_{popularity}(i) = \sum_{x \in X_i^p} \frac{1}{\log(1 + \text{day\_delta(x)})}.$$

3.4.3 Content-based model

Each item in the data set has assigned a set of attributes in the form of `(category_id, value_id)`. Two different items do not have to share an attribute of the same category, and one item can be assigned multiple attributes with the same category and different values.

We measured the similarity between two items based on these item attributes using the Jaccard similarity coefficient in two ways. Let C be a set of all attribute categories in the dataset, let V_c be a set of all values for a category $c \in C$, and let $A_i = \{(c, v) \mid c \in C \land v \in V_c\}$ be a set of all attributes assigned to an item i. The equations for the Jaccard similarity index between two items using attribute categories and attributes follow:

$$J_{categories}(i, j) = \frac{|\{c \mid (c, v) \in A_i\} \cap \{c \mid (c, v) \in A_j\}|}{|\{c \mid (c, v) \in A_i \cup A_j\}|},$$

$$J_{attributes}(i, j) = \frac{|A_i \cap A_j|}{|A_i \cup A_j|}.$$

Using the formulas mentioned above, we can precompute similarities between all pairs of items for quick access during inference. Since most of the items are quite different — they have a small Jaccard similarity coefficient — to be more memory efficient, it makes sense to for each item only save a certain number of most similar items and their corresponding scores. As already stated in Section 3.2, we store the 1000 most similar items for each item.

The content-based retrieval algorithm in our solution uses these computed similarities to calculate which items are the most similar to the items in a given session. The algorithm calculates the average similarity of an item to items in the session. We achieved an improvement in the MRR score on the validation data when we used the weighted average and only considered the n last items in each session. Note that sessions in our dataset correspond to sequences of item views. Let nth_last_item : $S \times \mathbb{N} \rightarrow I$ be a function that, given a session $s \in S$ and a position $m \in N$ returns the m^{th} last viewed item in the session s. A score for an item i, given a session s, is computed using the following formula:

$$w(s,p,i) = \begin{cases} J_{attributes}(i, \text{nth\_last\_item}(s,p)) & |s| \geq p \\ 0 & \text{otherwise} \end{cases},$$

$$\text{score}_{\text{content-based}}(s,i) = \frac{\displaystyle\sum_{p=1}^{n} \frac{w(s,p,i)}{p}}{\displaystyle\sum_{p=1}^{n} \frac{1}{p}}.$$

The Jaccard similarity on attributes performed much better than the Jaccard similarity on categories and was used in our content-based model. The hyperparameter $n = 5$ received the highest MRR score in the validation data. The advantage of the content-based model (and popularity-based model) over collaborative filtering retrieval models is that, given a session, it can always generate a set of candidate items since all items have some attributes based on which we can calculate similarity to other items. In collaborative filtering algorithms, in cases where the user interacted with cold items — items without interactions in training data — we end up with an empty candidate set.

3.4.4 ItemKNN

In our implementation of the ItemKNN, to calculate a similarity between items, we use a cosine similarity and rating matrix created from the training dataset as in Section 1.3.1. To speed up the ItemKNN computation, the 1000 most similar items for each item are precomputed and stored for fast access as described in Section 3.2.

When creating a user vector for the target user, we do not assign viewed items the same rating value for being the same type of interaction. In our task, a session corresponds to a user interaction history represented by a chronologically ordered sequence of item views. We can consider the more recently viewed items as more similar to the purchased item for being closer in the session to the purchased item. We give a viewed item i by the target user a rating of value $1/p$, where p is the first position in a reverse session item view sequence of the target user. E.g., p is one for the last viewed item in the sequence, two for the second last viewed item, and so on. Also, before we create the ratings, we truncate the session to the last n item views, where n is a hyperparameter. The formulas by which we compute ItemKNN scores follow:

$$w(s,p,i) = \begin{cases} sim_{cos}(i, \text{nth\_last\_item}(s,p)) & \text{session } s \text{ contains } p \text{ item views} \\ 0 & \text{otherwise} \end{cases},$$

$$score_{\text{itemKNN}}(s,i) = \frac{\sum_{p=1}^{n} \frac{w(s,p,i)}{p}}{\sum_{p=1}^{n} \frac{1}{p}}.$$

The hyperparameter n was fine-tuned to 6. Note that the k determining the size of the rated item neighborhoods is set to 1000 since we store 1000 of the most similar items and their similarities.

3.4.5 UserKNN

The UserKNN algorithm is well described in Section 1.3.3. In contrast to our ItemKNN implementation, we created the user vector by setting all the ratings of the viewed items, which were seen in the training dataset and thus have a corresponding column in the rating matrix, to $c_v = 1$, as used in the rating matrix.

To find the most similar users in the rating matrix using cosine similarity to the target user's rating vector, we utilized the NearestNeighbors class from the scikit-learn [57] open-source library for machine learning in Python.

3.4.6 Items-to-item models

An approach that proved effective in the candidate generation stage was the usage of items-to-item algorithms to recommend items somehow similar or relevant to the most recent items in a given session. The intuition behind this candidate generation approach is that often just before the user finds the wanted item they might be viewing another similar item.

Let us explain the method more formally. Let session $s = (i_1, \ldots, i_{n_s})$ be a vector of n_s items, which were viewed before the purchased item, ordered by their corresponding view timestamps. Let I_c be a set of all candidate items and let $R_s \subset I_c$ be a set of recommended candidate items recommended by our approach. It holds that

$$i \in R_s \implies |\{j \in I_c \mid \text{score}(i_{n_s}, j) > \text{score}(i_{n_s}, i)\}| < |R_s|,$$

where the function score : $I \times I \to R$ given items x and y as an input returns a relevancy value for the item y given the item x.

An advantage of generating candidates based on one item is fast inference since it consists of querying the pre-computed list of the most similar items for the given item. The item-to-item score computing methods used were primarily based on collaborative filtering and pair co-occurrence in the training sessions. An issue during inference occurs when the last item in the session has not appeared in the training set. As a fallback in such a situation, we would generate candidates based on the second last viewed item, if available.

In the following parts of this subsection, different item-to-item score function that we used to find the most relevant candidates will be covered.

3.4.6.1 Swing

We used the Swing score function described in Section 1.3.6, with only modification that penalizes long item sessions based on [31]. The original formula was changed by adding the following weights to the nominator:

$$w_u = \frac{1}{\mid I_u \mid},$$

$$\text{score}_{\text{SWING}}(i, j) = \sum_{a \in U_i \cap U_j} \sum_{b \in U_i \cap U_j} \frac{w_a \cdot w_b}{\alpha + \mid I_a \cap I_b \mid}.$$

3.4.6.2 ItemCF

The algorithm, which achieved the highest MRR on validation sessions, if we do not consider an ensemble that uses this algorithm, is the similarity-based model using ItemCF scores. The implementation is based on a modification of the ItemCF from [31].

The algorithm leverages the co-occurrence of items in session, indicating some similarity or mutual relevance. The degree of the mutual relevance of such co-occurrence in a session depends on the session length, which is mentioned in Section 1.3.5, and it further depends on how close the items are in the session interaction sequence and how much time separates the two item interactions. We further add consideration for the order of the items in a session and whether an item in a session was purchased, which further improved the MRR of this retrieval model on validation sessions. The intuition behind the consideration of the order of the items in the session is that items viewed after a particular item are more likely to be purchased as a result of seeing that item than those items seen before it.

Let $S_i \subseteq S$ denote a subset of sessions containing an interaction with an item i, let $P_{i,s} \subset \mathbb{N}$ denote a set of positions of interactions of the item

i in the session s, and let time : $\mathbb{N} \times S \to \mathbb{N}$ be a function with returns a timestamp in seconds of an interaction on a given position in a given session. The equation for computing ItemCF score follows:

$$w_{pos}(p_i, p_j) = \frac{1}{\log_2(|p_i - p_j| + c_1)}$$

$$w_t(s, p_i, p_j) = \frac{1}{\log_4(|\operatorname{time}(s, p_i) - \operatorname{time}(s, p_j)| + c_2)}$$

$$w_{len}(s) = \frac{1}{\log_2(|s|)}$$

$$w_b(s, p_j) = \begin{cases} \beta & p_j = |s| \\ \alpha & p_i < p_j < |s| \\ 1 & \text{otherwise} \end{cases}$$

$$\operatorname{score}(i, j) = \sum_{s \in S_i \cap S_j} \sum_{p_i \in P_{i,s}} \sum_{p_j \in P_{j,s}} w_b(p_i, p_j) \cdot w_{len}(s) \cdot w_t(s, p_i, p_j) \cdot w_{pos}(p_i, p_j),$$

where α, β are hyperparameters set to 2 and 3, and where $c_1 = 1$ and $c_2 = 5$.

3.4.6.3 Rating similarity

By rating similarity, we refer to the cosine similarity between item rating vectors from the rating matrix, as defined in Section 1.3.1. In order to identify the users in the rating matrix that are most similar based on cosine similarity, we employed the NearestNeighbors class from scikit-learn [57], which was also utilized in the UserKNN.

3.4.6.4 Word2vec

To predict the most similar items that appear in similar contexts to the last item in a session, we utilize word2vec [58] embeddings through the open-source Python library Gensim [59]. We experimented with different vector sizes and window sizes and found that combining a vector size 64 and a window size 4 gave the best results on the validation data.

3.4.6.5 Factorization

Another approach for recommending similar items utilizes a matrix factorization using alternating least squares. The item similarity score is then calculated as cosine similarity between item factor vectors. The Python library that we employed for factorization was Implicit [60]. We achieve the best results we tuned the number of factors and regularization on validation data.

3.4.6.6 Similarity-based ensemble model

Since we have implemented multiple score or similarity functions to predict the items most relevant to the last item in the session, we explored the idea of creating an ensemble of these individual models. To do this, we had to scale all the output scores and similarities with min-max scaling into a similar range. The range for a model was specified with a min value that was the same for all models and a max value, which we refer to as weight. The ensemble achieved the best performance when better-performing models were assigned higher weights than worse-performing models. Given m different methods, where each method has its score_a function and weight w_a, and a min value c_{min} such that $\forall a \in \{1, \cdots, m\} : c_{min} < w_a$, the equation for computing ensemble similarity follows:

$$\max_a(i) = \max\{\text{score}_a(i,j) \mid j \in I\}$$

$$\min_a(i) = \min\{\text{score}_a(i,j) \mid j \in I\}$$

$$scale_a(i,j) = \begin{cases} \dfrac{\text{score}_a(i,j) - \min_a(i)}{\max_a(i) - \min_a(i)} & \max_a(i) > \min_a(i) \\ 0 & \max_a(i) = \min_a(i) \end{cases}$$

$$\text{score}_{ensemble}(i,j) = \sum_{a=1}^{m} scale_a(i,j) \cdot (w_a - c_{min}) + c_{min}.$$

3.5 Ranking Stage

To address the ranking task, we chose to try pointwise and listwise approaches utilizing the gradient-boosting library LightGBM. In this section, we discuss the creation of the training dataset for our ranking model and the features that might aid the ranking model to achieve better performance.

3.5.1 Ranker Training

The training dataset for the ranking model is created from training sessions no older than one month from the start of the validation or testing month in the following manner. For each such training session, we employ retrieval algorithms from the retrieval stage to generate candidate items. On average, the number of candidates per session can be in the hundreds, e.g., around 500 candidate items. If the candidate items do not contain the corresponding purchased item for the training session, we discard the session and continue to the following session sample.

A situation when the candidate items do not include the purchased item might occur when the purchased item is not included in the provided whitelist (described in Section 3.3) and thus is filtered out. Furthermore, it can happen when the retrieval models fail to retrieve the purchased item. In the latter case, adding the purchased item to the candidate set for training could be possible. However, this did not show improvement in the overall performance of the model on the validation data, and it increased the training time, so we omitted to do so.

Therefore, a candidate item set for a training session contains a corresponding purchased item. The purchased items are then labeled as 1, and the other items are labeled as 0. In case a listwise model is trained, item candidates from the same training session are labeled with the same group id; for this, session-id can be used. One may notice that most target labels are zeroes, and the dataset is imbalanced. However, we do not handle this imbalance since, to rank the items, we are mainly interested in the relative order of the target variables. Finally, the training dataset consists of the feature vectors, labels, and possibly group ids of all generated candidate items. The features we can use to create the feature vectors will be covered in the following section.

3.5.2 Features for ranking algorithms

In the ranking stage, it is essential to create features that allow the ranking algorithm to rank items well. A ranking algorithm receives as input a set of candidate items represented by their feature vectors. As described in Section 1.4, features in this recommendation scenario can either entirely depend on the candidate item, entirely depend on the session and its context, or a combination of both.

Examples of features we can use that depend solely on the candidate item include:

- number of purchases in the last month,
- ratio between purchases and views in the last month,
- difference of purchases in the last two weeks of last month, or
- a score from the time-decay popularity model trained on the last month.

This category of features can be precomputed and accessed during training and inference. Let us highlight that we do not include interaction statistics of the current month, e.g., last week's purchase count, because the task of the final model is to predict purchases for the whole next month and thus cannot have such features available.

Even though the sessions are anonymous and no user attributes are available, it is still possible to create session-dependent features. The examples of features that we used are

- number of item views in the session,

- day of the week, or
- average time between item views.

We compute the session-dependent features at inference time.

The last category of features is based on the combination of the target user and the candidate item, and we will refer to them as cross-features. Specifically, similarities between the last item seen by the user and candidate items, which are the output scores of some of our retrieval models, fall into this category. Some examples in this category of features include:

- average attribute similarity between the item and the session items,
- ItemKNN score, or
- similarity to the last viewed item (Swing, Rating similarity, ItemCF, Ensemble similarity).

The cross-features are either computed during inference, as the first two examples are, or they can be precomputed during training and only retrieved during inference according to the information in the session. This is the case for the similarity to the last item in the session.

3.5.3 Hyperparamer tuning

To optimize the hyperparameters of the LightGBM models, we used the Optuna [61] hyperparameter optimization framework. The LightGBM model includes classical GBDT parameters such as the number of trees, max depth of trees, and the number of leaves, as well as hyperparameters that enforce regularization, some of which were described in Section 1.4.3.2. Specifically, L1 and L2 regularization coefficients and the bagging fraction and feature fractions that determine the size of randomly selected subsets of features and data samples that are used for building each tree. From the mentioned hyperparameters, we manually set the number of trees to 250 and max-depth to 10 and tune the rest from specified intervals. To speed up the computation of validation MRR for each hyperparameter combination, we used a random subsample of validation data containing 8,000 sessions.

Chapter 4

Experiments

In this chapter, we present an evaluation of the multi-stage recommender system proposed in the previous chapter. We begin by analyzing the performance of the retrieval models individually on validation data. Next, we explore how the retrieval models complement each other to create a robust retrieval stage. With the retrieval stage in place, we apply ranking algorithms and compare their performance against the validation data. Additionally, we evaluate the importance of the features used by the ranking algorithms. Moreover, we compare all the algorithms on the leaderboard and final sessions to determine their overall performance. Finally, we discuss further possible improvements to the recommender system to further enhance its performance, as well as potential research directions for future work.

4.1 Retrieval Models

Firstly, we studied the qualities of the retrieval models to see their strengths when used individually. The metrics used were MRR@N and HitRate@N for chosen values ranging from 1 to 200. In the context of the retrieval stage, HitRate can be viewed as the more critical metric since the goal of this stage is to retrieve possible candidates. However, the ranking stage can use the retrieval models' output for candidate ranking. Therefore, both metrics should be considered. The evaluation results of implemented retrieval algorithms are displayed in Figure 4.1 and Figure 4.2. The relative comparisons of HitRate@N and MRR@N results of different retrieval algorithms are illustrated in Figure 4.3 and Figure 4.4.

From Figure 4.4, it is apparent that relative rankings of different algorithms did not change for MRR@N at different values of N, unlike for the HitRate@N scores. This is mainly due to the definition of the MRR@N metric, where items ranked at higher positions do not contribute significantly to the overall MRR score.

To evaluate the MRR results, Figure 4.4 shows that the ensemble of similarities (ItemCF, Swing, rating similarity and word2vec, with weights 2, 2.5, 1, and 1 for the listed models in given order) was the best-performing model for the given metric. The model's great MRR performance hints that the ensembled similarity to the last item in the session might be a valuable feature for the ranking algorithms. The second and third places also belong to ItemCF and Swing, which are also algorithms that recommend based on the similarity to the last item and are included in the best-performing ensemble. Furthermore, ItemKNN placed fourth, and the rating similarity to the last item placed fifth. This shows that the ItemKNN that leverages the rating similarity but considers more items from the session outperforms a solution based only on the last item. Furthermore, the sixth place belongs to UserKNN despite it having the longest inference time. The popularity-based algorithm expectedly finished in last place, given the non-personalized nature of the algorithm, which always recommends the same items. Content-based algorithm placed second-to-last with an MRR@100 score four times higher then the popularity-based model score and almost half the MRR@100 score compared to the best-performing algorithm.

In relation to the HitRate@N scores of the items, we would like to highlight that the rankings of the implemented retrieval algorithms for HitRate@N, with N equal to 1, 3, and 5, correspond to the rankings of MRR@N. This connection stems from the definition of MRR, which is affected the most by the algorithm's ability to recommend the ultimately purchased items in the top positions. Regarding the HitRate@N, we will be primarily interested in higher values of N, such as 100, 150, and 200, since our goal in the retrieval stage is to retrieve hundreds of relevant candidate items, ideally including the ultimately purchased item.

For the N of 100, 150, and 200, the most significant HitRate@N is achieved by ItemKNN, which performs better than all other retrieval models from $N = 25$. Its HitRate@100 is 48.3%, and HitRate@200 is 54.1%. In that order, the following top positions belong to the last item similarity-based models: ensemble, ItemCF, word2vec, rating similarity, and Swing. Interestingly, the word2vec last item similarity-based model improves its HitRate@N with higher N much better than rating similarity and Swing models, which leads to it outperforming the mentioned algorithms. The next position belongs to the content-based model, which improves its hit rate well with larger values of N compared to other models and finishes with a HitRate@200 of 48.3%. The least performing last item similarity-based model is factorization regarding both HitRate and MRR scores. The UserKNN model does not improve its HitRate@N after $N = 100$, and it stops at 38%, which is caused by its inability to generate large numbers of candidates. Similarly to results regarding MRR scores, the popularity-based model achieves the lowest HitRate scores, with HitRate@200 at 28%.

N	HitRate (%)	MRR
1	5.2721	0.0527
3	10.7047	0.0761
5	14.0337	0.0838
10	18.7385	0.0900
25	25.2408	0.0942
50	31.0422	0.0958
100	36.2494	0.0966
150	39.7327	0.0968
200	42.2113	0.0970

(a) Factorization

N	HitRate (%)	MRR
1	6.8430	0.0683
3	14.2848	0.1000
5	18.2166	0.1090
10	24.1282	0.1169
25	32.2919	0.1221
50	38.7243	0.1239
100	45.1396	0.1248
150	49.1570	0.1252
200	51.8536	0.1253

(b) Word2vec

N	HitRate (%)	MRR
1	8.2813	0.0828
3	16.5258	0.1180
5	21.2429	0.1289
10	26.8642	0.1365
25	33.8663	0.1409
50	38.9755	0.1424
100	44.1104	0.1431
150	47.2592	0.1434
200	49.5319	0.1435

(c) Rating similarity

N	HitRate (%)	MRR
1	9.5102	0.0951
3	17.9299	0.1310
5	22.6715	0.1419
10	28.4337	0.1496
25	35.8095	0.1543
50	41.1025	0.1558
100	46.0119	0.1565
150	48.6082	0.1567
200	50.2941	0.1568

(d) Swing

N	HitRate (%)	MRR
1	10.2820	0.1028
3	18.5155	0.1382
5	23.0317	0.1485
10	28.6652	0.1561
25	35.9663	0.1607
50	41.2691	0.1622
100	46.6502	0.1630
150	49.8223	0.1633
200	52.0780	0.1634

(e) ItemCF

N	HitRate (%)	MRR
1	10.7011	0.1070
3	19.0963	0.1431
5	23.5181	0.1531
10	29.0108	0.1605
25	36.0411	0.1649
50	41.3487	0.1664
100	46.7764	0.1672
150	50.0564	0.1675
200	52.3855	0.1676

(f) Ensemble

Figure 4.1: HitRate@N and MRR@N for retrieval models based on similarity to the last viewed item

N	**HitRate (%)**	**MRR**
1	0.6972	0.0073
3	1.9064	0.0124
5	2.9246	0.0148
10	4.5676	0.0170
25	8.1930	0.0193
50	13.3929	0.0207
100	19.1000	0.0216
150	24.4358	0.0220
200	27.9595	0.0222

(a) Popularity-based model

N	**HitRate (%)**	**MRR**
1	4.4757	0.0448
3	9.4562	0.0663
5	12.3441	0.0728
10	16.6556	0.0785
25	24.4542	0.0833
50	31.3999	0.0853
100	39.5648	0.0864
150	44.6372	0.0868
200	48.3214	0.0871

(b) Content-based model

N	**HitRate (%)**	**MRR**
1	7.5118	0.0751
3	14.6156	0.1058
5	18.4077	0.1144
10	23.3502	0.1211
25	29.3293	0.1250
50	33.6592	0.1262
100	38.0002	0.1268
150	38.0002	0.1268
200	38.0002	0.1268

(c) UserKNN

N	**HitRate (%)**	**MRR**
1	8.7493	0.0875
3	17.5072	0.1251
5	22.3909	0.1363
10	28.6224	0.1447
25	36.6206	0.1498
50	42.5041	0.1515
100	48.2982	0.1523
150	51.6491	0.1526
200	54.1229	0.1527

(d) ItemKNN

Figure 4.2: HitRate@N and MRR@N for different retrieval models on validation data

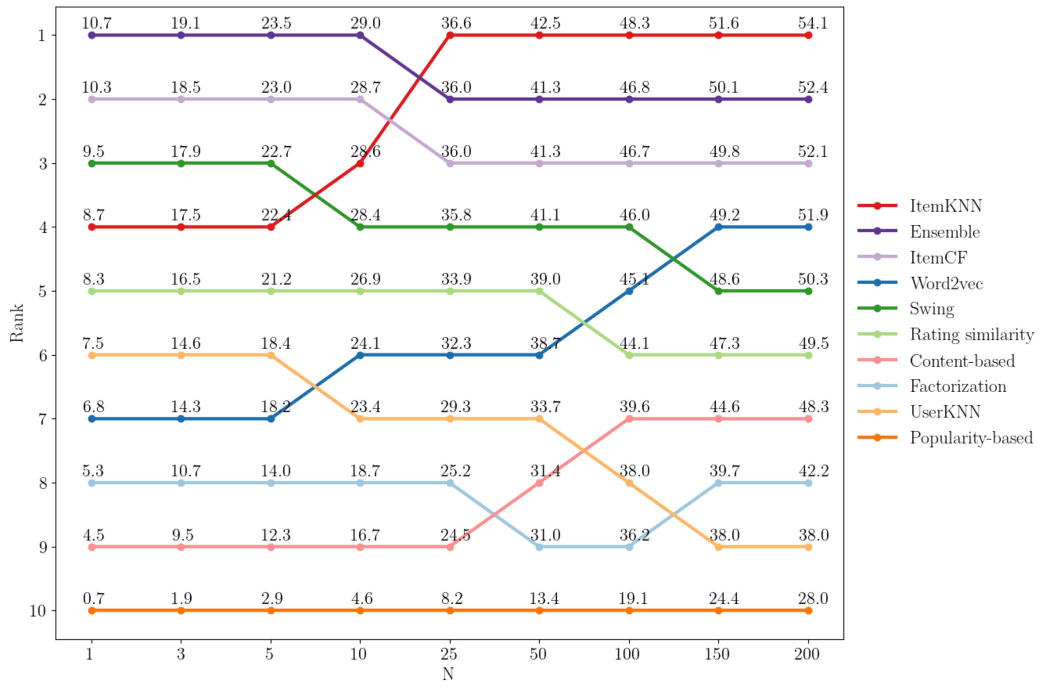


Figure 4.3: Bump graph for HitRate@N in % for retrieval models on validation data

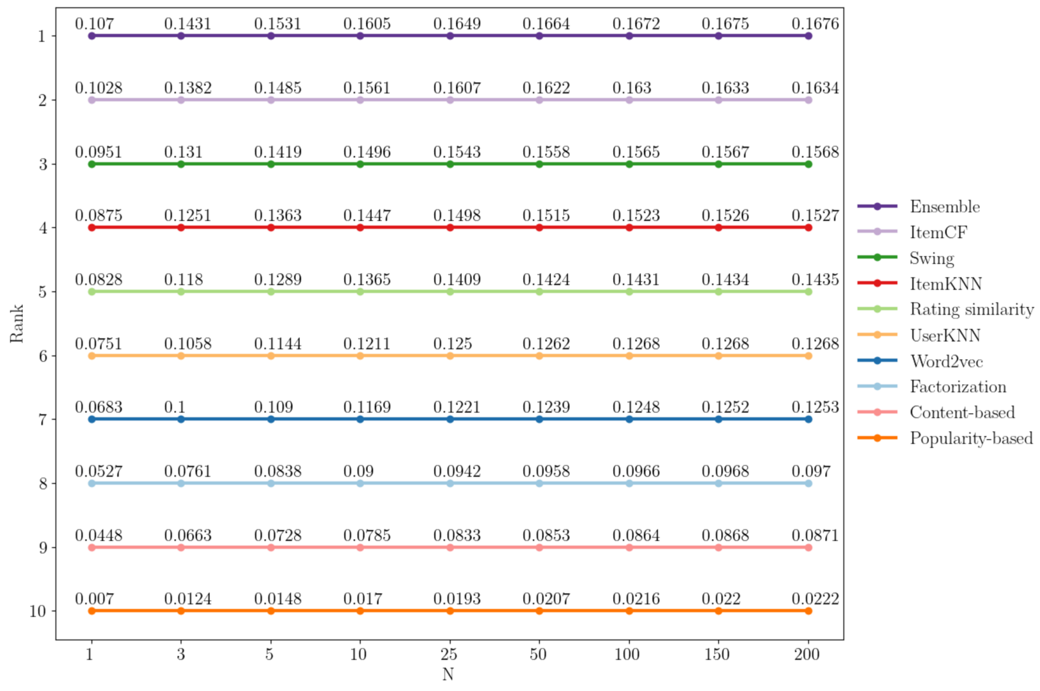


Figure 4.4: Bump graph for MRR@N for retrieval models on validation data

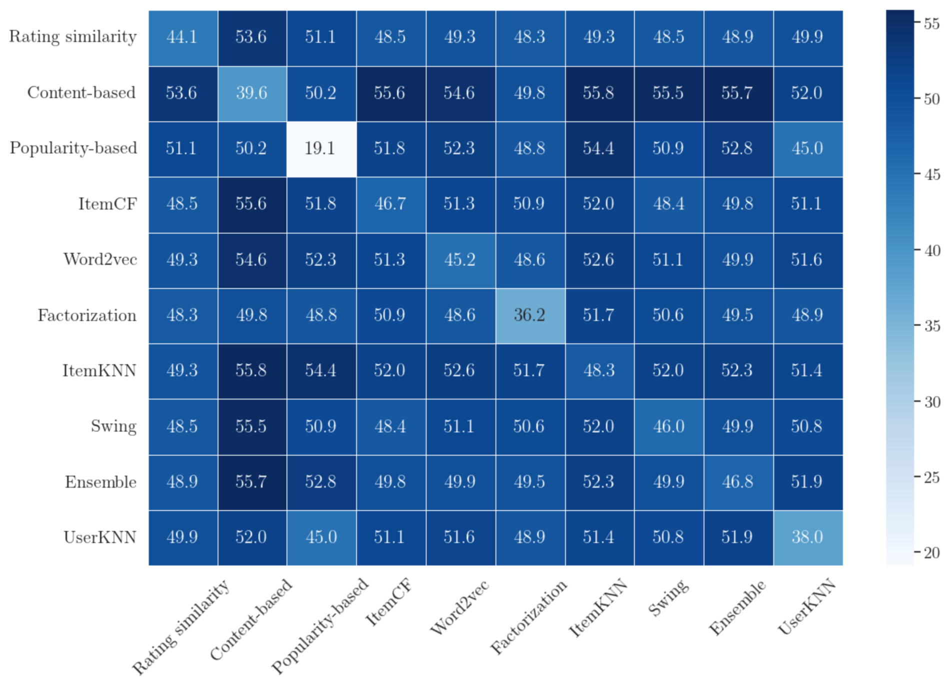
4.2 Selecting retrieval models

In this section, we will study how the retrieval models perform when used together in order to select a subset of retrieval models that will be employed together to generate a candidate set with a high hit rate. The motivation behind choosing a subset of retrieval models is mainly to save computational resources and increase the recommendations throughput, as including retrieval algorithms in the retrieval stage that do not contribute much to the final candidate set or do not help to improve the hit-rate set is inefficient. Furthermore, the more items are recommended in the retrieval stage, the more computationally intensive the following ranking stage. Therefore, to assess this tradeoff, we will aim to generate a retrieval stage with a high hit rate but still consider the number of total retrieved candidate items.

To evaluate how the individual retrieval models complement each other, trying all combinations of retrieval models and comparing their hit rates might take a lot of time and computational resources. Therefore, instead, we employ a more gradual approach. We start by studying how pairs of different retrieval models complement each other. We let each model recommend 100 and 200 candidates and then use the combined sets of candidates to measure HitRate. Depending on the number of the same items recommended by both algorithms, each set of candidates in the first variant contains between 100-200 items, and 200-400 items in the second variant. The HitRate@200 and HitRate@400 for pairs of retrieval models are depicted in Figure 4.5 and Figure 4.6. The column and row corresponding to the content-based model appear to contain the highest HitRate values. This means that the content-based model can retrieve candidates that other models do not, likely because the model uses content-based filtering while other models mainly use collaborative filtering methods.

ItemKNN and content-based model pair achieve the highest values in both variants. This combination of content-based and collaborative filtering models will be the base for candidate generation in the retrieval stage. The validation data shows that the HitRate is 62.9% when each of the two algorithms retrieves 200 candidates. To improve this, we will greedily try to add one more model to the selected retrieval models to see which combination results in the highest hit rate. Figure 4.7 shows HitRate@300, where item candidates are generated by combining ItemKNN and Content-based model with other implemented retrieval models, each recommending 100 candidates. Thus, the combined number of candidates is up to 300. Figure 4.9 shows HitRate@600 of the same combinations of retrieval algorithms, where each model recommends 200 candidates.

The results show that the highest scores in both variants are achieved by combining the popularity-based model with ItemKNN and the content-based model, which increases the total HitRate to 61.7% when each model recommends 100 candidates and 67% when each model recommends 200 items.



	Rating similarity	Content-based	Popularity-based	ItemCF	Word2vec	Factorization	ItemKNN	Swing	Ensemble	UserKNN
Rating similarity	44.1	53.6	51.1	48.5	49.3	48.3	49.3	48.5	48.9	49.9
Content-based	53.6	39.6	50.2	55.6	54.6	49.8	55.8	55.5	55.7	52.0
Popularity-based	51.1	50.2	19.1	51.8	52.3	48.8	54.4	50.9	52.8	45.0
ItemCF	48.5	55.6	51.8	46.7	51.3	50.9	52.0	48.4	49.8	51.1
Word2vec	49.3	54.6	52.3	51.3	45.2	48.6	52.6	51.1	49.9	51.6
Factorization	48.3	49.8	48.8	50.9	48.6	36.2	51.7	50.6	49.5	48.9
ItemKNN	49.3	55.8	54.4	52.0	52.6	51.7	48.3	52.0	52.3	51.4
Swing	48.5	55.5	50.9	48.4	51.1	50.6	52.0	46.0	49.9	50.8
Ensemble	48.9	55.7	52.8	49.8	49.9	49.5	52.3	49.9	46.8	51.9
UserKNN	49.9	52.0	45.0	51.1	51.6	48.9	51.4	50.8	51.9	38.0

Figure 4.5: HitRate@200 in % heatmap when combining 100 candidates from two retrieval models

Furthermore, Figures 4.8 and 4.10 show the average counts of recommended items by different combinations. We can observe that the models that generate together a more diverse set of items achieve a higher hit rate.

To try to further improve the hit rate, we repeat once again the previous step. This time, we take the ItemKNN, content-based and popularity-based models as the base ensemble and compare the remaining algorithms with HitRate@400 where each model recommends 100 items and HitRate@800 for each model recommends 200 items. The resulting hit rates do not increase dramatically. The best hit rates are achieved by adding word2vec, factorization, and ensemble. Even though the ensemble has the lowest hit rate out of the three, since it has the lowest number of average recommendations, we choose it as our final combination of retrieval models. To conclude this section, we picked 4 retrieval models, each of which we will recommend top-200 in the retrieval stage. This method results in a total HitRate of 71% on validation sessions and on average 540 items in the generated candidate set.

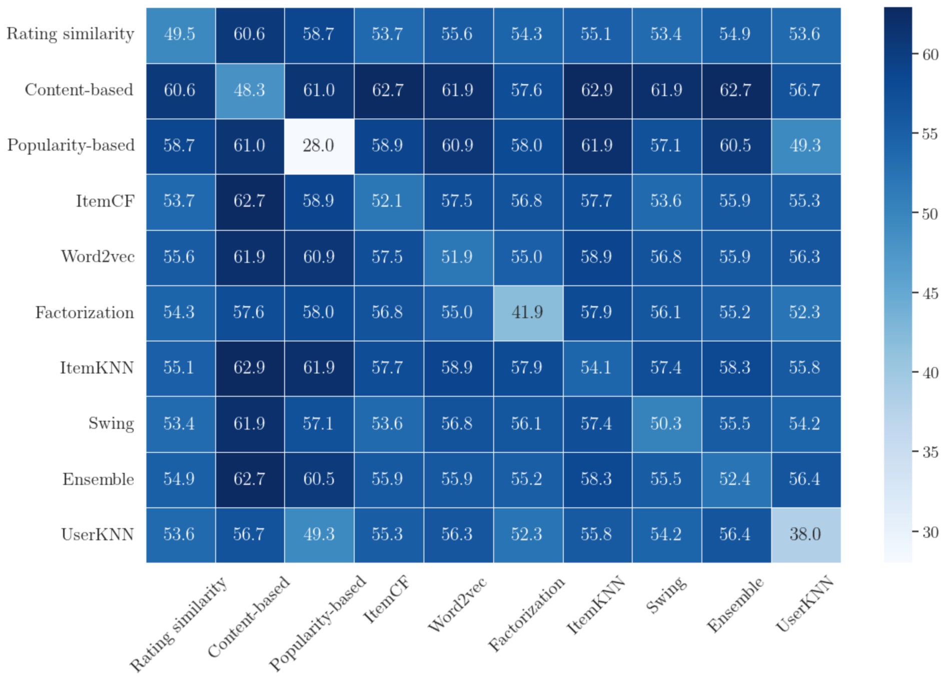


Figure 4.6: HitRate@400 in % heatmap when combining 200 candidates from two retrieval models

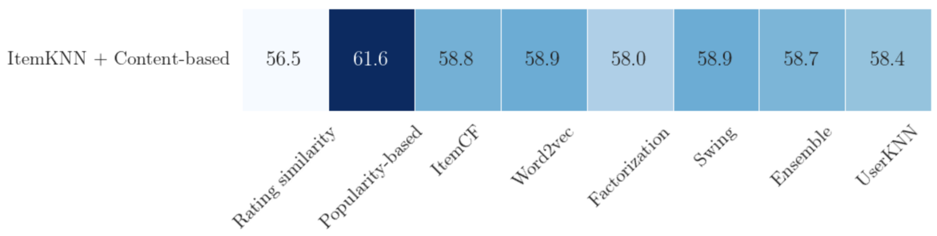


Figure 4.7: HitRate@300 in % when ItemKNN, Content-based model, and another retrieval model each retrieve top-100 items

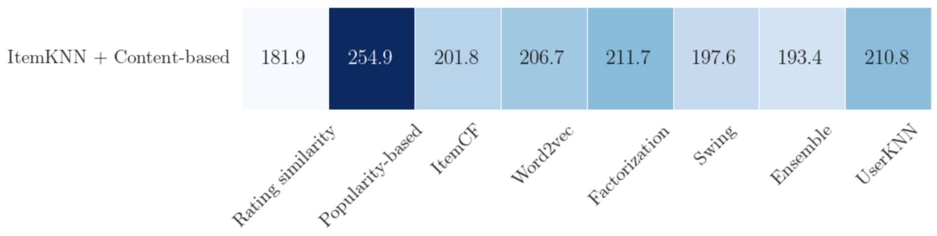


Figure 4.8: Average size of the candidate set when ItemKNN, Content-based model, and another retrieval model each retrieve top-100 items

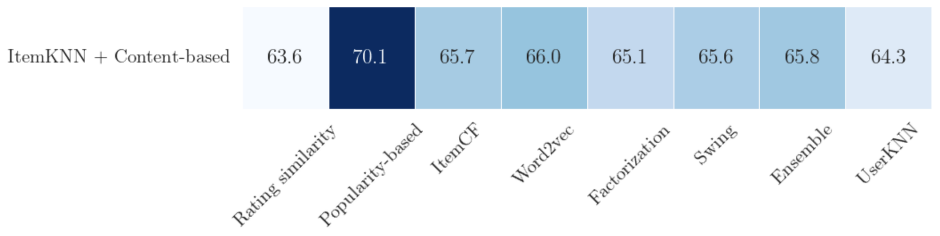


Figure 4.9: HitRate@600 in % when ItemKNN, Content-based model, and another retrieval model each retrieve top-200 items

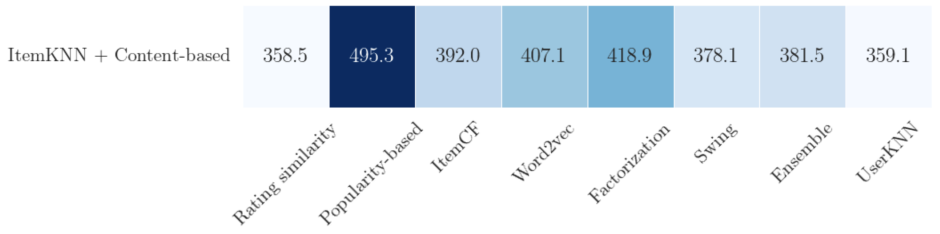


Figure 4.10: Average size of the candidate set when ItemKNN, Content-based model, and another retrieval model each retrieve top-200 items

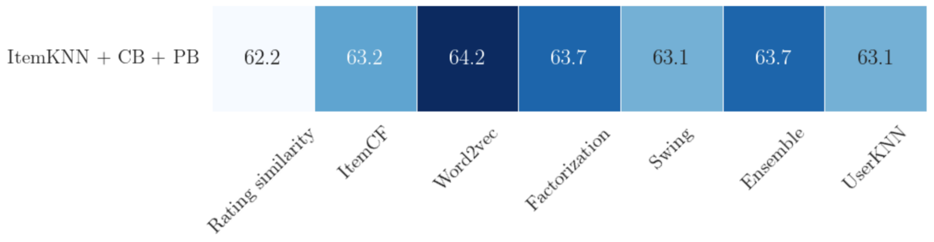


Figure 4.11: HitRate@400 in % when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-100 items

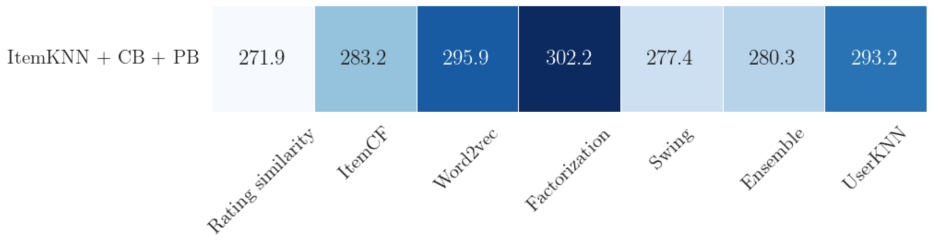


Figure 4.12: Average size of the candidate set when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-100 items

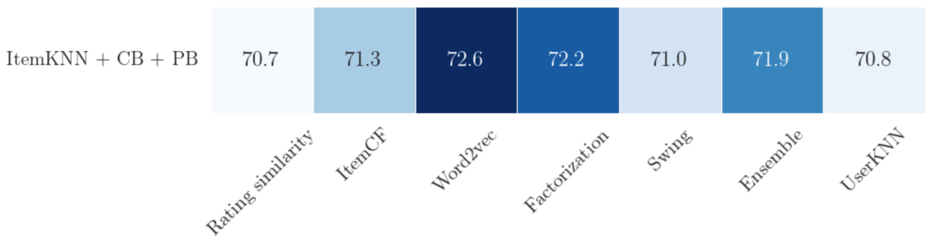


Figure 4.13: HitRate@800 in % when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-200 items

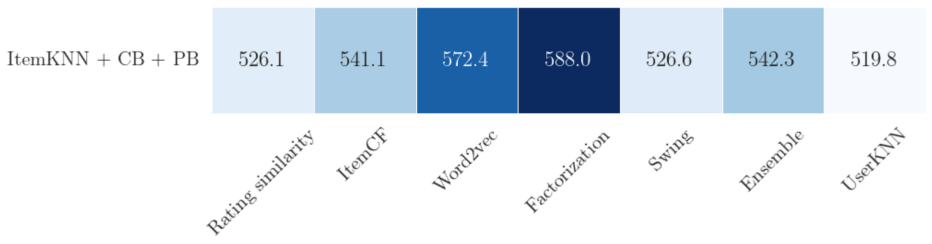


Figure 4.14: Average size of the candidate set when ItemKNN, Content-based, Popularity-based, and another retrieval model each retrieve top-200 items

Ranking model	**MRR@100 validation data**
Pointwise LighGBM	0.1829
Listwise LighGBM	0.1844
Ensemble of above approaches	0.1850

Figure 4.15: MRR@100 of the ranking models on validation sessions

4.3 Ranking algorithms

The ranking algorithms we employed were the pointwise LightGBM model trained as a binary classifier and the LightGBM model trained with the lambda rank objective. In both cases, inspired by the approach in [49], to achieve robust ranking predictions, we combine the output of 5 models. We split the training data into five folds and train each of the models on different four of those folds. The final ranking would than be based on the sum of predicted values by each of those models. Furthermore, we created an ensemble of the pointwise and listwise LightGBM models by applying a weighted average rank for the outputs of the two approaches.

The evaluation results on validation data are depicted in Figure 4.15. The pointwise model achieves MRR@100 of 0.1844, outperforming the listwise model that scored MRR@100 of 0.1829. And the employment of the ensemble of both models, where the pointwise model has an assigned weight of 0.65 and listwise a weight of 0.35, leads to the best MRR on validation data with an MRR@100 score of 0.185.

By observing Figures 4.16 and 4.17, we analyze the importance of the different features used by the listwise and pointwise models for the ranking of the items represented by their feature vectors. The gain-based feature importance for a given feature is the sum of gains from the splits that the feature is used for during the construction of the trees. Let us note that the feature importance is always based on one randomly chosen model from the five models that form the final rankings. When comparing the feature importance of the same models trained on different folds, we found them to be quite similar, mainly regarding the best-performing features. On the contrary, the importance of low-performing features varied much more.

The most essential feature of the listwise LightGBM was the ItemCF score and the ensemble similarity for the pointwise LightGBM. The overall best three features on all models turned about to be from the cross-feature category. The best item-based feature in both models was the conversion rate from the last month of the training data. We can observe that the session-based category of our features did not perform well as the pointwise used only three out of 5 session-based features, and the listwise did not use any of them. The most important category of features turned out to be the cross-feature category, in which the features depend on the combination of the candidate item and the target user. These can often be the most computationally requiring. However, we mostly utilized the item similarities and queried them according to the items in the given session. Furthermore, in the cross-feature category, the features based on collaborative filtering performed better than those based on feature similarities, such as average feature similarity between the candidate items and the target user's session.

4.4 Evaluation on the testing datasets

To get the final performance estimate of the recommendation quality of implemented individual retrieval models and multi-stage recommender systems, we use the versions of retrieval and ranking models parameterized by the best-performing hyperparameters on validation data. And we train those models on the training dataset that includes the validation sessions. Now, we will evaluate their MRR@100 on the leaderboard and final sessions.

Results of individual recommendation algorithms used as retrieval methods and multi-stage recommenders utilizing different ranking methods on the testing datasets are presented in Table 4.18, which is visualized in Figure 4.19. It is clear that the multi-stage recommender systems outperform the retrieval models. The rankings of the retrieval models match the order on the validation dataset, with the exception being UserKNN which improved and performed even better than ItemKnn. The rankings of the multi-stage recommenders with different ranking approaches are equivalent to those on validation data. In summary, the best performance out of the retrieval models was achieved by the ensemble, which utilized multiple similarity functions, and based on them recommended the items most similar to the last item in a given session. Regarding the ranking algorithms, the pointwise approach performed better than the listwise approach. Furthermore, the best MRR result 0.1947 on the final and leaderboard datasets, was achieved by the ensemble that applied a weighted average of the rankings predicted by the pointwise and listwise GBDT rankers.

4.5 Discussion

One way to potentially further improve the performance of the Dressipi recommender system would be to focus on feature engineering and creating more features. It is worth noting that we used a total of 33 features compared to around 500 features used by the 5^{th} placed team in the Recsys Challenge. Additionally, experimenting with different sampling approaches for training the ranker models could be beneficial. Currently, we generate item candidates for testing sessions based on the retrieval stage, but exploring different sampling strategies for the items with labels of 0 could be worth investigating.

In addition, it would be interesting to explore the use of graph neural networks (GNN) for feature creation and compare them with existing cross-features based on collaborative filtering. Furthermore, another potential avenue for exploration is the use of neural networks for ranking instead of the applied gradient-boosted trees.

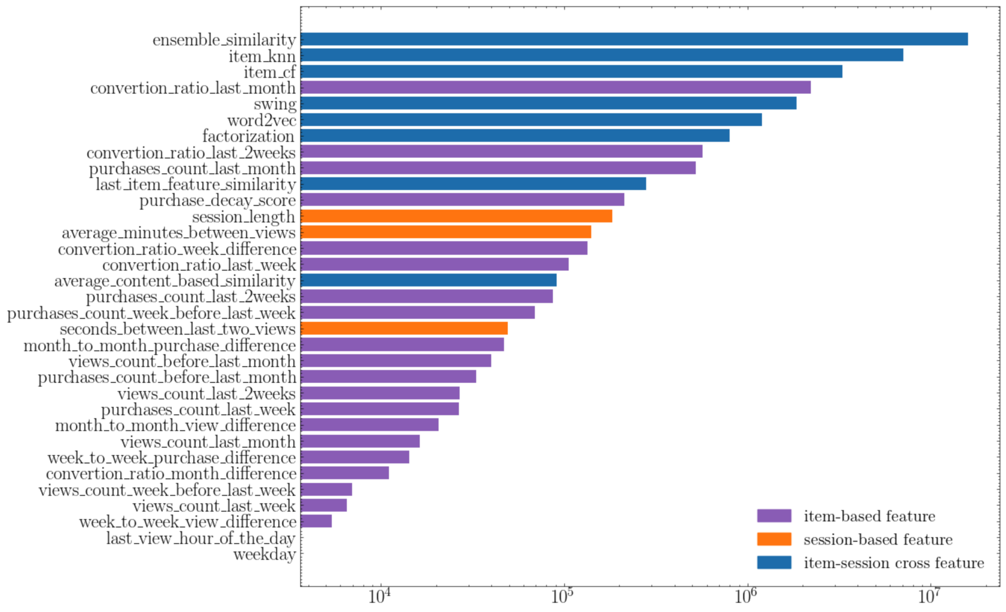


Figure 4.16: Gain-based feature importance on a logarithmic scale of a single pointwise LightGBM model

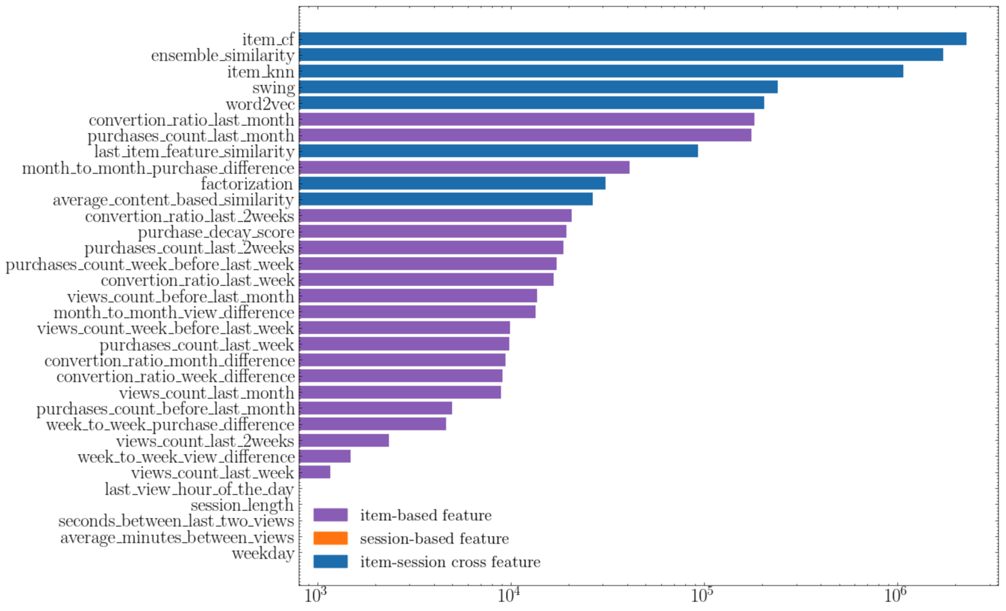


Figure 4.17: Gain-based feature importance on a logarithmic scale of a single listwise LightGBM model

Model name	MRR score leaderboard	MRR final
Rating similarity	0.1505	0.1504
Content-based	0.0786	0.0793
Popularity-based	0.0221	0.0214
ItemCF	0.1764	0.1758
Word2vec	0.1367	0.1391
Factorization	0.1077	0.1087
ItemKNN	0.1570	0.1585
UserKNN	0.1583	0.1611
Swing	0.1745	0.1730
Ensemble similarity	0.1831	0.1839
Multi-stage listwise	0.1915	0.1921
Multi-stage pointwise	0.1929	0.1926
Multi-stage ensemble	0.1947	0.1947

Figure 4.18: Final MRR@100 for different recommendation algorithms on testing datasets

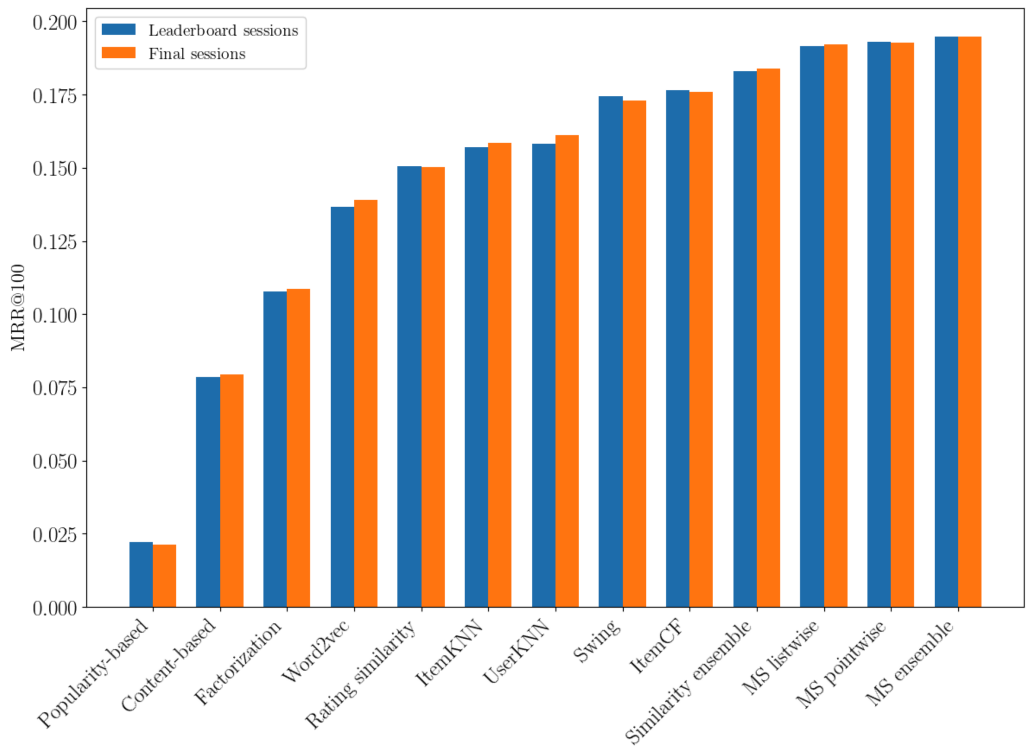


Figure 4.19: Vizualization of MRR@100 for different recommendation algorithms on leaderboard and final sessions

Conclusion

In this thesis, we have presented a comprehensive study of recommender systems, with a particular focus on multi-stage recommender systems. Our work aimed to provide a thorough understanding of the challenges and complexities involved in building such systems. We have examined the four-stage pipeline of the multi-stage recommender systems, and we dived into each of the stages to explain its responsibilities and motivation behind it. We have also provided an overview of various recommendation and retrieval algorithms, including state-of-the-art neural network architecture for retrieval. For the ranking stage, we introduced the concept of Learning to Rank and discussed the approaches used in this stage. We discuss in further detail the approach of employing gradient-boosted trees trained on feature vectors containing item, user, and cross features.

In the practical part of the thesis, we designed and implemented a multi-stage recommender tailored for the Dressipi dataset, using a range of retrieval models and LightGBM, the state-of-the-art implementation of gradient-boosted decision trees, for the ranking task. We carefully designed, implemented, and evaluated the retrieval models to achieve a high hit rate. On top of the retrieval stage, we trained and employed pointwise and listwise ranking algorithms to form the complete multi-stage recommender system. Moreover, we ensembled pointwise and listwise approaches using a weighted ranking average to achieve even better performance. We covered the importance of different kinds of features and evaluated all the algorithms on testing data. The results demonstrated that the multi-stage approach improves on using single recommendation algorithms or their ensembles.

Bibliography

[1] Simon Kemp. *Digital 2022: Global Overview Report*. May 2022. URL: https://datareportal.com/reports/digital-2022-global-overview-report.

[2] Chris Anderson. *The Long Tail*. Oct. 2004. URL: https://www.wired.com/2004/10/tail/.

[3] Alexandre Gilotte et al. "Offline A/B Testing for Recommender Systems". In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. WSDM '18. Marina Del Rey, CA, USA: Association for Computing Machinery, 2018, pp. 198–206. ISBN: 9781450355810. DOI: 10.1145/3159652.3159687. URL: https://doi.org/10.1145/3159652.3159687.

[4] Guy Shani and Asela Gunawardana. "Evaluating Recommendation Systems". In: *Recommender Systems Handbook*. Ed. by Francesco Ricci et al. Boston, MA: Springer US, 2011, pp. 257–297. ISBN: 978-0-387-85820-3. DOI: 10.1007/978-0-387-85820-3_8. URL: https://doi.org/10.1007/978-0-387-85820-3_8.

[5] Joeran Beel et al. "A Comparative Analysis of Offline and Online Evaluations and Discussion of Research Paper Recommender System Evaluation". In: *Proceedings of the International Workshop on Reproducibility and Replication in Recommender Systems Evaluation*. RepSys '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 7–14. ISBN: 9781450324656. DOI: 10.1145/2532508.2532511. URL: https://doi.org/10.1145/2532508.2532511.

[6] Zhe Wang et al. *COLD: Towards the Next Generation of Pre-Ranking System*. 2020. arXiv: 2007.16122 [cs.IR].

[7] J. Ben Schafer et al. "Collaborative Filtering Recommender Systems". In: *The Adaptive Web: Methods and Strategies of Web Personalization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, 291–324. ISBN: 978-

3-540-72079-9. DOI: 10.1007/978-3-540-72079-9_9. URL: https://doi.org/10.1007/978-3-540-72079-9_9.

[8] Kim Falk. *Practical recommender systems.* Manning Publications, 2019.

[9] Yehuda Koren, Robert M. Bell, and Chris Volinsky. "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42 (2009).

[10] Ramin Ebrahim Nakhli, Hadi Moradi, and Mohammad Amin Sadeghi. "Movie Recommender System Based on Percentage of View". In: *2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI).* 2019, pp. 656–660. DOI: 10.1109/KBEI.2019.8734976.

[11] Ziyou Yan. "Bandits for Recommender Systems". In: *eugeneyan.com* (May 2022). URL: https://eugeneyan.com/writing/bandits/.

[12] Nícollas Silva et al. "Multi-Armed Bandits in Recommendation Systems: A survey of the state-of-the-art and future directions". In: *Expert Systems with Applications* 197 (2022), p. 116669. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2022.116669. URL: https://www.sciencedirect.com/science/article/pii/S0957417422001543.

[13] Karl Higley et al. "Building and Deploying a Multi-Stage Recommender System with Merlin". In: *Proceedings of the 16th ACM Conference on Recommender Systems.* RecSys '22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 632–635. ISBN: 9781450392785. DOI: 10.1145/3523227.3551468. URL: https://doi.org/10.1145/3523227.3551468.

[14] Jake Brutlag. *Speed matters for google web search.* June 2009. URL: https://services.google.com/fh/files/blogs/google_delayexp.pdf.

[15] Paul Covington, Jay Adams, and Emre Sargin. "Deep Neural Networks for YouTube Recommendations". In: *Proceedings of the 10th ACM Conference on Recommender Systems.* RecSys '16. Boston, Massachusetts, USA: Association for Computing Machinery, 2016, pp. 191–198. ISBN: 9781450340359. DOI: 10.1145/2959100.2959190. URL: https://doi.org/10.1145/2959100.2959190.

[16] Christopher JC Burges. "From ranknet to lambdarank to lambdamart: An overview". In: *Learning* 11.23-581 (2010), p. 81.

[17] Mark Wilhelm et al. "Practical Diversified Recommendations on YouTube with Determinantal Point Processes". In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management.* CIKM '18. Torino, Italy: Association for Computing Machinery, 2018, pp. 2165–2173. ISBN: 9781450360142. DOI: 10.1145/3269206.3272018. URL: https://doi.org/10.1145/3269206.3272018.

[18] Pablo Castells, Neil J. Hurley, and Saul Vargas. “Novelty and Diversity in Recommender Systems”. In: *Recommender Systems Handbook*. Boston, MA: Springer US, 2015, 881–918. ISBN: 978-1-4899-7637-6. DOI: 10.1007/978-1-4899-7637-6_26. URL: https://doi.org/10.1007/978-1-4899-7637-6_26.

[19] G Geetha et al. “A Hybrid Approach using Collaborative filtering and Content based Filtering for Recommender System”. In: *Journal of Physics: Conference Series* 1000.1 (Apr. 2018), p. 012101. DOI: 10.1088/1742-6596/1000/1/012101. URL: https://dx.doi.org/10.1088/1742-6596/1000/1/012101.

[20] Tomáš Řehořek. “Manipulating the Capacity of Recommendation Models in Recall-Coverage Optimization.” PhD thesis. Czech Technical University in Prague, 2019. URL: https://dspace.cvut.cz/handle/10467/81823.

[21] Xiaoyong Yang et al. *Large Scale Product Graph Construction for Recommendation in E-commerce.* 2020. arXiv: 2010.05525 [cs.IR].

[22] Ziyou Yan. “Real-time Machine Learning For Recommendations”. In: *eugeneyan.com* (Jan. 2021). URL: https://eugeneyan.com/writing/real-time-recommendations/.

[23] Xinyang Yi et al., eds. *Sampling-Bias-Corrected Neural Modeling for Large Corpus Item Recommendations.* 2019.

[24] Xiangyang Li et al. *IntTower: the Next Generation of Two-Tower Model for Pre-Ranking System.* 2022. arXiv: 2210.09890 [cs.IR].

[25] Po-Sen Huang et al. “Learning Deep Structured Semantic Models for Web Search Using Clickthrough Data”. In: *Proceedings of the 22nd ACM International Conference on Information &amp; Knowledge Management.* CIKM ’13. San Francisco, California, USA: Association for Computing Machinery, 2013, pp. 2333–2338. ISBN: 9781450322638. DOI: 10.1145/2505515.2505665. URL: https://doi.org/10.1145/2505515.2505665.

[26] Alim Virani et al. *Lessons Learned Addressing Dataset Bias in Model-Based Candidate Generation at Twitter.* 2021. arXiv: 2105.09293 [cs.IR].

[27] Tie-Yan Liu. “Learning to Rank for Information Retrieval”. In: *Found. Trends Inf. Retr.* 3.3 (Mar. 2009), pp. 225–331. ISSN: 1554-0669. DOI: 10.1561/1500000016. URL: https://doi.org/10.1561/1500000016.

[28] David C. Liu et al. *Related Pins at Pinterest: The Evolution of a Real-World Recommender System.* 2017. arXiv: 1702.07969 [cs.IR].

[29] Heng-Tze Cheng et al. *Wide & Deep Learning for Recommender Systems.* 2016. arXiv: 1606.07792 [cs.LG].

[30] Ruoxi Wang et al. *Deep & Cross Network for Ad Click Predictions.* 2017. arXiv: 1708.05123 [cs.LG].

[31] Zzh, Wei Zhang, and Wentao. "Industrial Solution in Fashion-Domain Recommendation by an Efficient Pipeline Using GNN and Lightgbm". In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge '22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 45–49. isbn: 9781450398565. doi: 10.1145/3556702.3556850. url: https://doi.org/10.1145/3556702.3556850.

[32] Carlos García Ling et al. *H&M Personalized Fashion Recommendations.* 2022. url: https://kaggle.com/competitions/h-and-m-personalized-fashion-recommendations.

[33] Hamed Bonab et al. "Cross-Market Product Recommendation". In: *Proceedings of the 30th ACM International Conference on Information & Knowledge Management.* CIKM '21. Virtual Event, Queensland, Australia: Association for Computing Machinery, 2021, pp. 110–119. isbn: 9781450384469. doi: 10.1145/3459637.3482493. url: https://doi.org/10.1145/3459637.3482493.

[34] Olivier Chapelle and Yi Chang. "Yahoo! Learning to Rank Challenge Overview". In: *Proceedings of the 2010 International Conference on Yahoo! Learning to Rank Challenge - Volume 14.* YLRC'10. Haifa, Israel: JMLR.org, 2010, pp. 1–24.

[35] Chris Burges et al. "Learning to Rank Using Gradient Descent". In: *Proceedings of the 22nd International Conference on Machine Learning.* ICML '05. Bonn, Germany: Association for Computing Machinery, 2005, pp. 89–96. isbn: 1595931805. doi: 10.1145/1102351.1102363. url: https://doi.org/10.1145/1102351.1102363.

[36] Zhe Cao et al. "Learning to Rank: From Pairwise Approach to Listwise Approach". In: *Proceedings of the 24th International Conference on Machine Learning.* ICML '07. Corvalis, Oregon, USA: Association for Computing Machinery, 2007, pp. 129–136. isbn: 9781595937933. doi: 10.1145/1273496.1273513. url: https://doi.org/10.1145/1273496.1273513.

[37] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: Aug. 2016, pp. 785–794. doi: 10.1145/2939672.2939785.

[38] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems.* NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 3149–3157. isbn: 9781510860964.

[39] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. *CatBoost: gradient boosting with categorical features support.* 2018. arXiv: 1810.11363 [cs.LG].

[40] Dietmar Jannach, Gabriel Moreira, and Even Oldridge. “Why Are Deep Learning Models Not Consistently Winning Recommender Systems Competitions Yet?: A Position Paper”. In: Sept. 2020, pp. 44–49. DOI: 10.1145/3415959.3416001.

[41] Jerome H Friedman. “Greedy function approximation: a gradient boosting machine”. In: *Annals of statistics* (2001), pp. 1189–1232.

[42] Jerome H. Friedman. “Stochastic gradient boosting”. In: *Computational Statistics & Data Analysis* 38.4 (2002). Nonlinear Methods and Data Mining, pp. 367–378. ISSN: 0167-9473. DOI: https://doi.org/10.1016/S0167-9473(01)00065-2. URL: https://www.sciencedirect.com/science/article/pii/S0167947301000652.

[43] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. “Additive Logistic Regression: A Statistical View of Boosting”. In: *The Annals of Statistics* 28 (Apr. 2000), pp. 337–407. DOI: 10.1214/aos/1016218223.

[44] Tin Kam Ho. “The random subspace method for constructing decision forests”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.8 (1998), pp. 832–844. DOI: 10.1109/34.709601.

[45] Nick Landia et al. “RecSys Challenge 2022 Dataset: Dressipi 1M Fashion Sessions”. In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge ’22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 1–3. ISBN: 9781450398565. DOI: 10.1145/3556702.3556779. URL: https://doi.org/10.1145/3556702.3556779.

[46] Benedikt Schifferer et al. “A Diverse Models Ensemble for Fashion Session-Based Recommendation”. In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge ’22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 10–17. ISBN: 9781450398565. DOI: 10.1145/3556702.3556821. URL: https://doi.org/10.1145/3556702.3556821.

[47] Jiangwei Luo et al. “LightGBM Using Enhanced and De-Biased Item Representation for Better Session-Based Fashion Recommender Systems”. In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge ’22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 24–28. ISBN: 9781450398565. DOI: 10.1145/3556702.3556839. URL: https://doi.org/10.1145/3556702.3556839.

[48] Qi Zhang et al. “Fashion Recommendation with a Real Recommender System Flow”. In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge ’22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 4–9. ISBN: 9781450398565. DOI: 10.1145/3556702.3556792. URL: https://doi.org/10.1145/3556702.3556792.

[49] Pietro Maldini, Alessandro Sanvito, and Mattia Surricchio. "United We Stand, Divided We Fall: Leveraging Ensembles of Recommenders to Compete with Budget Constrained Resources". In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge '22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 34–38. ISBN: 9781450398565. DOI: 10.1145/3556702.3556845. URL: https://doi.org/10.1145/3556702.3556845.

[50] Yichao Lu et al. "Session-Based Recommendation with Transformers". In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge '22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 29–33. ISBN: 9781450398565. DOI: 10.1145/3556702.3556844. URL: https://doi.org/10.1145/3556702.3556844.

[51] Chendi Xue et al. "SIHG4SR: Side Information Heterogeneous Graph for Session Recommender". In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge '22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 55–63. ISBN: 9781450398565. DOI: 10.1145/3556702.3556852. URL: https://doi.org/10.1145/3556702.3556852.

[52] Costas Panagiotakis and Harris Papadakis. "Session-Based Recommendation by Combining Probabilistic Models and LSTM". In: *Proceedings of the Recommender Systems Challenge 2022*. RecSysChallenge '22. Seattle, WA, USA: Association for Computing Machinery, 2022, pp. 39–44. ISBN: 9781450398565. DOI: 10.1145/3556702.3556846. URL: https://doi.org/10.1145/3556702.3556846.

[53] Guorui Zhou et al. *Deep Interest Network for Click-Through Rate Prediction*. 2018. arXiv: 1706.06978 [stat.ML].

[54] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[55] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[56] Mukund Deshpande and George Karypis. "Item-Based Top-N Recommendation Algorithms". In: *ACM Trans. Inf. Syst.* 22.1 (Jan. 2004), pp. 143–177. ISSN: 1046-8188. DOI: 10.1145/963770.963776. URL: https://doi.org/10.1145/963770.963776.

[57] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[58] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[59] Radim Řehůřek and Petr Sojka. “Software Framework for Topic Modelling with Large Corpora”. English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* `http://is.muni.cz/publication/884893/en`. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[60] Ben Frederickson. *Implicit: Fast Collaborative Filtering for Implicit Feedback Datasets.* `https://github.com/benfred/implicit`. 2018.

[61] Takuya Akiba et al. “Optuna: A Next-Generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp; Data Mining.* KDD ’19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 2623–2631. ISBN: 9781450362016. DOI: `10.1145/3292500.3330701`. URL: `https://doi.org/10.1145/3292500.3330701`.

Appendix **A**

Acronyms

ANN Approximate nearest neighbors

CCR Click-through rate

CF Collaborative filtering

CTR Click-through rate

DIN Deep interest network

DNN Deep neural network

ERR Expected reciprocal rank

GBDT Gradient boosted decision trees

GNN Graph neural network

KNN K-nearest neighbors

KPI Key performance indicator

LSTM Long short-term memory

MPL Multi-layer perceptron

MRR Mean reciprocal rank

MS Multi-stage

NDCG Normalized discounted cumulative gain

QPS Queries per second

RT Return time

SOTA State-of-the-art

Appendix B

Contents of enclosed CD

```
│ readme.md ........................... description of the project structure
├─ multistage_recommender.ipynb ........ notebook with the experiments
└─ visualizations.ipynb.................... notebook with visualizations
```