Master Computer Science

A Multi-Stage Framework for Strict Item Cold-Start Recommendation with LLM-based Weak Supervision

Name: Cong Ning
Student ID: s2303299

Date: 25/03/2026

Specialisation: Data Science

1st supervisor: Zhaochun Ren
2nd supervisor: Qinyu Chen
Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Under a strict item cold-start setting, traditional ID-based sequential recommendation models usually perform well on warm items, but they often become almost ineffective for cold-start scenarios because they cannot generalize well to unseen products. To address this problem, this work proposes an offline recommendation framework that combines content-based retrieval, hierarchical structure constraints, LLM-based weak supervision, and sequential re-ranking.

The framework begins with a Two-Tower model that learns user and item representations in a shared semantic space. This gives cold items a content-driven entry point into the recall stage, so they can still appear in the candidate set even when they have no historical interactions. On top of this, a taxonomy-based hierarchical candidate space is built, and a Stage 1–5 process is designed so that the large language model can work in a more controlled way. In these stages, the model is used to summarize user interests, choose categories, filter subcategories, select items within subcategories, and finally perform a global re-ranking step with some diversity. The outputs of this process are not used only as direct ranking results. They are further transformed into pointwise pseudo edges and pairwise triplets, which are then injected into SASRec as weak supervision through multi-objective training. In this way, the semantic judgments made by the LLM are distilled into a ranking model that can generalize more effectively.

Experiments on the Amazon Beauty and Health datasets show that the proposed method can clearly improve cold-item recommendation. Using E0 and two models as the baselines, E3 (see Table5.1, 5.2) increases Recall@20 on Beauty-cold from 0.0029 to 0.0131, and on Health cold result from 0.0091 to 0.0216. It also achieves the best, or nearly the best, results on most cold NDCG metrics. Compared with E3, E5 delivers a more moderate cold-item improvement but preserves warm-item performance more effectively, indicating a better trade-off between cold-start enhancement and overall recommendation quality

These results indicate that combining content-based recall, hierarchical candidate filtering, and LLM-based weak supervision can substantially improve strict item cold-start recommendation.

Contents

Chapter 1

Background

1.1 Traditional Recommender System

The goal of a recommender system is to select items that a user is likely to prefer from a large item pool. In practice, the system uses user behavior and item information to estimate the matching score between a user and an item, and then recommends the top-ranked results[1].

A traditional recommender system usually includes several steps. Firstly, the system collects user-item interactions, such as clicks, views, purchases, ratings, and dwell time. It may also use user profile features and item features, such as category, brand, price, and text description. Secondly, traditional methods mainly include content-based recommendation and collaborative filtering. Content-based methods recommend items similar to those the user liked before. Collaborative filtering uses interaction patterns across users and items to discover potential preferences[2]. After recommendation, new user behaviors are collected and used to update the system continuously. Although traditional methods are effective, they often depend heavily on historical interactions. As a result, they may struggle with cold-start cases, complex context, and fine-grained preference modeling[3].

1.2 Modern Recommendation Model

With the development of deep learning, recommender systems have gradually moved away from methods that mainly depend on shallow features and simple similarity rules. Modern models focus more on learning user/item representations, and more complex relationships between them in an end-to-end way[4]. In other words, rather than relying only on fixed rules or simple statistics, modern recommender systems use neural networks to learn these patterns automatically[5].

In terms of system design, modern recommender systems still usually follow a multi-stage pipeline, but each stage has become powerful. In the recall stage, a common method is the Two-Tower model or other vector-based retrieval methods[6]. These models learn one vector for the user and another for the item, and then use the similarity between the two vectors to quickly find candidate items from a very large item pool. Because this turns large-scale matching into vector search, it works well in real-world systems that need fast retrieval.

In the ranking stage, modern models pay more attention to changes in user interest over time and more complex feature interactions. One direction is sequential recommendation. These models do not only look at what a user liked before, but also at the order of the user's actions, so they can better understand how interest changes over time. GRU4Rec, SASRec, and BERT4Rec are all examples of this line of work[7, 8, 9], although they model sequences in different ways. Another direction is

graph-based recommendation. In this setting, user-item interactions are treated as a graph, and the model updates user and item representations by using information from connected neighbors. This graph-based mechanism enables the system to exploit collaborative signals effectively, and LightGCN is a representative example[10].

Modern recommender systems also often combine different kinds of features and even different kinds of data. In e-commerce, for example, an item may have not only an ID, but also a title, description, category, brand, image, and review text. Deep models can bring these signals into a shared representation space, so the system can still make useful matches even when behavior data is limited. This is particularly valuable for cold-start items, whose behavioral evidence is often limited.

Modern recommender systems are typically built as a retrieval–ranking pipeline: an efficient retriever narrows down a candidate set, and a stronger model performs fine-grained ranking (Fig. 1.1). Compared with traditional methods, modern models have stronger representation ability, can make better use of sequential, graph, and text information, and are more suitable for large-scale systems with vector retrieval and multi-stage pipelines.

However, modern deep recommendation models do not solve every problem. They still depend a lot on the training data they have seen. When there are very few interactions for new items, when data comes from a new domain, or when the task needs more explicit reasoning, these models may still be limited. This is one reason why large language models have started to attract attention in recommender systems.

1.3 Modern Recommender System with Large Language Model

Building on the limitations of deep recommenders under data sparsity, large language models (LLMs) provide an additional semantic layer for recommendation.

Unlike traditional models that primarily rely on interaction-derived signals, LLMs can directly interpret item text, category information, and user-intent descriptions, which is particularly useful in cold-start scenarios where behavioral supervision is scarce[11]. More importantly, the LLM converts item titles, descriptions, reviews, and user histories into richer semantic representations for downstream models. For example, an original product title may be short and incomplete, but an LLM can rewrite or summarize it into a clearer semantic description. This gives later models, such as embedding models, Two-Tower models, or ranking models, better input. The second way is as a candidate filtering or re-ranking module. In this process, the base recall system first provides a set of candidate items, and then the LLM further filters, explains, or ranks them based on user history, candidate descriptions, and task instructions. Compared with traditional ranking models, the advantage of an LLM is that it does not only rely on numerical features. It can also use natural language understanding to analyze why an item may suit a user. This capability is particularly useful for modeling complex preferences, distinguishing between semantically similar items with different functions, and supporting recommendation tasks that require explicit reasoning.. The third way is as a knowledge generation module for cold start. This has become an important direction in recent years. When a new item has no interaction data, the system can ask the LLM to generate useful signals from the item text, attributes, and category information[11]. These signals may include preference judgments, pairwise comparisons, possible reasons why users may like the item, similarity relations with other items, or even

pseudo interaction links. In this way, cold items that originally have no behavior data can still be brought into later training through soft labels or pseudo labels generated by the LLM. In simple terms, the LLM helps make up for missing behavior data.

From the system point of view, an LLM-based recommender system usually follows several steps. The system generates an initial candidate set. This stage is still usually handled by traditional recommender models, such as Two-Tower models, text embedding retrieval, or collaborative filtering recall. This is mainly because direct full-corpus LLM reasoning is computationally prohibitive, so efficient models are used to narrow down the candidate space first[6].

Next comes structured constraint or semantic narrowing. If the system has a category tree, taxonomy, leaf clusters, or other knowledge structure, these can be used to further narrow the candidate range[12]. For example, the model may first decide which broad category the user is more likely to prefer, then choose a subcategory, and finally select items only within that subcategory. This both reduces the number of candidates the LLM needs to handle and makes the process closer to the layered decision logic of real recommender systems.

Then comes the LLM generation or reasoning stage. At this point, the LLM can do different tasks, such as summarizing user preferences, choosing category paths, comparing candidate items, producing ranked results, giving recommendation reasons, or generating pseudo supervised samples[11]. Unlike traditional models, the input here is usually not only item IDs or sparse features, but also text-based user history, semantic item information, and explicit task instructions. So this stage is not just about scoring items, but like carrying out a guided reasoning process.

The last step is training feedback or result fusion. The output of the LLM can be used directly as the final ranking result, or it can be turned into training signals to improve traditional recommender models. For example, if the LLM says that a user is likely to prefer item A than item B, this can be converted into a pairwise triplet. If it suggests that a new item may fit a certain group of users, this can be turned into pseudo edges or pseudo labels. Multi-stage filtering results from the LLM can also be used as supervision for re-ranking or as targets for knowledge distillation. In this sense, the LLM is not only an online generator, but also a source of knowledge in the offline training pipeline.

Bringing LLMs into recommender systems does not mean fully replacing traditional recommendation models. A practical approach is to combine the two. Traditional models are still better for efficient recall and scalable training, while LLMs are more useful for semantic understanding, cold-start support, and complex decision making. This kind of combination is especially suitable for the setting discussed in this work, because cold-start recommendation needs to deal with both limited interaction data and limited semantic reasoning at the same time.

1.4 Evaluation Metrics

To evaluate the effectiveness of a recommender system, a set of metrics is needed to reflect different goals. Unlike standard classification tasks, recommendation focuses more on whether relevant items appear at the top of the ranked list. Therefore, most commonly used metrics are ranking-aware.

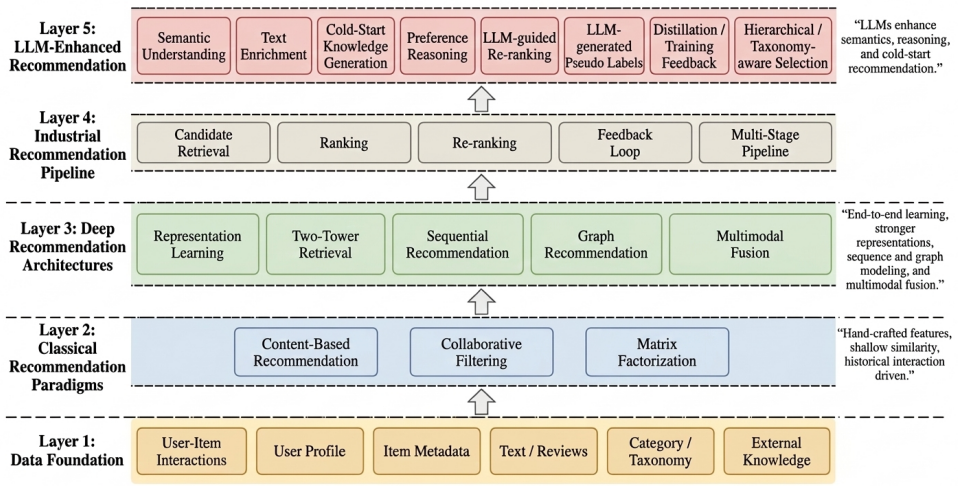

FIGURE 1.1: Overview of the evolution from traditional recommendation to modern deep and LLM-enhanced recommender systems

1.4.1 Recall@K

Recall@K measures whether the ground-truth item appears in the top-K recommended list, and is widely used to evaluate final ranking quality under implicit feedback settings[1]. For multi-stage recommender systems, it is also important to separately evaluate the candidate generation stage. Therefore, we additionally report *candidate recall* at the candidate cutoff M (e.g., Recall@M), which measures whether the ground-truth item is included in the retrieved candidate set before re-ranking[6]. This distinction makes it possible to determine whether performance bottlenecks arise from insufficient retrieval coverage or from the downstream ranking module.

For a user u, Recall@K is defined as:

$$\text{Recall@K}(u) = \frac{\left|R_u^K \cap G_u\right|}{|G_u|}, \tag{1.1}$$

where R_u^K denotes the top-K recommended items for user u, and G_u denotes the set of ground-truth relevant items.

In practice, if $|G_u| = 0$, Recall@K is usually defined as 0 for that user. A higher Recall@K indicates that the system is more likely to retrieve the correct items into the candidate set. In a multi-stage recommender system, this metric is particularly important, since the later ranking model cannot recover an item that was missed in the recall stage.

1.4.2 NDCG@K

NDCG@K is a widely used ranking metric in recommender systems. Unlike Recall@K, it not only considers whether the correct item appears in the recommendation list, but also whether it is ranked near the top.

First, the Discounted Cumulative Gain at K is defined as

$$\text{DCG@K}(u) = \sum_{i=1}^{K} \frac{rel_i}{\log_2(i+1)}, \tag{1.2}$$

where rel_i indicates the relevance of the item at position i. In implicit feedback settings, rel_i is usually set to 1 for relevant items and 0 otherwise.

Then NDCG@K is obtained by normalizing DCG@K with the ideal DCG:

$$\text{NDCG@K}(u) = \frac{\text{DCG@K}(u)}{\text{IDCG@K}(u)}. \tag{1.3}$$

.

NDCG@K usually ranges from 0 to 1. A higher value indicates that relevant items are ranked closer to the top, making it particularly suitable for evaluating ranking quality.

1.4.3 MRR

MRR, or Mean Reciprocal Rank, focuses on the position of the first relevant item in the ranked list. If the first correct item appears earlier, the score is higher.

For a user u_i let $rank_u$ denote the position of the first relevant item. The reciprocal rank is:

$$\text{RR}(u) = \frac{1}{rank_u}. \tag{1.4}$$

The mean reciprocal rank over all users is:

$$\text{MRR} = \frac{1}{|U|} \sum_{u \in U} \frac{1}{rank_u}. \tag{1.5}$$

where U is the set of users.

MRR is useful when the system is expected to present a correct result as early as possible.

1.4.4 Offline Evaluation and Real-System Objectives

Offline metrics can only approximate real-world recommendation performance. Metrics such as Recall@K, NDCG@K, and MRR mainly measure whether the model can better retrieve and rank historically interacted items. In real systems, however, recommendation quality is also affected by exposure bias, position bias, user context changes, and exploration strategies.

Even so, offline evaluation remains the most appropriate choice in this work. The main goal here is to study how model design and training strategies improve cold-start recommendation, rather than to optimize online feedback in a deployed system. For this reason, this project adopts standard offline ranking metrics and further separates cold and warm item performance to ensure fair and interpretable evaluation.

Chapter 2

Related work

2.1 Cold-start Recommendation with Content-based Retrieval

Cold-start recommendation has long been one of the main challenges in recommender systems, especially on platforms where new items are added all the time. For a new item, there is usually little or no historical interaction data, so it is hard for the system to learn a reliable representation from collaborative signals alone. Many studies can use item-side content information to ease the problem. Common choices include titles, descriptions, categories, brands, attributes, and even review text[3]. These signals can be used to build item representations and support the first stage of retrieval through content similarity or semantic search.

Compared with methods that only depend on item IDs or interaction matrices, content-based retrieval is more natural for item cold start. Even if an item has just been released, it usually still has some text and structured attributes that the system can use. With the development of representation learning, this line of work has gradually moved from simple keyword matching to embedding-based retrieval. In this setting, the model maps user preferences and item text into the same vector space, and candidate items are retrieved through similarity search. This idea is useful since it is both friendly to cold-start items and efficient for large-scale retrieval, so it has become a common solution in modern recommender systems.

However, content-based cold-start methods still have clear limits. Content features allow the model to characterize what an item is, but they do not always indicate which users the item is most suitable for.. In addition, many methods simply encode text into vectors and use them for retrieval, so the level of semantic understanding is still limited, especially for fine-grained preference matching. Moreover, content retrieval usually contributes only to candidate generation. For later steps such as filtering, explanation, and stronger ranking for cold items, the system often needs a more powerful semantic modeling component. For this reason, content-based retrieval is better seen as a strong starting point for cold-start recommendation, rather than a complete solution.

2.2 LLM-Enhanced Recommender Systems

With the rapid development of large language models, more and more studies have started to bring LLMs into recommender systems. The main reason is that they address problems that traditional methods often struggle with, such as interpreting item semantics, handling cold-start cases, and modeling user preferences more effectively. Compared with standard recommendation models, LLMs have a clear

advantage: they can directly work with natural language, including product information, user history, and task instructions. So, they can still provide useful semantic support even when interaction data is limited.

2.2.1 Data Augmentation for Recommendation

An important line of work treats LLMs as data enhancers for recommender systems. The main idea is simple: when real user-item interactions are limited, especially when cold items have very little supervision, LLMs can generate extra training signals from existing text, category information, and context. For example, they can produce pseudo preference pairs, pseudo click signals, pairwise preferences between candidates, or explanations of why a user may prefer one item over another. These outputs can then be turned into pointwise or pairwise supervision for downstream recommendation models[13].

The advantage of this idea is that the LLM does not need to handle the whole recommendation process by itself. It uses its semantic understanding to fill in the part that is most missing in the training data, which is especially useful for cold-item recommendation. For a new item, there may be no real interaction data yet, but there is usually still useful text, such as the title, description, reviews, and category. Based on this information, the LLM can infer what kind of users may like the item and how it may compare with other items. Compared with relying only on sparse behavior data, this gives cold items a richer source of weak supervision[11].

However, this direction also has clear limits. First, pseudo labels generated by an LLM are not the same as real user feedback, so they may contain noise and bias. Second, the quality of generated samples depends a lot on prompt design and candidate range control. If the context is too broad or the task is not well constrained, the output may become unstable. Third, many data augmentation methods focus more on what signals are generated, and less on how these signals can be combined with recall, ranking, and structural constraints in a full recommendation pipeline[6]. An open question is how to place LLM-generated supervision into a complete multi-stage recommender system.

2.2.2 User/Interaction Simulation for Cold Items

Another line of work goes a step further by using LLMs not only to generate static labels, but also to simulate user interests or interaction behavior. These methods usually summarize a user's past behavior into a preference profile, then combine it with the semantic description of a cold item to infer whether the user may be interested in it, or to simulate how the user would choose among several candidates[14]. Compared with simple label generation, this is closer to reconstructing behavior itself, so it is often more suitable for cold-start settings.

The value of this direction is that it lets the LLM act more like a user preference estimator or an interaction simulator. For cold items, the biggest problem is often not that the item has no information, but that the system does not know which users may like it. By simulating the matching process between users and items, the LLM can partly fill this gap and turn unobserved preference relations into pseudo interactions, preference edges, or ranking constraints[11]. This allows cold items to enter the training and retrieval process earlier.

This direction also introduces additional uncertainty. Once the LLM moves from text enhancement to behavior simulation, its output is no longer just descriptive support. It starts to directly affect the preference structure learned by the recommender

system. If the simulation is not properly constrained, it may introduce systematic bias. Also, many existing methods still use the LLM as a fairly separate generation module, without making full use of the strengths of traditional recommender systems in candidate control, retrieval efficiency, and structured filtering. In real recommendation tasks, a more practical choice is often to let the LLM make judgments or simulations under a limited candidate set and clear structural constraints, rather than asking it to perform recommendation on its own.

2.2.3 Hierarchical Retrieval

Besides data augmentation and interaction simulation, another related direction is to combine LLMs with hierarchical retrieval or layered filtering. In real systems, items often have a clear category structure, from broad categories to subcategories and then to fine-grained leaf nodes. For recommendation, this structure organizes the candidate space and provides the LLM with a clearer reasoning path. Instead of asking the model to judge directly over a very large candidate pool, it is often more natural to first decide which broad category the user may prefer, then narrow down to a subcategory, and finally make a choice within a much smaller local set. This better matches how real recommender systems usually work[15].

Hierarchical retrieval is useful in two main ways. First, it reduces the search space for later retrieval and re-ranking, which makes LLM reasoning more focused and stable. Second, for cold items, category information is often one of the earliest available signals, so hierarchical retrieval can place a new item into a reasonable semantic region even before it has enough behavior data. This suggests that LLM reasoning is often effective when it works together with structured retrieval, rather than being used alone as a black-box ranker.

Still, in much of the existing work, hierarchical retrieval and LLM reasoning are often studied separately. The former focuses more on retrieval structure and efficiency, while the latter focuses on generation and semantic judgment. Work that combines them in a systematic way for cold-item recommendation is still relatively limited. This is also where the present work is positioned: the hierarchical structure is used not only for candidate organization, but also as part of the LLM's multi-stage reasoning and cold-start enhancement process.

2.3 Positioning of This Project

Existing studies have provided useful ideas for cold-start recommendation from several directions, including content-based retrieval, LLM-based data augmentation, user or interaction simulation, hierarchical retrieval, and multi-stage recommender systems. Content-based retrieval brings cold items into the initial candidate set, LLM-based augmentation and simulation add semantic supervision and pseudo behavioral signals, and hierarchical retrieval provides structural guidance for candidate organization and multi-stage reasoning. Multi-stage recommendation frameworks further suggest that these components are most effective when integrated into a unified pipeline of retrieval, filtering, and ranking

The present work is built on this observation. Unlike cold-start methods that rely only on content retrieval, this work does not treat content-based recall as the final goal. Rather, content-based recall serves as the starting point for subsequent semantic filtering and cold-start enhancement. Unlike methods that use LLMs only as black-box rankers, this work finds how LLMs can summarize user preferences,

judge candidates, and generate pseudo supervision under a limited candidate set and clear hierarchical constraints. Unlike approaches that use taxonomy only for static grouping, this work further allows the hierarchical structure to take part in multi-stage candidate control and decision making.

In general, this work aims to bring together content-based retrieval, hierarchical retrieval, and LLM-based augmentation or simulation within one offline multi-stage recommendation framework, so as to improve cold-item recommendation in a more targeted way.

Chapter 3

Architecture and Data processing

3.1 System Architecture

This work presents an offline multi-stage framework for cold-item recommendation. Rather than relying on a single model, it combines semantic retrieval, sequential re-ranking, hierarchical structure, and LLM-based enhancement in one system. The main idea is to keep the strong retrieval ability of traditional recommender models, while improving how cold items are understood, filtered, and ranked.

At the retrieval stage, the framework uses a Two-Tower model as the backbone. It maps users and items into the same vector space and retrieves candidate items through vector similarity. This part is responsible for searching over the full item pool and quickly finding a manageable set of candidates that may match the user's interests.

On top of that, SASRec is used to re-rank the retrieved items by modeling the user's behavior sequence. SASRec goes beyond static matching by modeling the order of past user actions and capturing how user interests evolve over time. In this way, the model can make fine-grained ranking decisions within the candidate set.

The framework also includes a taxonomy or tree module to provide hierarchical organization and local constraints. Here, category information is not treated as just another feature. Items are grouped into leaf nodes or subcategories, and this structure is used to narrow the candidate space, organize local candidates, and give later filtering and comparison steps a clearer semantic range. This makes candidate organization efficient and also helps cold items appear in a suitable semantic region.

To deal with the lack of supervision for cold items, the system further introduces an LLM-based enhancement module. The LLM is not used to replace the recommender model directly. It works as a supporting component for summarizing user preferences, judging cold-item relevance, and generating pseudo supervision signals. These outputs are then converted into pointwise pseudo edges and pairwise triplets, which are fed back into model training and provide extra semantic supervision for cold items. The system also includes a Stage 1–5 hierarchical LLM process, where the model makes recommendation decisions step by step through category selection, subcategory filtering, local candidate choice, and final global re-ranking. This makes the reasoning process closer to the actual structure of a multi-stage recommender system.

Each module in the framework has a clear role. Two-Tower handles retrieval, SASRec handles re-ranking, the tree structure controls the candidate space, and the LLM strengthens cold-start modeling. These parts are not used separately. They work together around the same goal of improving cold-item recommendation: the retrieval model provides the candidate pool, the hierarchical structure reduces the search range, the LLM adds semantic supervision, and the sequential model learns from these signals to produce better final ranking results. In this sense, the whole

system can be seen as a unified cold-start recommendation framework that brings together efficient retrieval, structural control, and semantic enhancement.

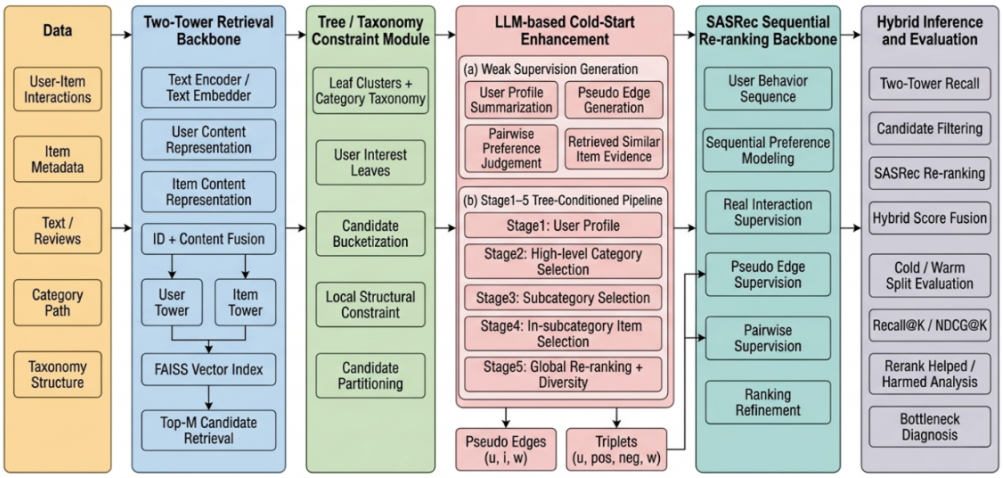

FIGURE 3.1: System architecture of the recommendation framework

3.2 Data Processing and Analysis

The data used in this project comes from the Amazon 5-core review data and meta data released by UCSD. In the Beauty subset, the 5-core datas contains 198,502 reviews, while the Health and Personal Care subset contains 346,355 reviews. The processing starts by reading the review file line by line and keeping the reviewer ID, item ID, review time, and rating. The original multi-level Amazon categories are then compressed into a single longest category path. The metadata file is then parsed to obtain item-side textual and structural information. For each product, the system keeps the title, a shortened description, and a normalized category path derived from the longest available category chain. Text fields are lightly cleaned by removing line breaks, redundant spaces, and other formatting noise, so that the resulting item representations can be directly reused in text encoding, prompt construction, and taxonomy building.

After the raw interactions and metadata are loaded, the pipeline performs a metadata-aware filtering step, where only interactions associated with items that can be found in the metadata table are retained. This design ensures that every item kept in the final dataset has both behavioral information and semantic side information. The filtered interaction table is then globally sorted by timestamp and split into training and test segments using a global time-based 70/30 split. Rather than directly treating all ratings as equivalent feedback, the pipeline further converts explicit ratings into implicit preference signals: ratings greater than or equal to the positive threshold are treated as strong positives, while 3-star ratings can optionally be retained as weak positives with a lower weight, and ratings below that threshold are not converted into explicit negative supervision. To be more specific, ratings of 4 or 5 are treated as strong positive feedback with weight 1.0, while rating 3 is treated as weak positive feedback with weight 0.3. Ratings below 3 are not used as positive training or evaluation signals. As a result, the final recommendation task is formulated as a temporally ordered implicit-feedback problem with weighted positive signals. This makes the processed data suitable for retrieval models, sequential

recommendation, and pairwise preference learning. In this work, a cold item is defined as an item that appears in the test set but never appears in the training set. A cold user is defined in the same way, meaning that the user appears in the test set but not in the training set.

Based on the split results, the pipeline then constructs a set of processed files that support the entire offline framework. It builds a shared user–item index space through mappings.json, generates user training sequences in chronological order for sequential models, creates item text files in the compact "title | category path" format for later prompting and candidate presentation, and produces a taxonomy.json file that organizes products into stage-level categories, subcategories, and item-to-subcategory mappings. In addition, the same processed outputs are later reused by the retrieval backbone, the taxonomy-aware tree module, and the Stage1–5 LLM pipeline, which means that the preprocessing stage is not only a data cleaning step but also the foundation of the entire system design.

TABLE 3.1: Data processing statistics of the Beauty subset

Metric	**Value**
Effective interactions (final)	198,502
Training interactions (time split)	138,951
Test interactions (time split)	59,551
Training positives / weak positives	123,498
Test positives / weak positives	53,022
Cold items	651
Cold users	2,722
Cold-item ratio in unique test items	6.37%
Users in training split	19,641
Users in test split	13,400
Users appearing in both splits	10,678

From the initial processing results, the Beauty dataset keeps 198,502 valid interactions in total, including 138,951 in the training period and 59,551 in the test period. After mapping ratings into positive feedback signals, the training set contains 123,498 positive or weakly positive samples, and the test set contains 53,022. In this dataset, 651 items are identified as cold items and 2,722 users are identified as cold users. Cold items account for about 6.37% of the unique items in the test set. The training set includes 19,641 users, while the test set includes 13,400 users, and 10,678 of them appear in both parts. This shows that the dataset covers not only the standard next-item setting, but also a group of users who do not appear in training. On average, each user has about 7.07 interactions in training and 4.44 interactions in testing.

The Health and Personal Care dataset is larger. It keeps 345,725 valid interactions, with 242,007 in training and 103,718 in testing. After converting ratings into positive feedback signals, the training set contains 217,480 samples and the test set contains 95,000. In this dataset, 926 items are identified as cold items and 3,740 users are identified as cold users. Cold items make up about 5.92% of the unique items in the test set. The user scale is also much larger, with 34,869 users in training and 25,425 in testing, among which 21,685 appear in both sets. In terms of average activity, each user has about 6.94 interactions in training and 4.08 in testing. Compared with Beauty, the Health dataset is larger in users, items, and total interactions, and is closer to a broader and sparser e-commerce recommendation setting.

TABLE 3.2: Data processing statistics of the Health and Personal Care subset

Metric	**Value**
Effective interactions (final)	345,725
Training interactions (time split)	242,007
Test interactions (time split)	103,718
Training positives / weak positives	217,480
Test positives / weak positives	95,000
Cold items	926
Cold users	3,740
Cold-item ratio in unique test items	5.92%
Users in training split	34,869
Users in test split	25,425
Users appearing in both splits	21,685

Chapter 4

Methodology

4.1 Semantic Retrieval Backbone

The framework uses a Two-Tower model as the retrieval backbone to quickly generate a candidate set for each user from a large item pool. The input representation, training objective, and treatment of cold items are extended in a targeted way, so that the model can support not only standard retrieval but also more stable semantic matching for cold-item recommendation.

As mentioned earlier, the input text comes from the short item text built in the preprocessing stage, where each item is represented by its title and category path. A pretrained text encoder, MiniLM, is used to map each item into a dense semantic vector. During encoding, the batch size and maximum truncation length are controlled separately. For an item i, suppose the encoder produces token-level hidden representations $\{h_1, \ldots, h_T\}$, with the corresponding attention mask $\{m_1, \ldots, m_T\}$. The item content vector is obtained by masked mean pooling:

$$\mathbf{c}_i = \text{norm}\left(\frac{\sum_{t=1}^{T} m_t \, \mathbf{h}_t}{\sum_{t=1}^{T} m_t}\right), \tag{4.1}$$

Here, $norm(\dot{})$ denotes L_2 normalization. The resulting c_i forms the content representation of item i, and all such vectors together make up the item content matrix, which is used as an valuable input to the item tower.

Once the item content vectors are prepared, the model further builds a user content representation from the user's historical interaction sequence. The idea is to use the content vectors of recently interacted items to form a lightweight semantic summary of the user's current interest. Let H_u denote the recent history of user u. The user content representation is defined as

$$\mathbf{c}_u = \text{norm}\left(\frac{1}{|H_u|} \sum_{i \in H_u} \mathbf{c}_i\right). \tag{4.2}$$

In the implementation, at most the recent 50 interacted items are kept for each user. Their content vectors are averaged and normalized to produce the user content matrix. This representation is then used together with the user ID embedding as the input to the user tower.

During representation learning, the model encodes users and items separately. The architecture includes user ID embeddings, item ID embeddings, and linear projection layers that map content vectors into a shared hidden space. The goal at this stage is to learn a common semantic space in which users are closer to the items they prefer. The Two-Tower model is trained for 50 epochs with the learning rate of 3×10^{-4}. The training data consists of positive feedback samples (u, i, w), where

w comes from the interaction weights defined in preprocessing. This means that strong positive samples and weak positive samples contribute to training with different strengths.

For a batch with B positive samples, the model computes the user representation matrix and the item representation matrix, and then builds a score matrix. During training, the diagonal entries are treated as positive pairs, while the other items in the batch are treated as negatives. The model is optimized with an in-batch negative cross-entropy loss:

$$\mathcal{L}_{\text{ce}} = \frac{1}{B} \sum_{b=1}^{B} w_b \cdot \text{CE}(S_b, y_b), \tag{4.3}$$

After training, the system does not only keep the Two-Tower parameters. It also exports all user vectors and item vectors so that later modules can reuse them directly, and saves them as *.npy* files. In addition, it stores metadata such as the number of users, the number of items, the vector dimension, and the data split setting, which makes it easier to reuse the vectors across experiments.

One design choice at this stage is especially important for cold items. For items that never appear in training, the item ID embedding part is set to zero when the item vector is exported, and only the content projection part is kept. For a cold item i, the exported vector can be approximately written as:

$$\mathbf{z}_i^{\text{cold}} \approx \text{norm}(W_i \mathbf{c}_i). \tag{4.4}$$

This avoids random and untrained item ID embeddings affecting cold-item retrieval, so the recall of cold items depends more on their text-based semantic representation.

The exported Two-Tower vectors are reused throughout the whole framework. The item vectors are first used to build a FAISS inner-product index, and the user vectors are then used to retrieve top-M candidates. This produces the first-stage candidate set of the whole system. The same vectors are also used to build user candidates and provide RAG evidence. In this sense, the Two-Tower module serves as the shared semantic retrieval layer and the base candidate space of the framework. The later tree module, Stage 1–5 label generation, and SASRec re-ranking are all built on top of its output.

4.2 Tree/Taxonomy-Constrained Candidate Organization

Once the Two-Tower model has built a shared semantic vector space and exported user and item embeddings, the tree or taxonomy module further organizes the item space into smaller local regions and passes this structure to the later stages. As a result, the following modules no longer work over the full flat item pool. On the contrary, they operate in a candidate space that is constrained, routable, and easier to filter locally.

One part of this structural layer is an explicit taxonomy built from the item category path. In preprocessing, the original multi-level Amazon categories are already compressed into one longest category path, so each item comes with a normalized hierarchical category string. The tree module first splits this path into levels, and then builds mappings between higher-level categories and subcategories based on a chosen depth range. In practice, the first two levels are used to form the stage2 category, while the subcategory is usually defined from the third to the fourth level.

This gives four main structures: the full set of higher-level categories, the mapping from each higher-level category to its subcategories, the set of items under each subcategory, and the reverse mapping from each item to its subcategory. Together, these structures form the explicit taxonomy used later in Stage 1–5.

Besides this explicit taxonomy, the system also builds a set of leaf clusters to create finer local buckets for items. Items with the same category path are first grouped together into the same coarse category group. For each group, if the number of items is no larger than the preset max leaf size, which is currently 200, the group is directly treated as one leaf node. When a group is larger than this threshold, the system splits it into several smaller buckets. In other words, if the number of items under one category path is greater than 200, that group is divided further.

To do this, the system first keeps the items in the group that can be mapped by item2idx, then retrieves their Two-Tower item vectors. The number of sub-buckets is decided by the group size and the max leaf size. A small embedding-based partition is then carried out. The method starts by randomly sampling several initial centers from the item vectors in the group. After that, it runs a very lightweight clustering loop: each item is assigned to the nearest center based on inner-product similarity, then the mean vector of each cluster is computed and normalized, and the process is repeated for a few rounds. In the end, each item is assigned to an embedding-based bucket. If this partition cannot be completed, for example because vectors are missing, the number of valid items is too small, or other problems appear, the system falls back to a hash-based bucketing strategy, which spreads items more evenly across several buckets. This process produces the mapping from leaf node IDs to item lists, together with readable text descriptions for the leaf nodes.

After the leaf nodes are built, the system further derives user interest leaves from the user's history sequence. For each user u, the system takes the most recent 50 interacted items and checks which leaf nodes they belong to. The leaf nodes are then deduplicated while keeping the order of first appearance, and the final list is truncated to at most five interest leaves. This makes the model more likely to construct pairwise comparison samples from the same leaf node or from nearby local regions.

4.3 LLM-based Cold-Start Enhancement

4.3.1 User Profile Summarization

The goal of this stage is not to let the large language model replace the recommender system and make the final online recommendation by itself. That would be too costly and would rely too heavily on the model. The LLM is used for what it is better at: semantic understanding and structured generation. Its role here is to produce extra supervision for cold items, and then turn these outputs into pseudo labels that can later be used to train the ranking model.

The first step is user profile generation. Here, the system compresses the user's recent behavior history into a structured semantic summary, which then serves as a shared user context for Stage 2 to Stage 5. A profile prompt is built from the short item texts prepared earlier, and the LLM is asked to return a structured json summary of the user's interests. The output usually contains fields such as $profile$, $keywords$, and $preferredcategories$. The generated profile is then parsed with a relaxed json parser. Only when it can be successfully parsed as a non-empty dictionary is it treated as a valid user profile and written into a json file.

4.3.2 Stage2–5 Processing Pipeline

The next step does not ask the model to compare all candidate items directly. It first performs routing over a predefined set of higher-level categories. The corresponding prompt asks the model to choose the most relevant categories from a given list. The parser does not simply trust the raw text output. It first tries to read the *categories* field from json, and then projects the result back to the allowed category set. If the json output is invalid, it falls back to fuzzy matching over the natural language text. In this sense, Stage 2 is a constrained category selection task rather than a free-text generation task.

After the higher-level categories are selected, the system moves down to their corresponding subcategory sets and chooses the top-K subcategories. The value of K is controlled by a parameter and is currently set to 8, meaning that at most 8 subcategories are kept for each user. This parameter directly affects the complexity and coverage of the local candidate construction in the next stage.

Stage 4 works inside each selected subcategory. At this point, the system builds a local candidate set and asks the LLM to choose the best items from it. To keep the prompt length manageable, the number of candidates shown to the LLM in each subcategory is capped at 80. Once the local candidates are prepared, they are converted into a scheme-style display format. Each item is written as a short text in the form of "title | category path". If there is related evidence for the item in the evidence map, such as similar-item support, an extra description is added as well. This creates a lightweight RAG-supported candidate presentation. The Stage 4 prompt then asks the model to select the top-k items from the given list. During parsing, the system first tries to read the asins field from json. If that fails, it falls back to extracting ranked indices, bracketed asins, or fuzzy matches from the output text. If no valid item can be recovered in the end, the system simply falls back to the first few items in the candidate order.

Once local preselection has been completed for all chosen subcategories, the selected items are merged and passed into Stage 5 for global re-ranking. The corresponding prompt explicitly asks the model to return strict json, containing only the top-K asins, while trying to keep some diversity across categories or brands. The parser first supports a compact output schema such as {"asins":[...]} and applies hard filtering, so that only items that actually appeared in the grouped preselection are accepted. This prevents the model from returning items outside the candidate pool.

4.3.3 Pseudo Edge and Pairwise Triplet Extraction

Pseudo edges and triplets are not generated separately outside Stage 1–5. They are extracted in a unified way from the final ranking. In other words, the output of Stage 1–5 is not treated as the final recommendation list shown to the user, but as trainable pseudo supervision. Once Stage 5 produces either a valid ranking or a fallback ranking, the system first rewrites each ranked item into a tuple of $(asin, score, keywords, subcat)$, and then converts these tuples into pseudo edges and triplets.

For pseudo edges, the conversion function scans through the final ranking and keeps only the items whose score is no lower than a preset minimum score. The score is then divided by score_scale and normalized into a floating-point weight in the range $[0, 1]$. Under the current default setting, $score\_scale = 5$ and $min\_score = 2$. This means that as long as an item receives a discrete score of at least 2 in the final ranking, a pseudo edge (u, i, w) will be created. Intuitively, such a pseudo edge

means that, for user u, the system believes item i has a certain level of potential positive preference. Its weight comes from the Stage 5 ranking score rather than from real user behavior.

The triplet construction follows the same ranking result but turns it into pairwise supervision. Instead of enumerating all possible positive-negative pairs, the system uses a controlled mixed sampling strategy. One part comes from hard negatives built from neighboring ranked items. If two adjacent items have different scores, the higher-ranked one is treated as the positive sample and the lower-ranked one as the negative sample, while the weight is determined by the score gap. When the number of triplets is still not enough, the system further samples head-versus-tail pairs between top-ranked and low-ranked items, which gives easier ranking constraints. If all items happen to have the same score, ranking position itself is used as a weak preference signal to add more triplets. The number of triplets generated for each user is limited by a parameter, which is currently set to 3. As a result, each user contributes only a small number of relatively high-quality pairwise supervision signals rather than all possible item pairs.

4.3.4 Robust Generation

To keep LLM-based cold-start label generation stable and controllable, the system does not directly use raw natural language outputs for training. It includes a set of safeguards, mainly covering RAG-style evidence augmentation, JSON-first parsing, allowed-set constraints, retry and fallback, and cache reuse.

For RAG-style evidence, the system may attach up to three similar-item evidence snippets when preparing the Stage 4 candidate display text, and these snippets are kept only when the similarity is above 0.35. The idea here is not traditional retrieval from an external knowledge base. It uses the in-domain item embedding space . Similar items are retrieved through FAISS, and their titles and category summaries are appended to the current item. This evidence allows the LLM to evaluate a candidate not only from its title also from a small group of semantically related items, improving its ability to infer the properties and potential use cases of cold items.

The parsing strategy is also designed to be as structured as possible. Across Stage 1–5, the model is always asked to return json before anything else, rather than free-form text. The parser first looks for a possible json block in the raw output and then applies relaxed parsing, so it can still handle semi-structured responses that contain a little extra explanation before or after the json. When json is available, the system uses its structured fields as the main source. Only when json cannot be recovered does it fall back to heuristic extraction from plain text, line numbers, or bracketed content. This keeps the output in a more structured space.

Output constraints become tighter as the process moves forward. In Stage 2 and Stage 3, the returned categories and subcategories must come from the allowed category or subcategory sets prepared in advance. In Stage 4, the returned asins must belong to the candidate list of that subcategory. In Stage 5, the returned asins must come from the allowed asin set in the grouped preselect; otherwise, the parser removes them directly. So the later the stage, the smaller and stricter the output space becomes. This greatly reduces output drift and the problem of the model inventing items outside the candidate pool. It also avoids any leakage from ground-truth items.

When json parsing fails in the first attempt, the system does not stop immediately. It gives the model one more retry. If that still fails, a fallback branch is used. In Stage 4, the fallback simply takes the first few items from the candidate list. In Stage

5, it falls back to the flattened grouped preselect ranking. Moreover, the Stage 5 fallback is not treated as a complete failure. The system keeps these fallback-derived labels with a very small weight, acting like a soft denoising gate. Therefore, the framework prefers to preserve at least weak supervision rather than leaving the user with no training signal at all after a json failure.

4.3.5 Sequential Re-ranking with Multi-source Supervision

In the SASRec stage, the system first builds a time-ordered sequence for each user from the training data. The actual input to SASRec is not the raw item ID, but the item index sequence after applying an offset of one. The pseudo edges and triplets are then mapped into the same user and item index space. An filtering step is applied here as well: only users whose history length is at least 2 are kept in the mapping. This is necessary because SASRec needs at least one history prefix and one target item to form a valid training sample. The system also builds a set of historical positive items for each user, which is later used during negative sampling so that already interacted items are excluded. Each user sequence is padded or truncated to a fixed length controlled by a parameter, which is currently set to 50. During training, the DataLoader uses a batch size of 256 and provides $(uid, seq, target)$ triples for the forward pass. The optimizer is AdamW with a learning rate of 3×10^{-4}.

For using pseudo edges, a sample is kept only when several conditions are met: the user u exists in user2idx, the item i exists in item2idx, and the user belongs to the valid user set, meaning that the user has a non-trivial real interaction sequence. Only after passing these checks is the sample mapped into pseudo indices. The same idea is used for triplets. A triplet is kept only if the user, the positive item, and the negative item can all be mapped into the shared index space, and the user also belongs to the valid sequence user set. This means the LLM-generated labels are not inserted into the model directly. They first go through user validity filtering and index mapping, so that all extra supervision stays in the same training space as the real behavior sequences.

The negative samples are not chosen in a purely random way. The system gives priority to negatives from the Two-Tower recall candidates. In practice, the code first builds a FAISS index from the Two-Tower item vectors, then uses the user vectors to retrieve a batch of recall candidates for each valid user. After removing the items the user has already interacted with, as well as cold items, the remaining ones form the user-specific negative pool. This means SASRec is trained with harder negatives that are closer to the real re-ranking setting.

The main SASRec loss is a candidate-level listwise classification task. In each batch, the DataLoader provides $(u_batch, seq_batch, tgt_batch)$, where seq_batch is the padded history sequence and tgt_batch is the true target item. For each user, the system samples a set of negative items and combines them with the target to form a candidate set. The model then computes scores for the whole candidate group based on the last position of the current sequence. Since the first column in the candidate matrix is always the true target, the label is always an all-zero vector, and cross-entropy loss is used to train the model to rank the target item highest. In effect, this main loss teaches the model to place the real next item at the top among a set of candidates given the user's history sequence.

The model can also use an additional loss called loss sim. Here, the pseudo edges are not simply mixed into the main training data. In each batch the system samples from the pseudo index set an extra mini-batch of pseudo edges, with size at most similar to the main batch. For each pseudo edge, the system rebuilds the user's

sequence input, samples a set of negatives for that user, and then places the pseudo-edge item in the first column as the positive candidate. This means pseudo edges act as an extra pointwise positive re-ranking signal, while still following the same training pattern as the main task, namely one positive item against a set of candidate negatives.

The triplets are used more directly as pairwise ranking constraints. In each training step, the code samples a fixed number of triplets and obtains (su, sp, sn, sw), representing the user, the positive item, the negative item, and the weight. The system then rebuilds the sequence input for these users and computes the score of the positive item and the score of the negative item under the same sequence context. The objective is to make the positive item score higher than the negative item score, with a larger margin being better. The weight allows different pairwise constraints generated by the LLM to contribute with different strength, depending on how reliable they are considered to be. Putting these parts together, the overall SASRec training objective is:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{main}} + \lambda_{\text{sim}}\mathcal{L}_{\text{sim}} + \lambda_{\text{pair}}\mathcal{L}_{\text{pair}}. \tag{4.5}$$

This shows that SASRec is not used here as a standard sequence predictor only. It is a re-ranker trained with multi-source supervision, where loss main learns from real user behavior sequences,loss sim brings in pointwise pseudo positive signals from the LLM, and loss pair adds pairwise ordering constraints from the LLM.

The trained SASRec model is not used to rank the entire item catalog by itself. During evaluation, it only re-ranks the candidate set that has already been retrieved by Two-Tower. The evaluation first uses the Two-Tower user vector to retrieve candidates from FAISS, then filters out seen items, and computes sas scores for the remaining candidates. The system also computes the corresponding tw scores, and the final ranking is obtained by combining them according to rerank mode. The current default setting is hybrid. The SASRec is not designed to replace the retrieval stage. Its role is to refine sequential preference at the candidate level inside the Two-Tower candidate space, and then work together with the retrieval score to produce the final ranking.

4.4 Implementation Tools

This project is implemented in Python 3.12. The experiments use two public subsets from the Amazon Review Data (2018) collection[16]: All Beauty and Health & Personal Care. For text representation, the model uses *all_MiniLM_L6_v2* to encode item text and build content-based embeddings[17]. For the generative language model, the framework uses $Qwen3\_4B - Instruct\_2507$ for user interest summarization, hierarchical filtering, ranking decisions, and weak supervision generation. According to the official model card, this model is a 4.0B-parameter causal language model that has gone through both pretraining and post-training, and it natively supports a context length of 262,144 tokens[18].

Chapter 5

Experiments and results

The experiments are designed to evaluate how different enhancement modules contribute to cold-item recommendation within the same retrieval–reranking backbone. Comparing with many unrelated baselines, the study focuses on four representative settings: E0, E1, E2, and E3. These four settings correspond to progressively stronger variants of the same system and allow a controlled analysis of the contributions of the tree module, pointwise pseudo supervision (sim), and pairwise supervision (pair). More specifically, the experiments aim to answer three questions. First, how strong is the retrieval–reranking backbone by itself when no tree, sim, or pair enhancement is used? Second, when the tree structure is enabled, does adding sim supervision or pair supervision lead to different types of improvements? Third, when both forms of LLM-generated supervision are used together, does the complete system achieve better cold-item performance than either partial variant? These questions are directly reflected in the four selected configurations

For comparison, additional baseline models, including GRU4Rec and BERT4Rec, were also evaluated. These baselines were tested under their original settings to compare their performance in warm-start and cold-start scenarios. All experiments mainly use Recall@5, Recall@10, Recall@20, NDCG@5, NDCG@10, and NDCG@20 as evaluate metrics.

5.1 Experiments

The importance of the tree structure in the overall design is not that it replaces the retrieval stage. Its real value is that it organizes the retrieved candidates into a structured decision space, where items can be filtered step by step, where higher-quality weak supervision can be generated, and where cold items have a better chance to compete. Without the tree structure, Stage 1–5 would become a flat item-ranking process. In that case, the burden on the LLM would be much heavier, the pseudo edges and triplets would become noisier, and the whole system would depend more strongly on the ceiling of the Two-Tower retrieval model. For this reason, the tree structure is integrated into Stage 1–5 and is always enabled in the main framework.

We also ran two settings without tree participation, but these are treated as comparison methods rather than part of the main pipeline. In one setting, the LLM generates synthetic preference signals for cold items during training, so that the recommendation model can learn better cold-item embeddings. More specifically, pairwise prompts are used to generate relative preferences, and these are then injected into training through an auxiliary BPR loss. In the other setting, the simulator method uses the LLM to simulate user sequences that would likely interact with each cold item. In this way, a cold item is turned into an item with synthetic behavior, and its embedding is then updated through the behavior-based optimization

pipeline. These variants will be compared later in the experimental results, rather than being presented as part of the main system flow.

In E0, all three enhancement switches are disabled. This means E0 uses only the shared retrieval–reranking backbone: Two-Tower for first-stage retrieval and SASRec for sequential reranking. No tree/taxonomy constraint is applied, no LLM-based pseudo edges are generated, and no pairwise triplets are used. Therefore, E0 serves as the backbone-only reference point for all later comparisons.

This E1 setting activates tree/taxonomy-based candidate organization and LLM-generated pseudo edges, but does not use pairwise triplets. The Stage1–5 pipeline is enabled. Its final ranking is converted only into pointwise pseudo supervision, which is then passed to SASRec as pseudo edges. Therefore, E1 isolates the contribution of pointwise LLM supervision under taxonomy constraints.

The E2 configuration still enables the tree/taxonomy layer and the Stage1–5 pipeline, but only uses the pairwise triplet labels extracted from the final ranking. No pseudo edges are injected. Therefore, E2 isolates the contribution of pairwise supervision under the same tree-constrained generation process.

The E3 stage is the full model. It uses tree module for candidate organization, runs the Stage1–5 hierarchical LLM pipeline, and extracts both pseudo edges and triplets from the same final ranking. These two supervision forms are then jointly used in SASRec training. Theoretically, E3 is the strongest and most complete version of the proposed framework.

TABLE 5.1: Expeirments settings

Setting	**Tree**	**Sim**	**Pair**	**Backbone**
E0	/	/	/	Only Two-Tower + SASRec
E1	on	on	/	Two-Tower + SASRec
E2	on	/	on	Two-Tower + SASRec
E3	on	on	on	Full model

5.2 Results Analysis

We further included BERT4Rec and GRU4Rec as additional baselines to examine how classic sequential recommendation models perform in cold-start settings. The results show a clear split on both datasets. For warm-item recommendation, these models achieved relatively good performance, with most metrics close to 0.1. For cold-item recommendation, however, Recall@5, Recall@10, and Recall@20 were almost zero. This issue will be discussed later.

Looking at the overall results in the Tables(5.2, 5.3), all three improved variants perform better than the baseline E0 on cold-item recommendation, although the size of the gain is different across methods.

On the Beauty dataset, the cold-item performance of baseline E0 is extremely low, which suggests that the cold-start task is particularly difficult in this dataset. For example, the Recall@5, Recall@10, and Recall@20 of E0 are only 0.0006, 0.0017, and 0.0029, while NDCG@5, NDCG@10, and NDCG@20 are only 0.0004, 0.0007, and 0.0010. This shows that under the baseline setting, the model has very limited ability to retrieve and rank cold items.

Against this baseline, E1 brings a clear improvement. On Beauty-cold, its Recall@20 increases from 0.0029 to 0.0093, which is about a 220.7% improvement over

TABLE 5.2: Cold-start result summary

Dataset	Stage	Recall@5	Recall@10	Recall@20	NDCG@5	NDCG@10	NDCG@20
beauty	E0	0.0006	0.0017	0.0029	0.0004	0.0007	0.0010
beauty	E1	0.0023	0.0029	0.0093	0.0013	0.0015	0.0031
beauty	E2	0.0017	0.0029	0.0075	0.0009	0.0013	0.0025
beauty	E3	0.0023	0.0040	0.0131	0.0012	0.0018	0.0040
health	E0	0.0000	0.0022	0.0091	0.0000	0.0007	0.0029
health	E1	0.0043	0.0119	0.0199	0.0020	0.0044	0.0065
health	E2	0.0011	0.0043	0.0165	0.0004	0.0014	0.0034
health	E3	0.0049	0.0108	0.0216	0.0021	0.0040	0.0067

E0. Its NDCG@20 also rises by about 210.0%. This suggests that E1 not only improves the chance of retrieving cold items, but also places them in better positions in the ranked list.

E2 also performs better than the baseline on Beauty-cold, but its gain is smaller than that of E1. E3 gives the strongest results on Beauty-cold. Its Recall@5 reaches 0.0023, which is the same as E1, while Recall@10 further rises to 0.0040, about 135.3% higher than E0. Recall@20 reaches 0.0131, which is about a 351.7% improvement over E0 and is the best result among all methods on this dataset. The corresponding NDCG@10 and NDCG@20 improve by about 157.1% and 300.0%, respectively. This means that the advantage of E3 is not only in the very top positions, but also in its stronger ability to keep cold items in the candidate list when the cutoff becomes larger. From an overall cold-item perspective, E3 is therefore the best method on Beauty. The warm-item results do not show clear improvement, but stay close to the baseline, which means the model can greatly improve cold-item recommendation while keeping warm-item performance almost unchanged.

TABLE 5.3: Warm-start result summary

Dataset	Stage	Recall@5	Recall@10	Recall@20	NDCG@5	NDCG@10	NDCG@20
beauty	E0	0.0126	0.0200	0.0313	0.0081	0.0105	0.0134
beauty	E1	0.0106	0.0189	0.0309	0.0072	0.0096	0.0128
beauty	E2	0.0127	0.0195	0.0322	0.0079	0.0101	0.0132
beauty	E3	0.0115	0.0186	0.0304	0.0075	0.0098	0.0127
health	E0	0.0107	0.0164	0.0256	0.0076	0.0094	0.0117
health	E1	0.0090	0.0142	0.0263	0.0062	0.0078	0.0096
health	E2	0.0112	0.0175	0.0258	0.0075	0.0095	0.0116
health	E3	0.0096	0.0149	0.0274	0.0065	0.0082	0.0101

To further visualize recall behavior across configurations, the following three figures report Recall@20 trends in combined and dataset-specific views.

A similar pattern appears on the Health dataset. The baseline E0 is again weak in the cold-item setting. E1 brings a very strong improvement, showing clear gains in both early ranking quality and overall cold-item coverage. E2 also performs better than the baseline on Health-cold, but the improvement is smaller than that of E1. E3 shows the strongest overall performance on Health-cold. It raises Recall@5 to 0.0049, slightly above E1's 0.0043, and Recall@20 to 0.0216, which is again higher than E1's 0.0199. Its NDCG@5 reaches 0.0021 and NDCG@20 reaches 0.0067, both of which are the best results on this dataset. The warm-item results are similar to those on Beauty, with only small fluctuations and no obvious loss in performance.

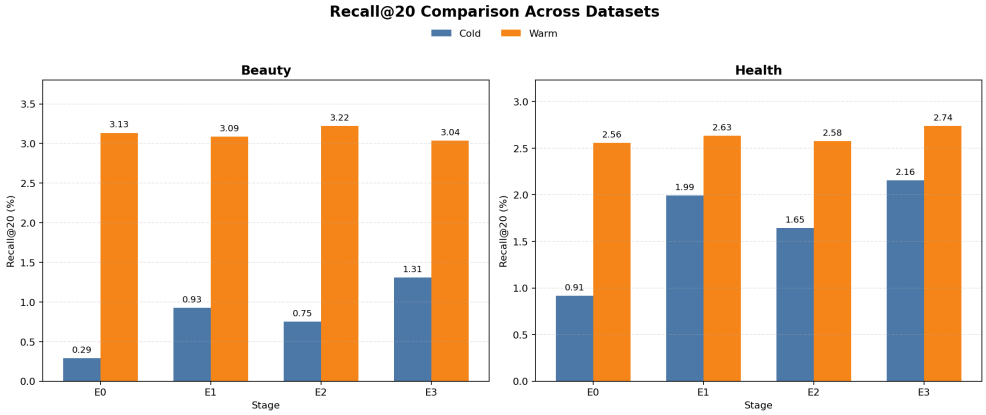

FIGURE 5.1: Combined Recall@20 comparison under cold and warm settings across experiments.

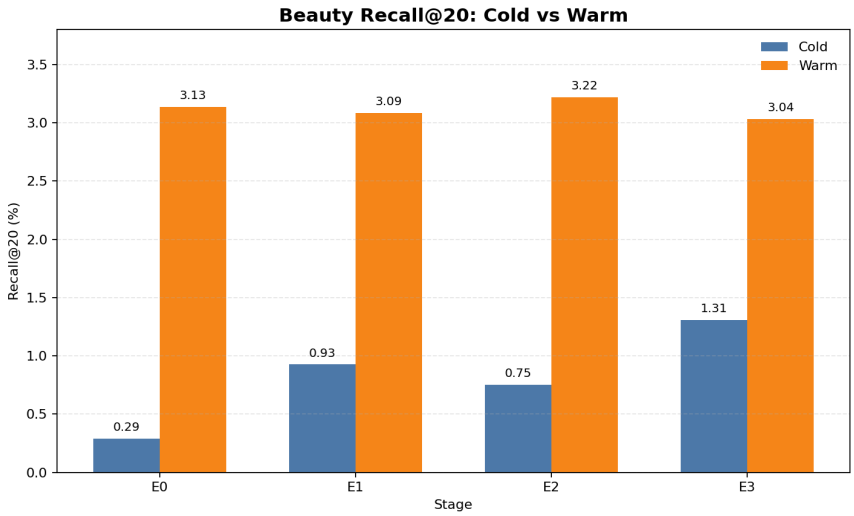

FIGURE 5.2: Recall@20 results on Beauty under cold and warm settings.

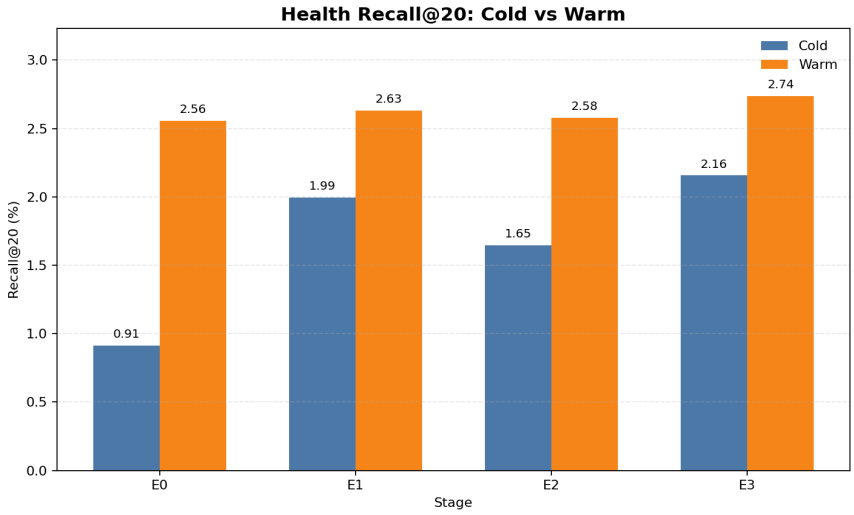

FIGURE 5.3: Recall@20 results on Health under cold and warm settings.

Overall, these results suggest that the proposed improvements consistently strengthen cold-item recommendation on both datasets. The gains are especially clear in E1 and E3, while warm-item performance remains largely stable. This is noteable, because it shows that the framework improves the difficult cold-start part of the task without noticeably hurting the standard warm-item recommendation quality.

Chapter 6

Discussion

As mentioned earlier, BERT4Rec and GRU4Rec were included as additional baselines. The results show a very clear pattern. In warm-item recommendation, their recall scores are generally close to 0.1. In cold-item recommendation, however, the recall values are almost zero.

This result is not surprising when we look at what these models are built to do. GRU4Rec is based on an RNN, while BERT4Rec is based on a Transformer with masked sequence modeling. Even though they differ in architecture, both are essentially ID-based sequential recommendation models that rely heavily on historical interaction sequences. They are usually very effective for warm items, but they are naturally weak when the target items are truly cold.

What these models take as input is the sequence of item IDs in the user's history. The model then learns the order relations between items. In other words, they are good at answering a question like this: if a user has interacted with these items before, which previously seen items are most likely to come next? This works well for warm items, because warm items have already appeared in training, so the model has a chance to learn their item embeddings or transition patterns.

Cold items are different. If an item has no interactions, or almost none, in the training data, the model has little chance to learn a useful embedding for it. These models do not make real use of content features, so there is very limited support for generalization beyond seen IDs. As a result, during testing, they are very unlikely to rank cold items near the top.

The main difference between GRU4Rec and BERT4Rec is in how they model the user sequence and how they use context. That difference matters for standard sequence recommendation, but it does not change the basic limitation here. The issue is not that these two models are weak by themselves. The problem is that pure ID-based sequential models are much more suitable for warm recommendation than for true item cold-start.

In that sense, this result actually supports the validity of the task setting. It suggests that the cold-item definition used in this work is strict and meaningful, since it really separates out the items that the model has not seen during training.

Chapter 7

Conclusion and future work

Under a strict item cold-start setting, the goal here is to improve recommendation for unseen items while keeping warm-item performance as stable as possible. Based on the results, this goal is largely achieved. Compared with the baseline E0, as well as the original BERT4Rec and GRU4Rec, E1, E2, and E3 all show clear gains in cold Recall and cold NDCG on both datasets. This suggests that once content-based retrieval, structured filtering, and LLM-based weak supervision are added, the system can indeed build extra recommendation signals for items that never appeared in training. Among these variants, E3 gives the strongest or nearly strongest cold performance on both Beauty and Health.

The reason is that the framework brings together several parts that complement the weaknesses of pure sequential models in cold-start recommendation. Two-Tower gives cold items a content-based entry point at the recall stage, so even without historical interactions they still have a chance to enter the candidate set. The taxonomy tree then organizes these flat candidates into a hierarchical space of category, subcategory, and item, which makes the LLM's job easier than asking it to reason directly over a large item list. On top of that, the ranking signals generated by the LLM are converted into pseudo edges and triplets, and these are injected into the parametric model through the multi-objective training of SASRec. In this way, cold items that originally lack behavior signals receive extra training support.

The results also make the current bottlenecks quite clear. The whole framework is still limited by the recall ceiling of the Two-Tower stage. No matter how refined the later tree and LLM modules are, if a cold item does not enter the candidate pool during recall, then Stage 1–5 and SASRec cannot recover it later. This means that the upper bound of cold-item improvement still depends heavily on the quality of the content representation learned by Two-Tower. One result makes this especially clear: the recall coverage within the top 2000 candidates is only about 60%. In other words, a noticeable part of the items and interactions never really enters the range that the later stages can work on.

The taxonomy tree helps by reducing the difficulty of LLM decision making, but it can also introduce its own risk. If Stage 2 or Stage 3 selects the wrong category or subcategory early on, then Stage 4 and Stage 5 can only continue filtering within the wrong local space. Once that happens, the final ranking quality is directly affected.

The weak supervision from the LLM improves cold-item performance, but it also changes the balance of ranking preference to some extent, making the model more inclined to protect cold items. This is the main reason why the warm metrics show some fluctuations.

Several directions could be explored in future work. One of the most important is to further improve the recall coverage of the backbone model. A stronger backbone would make cold items more visible at the first stage, so that more potential target items, relevant users, and useful interaction signals can enter the candidate space

and stay in the list at earlier positions. This would also reduce how much the later stages depend on the recall ceiling.

Another possible direction is to use a larger or stronger LLM for the analysis and generation steps. The current model is only a 4B open-source model, so there is still a clear upper limit in areas such as user interest understanding, step-by-step filtering, structured output, and weak supervision generation. A stronger language model may bring further gains in these parts. At the same time, better model ability does not automatically mean better recommendation performance. The final effect also depends on inference cost, reproducibility, and how well the model fits the existing candidate structure. It would be valuable to compare models of different sizes, as well as open-source and closed-source models, within the same framework, and then study the trade-off between model strength, system efficiency, and recommendation quality.

It would also be useful to test the framework on more raw, more complex, and sparser datasets. The current datasets already show a strict cold-start setting, but real-world platform data is often much harder. It usually has a stronger long-tail distribution, higher sparsity, more noise, and richer hidden information. Evaluating the method under such conditions would help us better understand its robustness and generalization ability in more challenging recommendation scenarios.

References

[1] Gediminas Adomavicius and Alexander Tuzhilin. "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions". In: *IEEE transactions on knowledge and data engineering* 17.6 (2005), pp. 734–749.

[2] Badrul Sarwar et al. "Item-based collaborative filtering recommendation algorithms". In: *Proceedings of the 10th international conference on World Wide Web*. 2001, pp. 285–295.

[3] Andrew I Schein et al. "Methods and metrics for cold-start recommendations". In: *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. 2002, pp. 253–260.

[4] Heng-Tze Cheng et al. "Wide & deep learning for recommender systems". In: *Proceedings of the 1st workshop on deep learning for recommender systems*. 2016, pp. 7–10.

[5] Shuai Zhang et al. "Deep learning based recommender system: A survey and new perspectives". In: *ACM computing surveys (CSUR)* 52.1 (2019), pp. 1–38.

[6] Paul Covington, Jay Adams, and Emre Sargin. "Deep neural networks for youtube recommendations". In: *Proceedings of the 10th ACM conference on recommender systems*. 2016, pp. 191–198.

[7] Balázs Hidasi et al. "Session-based recommendations with recurrent neural networks". In: *arXiv preprint arXiv:1511.06939* (2015).

[8] Wang-Cheng Kang and Julian McAuley. "Self-attentive sequential recommendation". In: *2018 IEEE international conference on data mining (ICDM)*. IEEE. 2018, pp. 197–206.

[9] Fei Sun et al. "BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer". In: *Proceedings of the 28th ACM international conference on information and knowledge management*. 2019, pp. 1441–1450.

[10] Xiangnan He et al. "Lightgcn: Simplifying and powering graph convolution network for recommendation". In: *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 2020, pp. 639–648.

[11] Likang Wu et al. *A Survey on Large Language Models for Recommendation*. 2024. arXiv: 2305.19860 [cs.IR]. URL: https://arxiv.org/abs/2305.19860.

[12] Yupeng Hou et al. *Large Language Models are Zero-Shot Rankers for Recommender Systems*. 2024. arXiv: 2305.08845 [cs.IR]. URL: https://arxiv.org/abs/2305.08845.

[13] Jianling Wang et al. "Large language models as data augmenters for cold-start item recommendation". In: *Companion Proceedings of the ACM Web Conference 2024*. 2024, pp. 726–729.

[14] Feiran Huang et al. “Large language model simulator for cold-start recommendation”. In: *Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining*. 2025, pp. 261–270.

[15] Wenlin Zhang et al. “Llmtreerec: Unleashing the power of large language models for cold-start recommendations”. In: *Proceedings of the 31st International Conference on Computational Linguistics*. 2025, pp. 886–896.

[16] Jianmo Ni, Jiacheng Li, and Julian McAuley. “Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2019.

[17] Sentence-Transformers. *sentence-transformers/all-MiniLM-L6-v2*. Hugging Face model card. 2025.

[18] Qwen Team. *Qwen3-4B-Instruct-2507*. Hugging Face model card. 2025.