NLP-based Recommender System

 [1]

Springboard Capstone Project

Sook hyun Lee

October 2025

[1] Image taken from https://www.rootstrap.com/blog/the-magic-behind-recommendation-systems

1. Introduction

1.1 Why?

Online retail and service businesses are now ubiquitous, and anyone who has purchased items on platforms such as Amazon, Target, Lowe's, or Wayfair has likely encountered personalized product and service recommendations. A few years ago, these recommendation systems often appeared less effective, for example, users frequently received suggestions for items like tapestries or home appliances immediately after purchasing those very products. This raised concerns about both the timing and accuracy of the underlying algorithms. In contrast, contemporary recommendation systems have become markedly more precise and context-aware, offering suggestions that enhance rather than frustrate the shopping experience. As a result, online browsing now closely mirrors the experience of navigating a thoughtfully organized warehouse store, where related and complementary items are intuitively presented.

In this project, we develop a recommender system inspired by embedding techniques commonly used in natural language processing (NLP), utilizing the PyTorch framework.

1.2 Value to client

This project addresses the problem of product discovery by implementing a recommendation algorithm that personalizes the shopping experience. The algorithm analyzes users' historical purchasing behavior to understand their preferences and patterns. Based on these insights, it recommends products tailored to each individual customer, thereby improving user engagement, increasing conversion rates, and enhancing overall customer satisfaction. The business will use insights from the recommendation system to guide strategic decisions:

Marketing - Personalized campaigns, segmented targeting, optimized promotions

Product - UX improvements, feature prioritization, product bundling

Leadership - Strategic investment, growth metrics, cross-team alignment

2. Data acquisition and cleaning

2.1 Data source

The dataset used for evaluation is the RetailRocket Recommender System Dataset, obtained from Kaggle. The four files provided in the dataset will be used to test and evaluate our models.

- events.csv
- category_tree.csv
- item_properties_part1.csv
- item_properties_part2.csv

The dataset includes the following required information: **transaction ID**, **item ID**, and **category information for each item**, along with transaction date and time, and customer ID. For the initial study, the context window will be limited to a single transaction for simplicity. In later stages, model performance will be evaluated using varying context windows and/or including non-transactional activities. This will involve generalizing individual **transactions** into broader **sessions**; sessions group customer activities based on temporal proximity, typically within a single day or across a week.

2.2 Data cleaning and organization

2.2.1 Event data

The event data in `events.csv` includes the following attributes: *timestamp*, *visitor ID*, *event* type, *item ID*, and *transaction ID*. An example sequence of customer activities using these attributes is shown in **Table 1**.

	timestamp	visitorid	event	itemid	transactionid
265573	1434403666570	90352	view	425758	NaN
271793	1434404197081	90352	transaction	425758	0.0
277295	1434403991902	90352	addtocart	425758	NaN
533488	1435331816929	90352	view	425758	NaN

Table 1. An example sequence of activities by a customer (ID 90352)

The following steps were taken to clean the data:

1. Convert timestamps into Python datetime objects for consistency and create a new column, *date*, for ease of manipulation.

2. Group user activities by day: For each customer, activities occurring on the same day were grouped together, and a new column, *session_by_day*, was created to enumerate these daily sessions. The session counts per customer are shown in **Figure 1**.

3. Impute missing values: The *transaction ID* field contains NaN for non-transactional events such as *view* and *add-to-cart*. These missing values were filled with -1 to indicate the absence of a transaction.

4. Data Types: All columns, except for *event* and *date*, have the data type int64

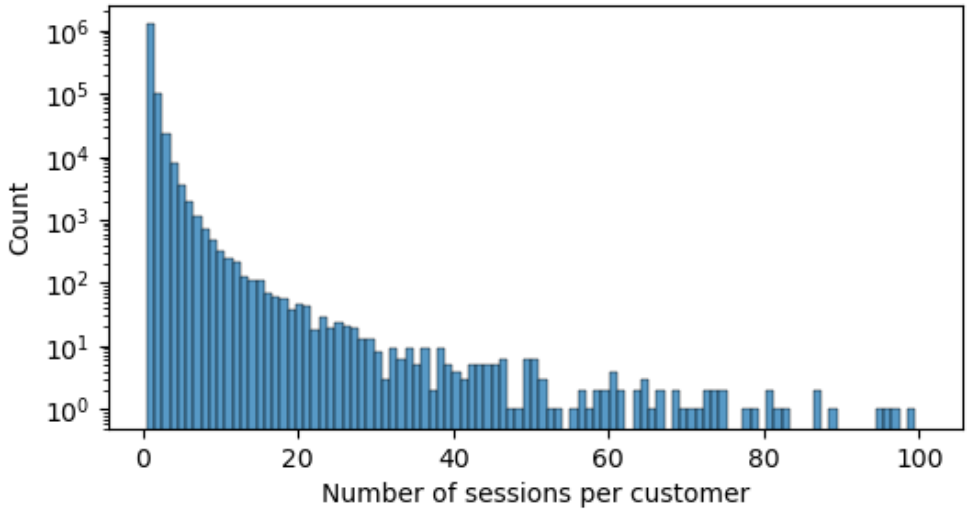

Figure 1. Distribution of total sessions by customer.

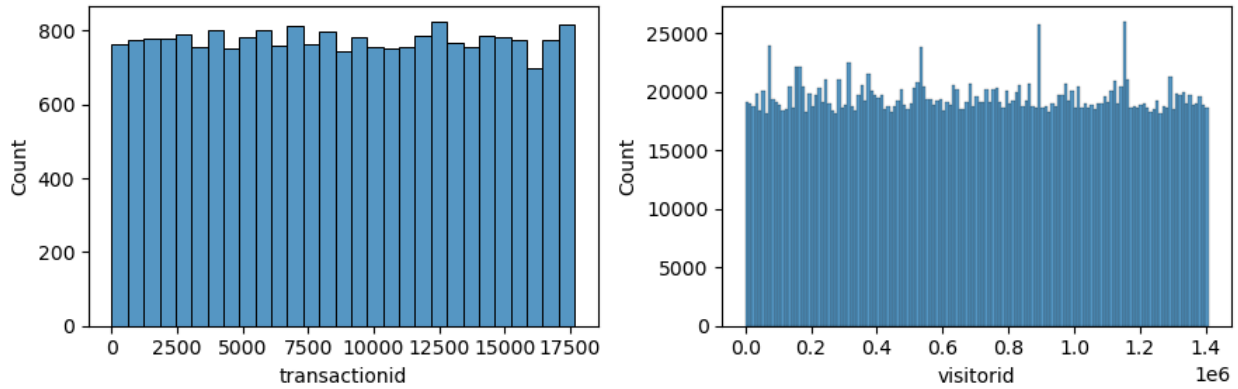

Figure 2. Distributions of *transaction ID* and *visitor ID*.

2.2.2 Item properties data

The item properties data in `item_properties_part1.csv` and `item_properties_part2.csv` contains category information at the most granular level along with other information as shown in **Table 2**. There are 417053 unique items with an associated *category ID*, and a total of 1242 unique *category ID*s in the dataset. We find that approximately **5.6%** of items are linked to more than one *category ID* (**Figure 3)**. This is due to occasional changes in category assignments within the date range between 2015-05-10 to 2015-09-13. To assign a **unique category ID** to each item, we extract the *date, item ID,* and *category ID* columns to track and resolve these changes. To simplify the analysis, only items with an initially assigned category ID are retained.

	timestamp	itemid	property	value
0	1435460400000	460429	categoryid	1338
1	1441508400000	206783	888	1116713 960601 n277.200
2	1439089200000	395014	400	n552.000 639502 n720.000 424566
3	1431226800000	59481	790	n15360.000
4	1431831600000	156781	917	828513

Table 2. Contents of item properties data in `item_properties_part1.csv.`

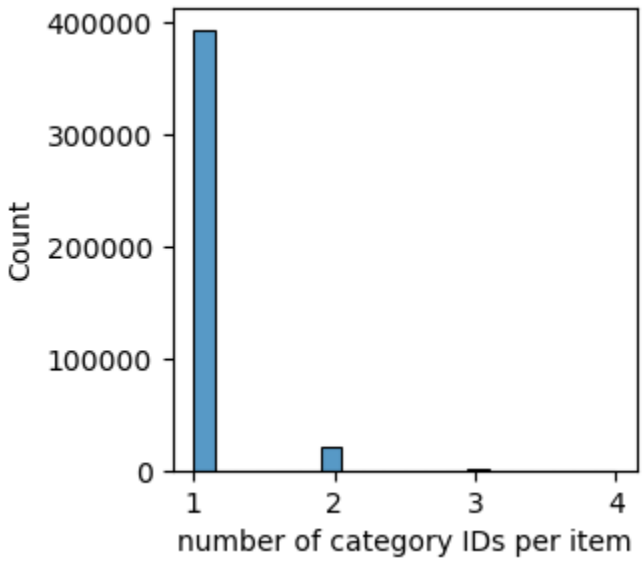

Figure 3. Distribution of category count per item.

After merging the unique *item ID–category ID* mapping with the event data, we find that **21.2%** of all unique items in the event data lack an associated category ID. In contrast, only **3.16%** of unique items involved in **transactions** are missing a category ID. For consistency, we assign a placeholder value of **-1** to items without a category ID.

2.2.3 Category data

The raw data in `category_tree.csv` includes hierarchical item categories represented in a pairwise format. All category IDs are found to be unique. The parent of the categories directly below the root node has no assigned ID. To address this, a unique value of **10000** is assigned as the parent ID of the root node.To construct a multi-layer tree structure suitable for training embedding weights, we convert the pairwise relationships into a chain format. We identified a total of 1,670 unique categories in the dataset. The table below shows the number of categories at each hierarchical level, from the top level (L0) down to the maximum depth (L6). The resulting dataframe is saved in a new file.

Level	L0	L1	L2	L3	L4	L5	L6
No. of categories	1	25	174	702	665	90	13

Table 2. Hierarchical category structure.

Results are saved in new files `category_df.csv` and `events_df.csv` for further analysis.

3. Data exploration and Tree building

3.1 Event data

3.1.1 Customer behavior

In this section, we examine various aspects of customer purchasing behavior. Figure 4 illustrates the number of unique items purchased per customer, while Figure 5 presents the distribution of purchase frequency by customer in terms of days (sessions). Figure 6 depicts the distribution of unique items purchased per session, and Figure 7 shows the fraction of transaction events per session. All figures exhibit long-tailed distributions, reflecting the wide variability in customer behaviors and purchasing patterns.

Figure 4. Distribution of unique Items purchased per customer.

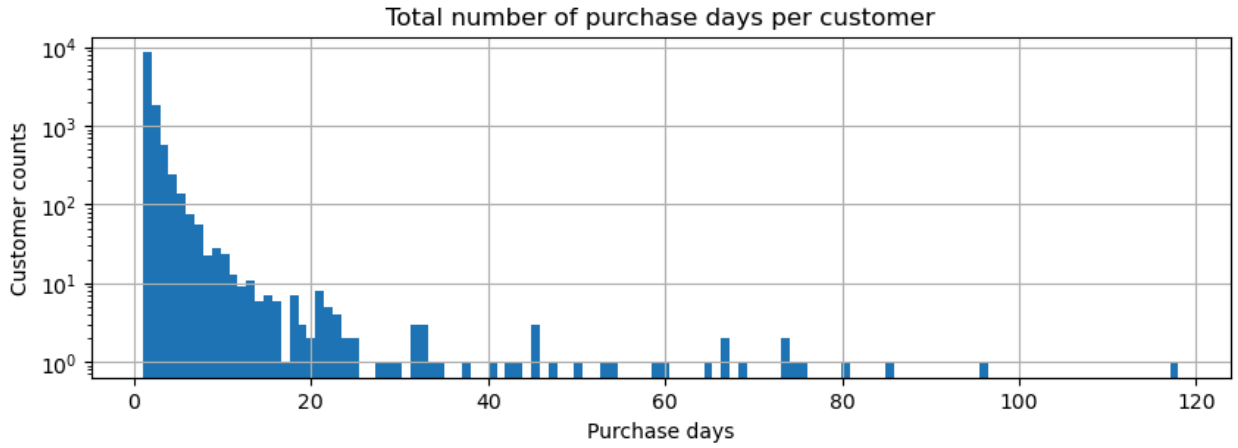

Figure 5. Distribution of purchase frequency by customer (in days).

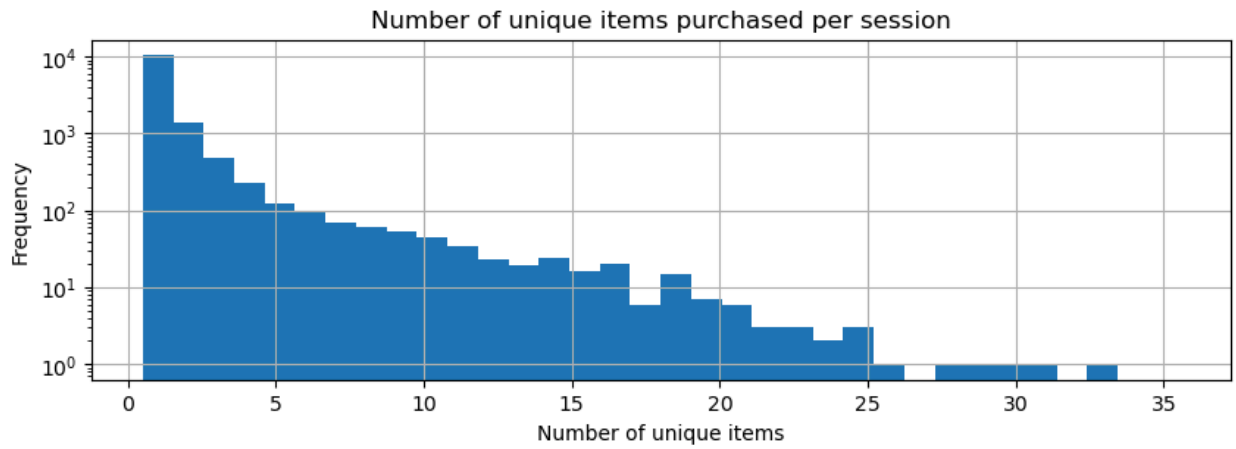

Figure 6. Distribution of unique items purchased per session.

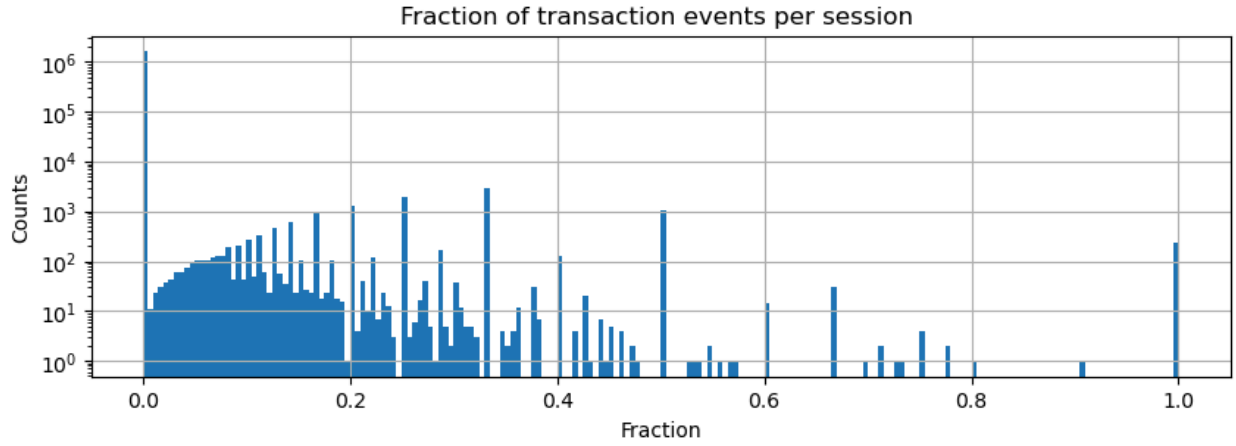

Figure 7. Fraction of transaction events per session.

3.2 Tree building with category data

There ere are two different types of trees we will be building:

- Huffman binary tree for hierarchical softmax
- Category tree for accessing category information associated with each item

We first prepare a dictionary of the format {category_id: {item_id: frequency}}. This frequency dictionary serves two purposes: (a) to construct a Huffman tree based on item frequencies, and (b) to augment the category hierarchy by integrating items as leaf nodes under their respective categories.

3.2.1 Huffman Tree

Huffman tree is a **binary tree** over items that are used for efficient probability estimation in **hierarchical softmax** training technique. It speeds up training and allows for a scalable output layer.

Sorting items by their frequency before constructing a Huffman tree ensures that more frequent items are assigned shorter codes, while less frequent items receive longer codes. This approach minimizes the total number of bits required to represent the dataset efficiently. The purchase frequency distribution is displayed in Figure 8. Figure 9 illustrates the Huffman tree constructed for the 10 (N) items under Category 0, which contains 9 (N-1) internal nodes.

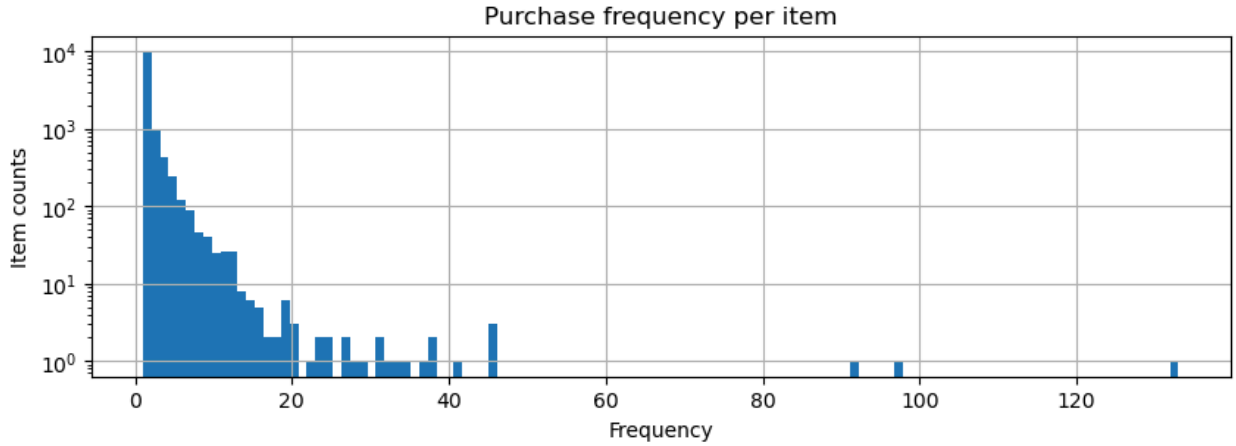

Figure 8. Distribution of item purchase frequencies.

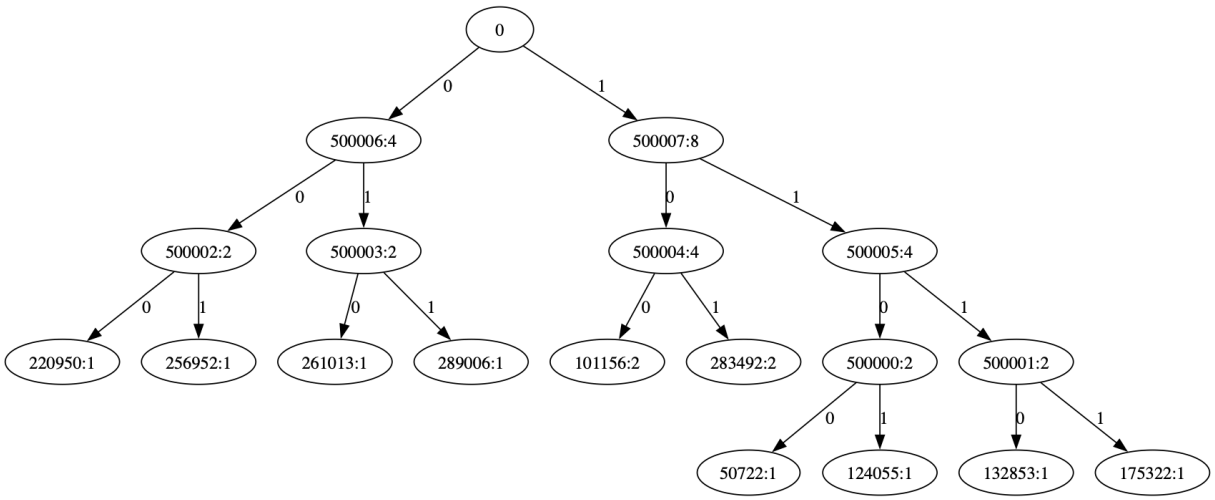

Figure 9. Example Huffman Tree Constructed for Items in Category 0.

3.2.2 Category Tree

We begin by first transforming the square category DataFrame into a hierarchical tree structure, with each category represented as a node. Once the category tree is constructed, we verify whether all categories associated with items correspond to terminal (leaf) nodes in the tree. Upon inspection, we find that some items are linked to non-terminal (intermediate) categories, indicating an inconsistency in the hierarchical structure.

To resolve this, we attach each item to the appropriate node in the category tree, regardless of whether the category is a terminal node. This ensures that all items are represented as leaf nodes in the tree, preserving a consistent structure where only items appear at the leaves, and all internal nodes represent categories.

The full path from an item (terminal) node to the root node can be obtained from the tree as shown in Figure 10.

```
item ID <------> root
[132853, 0, 605, 1482, 10000]
```

Figure 10. Full path from item 132853 to the root node 10000.

4. Feature engineering and train data development

4.1 Feature engineering

Only transaction data is used for the analysis. To ensure efficient model training and reduce complexity, we select only sessions that involve more than one item. In real-world purchase behavior, the relationship between items is often asymmetric, and the order of purchase matters, for example, a laptop is typically bought before a charger. Therefore, while this study focuses on multi-item sessions, modeling single-item sessions may further improve accuracy in the future.

4.2 Training data development

In total, **7547** unique items meet these criteria and appear across **3,055** distinct sessions. From each qualifying session, we construct all possible order-sensitive item-item pairs, reflecting the directional nature of item co-purchases. The dataset is then split into **80%** for training and **20%** for validation to evaluate model performance.

5. Modeling

5.1 Two primary approaches to recommender systems

Recommender systems suggest items to users based on their preferences, behavior, or similarities with others. They're widely used in platforms like Netflix, Amazon, Spotify, and YouTube. The primary goal is to predict a user's preferences and recommend items they are likely to engage with such as movies, products, news articles, or even people. There are mainly two core types of recommender systems, **collaborative filtering** and **content-based filtering**.

Collaborative filtering can be further divided into:

- *User-based filtering*, which identifies users with similar preferences and recommends items those users have liked.
- *Item-based filtering*, which finds items similar to those the user has previously liked and recommends them.

Similarity between users or items is typically computed using distance or similarity measures such as *Euclidean distance, Pearson correlation*, or *cosine similarity*. It's important to note that in collaborative filtering, item similarity is not based on the inherent features of the items, but rather on user interaction patterns. Item-based collaborative filtering is particularly effective in systems where the number of users significantly exceeds the number of items. However, collaborative filtering has several limitations:

- Data sparsity: User-item interaction matrices are often sparse, making it difficult to identify meaningful patterns.
- Cold start: The system struggles to make recommendations for new users or new items due to a lack of interaction data.
- Scalability: Performance can degrade as the number of users or items grows.

- Popularity bias: Tends to over-recommend popular items, reducing personalization and diversity.
- Lack of Interpretability: Recommendations are based on patterns in user behavior, not explicit item attributes.
- Gray sheep problem: Users with unique or atypical preferences may receive poor recommendations.

Content-based filtering, on the other hand, recommends items that are similar to those a user has liked in the past. These similarities are determined based on item attributes or features, such as categories, tags, genres, or other metadata. This approach can help address some of the limitations of collaborative filtering.

- No Need for Other Users' Data: Recommendations are based solely on the user's own preferences and item features.
- Handles Cold Start (User Side): Can recommend items to new users after only a few interactions.
- Interpretable Recommendations.
- Less prone to recommending only popular items.
- Privacy-Friendly: Doesn't require analyzing other users' data.

5.2 Deep learning-based recommender systems

In recent years, deep learning-based recommender systems have gained prominence due to their superior performance and ability to model complex, non-linear relationships between users and items. The following points summarize their key advantages over traditional methods:

- Better at Capturing Non-Linear and Complex Patterns
- Better at Cold Start and Sparse Data
- Effective Feature Representation (Embeddings): can automatically learn *dense, low-dimensional representations (embeddings)* of users and items from sparse interaction data
- Integration of Multiple Data Types (Multimodal Inputs)
- Personalized and Context-Aware Recommendations: Deep models can learn user-specific behaviors, preferences, and temporal patterns.

Deep learning models excel at capturing high-level patterns and personalization through architectures such as deep neural networks, convolutional neural networks (CNNs),

recurrent neural networks (RNNs), and attention-based mechanisms. As a result, they often outperform traditional approaches, particularly in large-scale and dynamic recommendation environments.

6. Modeling

6.1 Item2Vec and hierarchical Item2Vec models

This project focuses on the Item2Vec model, which is based on the Word2Vec architecture originally developed for learning optimized word embeddings in natural language processing (NLP). By drawing an analogy between sequences of words and sequences of user-item interactions, Item2Vec learns dense vector representations of items that capture similarity and co-occurrence patterns. This results in more effective recommendations through meaningful item embeddings.

Item2Vec is an item-based collaborative filtering technique and, as such, inherits some of the limitations discussed in the previous section, such as cold-start problems and a lack of content awareness. To address these issues, we explore a hybrid approach called Hierarchical Item2Vec, which integrates hierarchical content-based information into the embedding process. This method helps mitigate the shortcomings of purely collaborative approaches by incorporating category-level knowledge.

The implementation of the model, along with the necessary utility functions, can be found in the source files referenced below.

- `ItemMap.py` : contains category, item and frequency and methods to fetch index and item/category (TokenMap ~ ItemMap -> HuffmanTree)
- `HierarchicalItem2Vec.py` : main model and trainer, the model reduces to Item2Vec when Params.lambda_cat is set to zero.
- `parameters.py`

Next, we explore and fine-tune the model architecture and its components.

6.2 Hyperparameters

To optimize model performance, we consider the following hyperparameters:

- Regularization parameter λ_{cat}
- Embedding dimension dim_{embed}
- Threshold for item frequency f_{thresh}
- Learning rate lr
- Loss function = {Negative sampling, Hierarchical softmax}
- Similarity measure: cosine similarity, distance

6.2.1 Regularization parameter

The value of λ_{cat} regulates the strength of alignment between item embeddings and their respective category embeddings. We will select the following values during hyperparameter tuning:

λ_{cat} = {0, 0.1, ... , 1, 10}

λ_{cat} Value	Meaning / Effect	When to Use
0	No category alignment at all. Equivalent to Item2Vec.	Baseline comparison; when category data is noisy or not useful.
1 $\times 10^{-4}$ **to 1** $\times 10^{-3}$	Very weak alignment. Minor influence from category embeddings.	Categories are somewhat useful, but item-level patterns dominate.
1 $\times 10^{-2}$ **to 0.1**	Moderate alignment. Balanced influence between item behavior and category info.	Often a good starting point for tuning; works well in many scenarios.
0.1 to 1.0	Strong alignment. Item embeddings are pulled significantly toward category ones.	When categories are well-defined and strongly predictive.
>1.0	Very strong alignment. Item embeddings may lose individuality.	Only use if category structure is known to be highly reliable.

Table 3. λ_{cat} values: their meanings and proper usages

6.2.2 Embedding dimension

The optimal embedding dimension depends on both the number of items and the underlying characteristics of the data. In our case, the dataset contains approximately 7,000 items suitable for training. Given this scale, we experiment with

embedding dimension dim_{embed} = {32, ... , 128}

These values strike a balance between model capacity and generalization: smaller dimensions may not capture enough semantic information, while larger dimensions risk overfitting and increased computational cost. Through experiments, we aim to identify the dimension that provides the best trade-off between performance and efficiency.

6.2.3 Threshold for item frequency

For word embeddings in NLP, tokens are typically required to meet a minimum frequency requirement $f > f_{thresh}$ to improve model quality and efficiency. This helps reduce noise from rare words that lack sufficient contextual information for meaningful representation. In recommender systems, however, even items that co-occur only once with others can still provide valuable signals for model training. During model evaluation, we will examine the dependence of model performance on item frequency.

6.2.4 Learning rate and batch size

Learning rate and batch size are critical hyperparameters that significantly influence model convergence and overall performance. A smaller learning rate may lead to more stable convergence but slower training, while a larger batch size can improve training efficiency but may reduce generalization.

Learning rate = {0.005, 0.01, 0.02}

Batch size = {8, 16, 32, 64, 128}

6.2.5 Loss function

The **Negative Sampling** method updates only a small number of negative samples along with the true target, significantly reducing computational cost. While it is fast, simple, and easily parallelizable, it relies on a stochastic approximation of the softmax function. As a result, it is sensitive to the choice of sampling strategy and does not produce outputs with a clear probabilistic interpretation. It is particularly suitable for models with a large output space (e.g., over 10,000 items), where computing the full softmax would be computationally prohibitive.

The **Hierarchical Softmax** is an efficient alternative to standard softmax when dealing with a large number of items. Instead of computing the probability distribution over all items (which is expensive), hierarchical softmax organizes items into a binary tree where each leaf represents an item. Predicting an item then becomes a matter of traversing the tree from the root to the leaf, making a sequence of binary decisions. This reduces the computational cost from O(V) to O(logV), where V is the number of items.

Considering our training data contains fewer than 10K items, and part of our focus is on rare items and their structure, hierarchical softmax is a more effective choice due to its ability to leverage shared paths in the tree, improving representation for infrequent items.

6.2.5 Similarity measure

Since our model learns embeddings that encode semantic relationships between items, **cosine similarity** is the most suitable similarity measure for our study. It effectively captures the orientation (rather than magnitude) of embedding vectors. This is particularly important in high-dimensional spaces, where the angle between vectors provides a more meaningful measure of similarity than Euclidean distance. This makes it especially well-suited for tasks such as *ranking, retrieval, and recommendation*.

Furthermore, cosine similarity can be used to help determine the optimal embedding dimension. If the dimension is too small, item vectors tend to cluster tightly together, resulting in uniformly high cosine similarity even before training, which limits the model's ability to distinguish between items. A good strategy is to choose an embedding dimension large enough so that cosine similarities between random item pairs are below **0.4** *before* training, while ensuring that co-occurring items exhibit high similarity, typically above **0.7-0.8**, *after* training.

Cosine Similarity Score	Interpretation
> 0.8	Very high similarity (strong recommendation candidates)
0.6 – 0.8	High similarity (related items, likely of interest)
0.4 – 0.6	Moderate similarity (some shared context)
< 0.4	Low or weak similarity

Table 4. Cosine similarity scores and their interpretation.

6.3 Hyperparameters

6.3.1 Embedding dimension

To determine the optimal embedding dimension, we measure the average similarity between item pairs prior to training across a range of dimension values. We select the embedding dimension at which the average similarity between the five most frequent items and their five nearest neighbors first drops below 0.4, ensuring that item similarities are not inflated due to an overly dense latent space. The value **72** is selected as the optimal embedding dimension based on the results shown in Figure 11.

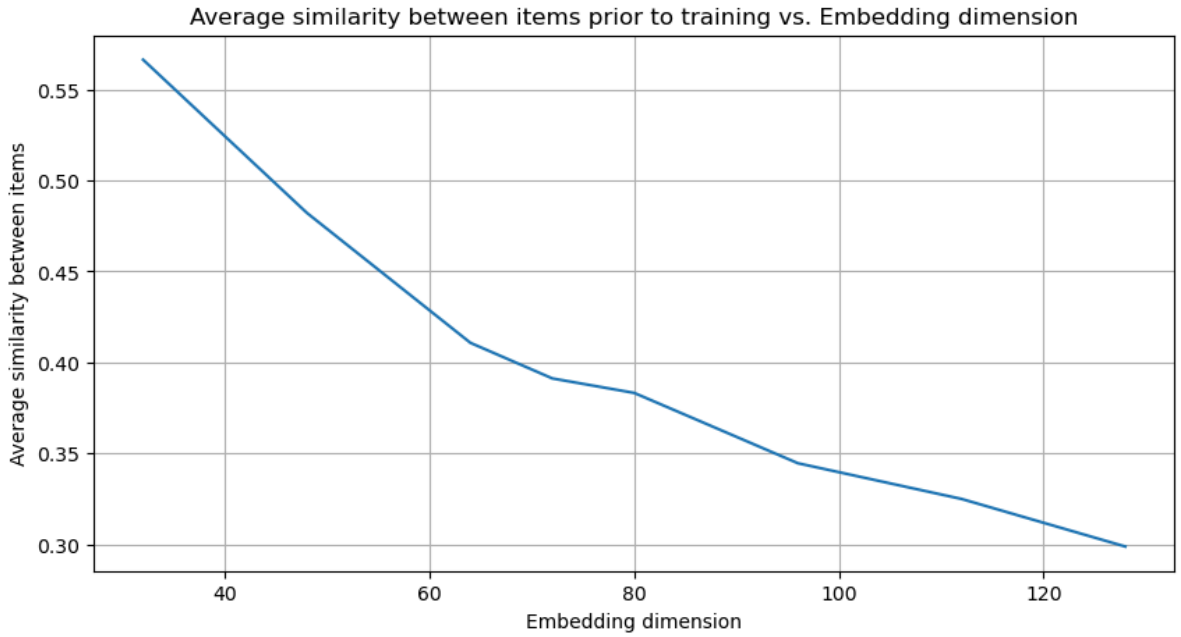

Figure 11. Average similarity between item pairs prior to training vs. embedding dimension.

6.3.2 Learning rate and batch size

We optimize the batch size and learning rate while keeping the regularization parameter λ_{cat} set to zero. Training and validation accuracy results for various learning rate and batch size combinations are presented in Figure 12. A batch size of **64** and a learning rate of **0.01** are identified as optimal.

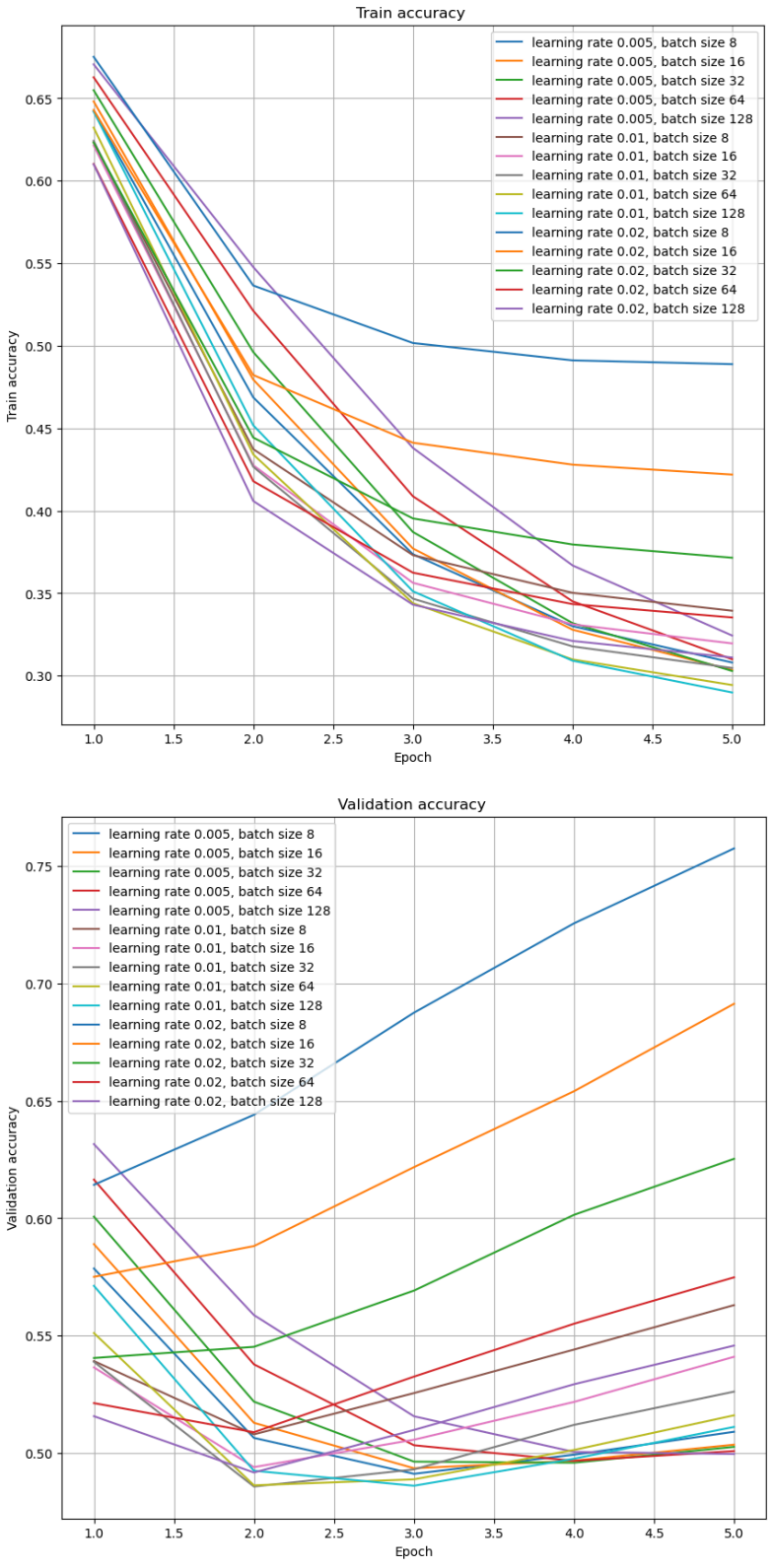

Figure 12. Training and validation accuracy across different combinations of learning rates and batch sizes.

6.3.3 Regularization parameter

We investigate the effect of different λ_{cat} values on recommendation performance. Fraction of recommended items in the top 15 that share the same category as the target item is measured with varying λ_{cat} values and shown in Figure 13.

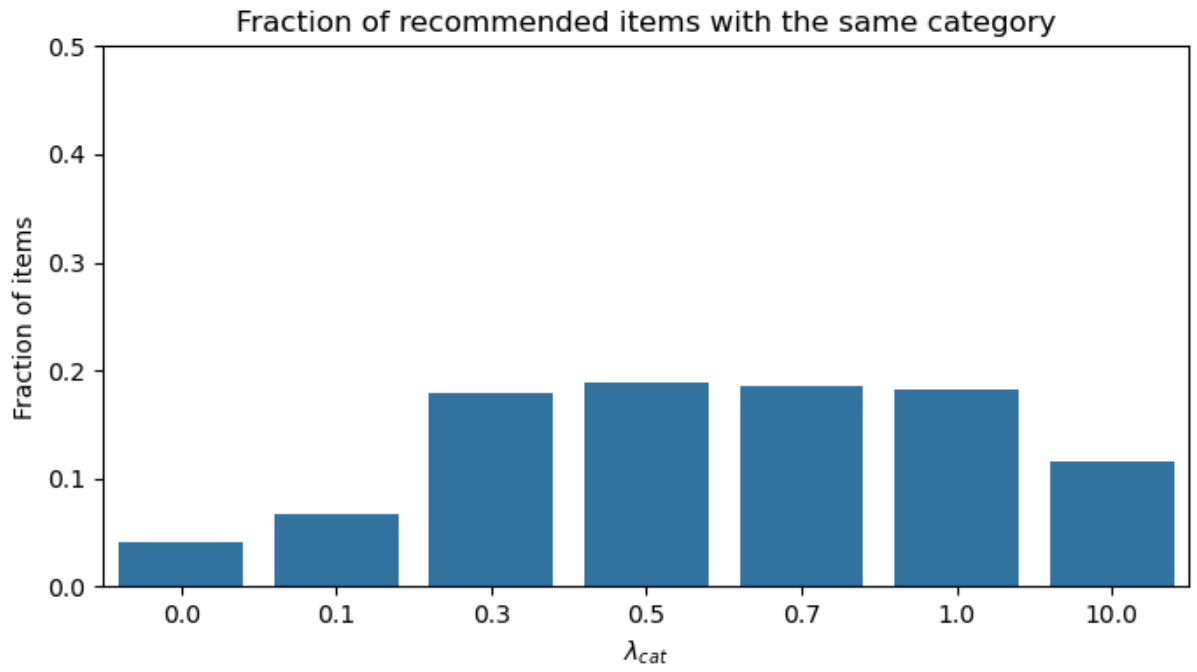

Figure 13. Fraction of top-15 recommendations within target item's category.

As expected, the fraction of items sharing the same category increases with the regularization strength up to a point. However, setting the parameter as high as 10 proves counterproductive, as it seems to suppress meaningful user-item interaction patterns.

7. Evaluation

7.1 Evaluation metrics

For problems at hand, ranking-based evaluation metrics such as Precision@K, Recall@K, Normalized Discounted Cumulative Gain (NDCG), MAP are most appropriate. Additionally, the similarity of the closest item serves as an effective evaluation metric.

☐ Recall at K $= \frac{Number\ of\ relevant\ items\ in\ top\ K}{Total\ number\ of\ relevant\ items}$

➔ Recall at K measures the proportion of all relevant items that are retrieved in the top k predictions. It is used when one cares about coverage, retrieving as many relevant items as possible.

☐ Precision at K = $\frac{Number\ of\ relevant\ items\ in\ top\ K}{K}$

➔ Precision at K measures the proportion of the top k predicted items that are relevant. It is used when one cares about the accuracy of the top-k results. Precision and Recall do **not** consider the relative ranking of items within the top-k prediction.

☐ NDCG at K = $\frac{Discounted\ Cumulated\ Gain\ (DCG)\ at\ K}{Ideal\ Discounted\ Cumulated\ Gain}$

, where DCG at K = $\sum_{i=1}^{K} \frac{Number\ of\ relevant\ items\ in\ top\ K}{log_2(i+1)}$

➔ NDCG accounts for both relevance and ranking order of items. More relevant items ranked higher give more gain. It is used when one cares not just about which items are retrieved, but how well they are ranked.

7.2 Models to evaluate

For evaluation, we select a moderate value of λ_{cat} = **0.1** to balance the contribution of the categorical loss in *Hierarchical Item2Vec*, and compare the results against a baseline *Item2Vec* model without regularization.

7.3 Ground truth co-occurrence frequency

To compute the evaluation metrics, we construct a co-occurrence frequency dictionary:

`{item1(i): [(item2(1):f`$_1$`), ... , (item2(m):f`$_m$`)]}.`

For each item `item1(i)` (where i = 1, …, n) in the item space, we compute the co-occurrence frequency f_j (where j = 1, …, m) defined as the number of times item `item2(j)` appears alongside other items.

7.4 Data grouping by item frequency

To better understand model performance across different item popularity levels, items are grouped based on their frequency of occurrence in the dataset (Table 5). This frequency-based segmentation allows us to evaluate how well the model performs on **frequent**, **moderately frequent**, and **rare** items. Such analysis is particularly important in recommendation systems, where models often perform well on popular items but struggle with long-tail or infrequent ones. By evaluating metrics such as precision, recall, or similarity within each frequency group, we gain deeper insight into the model's ability to generalize and handle data sparsity. This approach also highlights potential trade-offs between accuracy on frequent items and coverage of rare items.

Frequency	Group	Counts
0	Cold items	
1	Low frequency	4025
2-4	Moderate frequency	2884
5+	High frequency	638

Table 5. Item frequency-based data segmentation.

7.5 Results

The four evaluation metrics are computed on top K =15 items across three item frequency groups, and the results are presented in Figures 14 and 15 for both the Item2Vec model and Hierarchical Item2Vec model with a regularization parameter of 0.1.

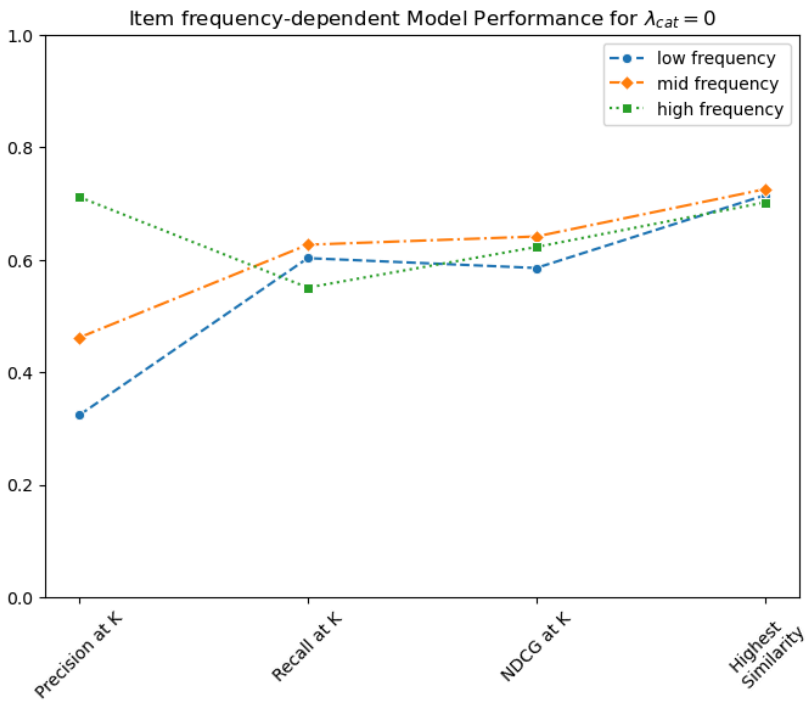

Figure 14. Item frequency-dependent model performance for baseline Item2Vec.

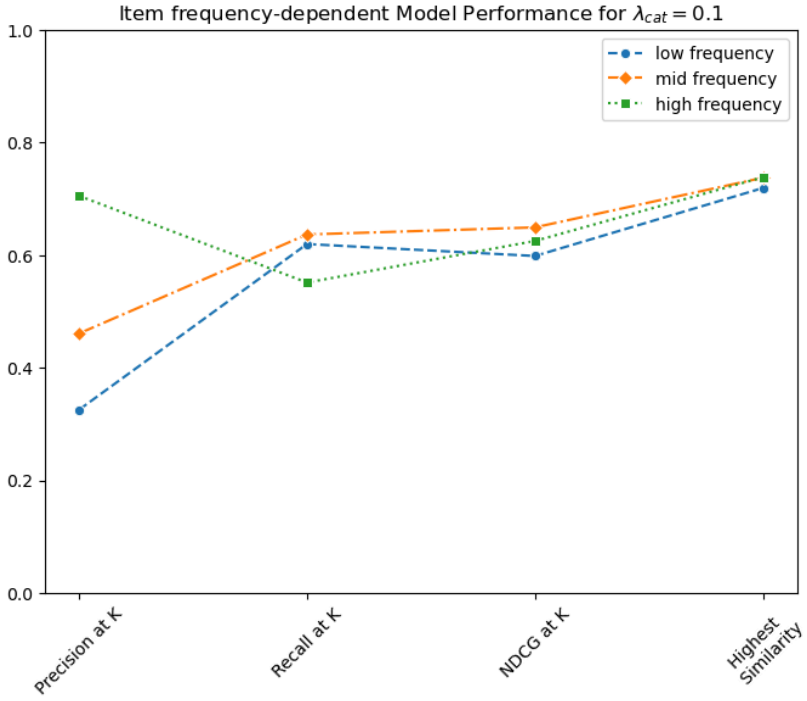

Figure 15. Item frequency-dependent model performance for Hierarchical Item2Vec with $\lambda_{cat} = 0.1$.

Precision at K tends to be lower for rare items, but this effect can be misleading: when fewer than K relevant items exist in the ground truth, even an ideal recommendation list cannot achieve high precision. Conversely, frequent items may show low recall at K, not because of poor recommendation quality, but because the number of relevant items exceeds K, limiting the achievable recall. In both cases, the fixed value of K imposes artificial constraints on the evaluation metrics.

NDCG at K, on the other hand, accounts for the internal ranking of items within the top-K recommendations and evaluates performance relative to the ideal ranking. This helps eliminate the artificial constraints present in precision at K and recall at K, making it a more robust metric across different item frequency groups.

The highest similarity score assesses the semantic closeness between a target item and its most similar recommended counterpart, providing insight into the model's ability to capture meaningful relationships. A similarity score of around 0.7 for both models suggests strong alignment in the embedding space, reflecting high-quality and semantically meaningful representations.

Figure 16 compares the overall performance of the two models. Hierarchical Item2Vec slightly outperforms Item2Vec, strengthening the alignments between relevant items. It also provides an effective method for handling cold items, such as by averaging the embeddings of all items within the same category.

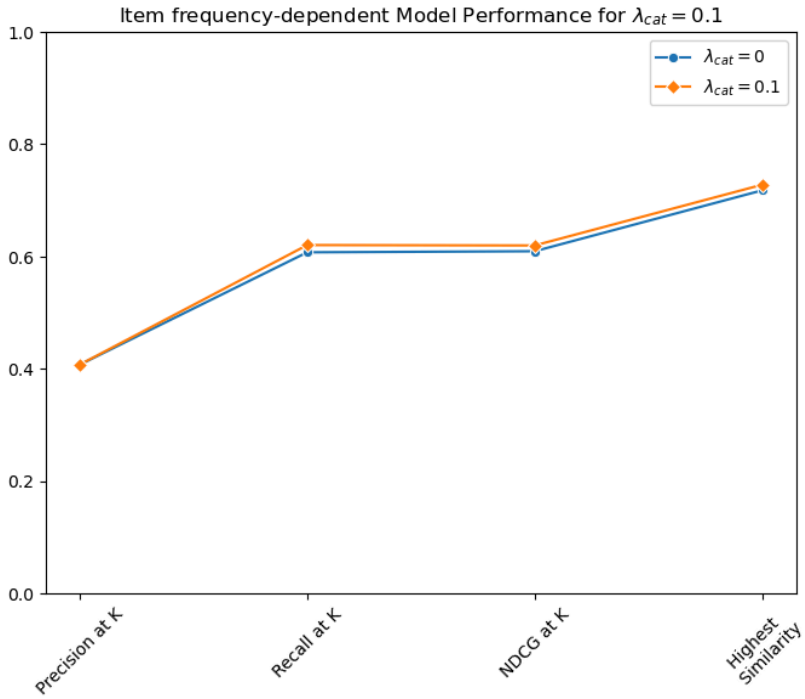

Figure 16. Overall performance comparison: Item2Vec vs. Hierarchical Item2Vec.

8. Summary and outlook

We built a recommender system that learns item embeddings using both baseline and hierarchical approaches, aiming to improve recommendation quality by capturing item relationships more effectively. Our models leverage transaction data to generate meaningful vector representations of items, which can then be used to predict and recommend related products.

This study compares the performance of Item2Vec and Hierarchical Item2Vec models in learning item embeddings for recommendation tasks. The evaluation is conducted using multiple metrics, including precision at K, recall at K, NDCG at K, and highest similarity scores, across different item frequency groups. While precision and recall metrics show limitations for rare and frequent items due to the varying number of relevant items, NDCG at K effectively addresses these issues by accounting for ranking positions relative to ideal recommendations.

Hierarchical Item2Vec demonstrates a slight performance advantage over the standard Item2Vec model by better capturing relationships between relevant items through the incorporation of hierarchical category information. Both models achieve high similarity scores, reflecting quality embeddings. Optimal training parameters, such as a batch size of 64 and learning rate of 0.01, contribute to effective model learning. Overall, the hierarchical approach enhances embedding quality and recommendation accuracy, particularly in aligning semantically related items.

Future work could explore incorporating temporal dynamics and user context to further refine recommendations. Investigating methods to better handle rare items and cold-start scenarios will also be important for enhancing recommendation coverage and robustness. Lastly, experimenting with adaptive evaluation metrics that account for variable relevance could provide a more nuanced assessment of model performance.