A Recommendation Approach to the Cold Start Problem ——Based on Implicit Feedback from Instacart Orders

Final Project Report for Project Group PG13

1st HUANG, 2nd LYU, 3rd YUAN, 4th YANG, 5th LI

***Abstract*—This project develops a personalized recommendation system for Instacart, targeting two core challenges in modern recommender systems: modeling implicit user feedback and addressing the cold-start problem. Using Instacart's large-scale public dataset, we engineered features at user, item, and interaction levels, and built a modular framework combining memory-based collaborative filtering, a hybrid LightFM model, and a semantic content-based approach using fine-tuned Sentence-BERT with FAISS indexing. To enhance collaborative filtering, we introduced optimizations such as time-decay and frequency-based weighting and SVD-based dimensionality reduction. Evaluated by Mean Precision@K, LightFM (tags-only) achieved the highest score (0.95 at K=5), while the SBERT+FAISS pipeline demonstrated strong performance in cold-start scenarios. Our results show that semantic and hybrid models significantly outperform traditional CF in sparse environments. While LightFM excels with interaction data, adding features provided no further gain—likely due to the absence of demographic user features. This project underscores that semantic-based methods (SBERT+FAISS) effectively address the cold-start problem for new items.**

I. Introduction

A. Context

Personalized recommendation systems are critical for e-commerce platforms, especially in the rapidly expanding online grocery sector. According to McKinsey's 2023 report [1], e-commerce accounted for 7.2% of all grocery spending in May 2023, which is over 35% higher than pre-pandemic levels. This shift underscores the growing consumer reliance on digital platforms for grocery shopping. Instacart, a leading online grocery delivery platform that allows customers to shop from local stores via a web or mobile app, exemplifies this trend by leveraging recommendation systems to enhance user experience, increase order frequency, and boost customer lifetime value. Our study utilizes Instacart's publicly available dataset on Kaggle, which includes over 3 million grocery orders from more than 200,000 users across approximately 50,000 products. Each user's historical data ranges from 4 to 100 orders, encompassing detailed metadata such as product categories, order timing, and reorder status. Due to the absence of explicit user ratings, the dataset primarily captures implicit user behaviors like reorder frequency and browsing patterns, making it suitable for developing recommendation strategies based on implicit feedback.

A significant challenge in such systems is the cold start problem, which arises when new users or products lack sufficient historical interaction data to support accurate recommendations. This issue is particularly salient on Instacart, where users often place only a few initial orders before determining long-term engagement, and where many products—such as seasonal items or limited-time promotions—have short lifespans with minimal prior data. Addressing the cold start problem in the context of implicit feedback is therefore central to ensuring early user retention and effective product discoverability.

Our research focuses on delivering meaningful recommendations for users with limited historical data by exploring effective techniques that operate on implicit feedback without access to user demographics or ratings. Through simulations of early-stage user behavior, we compared multiple recommendation techniques. We found that integrating semantic product information through Sentence-BERT embeddings provided meaningful improvements in cold start scenarios. These results highlight the importance of tailoring recommendation strategies to real-world platform conditions and demonstrate that implicit signals, when carefully modeled, can still enable effective personalization.

B. Use Case

1) Business Context: As users often reorder the same products and expect tailored suggestions, effective recommendation systems are crucial for enhancing user satisfaction and increasing revenue. On Instacart, a typical user session involves browsing, reordering frequently purchased items, and occasionally discovering new products through search or recommendation banners. These touchpoints represent key opportunities for recommendations to support product discovery and increase basket size.

One key operational challenge we aim to address is the cold start problem, where newly onboarded users or newly added products lack sufficient historical data for conventional recommendation algorithms. This issue is particularly important for Instacart because of the platform's specific vulnerability to high customer acquisition costs and declining retention rates. For example, while Instacart continues to invest heavily in acquiring new users, studies report that its month-six retention

rate has declined from approximately 21% to 15%, indicating that failure to engage new users early could result in significant wasted marketing expenditure [2].

To explore potential solutions within our recommendation framework, we implemented and evaluated two models targeting the cold start problem. First, we experimented with a hybrid matrix factorization model using LightFM, which incorporates user and item metadata to mitigate both user and item cold start scenarios. While preliminary results did not yield significant improvements, this approach builds a foundation for future enhancement using auxiliary features.

Separately, we developed a content-based recommendation pipeline leveraging Sentence-BERT and FAISS to compute semantic similarity between product names. This model addresses the item cold start issue by enabling recommendations based on semantic similarity, even without prior interactions. These two methods aim to improve early-stage user engagement and product discoverability, which are critical levers for maximizing return on acquisition investment and sustaining long-term growth.

2) Key Stakeholders/Users:

- **E-commerce platforms especially Instacart**: Our recommendation system tries to handle the cold start problem which improves product recommendation and may lead to potentially higher sales for e-commerce platforms especially Instacart .
- **Users especially new users**: We help them discover relevant products more efficiently, reduce search effort, and improve overall shopping satisfaction by solving the cold start problem.

3) Target Variable: The core objective of our project is to recommend products that a user is most likely to want in their next grocery delivery, based on implicit behavioral signals. Rather than predicting a binary label, we generate a relevance score for each candidate product and rank them accordingly.

We formulate this as a top-k ranking task, where the model selects the most relevant items from a candidate pool composed of previously purchased products or those frequently co-occurring with them. These relevance scores reflect the likelihood that a product matches the user's current intent, inferred from patterns in order timing, reorder frequency, and historical co-purchases.

A key focus of our approach is addressing cold start challenges, where recommendations must be generated despite minimal or no historical interaction data. To support this goal, we apply several models with different perspectives: item- and user-based collaborative filtering estimate item relationships based on interaction history; LightFM incorporates user and item metadata to support cold-start scenarios; and Sentence-BERT with FAISS captures semantic similarity between product names to recommend newly added items. Each model outputs a personalized ranked list evaluated using Mean Precision@K.

4) Time Window: Our recommendation task focuses on predicting products a user is likely to include in their next order. While we do not have explicit timestamps or order intervals in the dataset, we frame this as a next-event prediction task based on a user's historical purchasing behavior.

Rather than relying on time-based features or forecasting over a fixed interval (e.g., next 7 or 30 days), we aim to generate a ranked list of products that match a user's general preferences and reorder patterns. The underlying assumption is that users exhibit consistent shopping behaviors across sessions, even without knowing the exact timing of future purchases.

This approach allows the model to generalize across users with varying order frequencies, and supports practical use cases such as cart recommendations or homepage suggestions whenever the user returns to the platform.

C. Problem Statement

Instacart operates in a fast-moving online grocery market where platforms must deliver timely and accurate recommendations to sustain user engagement and drive revenue. However, two persistent challenges limit current recommender systems: the ambiguity of implicit feedback and the cold start problem.

Implicit signals such as clicks or past purchases are abundant but noisy—buying an item once does not always indicate genuine preference. Without careful modeling, systems can misinterpret these signals, resulting in irrelevant suggestions and reduced user satisfaction.

Cold start issues further hinder recommendation coverage for new users or products lacking historical interactions, which are common during user onboarding or the launch of seasonal items like holiday snacks or summer beverages. These high-turnover items are frequent in Instacart's dynamic catalog and especially vulnerable to under-recommendation without prior behavioral data.

In particular, our task involves generating top-k product recommendations for users based on recent behavioral signals.

To address these limitations, we develop a multi-model recommendation framework composed of independently implemented models, each targeting a specific aspect of the problem. LightFM incorporates metadata to improve recommendations in cold start scenarios. Sentence-BERT with FAISS supports a content-based pipeline that captures semantic similarity between product names to recommend unseen or infrequently purchased items. Additionally, we build user- and item-based collaborative filtering models that leverage implicit interaction patterns, adjusted using frequency and time-decay weighting.

While these models are evaluated independently, they provide complementary perspectives for assessing how different modeling strategies perform under varying levels of data sparsity and semantic availability.

D. Objectives

By the end of this project, our objective is to deliver a robust and scalable recommendation system for Instacart that addresses the dual challenges of implicit feedback modeling and cold start scenarios. The system aims to improve product

discoverability, personalize the user experience, and increase order volume.

To achieve this, we define the following SMART objectives:

- Build a modular recommendation framework to generate top-k product recommendations, consisting of four independently implemented models: a hybrid model (LightFM), a content-based method (Sentence-BERT + FAISS), and two collaborative filtering baselines. These models are evaluated under different data conditions using Mean Precision@K to assess performance trade-offs.
- Enhance LightFM performance to address cold start scenarios by incorporating user/item metadata and frequency-based features, performing hyperparameter tuning, and optimizing top-k accuracy.
- Enable item cold start recommendations by using Sentence-BERT to generate semantic embeddings of product names and FAISS to retrieve similar items in the absence of behavioral interactions.

These objectives align closely with business needs, including improved onboarding experience, reduced user churn, and increased catalog visibility. The final system is designed to deliver cold-start-resilient recommendations grounded in behavioral signals and product semantics.

E. Assumptions

The following assumptions were made to guide the methodology and scope of our project:

- Data quality and representativeness: We assume the Instacart dataset is sufficiently clean and complete for modeling purposes, and that its structure (e.g., reorder labels, order timestamps) reasonably reflects real-world user behavior.
- User preference inference: We assume past purchasing behavior reflects user preferences to a reasonable extent, although not all purchases indicate strong intent or satisfaction.
- Collaborative filtering assumptions: Memory-based CF assumes that users with similar past behavior will exhibit similar future preferences [3], which underpins our similarity-based recommendation methods.
- Cold start handling: For cold-start products, we assume that product names carry sufficient semantic information to support embedding-based similarity using Sentence-BERT.
- Implicit feedback signals: We treat reorder behavior as a strong implicit proxy for user preference in the absence of explicit ratings, enabling collaborative and hybrid methods to operate effectively.

II. Related Work

A. Implicit Feedback

Mingyue et all. [4] proposed SoftRec with GRU4Rec, a new RS optimization framework to enhance item recommendation from implicit feedback. Authors aimed to improve the method for the one-hot vectors, such as normalize the popularity distribution with temperature softmax. The model overperformed but it was limited by the softmax loss function. Hu et all. [5] introduced a factors Collaborative Filtering model which is especially tailored for implicit feedback. While more accurate than content based techniques, CF suffers from the cold start problem, which content based approaches would be adequate. In our work, We apply stratified sampling to include a more balanced set of candidate items across product categories, which improves diversity and allows the model to generalize better to users with less mainstream preferences.

B. Cold-Start Recommendation

Zhang et all. [6] applied embedding-based MF-based model. Authors generated the user embedding vectors and item embedding vectors using the user-item interaction matrix. Despite the high accuracy, these methods often require extensive computational resources, having weak practicality for larg-scale datasets. Wei et all. [7] introduced a new objective function and a simple and efficient cold-start recommendation framework CLCRec based on contrastive learning. Their approach primarily focused on collaborative filtering while ignore rich metadata. Chen et all. [8] proposed a graph neural network to patch GNP for cold start, a dual-functional GNN, yet such methods faced scalability challenges in real-time recommendation scenarios. In our work, Unlike Wei et all. [7] that relies only on interaction data, we integrate multi-level features in LightFM model and maintain interpretability. Besides, FAISS-based Sentence-BERT in our work updates achieve sub-second latency, addressing the efficiency-accuracy trade-off.

C. Hybrid Collaborative Filtering

Kula [9]introduced LightFM, a hybrid model combining collaborative filtering (CF) and metadata embeddings to handle cold-start. While this approach significantly improved coverage in sparse datasets, its performance heavily relies on metadata quality. Jiang et al. [10]proposed xLightFM, using tensor decomposition to reduce embedding dimensions. Despite their work achieved more memory savings on large-scale datasets, the precision decreased for niche items because of the simplified feature interactions. Patoulia et al. [11]explored the CF methods in e-commerce. However their work ignored hybrid approaches and underperformed in cold-start problem. In our work, our feature ablation ensures the metadata robustness compared with Kula's LightFM. Extending Patoulia's CF analysis, we prove that hybrid models perform better than content-based baselines.

III. Methodology

Our project followed a structured, iterative methodology as illustrated in Fig 1. The detailed description is as follows.

- Step1: Topic Selection
  - Determined the project topic as e-commerce recommendation system.
- Step2: Literature Review
  - Established domain knowledge of recommendation system.

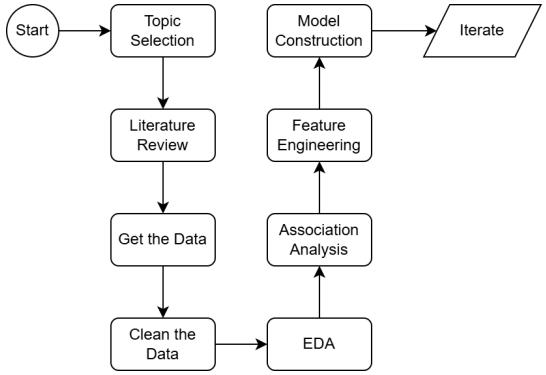

Fig. 1: Project Process Outline

  - Identified research issues on recommendation system.
- Step3: Data Collection
  - Decided Instacart Order Data as source data.
  - Identified stakeholder as Instacart and use case of recommendation.
  - Identified problem statement of implicit feedback and cold start problem.
- Step4: Data Cleaning
  - Rigorous data validation and cleaning.
  - Made the project plan, including EDA, feature engineering and recommendation model construction.
- Step5: Comprehensive Analysis
  - Explored data distribution and user behavior through in-depth exploratory data analysis (EDA).
  - Studied relationship between different products with association analysis such as Aprior and FP-growth.
- Step6: Feature Engineering
  - Extracted interactive information from implicit feedback through feature crossing.
  - Undertook classified statistic to expanded original 6 features into 31 new features, which consist of three dimensions: users, items, and user-item interactions.
- Step7: Collaborative Filtering Recommendation
  - Established baseline recommendation model using user-based and item-based collaborative filtering to find the most similar k products for recommendation based on user-item interaction matrix.
  - Optimized collaborative filtering by stratified sampling, different weighting methods on elements of interaction matrix, using Singular Value Decomposition(SVD) to remove noise and sparsity, applying K-nearest weighted average on recommenders to enhance diversity.
  - Determine Precision@K as the major evaluation indicator.
- Step8: LightFM Hybrid Recommendation Based on Implicit Feedback
  - Established advanced LightFM by combining user-item interaction matrix with features from feature engineering to recommend products based on implicit feedback.
- Step9: Sentence Bert + FAISS Recommendation for Cold Start Problem of New Products
  - Established Sentence Bert and FAISS model to find the most similar k products for recommendation merely based on product's name and aisle, solving cold start problem of new products.
  - Fine tuned Sentence Bert by constructing positive and negative sample pairs for input and minimizing cosine similarity loss function.
- Step10: Model Evaluation and Optimization
  - Evaluated four recommendation systems using Mean Precision@K metric.
  - Repeated step2 to step10 for project improvements.

A. Code Repository and Replication Instructions

For the structure of our code repository, our project is organized into the following folders and files, hosted on our private GitHub repository (`https://github.com/nus-st5188/ay2425-s2-pg13`).

- `data/`
  Contains raw data files used throughout the project. There are 6 files in csv format: aisles, departments, order_products__prior, order_products__train, orders and products.
- `figure/`
  Stores output plots generated from exploratory data analysis.
- `results/`
  Contains intermediate data files and saved model outputs, including products features, user features, configuration files of fine-tuned Sentence Bert, embedding results from fine-tuned Sentence Bert, recommendation results.
- Jupyter Notebooks
  The number prefix indicates the running order.
  - 01-eda.ipynb: Exploratory Data Analysis, including distributions, purchase behavior, top items/users and product features.
  - 02-feature.ipynb: Data Cleaning, feature engineering and construction of input features used in collaborative filtering and hybrid models.
  - 03-ItemSimilarity_Recommendation.ipynb: Baseline Item-based collaborative filtering recommendation logic, including model construction and evaluation.
  - 04-User-based CF for Instacart Recommendation.ipynb: Baseline User-based collaborative filtering recommendation logic, including model construction and evaluation.
  - 05-data analysis yangqing.ipynb: Additional analysis, likely supporting earlier feature design or validation.
  - 06-lightfm.ipynb: Use LightFM (hybrid matrix factorization) with user-item interactions and metadata for recommendation.
  - 07-Fine Tuned Sentence Bert+FAISS.ipynb: Fine-tuning of a Sentence-BERT model using PyTorch on

product names. And builds FAISS index to find the most similar k products.

– 08-SBERT+FAISS evaluation.ipynb: Evaluates FAISS similarity search results using precision@K.

In terms of replication instructions, configuration information is listed in Table VII. Dependencies are listed in `requirement.txt`, which includes all Python packages required for running the project. For your convenience, we have included the configuration information in the appendix.

IV. Data Collection and Pre-processing

A. Data Collection

The dataset is from Kaggle: Instacart Market Basket Analysis. Instacart is a popular online grocery delivery platform. The dataset consists of six tables: orders, order_products__train, order_products__prior, products, aisles, and departments. The dataset includes over 32 million orders made by approximately 200,000 users,49,688 unique items, 134 aisles, and 21 departments. Meanwhile, the purchasing behavior are implicit feedback such as add to cart, reorder, and the number of days since the previous order, etc.

The choice of the Instacart dataset is due to two reasons: Firstly, despite having 30 million purchase records, the dataset still contains items with less than one purchase and users with only a few interactions, which creates the cold start problem. Meanwhile, when a new user arrives or a new product is introduced, how to recommend based on Instacart? Secondly, the dataset provides clear product-level features and large implicit feedback, which makes it an ideal dataset for developing methods to tackle the cold start problem.

B. Data Variables

TABLE I: Product-level features after feature constructing

No.	Variable Name	Description	Data Type
1	product_id	Product ID	int
2	p_users_unique	Total unique users of a product	int
3	p_users_total	Total times the product was ordered	int
4	p_reordered_total	Total times the product was reordered	int
5	p_reordered_percentage	Reorder percentage of a product	float
6	p_avg_cart	Product's avg add-to-cart-order	float
7	p_order_first_cnt	Times product was first purchased	int
8	p_order_second_percent	Second purchase rate of the product	float
9	aisle_id	Aisle ID	int
10	d/a_users_unique	Unique users of aisle/Department	int
11	d/a_users_total	Total orders of aisle/Department	int
12	d/a_reordered_total	Total reorders of aisle/Department	int
13	d/a_reordered_percentage	Reorder percentage of the aisle/Department	float
14	d/a_avg_cart	Avg. add-to-cart position of aisle/Department	float
15	department_id	Department ID	int
15	product_name	Product name	object
15	department	Department name	object
15	aisle	Aisle name	object

TABLE II: User-level features after feature constructing

No.	Variable Name	Description	Data Type
1	user_id	User ID	int
2	u_avg/std_dow	User's avg/std day-of-week of order	float
3	u_orders_total	Total orders by a user	int
4	u_products_total	Total products user has bought	int
5	u_products_unique	Unique products purchased	int
6	u_reordered_total	User's total reordered products	int
7	u_reordered_percentage	User's overall reorder percentage	float
8	u_avg_order_size	Avg. order size of a user	float
9	u_avg_reordered_orders	Avg. reorders per order	float

TABLE III: User-Product-level features after feature constructing

No.	Variable Name	Description	Data Type
1	u_p_avg_cart	Avg. add-to-cart position for this product	float
2	u_p_avg_days_since_prior	Avg. days between purchases of the product	float
3	u_p_orders_total	Total times user ordered this product	int
4	u_p_reordered_total	Total reorders of product by the user	int
5	u_p_reordered_percentage	Reorder rate of this product by the user	float
6	u_p_last_order	Order number in which product last appeared	int
7	is_reorder_3	Whether reordered in the last 3 orders	float

The features listed above are constructed specifically for our model. The original features from the dataset can be found in Table VI of the appendix and the feature engineering process will be introduced in detail later.

C. Exploratory Data Analysis (EDA)

The dataset contains no missing values, except for certain reorder indicators which are absent by products that were not reordered by users. Therefore, we did not perform any imputation on these fields.

We set out to uncover temporal purchasing patterns.Fig. 2 reveals that orders are distributed relatively evenly across the week, with Sunday showing the highest volume of sales. Fig. 3 highlights the order distribution throughout one day, showing notable fluctuations. The data illustrates a spike in order volume from around 10 a.m. to 4 p.m, while activity significantly diminishes between 2 a.m. and 6 a.m. In summary, our analysis indicates that the peak purchasing period is between 10 a.m. and 4 p.m. on Sundays, while the lowest activity is between 2 a.m. and 6 a.m. on Wednesdays.

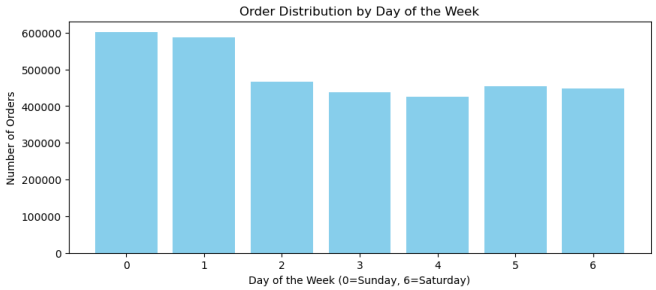

Fig. 2: Order Distribution by Day of the Week

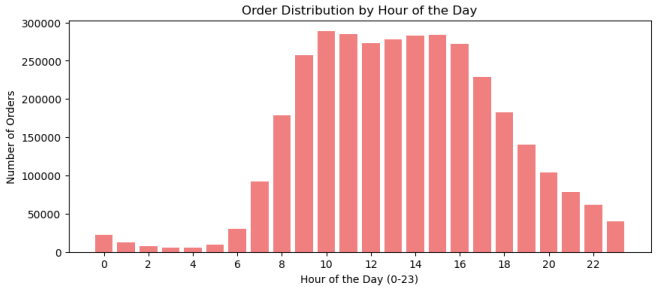

Fig. 3: Order Distribution by Hour of the Day

To facilitate a clearer understanding of products and departments, we construct visualization figures. The departments are

illustrated in Fig. 5 in the appendix and the product names are presented in Fig. 6.

Additionally, we apply association rule to explore potential relationships between products. We apply both the Apriori and FP-Growth algorithms to generate association rules from the last 500,000 unique orders. To cover of products across all departments, we filter products within each department by computing a composite score based on reorder rate and purchase frequency. The top 100 products per department are selected and subsequently merged to form the candidate set for association rule. As Apriori algorithm needs to scan the entire dataset and wastes a lot of time, we try to use FP-Growth, particularly for large datasets, by employing chunking and sparse matrix representation. We identify several interesting product associations. For example, customers who purchase boneless skinless chicken breasts are also likely to buy banana, honeycrisp apple, and organic avocado.

D. Data Pre-processing

In the data preprocessing phase, we use feature creation, binning of features, and apply PCA for dimensionality reduction. During the PCA stage, we test 99% and 99.9% variance, which significantly reduce the feature dimensions and allowe for faster processing of the training data.For further details, see the Feature Engineering section.

V. Core Concept, Methods, and Models

A. Feature Engineering

To build a strong predictive model, we constructed features from three levels: product-level, user-level, and user-product interaction level, also incorporating aisle and department metadata. We prepare the data using the Prior dataset. Use an inner join between *order_products_prior* and *orders*, keeping only the orders labeled as 'prior'. Then join with the *products* data.

To recognize the product popularity and reordering trends, we compute ratio of total reorders to total purchases for each product, and counts of first and second time purchases, help us assess product loyalty and retention. The average add-to-cart position reflect how widely a product is accepted and how visible it is in the cart. Finally, we add aisle and department-level features into product features.

To capture user behavior patterns, we compute the aggregated statistics, included the user's average day of the week and hour of ordering, average time gap between orders, and the total number of orders. We also captured diversity and intensity in purchasing through the number of unique and total products bought, average order size, and overall reorder rates. To account for recent behavior, we further measured reorder tendencies within the user's last three orders.

To enrich the historical interactions, we constructed the cross features that describe how each user interacts with each specific product. To account for recent behavior, we further measured reorder tendencies within the user's last three orders.

We performed feature engineering (Table VI, Table II and Table III) through binning and PCA, and then added user features, item features, and interaction features into LightFM.

B. Memory-based Collaborative Filtering Model Construction

Memory-based CF algorithms, including the user-based CF and the item-based CF, utilize the complete or a sample of the original user-item database to generate predictions. Each user belongs to a group of individuals with similar interests. By identifying the so called neighbors of the active user, a forecast of preferences on new items for him or her can be generated [12]. Specifically, if the task is to obtain a recommendation list containing N items for the active user, collaborative filtering will firstly figure out k most similar users or items after calculating the similarities within user-product dataset, subsequently aggregating the k neighbors to get the N most possible items as the final recommendations.

To achieve this, this neighborhood-based CF algorithm can be divided into two main steps:

1) **Step1 Similarity Computation**: calculate the similarity or weight $w_{u,v}$, which reflects correlation, distance, or weight, between two users or two products, u and v.
2) **Step2 Prediction and Recommendation Computation**: produce predictions for the particular user through aggregating all the ratings of the k most similar users or items [13].

1) Similarity Calculation: Similarity calculation within users or products is a significant component of collaborative filtering algorithms. The similarity between two documents(users or items) can be accessed by representing each document as a vector of word frequencies and employing different techniques to calculate the weights between these frequency vectors [14]. Formally, let $R \in \mathbb{R}^{m \times n}$ indicates a $m \times n$ user-item interaction matrix, therefore the similarity between user u and user v can be calculated with the n dimensional vectors corresponding to the uth and vth row of interaction matrix R; and the similarity between product i and product j, can be identified using the m dimensional vectors corresponding to the ith and jth column of matrix R.

First, to accurately capture the relationships between users and items of Instacart dataset, our project employs three different methods to construct the $m \times n$ interaction matrix, and compare their performances in subsequent predictions. Manifested below is the table IV summarizing these matrices:

TABLE IV: Different user-item interaction matrix

Method	Formula	Captures
Binary	$R_{ui} = 0, 1$	Order or not
Frequency-based	$F_{ui} = \text{count}(u, i)$	Purchase intensity
Time-decay-based	$W_{ui} = \sum_k e^{-\lambda \Delta t_{ui}(k)}$	Recency-weighted behavior

Since the raw binary matrix treats all purchases equally, it fails to capture many implicit preferences of users which are essential for subsequent personalized recommendation suggestions. To address this limitation, we incorporate frequency-weighted and time-decay weighting methods as optimizations into our model. And frequency weighting can highlight frequently purchased items reflecting personal interest, while time-decay weighting reflects the user's current interests by

prioritizing recent behavior over older actions. Both methods significantly enhance model performances in later experiments.

Second, after getting the matrix and corresponding vectors, we try different techniques to calculate the similarities in user-based model and item-based model.

1) **user-based CF algorithm**:
For the user-based CF algorithm, we calculate the **Cosine-Based Similarity**, $w_{u,v}$, between the user vectors $\vec{u}$ and $\vec{v}$:

$$w_{u,v} = \cos(u,v) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \times \|\vec{v}\|} \tag{1}$$

$$= \frac{\sum_{i \in I} R_{iu} R_{iv}}{\sqrt{\sum_{i \in I} R_{iu}^2} \sqrt{\sum_{i \in I} R_{iv}^2}} \tag{2}$$

where R_{iu} denotes the uth row and ith column document of the matrix R; $\|\vec{u}\|$ denotes the L_2 norm of vector $\vec{u}$.

2) **item-based CF algorithm**:
For the item-based CF algorithm, we compute the Correlation-Based Similarity to measure the extent to which two variables linearly relate with each other [5]. For item i and j, the **Pearson Correlation Similarity** $w_{i,j}$:

$$w_{i,j} = \frac{\sum_{u \in U} (r_{u,i} - \bar{r}_i)(r_{u,j} - \bar{r}_j)}{\sqrt{\sum_{u \in U} (r_{u,i} - \bar{r}_i)^2} \sqrt{\sum_{u \in U} (r_{u,j} - \bar{r}_j)^2}} \tag{3}$$

where $r_{u,i}$ is the rating of user u on item i, $\bar{r}_i$ is the average rating of the ith item by those users [13]. As a matter of fact, Pearson correlation similarity generates cosine similarity while normalizing the item ratings across users. Therefore, the Pearson correlation can yield negative values, while cosine similarity will not produce such results.
The estimated preference scores for an active user a over all candidate items are computed as a linear combination of items the user has previously interacted with, weighted by their similarities with other items:

$$\hat{\vec{r}}_a = \vec{r}_a \cdot S \tag{4}$$

Here, $\vec{r}_a$ is the user's implicit interaction vector (e.g., purchase counts), and S is the item-item similarity matrix computed using Pearson correlation. This approach focuses on item affinity and bypasses the need for user-user comparison.

3) **Truncated SVD-Based Similarity Computation**:
However, Instacart is a extremely large-scale dataset, which causes the user-item matrix sparse and high-dimensional. This sparsity makes it difficult to compute meaningful user similarities, and increase the computational cost during training. To handle this problem, we introduce the Singular Value Decomposition (SVD) as a dimensionality reduction technique in user-based CF model. By projecting users and items into a lower-dimensional latent space, SVD helps to uncover hidden patterns in user behavior rather than solely rely on co-rated items. This results in more robust and improves the quality of similarity computations. The detailed process is :

- Step1: SVD decomposes the initial matrix as:

$$\tilde{R} \approx U_k \Sigma_k V_k^\top \tag{5}$$

Where $U_k \in \mathbb{R}^{m \times k}$ denotes user latent feature matrix (left singular vectors); $\Sigma_k \in \mathbb{R}^{k \times k}$ denotes diagonal matrix of top-k singular values; $V_k \in \mathbb{R}^{n \times k}$ denotes item latent feature matrix (right singular vectors) ;$k \ll \min(m,n)$ is the number of components (dimensionality of latent space)

- Step2: keep only the top-k components for dimensionality reduction. The user embedding is:

$$F = U_k \Sigma_k \tag{6}$$

Where $F \in \mathbb{R}^{m \times k}$ is the reduced user latent representation matrix.

- Step3: calculate the Cosine Similarity using Latent Space vectors:

$$\text{sim}(u,v) = \frac{F_u \cdot F_v}{\|F_u\| \cdot \|F_v\|} \tag{7}$$

Where F_u is the latent vector for user u; $\|F_u\|$ is L_2 norm of F_u

- Step4: Get the User Similarity Matrix $S \in \mathbb{R}^{m \times m}$, where $S_{uv} = \text{sim}(u,v) \in [-1,1]$ denotes the similarity score between users u and v.

2) Prediction and Recommendation Generation: Identifying personalized recommendations for the active user is the most critical step in a collaborative filtering algorithm. In the neighborhood-based CF algorithm, a subset of nearest neighbors of the active user are chosen based on the similarities calculated in the Similarity Computation part. Then, CF will generate a series of recommendations for the active user with a weighted aggregation of these neighbors' ratings [15]. In our project, we implement the "Top-N Recommendation with k nearest neighbors" techniques to achieve the aggregation.

To trade off granularity for speed and relevance, making it more practical in large-scale Instacart dataset, we adopt a more focused and computationally efficient strategy: Top-N recommendation with k nearest neighbors which selects the top-k most similar users for the active user and generates recommendations based on the items that these close neighbors have interacted with. Manifested below is the clear workflow of the Recommendation technique generating a top-N item list:

- Firstly determine the k most similar users (closest neighbors) to the active user a by utilizing the cosine similarities between the active user and other users.
- After identifying the k most similar neighbors, their respective rows in the user-item matrix R are combined

to derive a set of estimated ratings over all items by the active user a:

$$\hat{\vec{r}}_a = \frac{\sum_{j \in U_a} w_{a,j} \cdot \vec{r_j}}{\sum_{j \in U_a} w_{a,j}} \tag{8}$$

where $\vec{r_j}$ is the rating over all items by user j; set U_a is the K-nearest neighborhood of the active user a which can be filtered through the top-k rank of all similarities between active user a and others.

- With the estimated rating $\hat{r}_a$, user-based collaborative filtering technique then will suggest the top-N most frequent items as the final recommendations by ranking their rating vectors.

However, the Top-N recommendation algorithm still has limitations related to scalability and real-time performance [16].

In summary, the item-based approach closely resembles the user-based method in overall structure, as both utilize a top-N recommendation system based on a similarity matrix calculated from the user-item interaction matrix. However, there are some notable distinctions in their detailed architectures. Firstly, and most importantly, the methods differ in how similarities are calculated—cosine similarity is used for users in the user-based CF model, while the item-based CF model employs Pearson correlation for item vectors, then filtered by minimum co-occurrence and popularity. Secondly, the aggregation methods vary: user-based collaborative filtering computes a weighted sum of the interactions from neighboring users, whereas item-based one propagates user preferences through similar items via matrix multiplication. Lastly, the user-based model additionally integrates several optimization techniques, including stratified sampling instead of top-N filtering, the addition of time-decay weights to the interaction matrix, and singular value decomposition to compress the vectors before calculating similarities, significantly enhancing its performance.

C. LightFM Model Construction

LightFM is a hybrid model which combines collaborative filtering (based on user behavior) and content-based filtering (based on features).It has three components: user-item interaction matrix,User and item features.

The cold start problem happens when a new user or a new item appears in the system, but there is no interaction data for them, so the model doesn't know what to recommend. LightFM solves this by using user or item features.

The following is description of the LightFM algorithm: User representation q_u is obtained by summing the embeddings of all features associated with that user:

$$q_u = \sum_{f \in f_u} \mathbf{e}_f^U \tag{9}$$

Similarly, the item representation p_i is computed as:

$$p_i = \sum_{f \in f_i} \mathbf{e}_f^I \tag{10}$$

Each feature also contributes a scalar bias. The total bias for a user u and an item i is given by:

$$b_u = \sum_{f \in f_u} b_f^U, \quad b_i = \sum_{f \in f_i} b_f^I \tag{11}$$

The model predicts the strength of interaction between a user and an item by combining their vector dot product with the bias terms:

$$\hat{y}_{ui} = \sigma(q_u^\top p_i + b_u + b_i) \tag{12}$$

where $\sigma(\cdot)$ denotes the sigmoid function, suitable for binary outcome modeling.

LightFM solves cold start problem by estimating embeddings from metadata features, and when only indicator features are used, its performance aligns with traditional matrix factorization [9].

Before building the model, we perform feature engineering to process the features, which has been explained in detail earlier, so it will not be repeated here.We built four LightFM-based models with different feature combinations:

- Model 1: Only interaction feature
- Model 2: Interaction features and user features
- Model 3: Interaction features and item features
- Model 4: Interaction features, user features, and item features

We use grid search to tune the hyperparameters and select the best model based on precision.

D. Sentence BERT and FAISS Model Construction

Collaborative filtering algorithms, which are frequently used in recommendation system, usually rely on the item's interactions to make recommendations. However, there is little or no interaction information on new products or new users, which leads to the cold start problem.

Even without interaction information, we can still make recommendations based on product information, such as product's name and category. In this project, we employed Sentence BERT (SBERT) model to vectorize product information and used FAISS model to find the most similar k products for recommendation.

1) Sentence BERT: Bidirectional Encoder Representations from Transformers (BERT) model is a bidirectional transformer model pretrained using a masked language modeling objective in deep learning. Structurally, it stacks encoders from Transformers to generate an embedding for each input word. Compared with other natural language processing models, such as TF-IDF, Word2Vec, etc., BERT has obvious advantages in contextualized embedding, transfer learning, full-scale understanding on sentence meaning, due to its bidirectional attention mechanism and pre-training. Although BERT excels at understanding natural language, it is not optimized for generating semantically meaningful sentence embeddings directly using standard distance metrics.

Sentence-BERT (SBERT) [17] overcomes this limitation by adapting the BERT architecture into siamese and triplet

network frameworks, enabling the generation of semantically rich sentence embeddings that are directly comparable using cosine similarity. This transformation significantly accelerates similarity searches, reducing computation time from approximately 65 hours with BERT or RoBERTa to around 5 seconds using SBERT while preserving the original accuracy [17].

Architecturally, SBERT introduces a pooling layer on top of BERT/RoBERTa outputs to produce fixed-size sentence embeddings. It offers three pooling strategies: utilizing the CLS-token output, averaging all token embeddings (MEAN strategy), and applying a max-over-time operation (MAX strategy). By default, the MEAN strategy is employed. During fine-tuning of BERT/RoBERTa, siamese or triplet networks are used to adjust the model weights, ensuring the resulting embeddings capture semantic meaning and remain compatible with cosine similarity comparisons. Depending on the task, SBERT supports three types of objective functions: classification, regression, and triplet loss.

In the training stage in our project, we chose MEAN-strategy of pooling which can aggregate information from all the input. For the training data, we followed siamese and triplet network structure by constructing positive and negative samples based on product's name and aisle, where product pair with the same aisle was labled positive while with different aisles was labeled negative. As for the objective function, given that our aim with SBERT is to generate embeddings for product names in order to retrieve the top k most similar products via similarity search, we selected the regression objective for this project. This objective function utilizes the cosine similarity between two sentence embeddings u and v (Fig. 4), optimized through a mean squared error loss function.

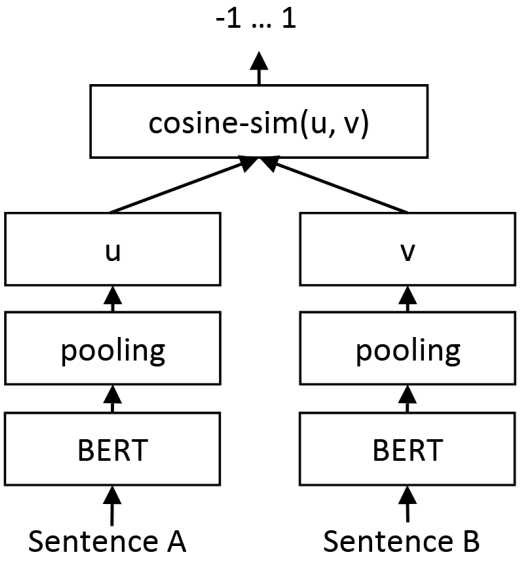

Fig. 4: The SBERT architecture for similarity scores computation.The same structure is used when applying the regression objective function. [18].

2) FAISS: After obtaining embeddings from SBERT, we used FAISS to search for the most similar k products for every anchor product.

FAISS is specifically designed to perform approximate nearest neighbor (ANN) searches by representing data points as vectors and rapidly identifying similar items. It significantly outperforms traditional methods in terms of speed and memory efficiency through the use of advanced techniques such as quantization, indexing structures, and optimized distance calculations, allowing it to scale effectively to large datasets. The FAISS workflow consists of two primary steps: index construction and search. Several indexing strategies are available, including Flat, Locality-Sensitive Hashing (LSH), Hierarchical Navigable Small World (HNSW), Inverted File Index (IVF), etc. The search process is aligned with the chosen indexing method.

For the purpose of model comparison, we used one of the basic index in baseline FAISS, which is Flat. Flat performs an exhaustive search, which means it compares the query vector to every vector in the database to find the most similar ones. It is the only index that can guarantee exact results and provides the baseline for results for the other indexes. Flat does not compress the vectors, and does not add overhead on top of them. It also does not require training and does not have parameters.

VI. Evaluation of Results and Findings

A. Evaluation Metrics

To quantitatively assess the performance of our recommendation models, we employed a commonly used evaluation metric- Mean Precision@K to figure out the best recommendation model. Specifically, Precision@K measures the proportion of recommended items in the top-K list that are actually relevant to the user. It is defined as:

$$\text{Precision@}K = \frac{|\text{Recommended}_K \cap \text{Relevant}|}{K} \tag{13}$$

where Recommended_K indicates the set of top-K items recommended to the user in model; Relevant represents the set of items that are truly relevant or that the user has interacted with in the test set; and K is the number of top items considered.

Since Precision@K is just computed per user, we use Mean Precision@K to summarize performance across all test users which can reduce the random bias and give more accurate evaluation to recommendation models. Mean Precision@K is defined as the average of individual Precision@K values:

$$\text{Mean Precision@}K = \frac{1}{|U|} \sum_{u \in U} \text{Precision@}K_u \tag{14}$$

where U is the set of all users in the test set, and $\text{Precision@}K_u$ is the corresponding Precision@K for user u.

A higher Mean Precision@K indicates that the model performs better.

B. Results and Insights

TABLE V: Mean Precision@K scores of Different Recommendation Models

Model	k=5	k=10	k=20
Item-based CF	0.0400	0.0200	0.0100
User-based CF (stratified sampling + svd + time-decay weights)	0.1995	0.1300	0.0805
Lightfm(tags)	0.9500	0.9500	0.9300
Lightfm(tags + ids)	0.0600	0.0600	0.0300
Lightfm(tags + uds)	0.0700	0.0600	0.0100
Lightfm(tags + about)	0.0500	0.0500	0.0100
SBert+FAISS	0.6016	0.5696	0.5336

The Mean Precision@K scores presented in the table V show variations in the recommendation performance across different models. The traditional item-based CF model yields relatively low precision, while user-based CF enhanced with optimization techniques, including stratified sampling, svd, time-decay weighting, demonstrates substantial improvements compared with the item-based one. Among the models tested, the LightFM with only tag features achieves the highest precision, whereas incorporating additional metadata (e.g., ids, uds, about) results in drops in performance. Although we explored different strategies for feature construction, selection, and engineering, the performance improvement remained limited and gradual. The mainly reason about the change is that the metadata does not include original user attributes such as age, gender, or location which allow for clear differentiation between users. The user features we constructed are solely based on implicit shopping behavior, as we do not have access to users' personal information due to privacy limitations. The SBERT+FAISS model shows strong performance overall, demonstrating the effectiveness of semantic representation in capturing item similarities. These findings align with our objective to solve the cold start problem in Instacart dataset because the SBERT+FAISS pipeline consistently yield higher scores compared to traditional CF models that are often hindered by cold start issues.

C. Lessons Learned from Analysis

Our results presented in Table V highlight the critical impact of the cold start problem on recommendation accuracy in Instacart dataset. Despite applying multiple enhancements, traditional memory-based CF models performed poorly compared to LightFM and SBERT+FAISS, reinforcing the necessity of hybrid and content-based approaches for handling sparse user-item interactions. We also found that not all engineered metadata improved model performance, suggesting that limited implicit behavior data and basic item attributes are insufficient. This insight points to the need for richer external pre-trained user embeddings to enhance personalization. Finally, although SBERT+FAISS demonstrated strong performances, its high computational cost posed a practical challenge. These findings underscored the importance of balancing model complexity, scalability, and available resources when designing real-world recommender systems.

VII. Discussion of Challenges & Lessons Learned

A. Key Challenges

In this section, we will outline threee main challenges encountered during the project and the strategies we implemented to overcome them.

1) Data Sparse Issue:
   - One of the primary challenges we faced in the project was handling the sparsity of the Instacart dataset which contains millions of orders from over 200 thousand users and a vast majority of users just purchasing a small set of items. This caused the raw collaborative filtering models struggling to suggest accurate recommendations due to the presence of many low-frequency users and products.
   - To mitigate this issue, we adopted several strategies: Firstly, we filtered out low-frequency users and products when applying CF and LightFM models to reduce matrix size. Secondly, we introduced the SVD, ALS, known as the matrix factorization methods, to compress a user-item matrix into a more informative, low-dimensional representation in terms of latent factors. These ultimately enhanced both the accuracy and scalability of our model.
2) Diversity and Long-Tail Bias:
   - Another challenge we encountered is the long-tail problem. Because in the early stages of CF models, we applied a filtering step to remove low-frequency products and it unintentionally introduced a popularity bias. The CF models tended to over-recommend popular items and failed to capture personalized interests, especially those related to niche or less frequently purchased products.
   - To mitigate this issue, on the one hand we revised our preprocessing pipeline by adopting a stratified sampling approach instead of top-n filtering which allowed us to preserve some long-tail signals while still controlling for matrix sparsity. On the other hand, we later introduced the LightFM hybrid model, which incorporates both collaborative and content-based signals. This helped in promoting more diverse recommendations, as LightFM can leverage item metadata and user features to make predictions even for rarely interacted items.
3) Team Collaboration and Task Coordination:
   - At the beginning of this semester, since everyone selected different courses and had hers own business, we faced big challenges in aligning our team members' schedules and maintaining consistent progress. Especially, due to miscommunication and unclarified responsibilities, we failed to write a qualified progress

report which disappointed us and reminded us to make some changes.
- To address these issues, we established a clear communication routine after the midterm by holding weekly meetings to set goals, assign tasks, and track progress. We also encouraged open discussion and flexibility among members to balance individual workloads. Over time, this fostered a more efficient workflow and stronger collaboration, which significantly contributed to the completion of our final deliverables.

B. Lessons Learned

Throughout this project, we encountered several practical challenges—from data sparsity and long-tail bias to coordination within a diverse team. These experiences offered valuable insights that will guide our future work.

First, we learned the importance of early planning and flexible communication within a team. After facing coordination issues early in the semester, we adopted regular meetings, clearer role assignments, and more structured workflows, which greatly improved our efficiency and team morale. Second, we gained a deeper understanding of the trade-offs between personalization and popularity bias in recommendation systems. In real-world scenarios like the Instacart dataset, where user behavior is heavily skewed toward popular products, it is critical to balance diversity and relevance. Our work with hybrid models and thoughtful sampling strategies taught us the value of preserving long-tail signals to deliver more personalized and meaningful recommendations. This project not only enhanced our technical proficiency in recommendation systems but also fostered essential project management and teamwork skills that will benefit us in both academic and professional contexts.

VIII. Conclusion and Future Work

A. Conclusion

In this project, we set out to address two long-standing challenges in personalized recommendation systems for online grocery platforms like Instacart: one is the ambiguity of implicit feedback signals and the other is the cold start problem for new users and items. These issues are particularly crucial in a dynamic retail environment where accurate recommendations can directly affect customer satisfaction, retention, and overall sales performance.

To tackle these problems, we proposed a multi-model recommendation framework that integrates memory-based collaborative filtering, hybrid matrix factorization (LightFM), and semantic content-based retrieval using Sentence-BERT and FAISS. Each approach targeted different aspects of the recommendation challenge: enhanced user-based CF served as a strong baseline with various optimizations such as SVD, frequency or time-decay weighting, and KNN aggregation; LightFM model leveraged engineered user and item features to mitigate data sparsity; SBERT + FAISS provided semantic matching capabilities for cold-start items.

Our experimental results, summarized in Table V, confirmed that traditional collaborative filtering models struggle under extreme data sparsity, despite optimization efforts since there is a huge gap between Mean Precision@K scores of memory-based CF and LightFM or SBERT + FAISS pipeline. Besides, the LightFM model, using interaction matrix, achieves the highest Mean Precision@K score, while it fails to improve performance with adding features. There are two reasons about the conclusion. Firstly, metadata does not include original user features such as age, gender, location, etc. Secondly, LightFM performs better in AUC instead of Mean Precision@K [9]. Meanwhile, the SBERT + FAISS pipeline proves effective in handling cold start scenarios by utilizing semantic textual similarity, offering a viable alternative when historical interactions are unavailable.

By directly aligning our implementation with the SMART objectives laid out at the beginning of the report, our project not only demonstrated the strengths and weaknesses of various recommendation paradigms, but also provided actionable insights into designing scalable and cold-start-resilient systems for real-world e-commerce applications.

The key contributions of this project work include: (1) a comparative analysis of collaborative filtering, hybrid, and content-based recommenders under implicit feedback constraints; (2) novel preprocessing and optimization techniques tailored to Instacart's large-scale, sparse dataset; and (3) practical design principles for feature engineering and model selection under resource limitations.

Looking ahead, this project lays a solid foundation for future improvements. More advanced evaluation metrics such as Normalized Discounted Cumulative Gain (NDCG), Mean Average Precision (MAP), and Recall@K which could assess both the relevance and the ranking quality of outputs will offer a more nuanced understanding of model quality beyond Mean Precision@K. Additionally, incorporating richer contextual signals—like user session patterns, temporal trends, or pretrained cross-domain embeddings—could further improve personalization and robustness.

Ultimately, this experience deepened our understanding of building real-world recommendation systems, striking a balance between model complexity, data sparsity, and computational feasibility. Our results offer a promising direction for platforms seeking to enhance user engagement and product discovery in large-scale online retail environments.

B. Recommendations for Future Work

Based on our project's outcomes, several promising directions can be pursued to improve our recommendation performances, address unsolved challenges in cold start problem and explore new fields. Manifested below are some main insights:

- **Adopt Advanced Model-based Approaches:** We can explore more model-based approaches and hybridization techniques to advance the recommendation accuracy. Model-based approaches include the Matrix Factorization with Bayesian Personalized Ranking (BPR), Deep Autoencoders, or Neural Collaborative Filtering (NCF),

which may address data sparsity and capture complex user-item interactions better than our models [19]. Hybridization techniques like cascading or ensemble models may also improve robustness.

- **Enhance Cold Start Solutions for Users:** While we incorporated Sentence-BERT and FAISS to mitigate the item cold-start problem, strategies such as user-side embedding generation, demographic-driven modeling, or interactive onboarding surveys could be explored to address the new-user cold-start issue more effectively.
- **Incorporate Richer User-side Features :** Since integrating implicit behavioral feature in the LightFM model shows limited improvements. Our future extension may focus on leveraging pre-trained user embeddings from external sources in e-commerce platforms or incorporating multi-view learning techniques which could further enhance user representation and improve model expressiveness, particularly in sparse settings where behavioral features alone are insufficient.
- **Explore More Comprehensive Evaluation Metrics:** Currently, our evaluation relies solely on Mean Precision@K to measure the quality of recommendations which neglected some important factors when evaluating recommendations, such as the ranking position, diversity, and recall. In future works, we may try to incorporate a broader set of metrics, including Recall@K, Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (NDCG) which could assess both the relevance and the ranking quality of outputs more accurately. In addition, some diversity-oriented metrics such as Intra-list Diversity will also be considered.
- **Scalability optimization for large datasets:** Distributed training frameworks (e.g., PySpark, Dask) and approximate nearest neighbor search (e.g., Annoy, ScaNN) could be leveraged to improve scalability and training speed for large-scale datasets like Instacart.

IX. Group Member Contributions

Our project was driven by a collaborative and task-oriented structure, with each member taking initiative in different stages of the workflow. Below is a structured overview of individual contributions, organized according to major milestones in the project lifecycle.

1) Project Ideation & Proposal Development
   - Yang: conducted background research and identified the Instacart dataset.
   - Yuan & Lyu& Li: collaborated on framing the problem statement, defining project objectives, and selecting evaluation metrics in proposal.
   - Huang: coordinated the planning process and distributing tasks for each member.
2) Exploratory Data Analysis (EDA)
   - Yang: Performed order-level analysis, including purchase frequency, repurchase rate, and product popularity; applied K-Means for RFM modeling and conducted association rule mining.
   - Lyu: Optimized memory usage of dataframes; carried out user-level analytics and variable exploration (e.g., reorder ratios vs. user count).
3) Progress Report & Midterm Presentation
   Yuan, Li, and Huang: collaboratively refined the project's direction and updated key components, including objectives, task planning, and model development status. They jointly contributed to the writing of the midterm progress report.
4) Model Construction & Tuning
   - Lyu: Engineered item and user features later used in the LightFM hybrid model.
   - Yuan: Built the item-based collaborative filtering pipeline and tested its recommendation performance.
   - Li: Developed the user-based collaborative filtering model and introduced multiple optimization strategies (e.g., stratified sampling, frequency and time-decay weighting, svd and k-nearest).
   - Yang: Implemented the LightFM model and performed hyperparameter tuning for better scalability and accuracy.
   - Huang: Constructed and fine-tuned the SBERT + FAISS pipeline to tackle the item cold-start problem.
5) Evaluation & Performance Analysis
   Both Li and Huang conducted model evaluation using Mean Precision@K as evaluation, performed comparative analysis, and visualized key benchmarking results to support interpretation and final conclusion.
6) Final Report & Documentation
   - Yuan: Wrote introduction part, including contexts, problem statement, use case and project assumptions.
   - Lyu: Composed related work section and described feature engineering in detail.
   - Huang: Documented methodology of the project, and introduced SBERT + FAISS model construction.
   - Li: Wrote collaborative filtering model construction, evaluation and results section, key challenges, and future directions.
   - Yang: Described data preprocessing steps and wrote the LightFM modeling section.

Through this project, all members not only strengthened their technical and analytical skills, but also gained valuable experience in interdisciplinary collaboration, task management, and problem solving under real-world constraints.

References

[1] McKinsey & Company. (2023) *The State of Grocery in North America 2023*, *McKinsey Insights*.

[2] Goodwater Capital. (2023). *Understanding Instacart: Serving the best grocery customers in the industry*. Goodwater Thesis.

[3] Goldberg, K., Roeder, T., Gupta, D., & McCallum, A. (2001). Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, *4*, 133–151.

[4] Cheng, Mingyue, et al. “Learning recommender systems with implicit feedback via soft target enhancement.” *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 2021.

[5] Hu, Yifan, Yehuda Koren, and Chris Volinsky. “Collaborative filtering for implicit feedback datasets.” *2008 Eighth IEEE international conference on data mining.* Ieee, 2008.

[6] Zhang, Yiding, et al. “Geometric disentangled collaborative filtering.” *Proceedings of the 45th international ACM SIGIR conference on research and development in information retrieval.* 2022.

[7] Wei, Yinwei, et al. “Contrastive learning for cold-start recommendation.” *Proceedings of the 29th ACM international conference on multimedia*. 2021.

[8] Chen, Hao, et al. “Graph Neural Patching for Cold-Start Recommendations.” *Australasian Database Conference. Springer, Singapore*, 2025.

[9] Kula, M. (2015). Metadata embeddings for user and item cold-start recommendations. *arXiv preprint arXiv:1507.08439*.

[10] Jiang, Gangwei, et al. “xLightFM: Extremely memory-efficient factorization machine.” *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 2021.

[11] Patoulia, Agori Argyro, et al. “A comparative study of collaborative filtering in product recommendation.” *Emerging Science Journal* 7.1 (2023): 1-15.

[12] Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, *2009*(1), 421–425.

[13] Sarwar, B. M., Karypis, G., Konstan, J. A., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web* (pp. 285–295). ACM.

[14] Salton, G., & McGill, M. (1983). *Introduction to modern information retrieval*. McGraw-Hill.

[15] Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 230–237). ACM.

[16] Karypis, G. (2001). Evaluation of item-based top-N recommendation algorithms. In *Proceedings of the International Conference on Information and Knowledge Management* (pp. 247–254). ACM.

[17] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *arXiv preprint arXiv:1908.10084*.

[18] Adomavicius, G., & Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, *17*(6), 734–749. https://doi.org/10.1.1.107.2790

[19] Blanda, S. (2015, May 25). Online recommender systems—How does a website know what I want? *American Mathematical Society*. Retrieved October 31, 2016, from https://www.ams.org/publicoutreach/feature-column/fcarc-recommender

[20] Castells, P., Hurley, N. J., & Vargas, S. (2015). Novelty and diversity in recommender systems. In F. Ricci, L. Rokach, & B. Shapira (Eds.), *Recommender systems handbook* (2nd ed., pp. 881–918). Springer. https://doi.org/10.1007/978-1-4899-7637-6_26

[21] Fleder, D., & Hosanagar, K. (2009). Blockbuster culture’s next rise or fall: The impact of recommender systems on sales diversity. *Management Science*, *55*(5), 697–712. https://doi.org/10.1287/mnsc.1080.0974

APPENDIX

A. Extended Description of Dataset Variables

The original data variables are displayed as follows.

TABLE VI: Original Data variables

No.	Variable Name	Description	Data Type
1	product_id	Product ID	int
2	order_id	Order ID	int
3	user_id	User ID	int
4	aisle_id	Aisle ID	int
5	department_id	Department ID	int
6	product_name	Product name	object
7	department	Department name	object
8	aisle	Aisle name	object
9	order_dow	Day of week of the order	int
10	order_hour_of_day	Hour of the order	int
11	days_since_prior_order	Days since the previous order	float
12	order_number	Order sequence of the user	int
13	add_to_cart_order	Add-to-cart position for the product	int
14	eval_set	Dataset type the order belongs to (prior, train, or test)	object

B. EDA Visualizations

Distribution of departments shows the relative size and popularity of each category:

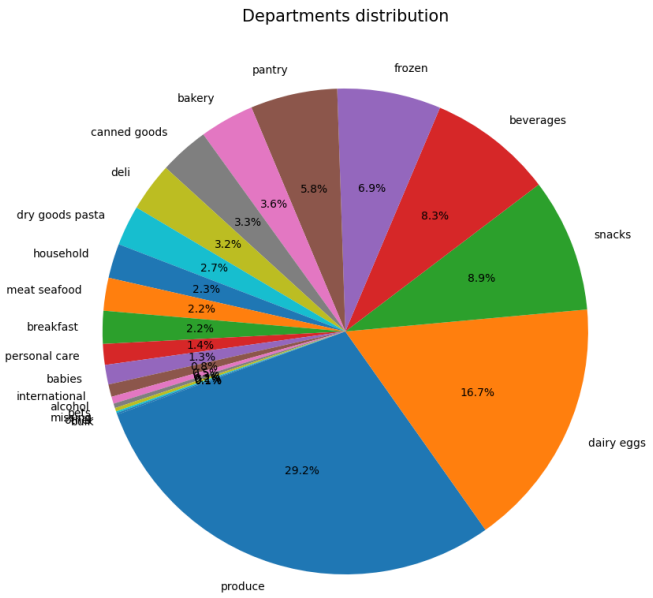

Fig. 5: Departments Distribution

Distribution of products highlights their overall frequency of purchase:

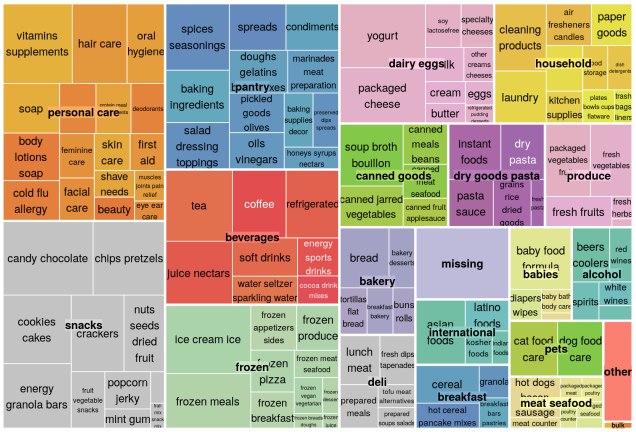

Fig. 6: Products Distribution

C. Environment Configurations and Code Snippets

TABLE VII: Configuration Information

Name	Configuration Information
Operation System	Windows 11
Development Language	Python 3.12.3
Framework	Pytorch 2.6.0+cpu
CPU	Intel(R) Core(TM) i5-10210U
Memory	12G

1) Environment Setting: Dependencies are listed in `requirement.txt`, which includes all Python packages required for running the project. We use following codes to install the exact versions specified. In addition, make sure that Microsoft Visual Studio 2022 and Microsoft C++ Build Tools is installed before installing `lightfm` package.

```
# Create and activate a virtual environment (optional but
    recommended)
conda create -n recommender_env python=3.12
conda activate recommender_env

# Install dependencies
pip install -r requirement.txt
```

2) Data Preprocessing:

```
# step 1:Prepare the data using the Prior dataset. Use an
    inner join between order_products__prior and orders,
    keeping only the orders labeled as 'prior'. Then join
    with the products table
df_prior = pd.merge(order_products_prior, orders,
    on='order_id', how='inner')
df_prior = pd.merge(df_prior, products, on='product_id',
    how='left')

df_prior['times_product_user_ordered'] =
    df_prior.groupby(['user_id', 'product_id']).cumcount() +
    1

dict_agg1 = {
    'user_id': [lambda x: x.nunique()],
```

```
    'reordered': ['count', 'sum', 'mean'],
    'add_to_cart_order': ['mean'],
    'times_product_user_ordered': [lambda x: sum(x == 1),
        lambda x: sum(x == 2)]
}

prod_f1 = df_prior.groupby('product_id').agg(dict_agg1)


prod_f1.columns = ['p_users_unique', 'p_users_total',
    'p_reordered_total',
                   'p_reordered_percentage', 'p_avg_cart',
                   'p_order_first_cnt', 'p_order_second_cnt']

prod_f1.reset_index(inplace=True)
prod_f1.head()

prod_f1['p_order_second_percent'] =
    prod_f1.p_order_second_cnt / prod_f1.p_order_first_cnt

dict_agg2 = {
    'user_id':[lambda x: x.nunique()],
    'reordered':['count', 'sum', 'mean'],
    'add_to_cart_order': ['mean']
}
ailse_f = df_prior.groupby('aisle_id').agg(dict_agg2)
ailse_f.columns = ['a_users_unique', 'a_users_total',
    'a_reordered_total',
                   'a_reordered_percentage', 'a_avg_cart']
ailse_f.reset_index(inplace=True)
ailse_f.head()

depart_f = df_prior.groupby('department_id').agg(dict_agg2)
depart_f.columns = ['d_users_unique', 'd_users_total',
    'd_reordered_total',
                   'd_reordered_percentage', 'd_avg_cart']
depart_f.reset_index(inplace=True)
depart_f.head()

prod_f1 = prod_f1.merge(products, on='product_id', how='left')
prod_f1 = prod_f1.merge(aisles, on='aisle_id', how='left')
prod_f1 = prod_f1.merge(ailse_f, on='aisle_id', how='left')
prod_f1 = prod_f1.merge(depart_f, on='department_id',
    how='left')
prod_f1 = prod_f1.merge(departments, on='department_id',
    how='left')
prod_f1.head()

#free some memory
del ailse_f, depart_f, aisles, departments
gc.collect()

#User level features
#(1) u_mean_dow :User's average and std day-of-week of order
#(2) u_std_dow
#(3) u_avg_hour :User's average and std hour-of-day of order
#(4) u_avg_days_since_prior:User's average and std
    days-since-prior-order
#(5) u_orders_total: Total orders by a user
#(6) u_products_unique: Total unique products user has bought
#(7) u_products_total:Total products user has bought
#(8) u_reordered_total: user's total reordered products
#(9) u_reordered_precent:User's overall reorder percentage
#(10) u_avg_order_size: Average order size of a user
#(11) u_avg_reordered_orders:User's mean of reordered items
    of all orders
dict_agg3 = {
    'order_dow':['mean','std'],
    'order_hour_of_day':['mean','std'],
    'days_since_prior_order':['mean','std'],
    'order_number':[lambda x: x.nunique()],
    'product_id':['count', lambda x: x.nunique()],
    'reordered':['sum', 'mean']
}

u_f = df_prior.groupby('user_id').agg(dict_agg3)
u_f.columns = ['u_mean_dow', 'u_std_dow', 'u_avg_hour',
    'u_std_hour',
               'u_avg_days_since_prior',
                   'u_std_days_since_prior',
               'u_orders_total', 'u_products_total',
                   'u_products_unique',
               'u_reordered_total', 'u_reordered_percentage']
u_f.reset_index(inplace=True)
u_f.head()

u_f2 = df_prior.groupby(['user_id',
    'order_number']).agg({'reordered':['count', 'mean']})
u_f2.columns = ['u_avg_order_size', 'u_avg_reordered_orders']
u_f2.reset_index(inplace=True)
u_f3 =
    u_f2.groupby(['user_id']).agg({'u_avg_order_size':['mean'],
    'u_avg_reordered_orders':['mean']})
u_f3.columns = ['u_avg_order_size', 'u_avg_reordered_orders']
u_f3.reset_index(inplace=True)
u_f3.head()

u_f = u_f.merge(u_f3, on='user_id', how='left')
u_f.head()

# (12) Percentage of reordered itmes in user's last three
    orders
#(13) Total orders in user's last three orders
u_last_three_orders =
    u_f2.groupby('user_id')['order_number'].nlargest(3).\
    reset_index() # extract the last three orders from u_f2
u_last_three_orders.head()
u_last_three_orders = u_f2.merge(u_last_three_orders,
    on=['user_id', 'order_number'], how='inner') # merge the
    last three orders with u_f2
u_last_three_orders['rank'] =
    u_last_three_orders.groupby('user_id')['order_number'].\
    rank('dense', ascending=True)
u_last_three_orders_f = u_last_three_orders.pivot_table(index
    = 'user_id', columns = ['rank'],
    values=['u_avg_order_size',
    'u_avg_reordered_orders']).reset_index(drop = False)
u_last_three_orders_f.columns = ['user_id','orders_3',
    'orders_2', 'orders_1', 'reorder_3', 'reorder_2',
    'reorder_1']
u_last_three_orders_f.head()

u_f = u_f.merge(u_last_three_orders_f, on='user_id',
    how='left')
u_f.head()

# User and Product level features:
#(1) u_p_avg_cart: User's avg add-to-cart-order for a product
#(2) u_p_avg_days_since_prior: User's avg
    days_since_prior_order for a product
#(3) u_p_reordered_total,u_p_reordered_percentage: User's
    product total orders, reorders and reorders percentage
#(4) User's order number when the product was bought last
```

```
dict_agg5 = {
            'add_to_cart_order': ['mean'],
            'days_since_prior_order': ['mean'],
            'reordered': ['count', 'sum', 'mean'],
            'order_number': ['max']
            }
u_prod_f = df_prior.groupby(['user_id',
    'product_id']).agg(dict_agg5)
u_prod_f.columns = ['u_p_avg_cart',
    'u_p_avg_days_since_prior',
                    'u_p_orders_total','u_p_reordered_total',
                        'u_p_reordered_percentage',
                    'u_p_last_order']
u_prod_f.reset_index(inplace=True)
u_prod_f.head()

u_last_three_orders.head()

last_three_orders = df_prior.merge(u_last_three_orders,
    on=['user_id', 'order_number'], how='inner')
last_three_orders.head()
last_three_orders['rank'] =
    last_three_orders.groupby(['user_id',
    'product_id'])['order_number'].rank("dense",
    ascending=True)

product_last_three_orders =
    last_three_orders.pivot_table(index = ['user_id',
    'product_id'],columns='rank', values =
    'reordered').reset_index()
product_last_three_orders.columns = ['user_id', 'product_id',
    'is_reorder_3', 'is_reorder_2', 'is_reorder_1']
product_last_three_orders.fillna(0, inplace = True)
product_last_three_orders.head()

u_prod_f = u_prod_f.merge(product_last_three_orders,
    on=['user_id', 'product_id'], how='left')
u_prod_f.head()
u_prod_f.fillna(0, inplace = True)

# Saving all features
prod_f1.to_pickle(save_path+'product_features.pkl')
u_f.to_pickle(save_path+'user_features.pkl')
u_prod_f.to_pickle(save_path+'user_product_features.pkl')
```

3) ItemSimilarity Recommendation:

```
# Creating User Product Df
user_product_df =
    common_products.pivot_table(index=['user_id'],
    columns=['product_name'], values="u_p_orders_total")
user_product_df.head()

df_train = pd.merge(order_products_train, orders,
    on='order_id', how='inner')
df_train = pd.merge(df_train, products, on='product_id',
    how='left')

dict_agg = {
            #'add_to_cart_order': ['mean'],
            #'days_since_prior_order': ['mean'],
            'reordered': ['count']
            #'order_number': ['max']
            }
u_prod_f_train = df_train.groupby(['user_id',
    'product_id']).agg(dict_agg)
u_prod_f_train.columns = ['u_p_orders_total']
u_prod_f_train.reset_index(inplace=True)
u_prod_f_train = pd.merge(u_prod_f_train,
    products[['product_id', 'product_name']],
    on='product_id', how='left')
u_prod_f_train.head()

def compute_filtered_item_similarity(user_product_df,
    min_common_users, min_product_popularity):
    """
    Build item-item similarity matrix, keep similarities if
        items have at least min_common_users
    Skip items bought by fewer than min_product_popularity
        users
    """
    # Calculate how many users bought each product
    product_popularity = user_product_df.notna().sum()
    popular_items = product_popularity[product_popularity >=
        min_product_popularity].index.tolist()

    filtered_corr = pd.DataFrame(index=popular_items,
        columns=popular_items, dtype=float)

    for i, item_i in enumerate(tqdm(popular_items,
        desc="Computing item similarities")):
        vec_i = user_product_df[item_i]

        for j in range(i, len(popular_items)):
            item_j = popular_items[j]
            vec_j = user_product_df[item_j]

            mask = vec_i.notna() & vec_j.notna()
            common_users = mask.sum()

            if common_users >= min_common_users:
                x = vec_i[mask].values
                y = vec_j[mask].values
                if np.std(x) > 0 and np.std(y) > 0:
                    corr = np.corrcoef(x, y)[0, 1]
                    filtered_corr.at[item_i, item_j] = corr
                    filtered_corr.at[item_j, item_i] = corr

    return filtered_corr

item_similarity_matrix = compute_filtered_item_similarity(
    user_product_df,
    min_common_users=50,
    min_product_popularity=1000 # Skip rare products that
        only been bought 1000 times
)

class ItemBasedRecommender:
    def __init__(self, user_product_df, similarity_matrix):
        self.user_product_df = user_product_df.fillna(0)
        self.similarity_matrix = similarity_matrix

    def recommend_products(self, user_id, n=10):
        user_vector = self.user_product_df.loc[user_id]
        scores = self.similarity_matrix.dot(user_vector).\
            sort_values(ascending=False)

        purchased = set(self.user_product_df.\
            loc[user_id][self.user_product_df.loc[user_id] >
                0].index)
        scores = scores.drop(labels=purchased,
            errors='ignore')
```

```
        return scores.head(n).index.tolist()

user_gt_dict = u_prod_f_train.groupby("user_id")["product_name"].\
    apply(set).to_dict()

class RecommenderEvaluator:
    def __init__(self, ground_truth, recommendations):
        self.ground_truth = ground_truth
        self.recommendations = recommendations

    def mean_precision_at_k(self, k=10):
        precisions = []
        for user, recs in self.recommendations.items():
            gt = self.ground_truth.get(user, set())
            hit_count = sum([1 for item in recs[:k] if item
                in gt])
            precisions.append(hit_count / k)
        return sum(precisions) / len(precisions)

    def mean_avg_precision(self):
        aps = []
        for user, recs in self.recommendations.items():
            gt = self.ground_truth.get(user, set())
            hits, score = 0, 0
            for i, item in enumerate(recs):
                if item in gt:
                    hits += 1
                    score += hits / (i + 1)
            if gt:
                aps.append(score / len(gt))
        return sum(aps) / len(aps)

    def mean_auc(self, N=10):
        aucs = []
        for user, recs in self.recommendations.items():
            gt = self.ground_truth.get(user, set())
            y_true = [1 if item in gt else 0 for item in recs]
            y_score = [1 / (i + 1) for i in range(len(recs))]
            if len(set(y_true)) == 2:
                auc = roc_auc_score(y_true, y_score)
                aucs.append(auc)
        return sum(aucs) / len(aucs)

# Get valid users
valid_users = list(set(user_product_df.index).\
    intersection(user_gt_dict.keys()))

# Initialize the recommender
recommender = ItemBasedRecommender(user_product_df,
    item_similarity_matrix)

# Generate recommendations
sample_users = valid_users[:10]  # take 10
recommendations = {
    user_id: recommender.recommend_products(user_id, n=10)
    for user_id in sample_users
}

# Evaluate recommendations
evaluator = RecommenderEvaluator(user_gt_dict,
    recommendations)
k_values = [5, 10, 20]
results = {}

for k in k_values:
    results[f"Precision@{k}"] =
        evaluator.mean_precision_at_k(k=k)

# Add MAP as a fixed metric (doesn't depend on cutoff)
results["MAP"] = evaluator.mean_avg_precision()

print(results)
```

4) Pipeline of User-based CF for Instacart Recommendation:

```
class UserSimilarityRecommendationPipeline_new:
    def __init__(
        self,
        prior_orders_merged: pd.DataFrame,
        df_products: pd.DataFrame,
        df_orders_meta: pd.DataFrame = None
    ):
        self.orders = prior_orders_merged
        self.df_products = df_products
        self.df_orders_meta = df_orders_meta if
            df_orders_meta is not None else
            df_orders[['order_id', 'user_id',
            'order_number']].copy()

    def run(
        self,
        similarity_method: str = 'svd',# 'none', 'svd', or
            'als'
        sample_method: str = 'stratified', # 'top_n' or
            'stratified'
        interaction_metod: str = 'frequency', # 'frequency'
            or 'time' or 'none'
        scale_method = 'log', # 'log' or 'orders'
        recommender_method = 'knn', # 'most_similar' or 'knn'
        k_nearest = 10,
        n_products: int = 1000,
        n_sample_users: int = 10000,
        n_recommendations: int = 10,
        stratum_ratios: tuple = (0.5, 0.3, 0.2),
        n_components: int = 50,    # for SVD
        lambda_decay: float = 0.1, # for time decay
        test_orders_merged: pd.DataFrame = None # Test orders
            for evaluation
    ):
        if test_orders_merged is None:
            raise ValueError("test_orders_merged must be
                provided for evaluation.")

        # Step 1: Preprocessing
        preprocessor =
            FlexibleRecommenderPreprocessor(self.df_products)

        # Step 2:
        df_selected_products = preprocessor.select_products(
            self.orders,
            sample_method,
            n=n_products,
            stratum_ratios = stratum_ratios
        )

        # Step 3:
        df_filtered_orders =
            preprocessor.get_top_n_products_in_orders(self.orders,
            df_selected_products)

        # Step 4: Build weighted interaction matrix
```

```
if interaction_metod == 'frequency':
    builder =
        WeightedInteractionMatrixBuilder ( self . orders ,
        self . df_orders_meta)
    interaction_matrix = builder .\
        get_frequency_weighted_matrix( df_selected_products )
elif interaction_metod == 'time':
    builder =
        WeightedInteractionMatrixBuilder ( self . orders ,
        self . df_orders_meta)
    interaction_matrix =
        builder . get_time_weighted_matrix( df_selected_products ,
        lambda_decay=lambda_decay)
elif interaction_metod == 'none':
    interaction_matrix = preprocessor .\
        get_user_product_interaction_matrix ( df_filtered_orders )
else:
    raise ValueError("Invalid interaction_metod .
        Choose from: 'frequency', 'time', or
        'none'.")

# user_product_dict =
    preprocessor . get_user_product_dict ( df_filtered_orders )
# user_purchase_count =
    preprocessor . get_user_purchase_count ( df_filtered_orders )

# Step 5: Choose similarity computation strategy
if similarity_method == 'none':
    sim_processor =
        SimilarityPreprocessor ( interaction_matrix )
    normalized_matrix =
        sim_processor. normalize_interaction ()
    sample_users =
        sim_processor. get_sample_users(n_sample_users)
    similarity_matrix = sim_processor.\
        calculate_cosine_similarity (sample_users)

elif similarity_method == 'svd':
    sim_processor =
        SVDSimilarityPreprocessor_new( interaction_matrix ,
        orders=self . orders ,
        products=self . df_products)
    normalized_matrix =
        sim_processor. normalize_interaction (method
        = scale_method)
    sample_users =
        sim_processor. get_sample_users(n_sample_users)
    similarity_matrix = sim_processor.\
        calculate_svd_cosine_similarity (sample_users,
            n_components=n_components)

else:
    raise ValueError("Invalid similarity_method .
        Choose from: 'none', 'svd', 'als'.")

# Step 6: Recommendation
recommender = UserBasedRecommender_new(
    user_product_interaction =normalized_matrix,
    similarity_matrix = similarity_matrix ,
    products_df=self . df_products
)

# Step 7: Build a dictionary of user purchases and
    purchase counts from test_orders_merged
test_filtered =
    test_orders_merged[ test_orders_merged['product_id'].\
    isin ( df_selected_products ['product_id'])]
test_user_product_dict =
    preprocessor . get_user_product_dict ( test_filtered )
test_user_purchase_counts =
    preprocessor . get_user_purchase_count ( test_filtered )

# Step 8: Identify valid users who exist in the
    normalized matrix and similarity matrix
valid_users = list (
    set( test_user_purchase_counts . index)
    . intersection (normalized_matrix . index)
    . intersection ( similarity_matrix . index)
)
# Step 9: Generate recommendations for valid users
    using the specified recommender method
recommendations = {
    user_id : recommender.recommend_products(user_id,
        n=n_recommendations, recommender_method=
        recommender_method, k_nearest= k_nearest)
    for user_id in valid_users
}

# Step 10: Create a series for test purchase counts
    evaluation and a dictionary for test purchase
    dictionary evaluation
test_purchase_counts_eval = pd. Series (
    {user: test_user_purchase_counts [user] for user
        in valid_users }
)
test_purchase_dict_eval = {
    user: test_user_product_dict [user] for user in
        valid_users
}

# Step 11: Evaluation about test recommendations
evaluator =
    RecommenderEvaluator(test_purchase_dict_eval ,
    test_purchase_counts_eval , recommendations)
map_score = evaluator . mean_avg_precision()
auc_score = evaluator . mean_auc(N=n_recommendations)
precision_at_k =
    evaluator . mean_precision_at_k(k=n_recommendations)

# Step 12: Return evaluation metrics
return {
    "MAP": map_score,
    "AUC": auc_score,
    "Precision@K": precision_at_k
}
```

5) LightFM:

```
# Step 1: Prepare the interaction matrix
# Use the user-product interaction data (u_prod_f) to create
    the interaction matrix
# Create index mappings for product_id
unique_product_ids = u_prod_f['product_id'].unique()
id_index = {product_id: idx for idx, product_id in
    enumerate(unique_product_ids)}
index_id = {idx: product_id for product_id, idx in
    id_index . items ()}

# Map product_id to integer indices
u_prod_f['product_id'] = u_prod_f['product_id'].map(id_index)
u_prod_f = u_prod_f.sample(frac=1, random_state=42)
interaction_data = u_prod_f[['user_id', 'product_id',
    'u_p_orders_total']]
```

```
# Create the interaction matrix
interaction_matrix = csr_matrix(
    (interaction_data['u_p_orders_total'],
     (interaction_data['user_id'],
      interaction_data['product_id'])),
    shape=(num_unique_users + 1, num_unique_products +
    1)
)
# Split the interaction matrix into train and test sets
train_data, test_data = train_test_split(interaction_data,
    test_size=0.2, random_state=42)

train_interaction_matrix = csr_matrix(
    (train_data['u_p_orders_total'],
     (train_data['user_id'], train_data['product_id'])),
    shape=(num_unique_users + 1, num_unique_products +
    1)
)

test_interaction_matrix = csr_matrix(
    (test_data['u_p_orders_total'],
     (test_data['user_id'], test_data['product_id'])),
    shape=(num_unique_users + 1, num_unique_products +
    1)
)

# Step 2: Initialize the LightFM model
model2 = LightFM(loss='warp')

# Step 3: Fit the model with user and product features
model2.fit(
    train_interaction_matrix,
    user_features=user_features_row_shifted,
    item_features=item_features_row_shifted,
    epochs=5,
    num_threads=1
)

start = time.time()
model_precision_hybrid2=precision_at_k(model=model2,
    test_interactions=test_interaction_matrix, k=10,
    user_features=user_features_row_shifted,
    item_features=item_features_row_shifted, num_threads=
    4, check_intersections=False)
end = time.time()
#pickle.dump(model_precision_hybrid2,
    open("model_precision_hybrid2.p", "wb"))
print("precision score for hybrid method=
    {0:.{1}f}".format(model_precision_hybrid2.mean(), 2))
print("time taken for precision at k evaluation = {0:.{1}f}
    seconds".format(end - start, 2))
```

6) Fine Tuned Sentence Bert+FAISS:

```
# Load pre-trained model to CPU
model =
    SentenceTransformer('distilbert-base-nli-mean-tokens')

# Creating Mappings from Product ID to Name
product_map = dict(zip(df_products.product_id,
    df_products.product_name))

# Generate Fine Tuning Data
#Construct positive and negative sample pairs. The pairs with
    the same aisle are positive samples, and the pairs with
    different aisles are negative samples.
df_prod_train = df_products.sample(frac=0.8, random_state=42)
df_prod_test = df_products.drop(df_prod_train.index)

# To speed up sampling, construct lookup
aisle_groups = df_prod_train.groupby("aisle_id")

positive_pairs = []
negative_pairs = []

# Constructing positive samples: two-by-two combinations
    within the same aisle_id
for aisle_id, group in tqdm(aisle_groups, desc="Positive
    sample construction"):
    names = group["product_name"].unique().tolist()
    if len(names) < 2:
        continue
    combs = list(combinations(names, 2))
    sampled = random.sample(combs, min(50, len(combs)))  #
        control quantity
    for name1, name2 in sampled:
        positive_pairs.append((name1, name2, 1))

# Construct negative samples: take one product combination
    under each different aisle_id.
product_by_aisle = {
    aisle_id: group["product_name"].tolist()
    for aisle_id, group in aisle_groups
    if len(group) >= 2
}

aisles = list(product_by_aisle.keys())
for _ in tqdm(range(len(positive_pairs)), desc="Negative
    sample construction"):  # Keeping positive and negative
    quantities consistent
    d1, d2 = random.sample(aisles, 2)
    p1 = random.choice(product_by_aisle[d1])
    p2 = random.choice(product_by_aisle[d2])
    negative_pairs.append((p1, p2, 0))

# Combining positive and negative samples
df_pairs = pd.DataFrame(positive_pairs + negative_pairs,
    columns=["product_1", "product_2", "label"])
df_pairs = df_pairs.sample(frac=1).reset_index(drop=True)  #
    Shuffle the order

# preserve
df_pairs.to_csv(save_path+"product_pairs_for_finetune.csv",
    index=False)
print("Sample construction is complete and the total number
    of samples generated is", len(df_pairs))

# Fine Tuing
df_pairs =
    pd.read_csv(save_path+"product_pairs_for_finetune.csv")

train_examples = [
    InputExample(texts=[row['product_1'], row['product_2']],
        label=float(row['label']))
    for _, row in df_pairs.iterrows()
]

# Build a training data loader
train_dataloader = DataLoader(train_examples, shuffle=True,
    batch_size=64)

# Define the loss function (contrast learning)
```

```
# You can also choose TripletLoss, ContrastiveLoss, etc.,
    which are suitable for data with triplets.
train_loss = CosineSimilarityLoss(model=model)

model_save_path = save_path+'fine_tuned_model'

# Fine tuning
# Running time is 2:20:20.
start = time.time()
os.environ["WANDB_DISABLED"] = "true" # Disable wandb
    logging
model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    epochs=3,
    warmup_steps=100,
    output_path=model_save_path # Save path of the
        fine-tuned model
)
end = time.time()
print('Running time is {:.2f} seconds.'.format(end-start))

# Use Fine-tuned Model
# Loading the fine-tuned model
model = SentenceTransformer(model_save_path)

# Generate embeddings
def generate_embeddings(texts, batch_size=64):
    embeddings = []
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i+batch_size]
        batch_emb = model.encode(batch,
            show_progress_bar=True)
        embeddings.append(batch_emb)
    return np.vstack(embeddings)

# Generate embeddings for all products
# Running time is 22 minutes.
start = time.time()
product_embeddings =
    generate_embeddings(df_products.product_name.tolist())
end = time.time()
print('Running time is {:.2f} seconds.'.format(end-start))

embeddings_df = pd.DataFrame(product_embeddings, columns=[i
    for i in range(product_embeddings.shape[1])])

# If the original product name needs to be preserved, a
    column can be added
embeddings_df['product_id'] = df_products['product_id'].values
embeddings_df['product_name'] =
    df_products['product_name'].values

# Save to CSV file
embeddings_df.to_csv(save_path +
    'fine_tuned_sentence_bert_product_embeddings.csv',
    index=False)

# FAISS-cpu
# Create FAISS indexes
dimension = product_embeddings.shape[1]  # Dimensions of
    embedded vectors
index = faiss.IndexFlatL2(dimension)  # L2 distance
    (Euclidean distance)

# Adding vectors to indexes
index.add(product_embeddings)

# Save index and product ID mapping
faiss.write_index(index, save_path+"product_index.faiss")
np.save(save_path+"product_ids.npy",
    df_products.product_id.values)


# Loading Indexes and Product IDs
index = faiss.read_index(save_path+"product_index.faiss")
product_ids = np.load(save_path+"product_ids.npy")

# Recommended Similar Product Functions with CPU
def recommend_similar_products(query, k=5):
    # Generate query text embedding
    query_embedding = model.encode([query])

    # Search for the most similar k products
    distances, indices = index.search(query_embedding, k)

    # Get product ID and name
    similar_ids = product_ids[indices[0]]
    similar_products = [product_map[pid] for pid in
        similar_ids]

    return list(zip(similar_ids, similar_products,
        distances[0]))

# Test the recommendation function
# CPU
# Running time is 1 hour.
start = time.time()
k_ = 21
df_recom = pd.DataFrame()
for ix,row in df_products.iterrows():
    tmp_dict = {"product_id": row['product_id'],
        'product_name':row['product_name']}

    query_product = row['product_name']
    recommendations =
        recommend_similar_products(query_product, k=k_)
    for i, (pid, name, dist) in enumerate(recommendations, 0):
        tmp_dict['nearest_'+str(i)] = name
    df_recom =
        pd.concat([df_recom,pd.Series(tmp_dict).to_frame().T])
end = time.time()
print('Running time is {:.2f} seconds.'.format(end-start))

df_recom.to_csv(save_path +
    "fine_tuned_sentence_bert_FAISS_recommendation.csv",
    index=None)
```