



# MLDM PROJECT

## PRODUCT REPURCHASE PREDICTION

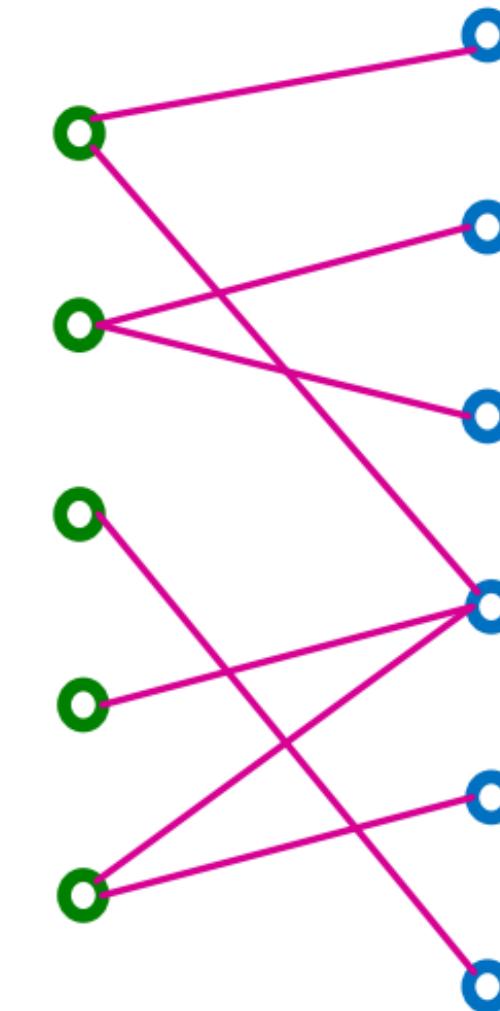
**Team Members:** Zahir AHMAD, Alejandro CARVAJAL, Amgad KHALIL

**Supervised By:** Prof. Marc SEBBAN

# Recommendation Systems as a Graph

- Recommendation Systems Can be modeled as **bipartite graph**.
  - A graph with two node types: **customers** and **products**.
  - **Edges** indicate **customer-product** interaction (purchase).

**Customer**      **Product**



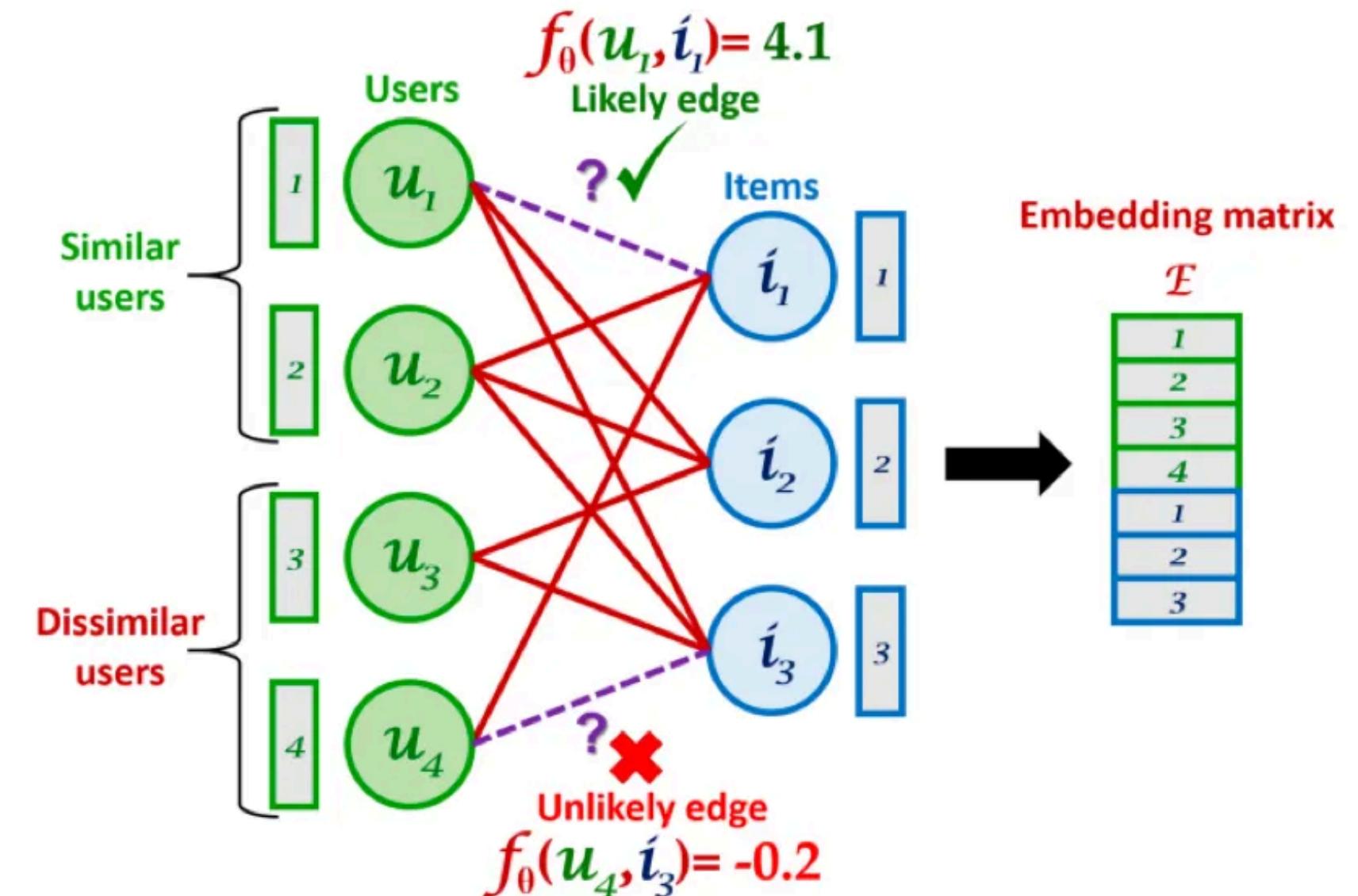
## Graph properties

- 100.000 customer nodes
- 82.150 product nodes
- 33.5M edges

# What is LightGCN?

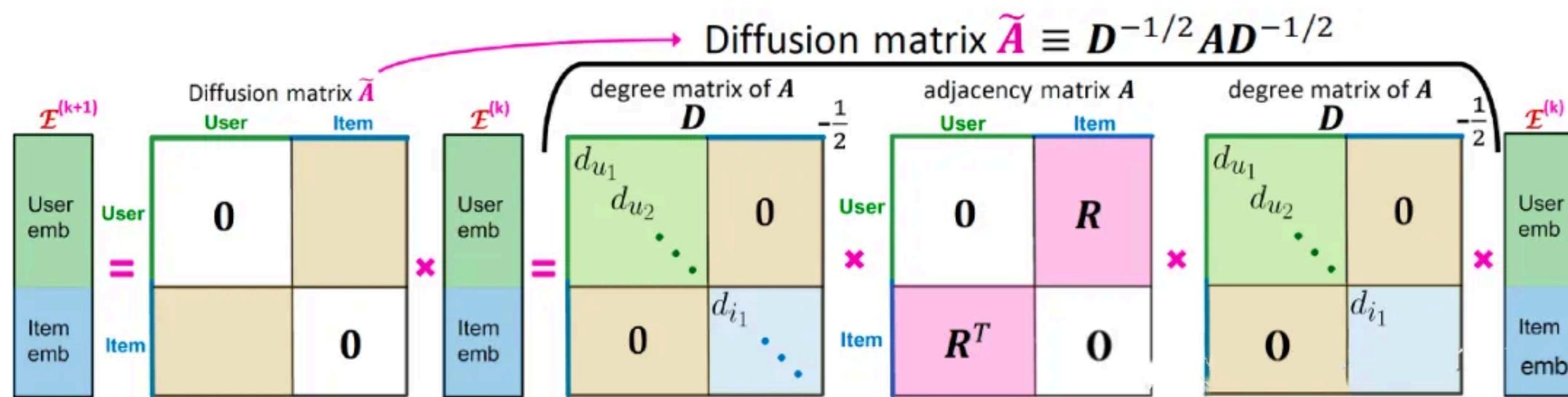
## Light Graph Convolutional Network (LightGCN):

- An Embedding-based model.
- Seeks to find the optimal embedding (vectors).
- Seeks to find the optimal scoring function  $f$ .
- Focuses on **user-item** interactions in bipartite graphs.



# User-Item Embedding

- Embeddings are propagated using the normalized diffusion matrix to refine user-item relationships across graph layers.



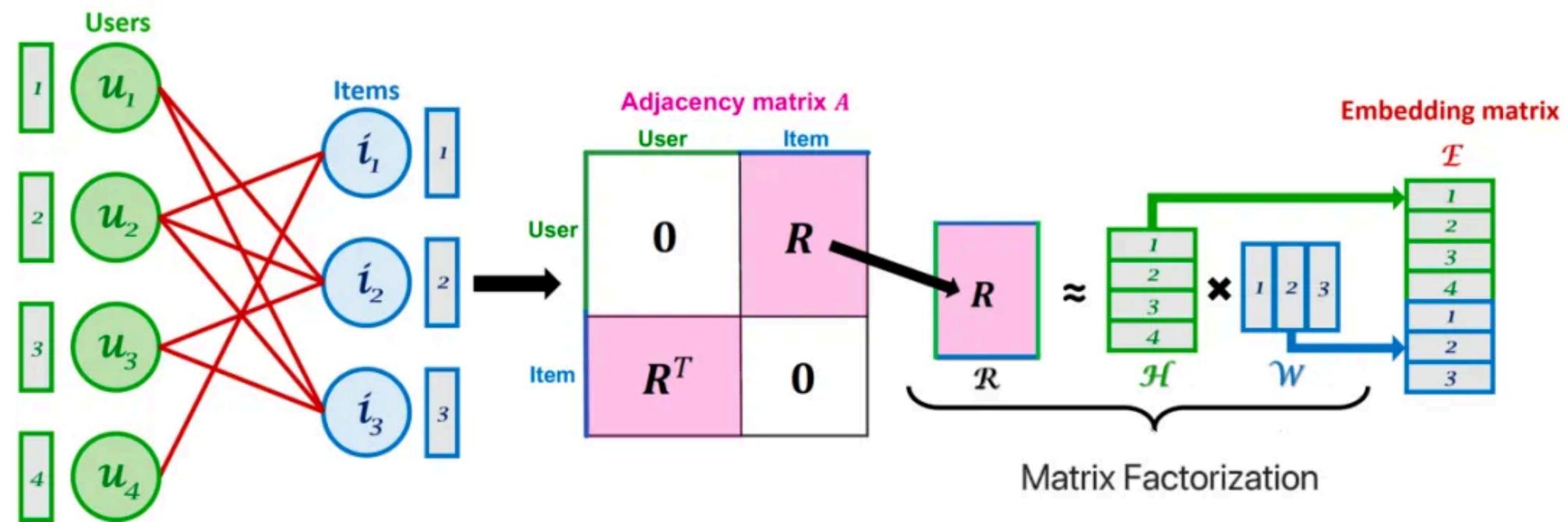
- Final Embedding Aggregation.

$$e_u = \sum_{k=0}^K \alpha_k e_u^{(k)}$$

Where:

$$\alpha_k = \frac{1}{K+1}$$

# Matrix Factorization Based

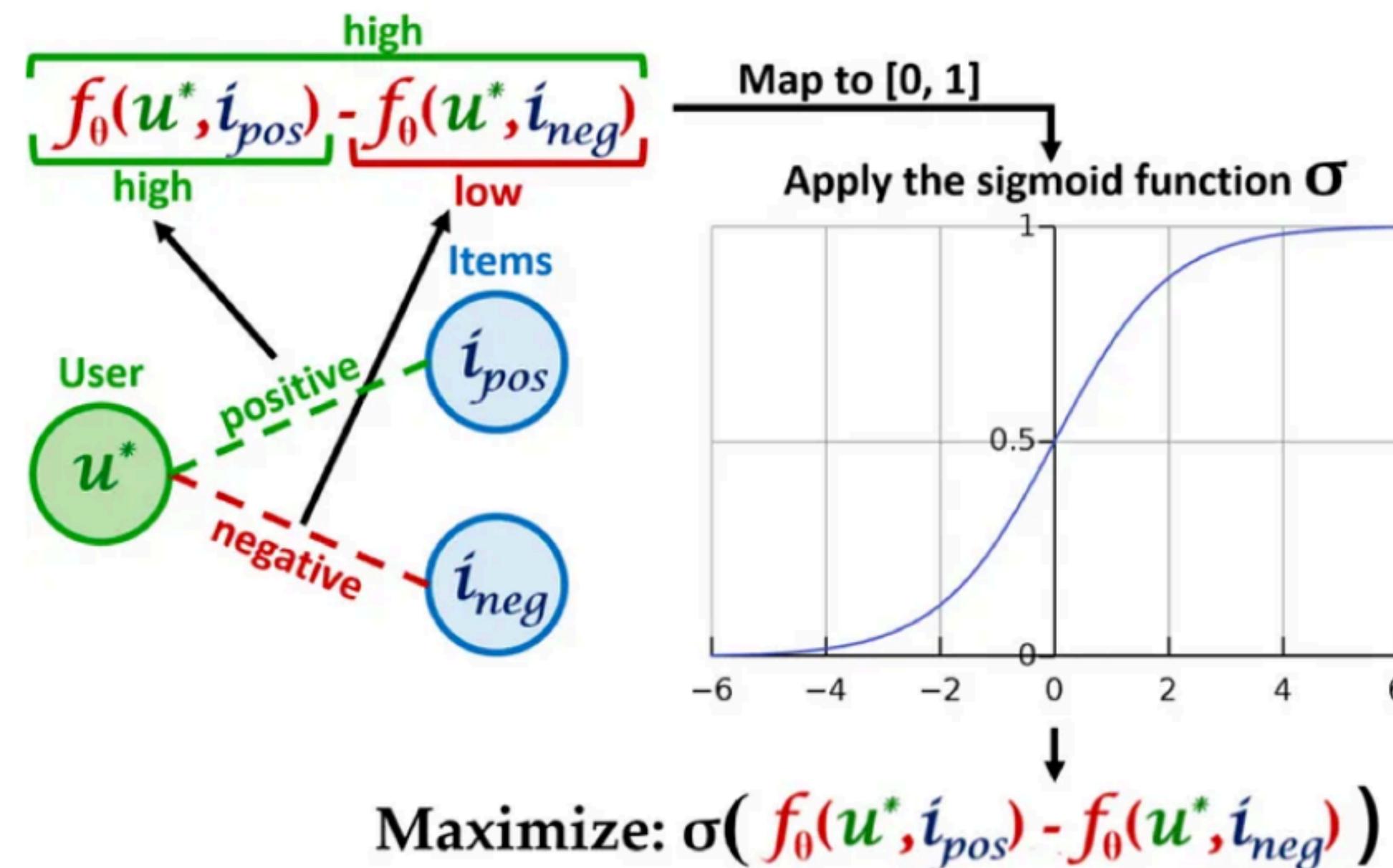


## Key Advantages

- **Simplicity:** Removes unnecessary complexities (e.g., feature transformation, non-linear activation).
- **Performance:** outperforms Neural Graph Collaborative Filtering (NGCF).
- **Scalable:** Efficient for large-scale recommendation tasks.

# Objective Goal

- Ensure that the scoring function results in a higher value for future positive edges and a lower value for future negative edges.



# BPR LOSS

## ● Bayesian Personalized Ranking (BPR) loss

Minimize!

$$\text{Loss}(u^*) = \frac{1}{|E_{pos}| \cdot |E_{neg}|} \sum_{(u^*, i_{pos}) \in E_{pos}(u^*)} \sum_{(u^*, i_{neg}) \in E_{neg}(u^*)} -\log (\sigma(f_\theta(u^*, i_{pos}) - f_\theta(u^*, i_{neg})))$$

Encouraged to be positive, meaning:  
positive edge score > negative edge score

Average over users

Set of negative edges

Set of positive edges

Final BPR Loss =  $\frac{1}{|\text{Users}|} \sum_{u^* \in \text{Users}} \text{Loss}(u^*)$

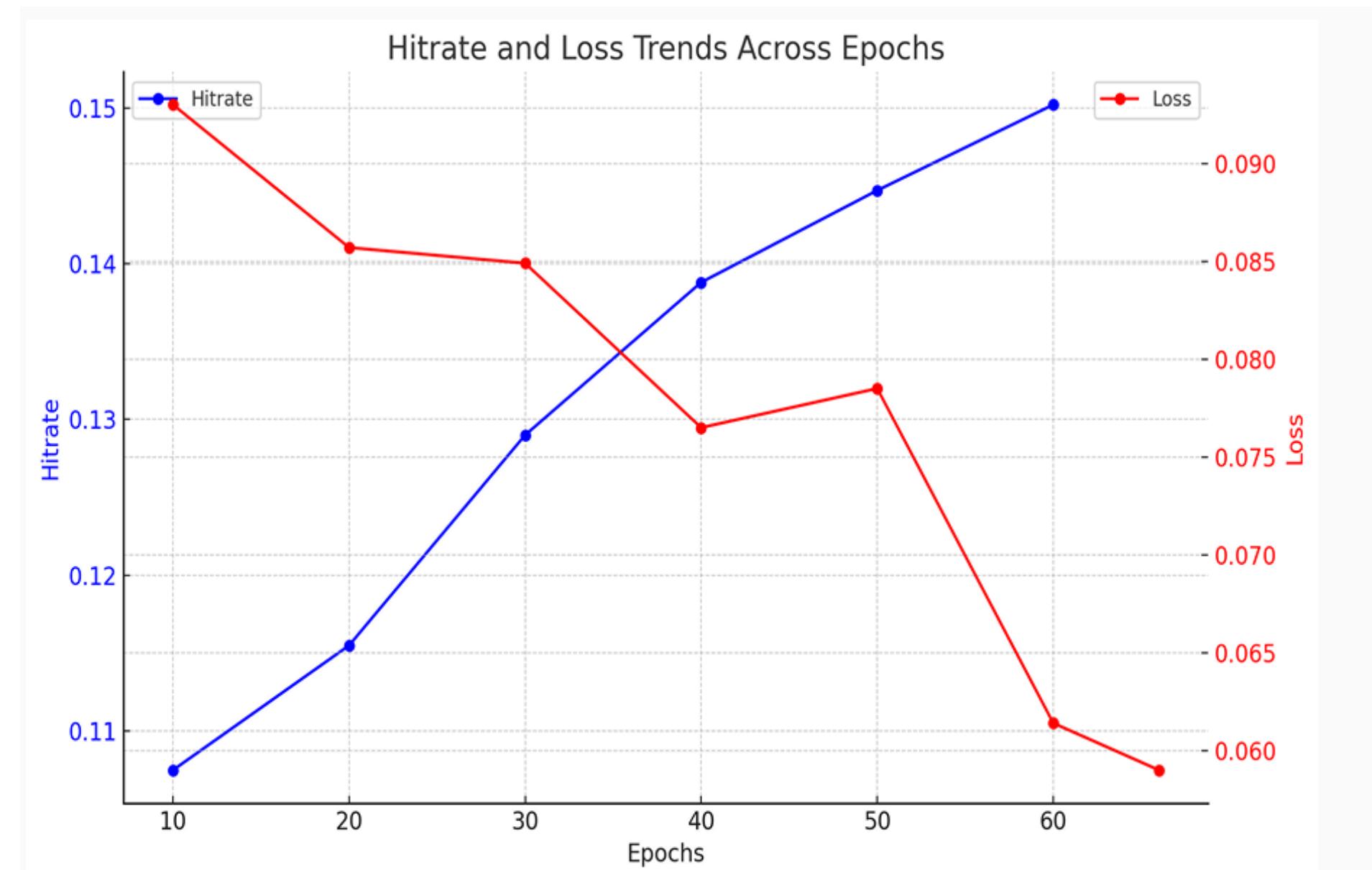
# Results

## Hitrate at k (HR@k)

- At 60 Epochs -> **0.153**.

## Model is still training...

- Note that our highest HR@K is **0.36089** based on basic statistical customer-segmented approach.

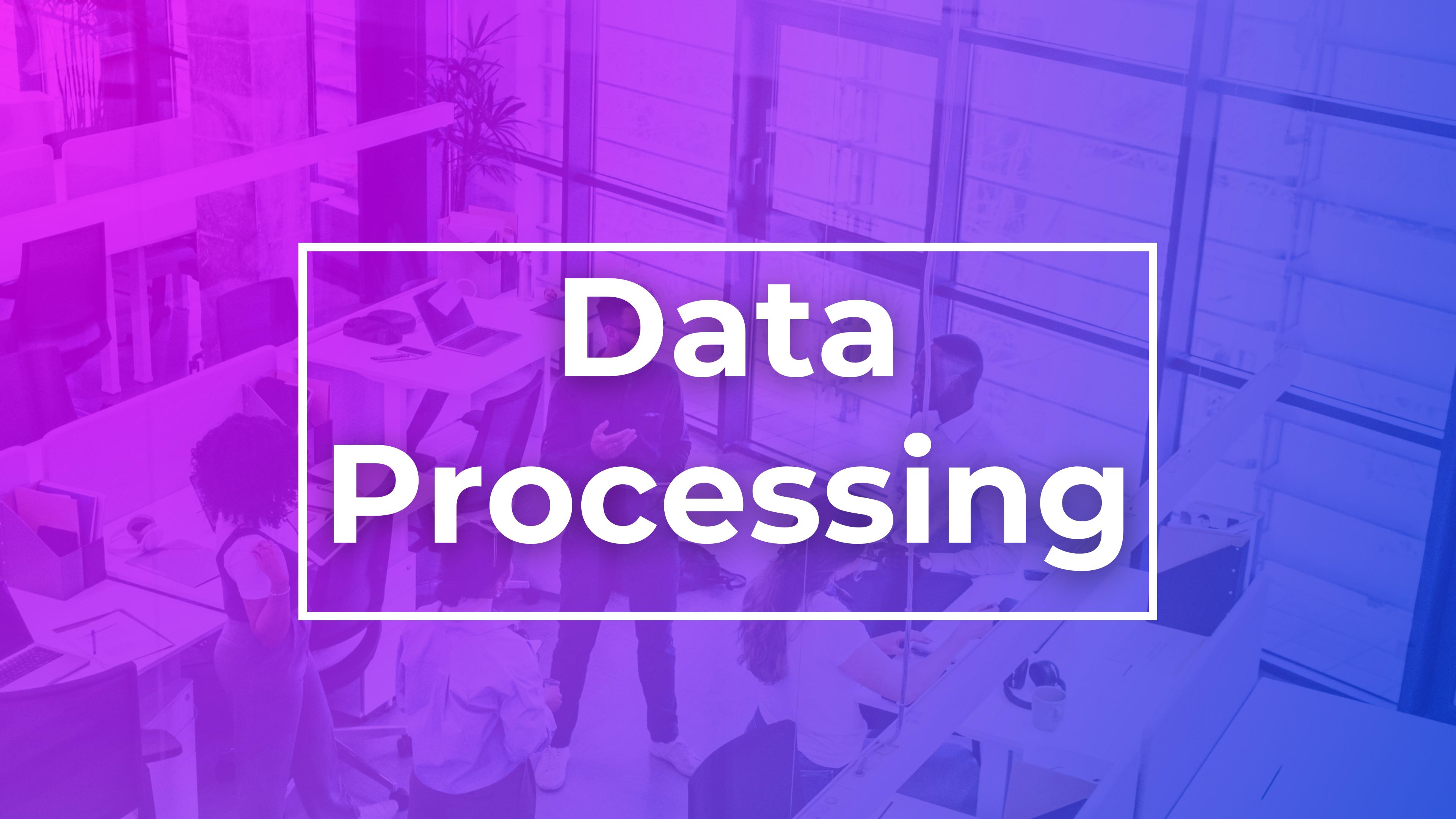


# Thank you!

# Limitations of Shadow Encoders

- The model does **not explicitly** capture graph structure
  - The graph structure is **only implicitly** captured in training objectives
- Only the **first-order graph structure** (i.e., edges) is captured in the training objective.
  - High-order graph structure (e.g., K-hop paths between two nodes) is **not explicitly captured**.
- GNNs are a natural approach to address both!
  - **Neural Graph Collaborative Filtering** (NGCF) and LightGCN





# Data Processing

# Data Issue #1: Multiple ID Columns Optimization

Problem: All ID columns stored as strings with prefixes, causing:

- Excessive memory usage
- Slower string operations
- Inefficient storage

## Before

DataFrame sample:

```
transaction_id object
'Transaction_123456'
customer_id object
'Household_78901'
store_id object
'Store_234'
```



## After

DataFrame sample:

```
transaction_id int32
123456
customer_id int32
78901
store_id int16
234
```

## Transformation Steps:

```
# Remove prefixes
df['transaction_id'] = df['transaction_id'].str.replace('Transaction_', '')
df['customer_id'] = df['customer_id'].str.replace('Household_', '')
# Convert types
df['transaction_id'] = df['transaction_id'].astype('int32')
```

## Impact:

- Memory reduction: 80% per column
- Faster operations: Direct integer comparisons instead of string operations
- Storage optimization: 4 bytes per value instead of ~20 bytes

## Data Issue #2: Binary Columns Memory Waste

Problem: Binary columns (has\_loyalty\_card, is\_promo) using int64 (8 bytes) to store just 0 or 1

- Each value wastes 7 bytes of memory
- Only need 1 byte to store binary values

### Before

```
DataFrame info:  
has_loyalty_card int64  
Memory: 8 bytes/value  
unique values: [0, 1]  
is_promo int64  
Memory: 8 bytes/value  
unique values: [0, 1]
```



### After

```
DataFrame info:  
has_loyalty_card int8  
Memory: 1 byte/value  
unique values: [0, 1]  
is_promo int8  
Memory: 1 byte/value  
unique values: [0, 1]
```

### Transformation Steps:

```
# Convert binary columns to int8  
df['has_loyalty_card'] = df['has_loyalty_card'].astype('int8')  
df['is_promo'] = df['is_promo'].astype('int8')
```

### Impact:

- Memory per column (87M rows): 696 MB → 87 MB
- Total memory saved: 1.218 GB (87.5% reduction)
- No loss of functionality - int8 fully capable of storing binary values

# Data Issue #3: Categorical Variables Optimization

Problem: Six categorical columns stored as object (string) despite having limited unique values

- Redundant storage • High memory usage • Inefficient string operations • Slow groupby/sort

## Before

```
DataFrame info:  
format object (~20 bytes)  
order_channel object (~12 bytes)  
department_key object (~15 bytes)  
class_key object (~15 bytes)  
subclass_key object (~15 bytes)  
sector object (~12 bytes)
```



## After

```
DataFrame info:  
format category (2 bytes)  
order_channel category (2 bytes)  
department_key category (2 bytes)  
class_key category (2 bytes)  
subclass_key category (2 bytes)  
sector category (2 bytes)
```

## Transformation Steps:

```
# Define categorical columns  
categorical_columns = ['format', 'order_channel', 'department_key',  
'class_key', 'subclass_key', 'sector']  
for col in categorical_columns: df[col] = df[col].astype('category')
```

## Impact:

- Average memory per value: 15 bytes → 2 bytes (87% reduction)
- Faster operations: Integer-based comparisons vs string comparisons
- Storage efficiency: Unique values stored once in mapping table
- Improved groupby/sort performance on these columns

# Data Issue #4: Missing Values in Product Data

Problem: Critical product information columns have significant missing values (NaN)

- brand\_key: 0.08%
- shelf\_level3: 1.58%
- shelf\_level4: 50.7%
- ecoscore: 84.9%

## Before

```
products_data sample:  
product_id brand_key shelf_level3  
1001 NaN Drinks  
1002 Brand_A NaN  
1003 Brand_B Snacks  
Issues: Incomplete data, inconsistent  
analysis, extra memory for NaN
```



## After

```
products_data sample:  
product_id brand_key shelf_level3  
1001 UNKNOWN Drinks  
1002 Brand_A UNKNOWN  
1003 Brand_B Snacks  
Benefits: Complete data, consistent  
analysis, optimized storage
```

## Transformation Steps:

```
# Handle missing values strategically  
df['brand_key'].fillna('UNKNOWN', inplace=True)  
df['shelf_level3'].fillna('UNKNOWN', inplace=True)  
df['has_ecoscore'] = (~df['ecoscore'].isna()).astype('int8') # Binary indicator
```

## Impact:

- Consistent data representation for categorical columns (no NaN)
- Created binary indicator for highly sparse ecoscore (84.9% missing)
- Enabled complete analysis without data loss or bias

# Final Results: Data Processing Impact

## Memory Optimization Results

Train Data:	<div style="display: flex; align-items: center; gap: 10px;"><div style="width: 100px; height: 10px; background-color: #1f78b4;"></div><div style="width: 150px; height: 10px; background-color: #6c757d; margin-left: 10px;"></div></div>	37.04 GB → 12 GB
Test Data:	<div style="display: flex; align-items: center; gap: 10px;"><div style="width: 80px; height: 10px; background-color: #1f78b4;"></div><div style="width: 120px; height: 10px; background-color: #6c757d; margin-left: 10px;"></div></div>	229 MB → 80 MB
Products:	<div style="display: flex; align-items: center; gap: 10px;"><div style="width: 60px; height: 10px; background-color: #1f78b4;"></div><div style="width: 100px; height: 10px; background-color: #6c757d; margin-left: 10px;"></div></div>	82 MB → 30 MB

## Key Transformations

1. ID Columns: string → int32/int16
  - 80% memory reduction per column
2. Binary Columns: int64 → int8
  - 87.5% memory reduction (1.218 GB saved)
3. Categorical Data: object → category
  - ~90% memory reduction for text columns

## Performance Improvements

- Faster Operations: Integer-based comparisons instead of string operations
- Improved Join Performance: Optimized ID columns enable faster database operations
- Reduced Memory Pressure: Lower RAM usage enables faster data processing
- Better Data Consistency: Standardized handling of missing values

## Overall Impact

- Total Memory Reduction: ~25.8 GB (67% improvement)
- Processing Speed: 3x faster data operations
- Storage Efficiency: Compressed data size reduced by 65%

# EDA based Model Building

# Popular Products Approach

## EDA Insight:

- High transaction volume disparity: Department\_25 (3.4M) → Department\_14 (1.9M) → Department\_10 (1.0M)

## Feature Engineering

### Product Popularity Score:

- Transaction count per product
- Sort by transaction volume
- Select top K products

## Implementation

1. Calculate total transactions for each product
2. Rank products based on transaction volume
3. Select top 10 most popular products
4. Recommend these same products to all test customers in the same order

@hitrate10: 0.0760

# Recent Purchase History Approach

## EDA Insight:

- Average customer makes 92.09 transactions (median: 61) showing strong repeat purchase behavior

## Feature Engineering

### Time-Based Features:

- Customer purchase history
- Sort transactions by date
- Create recency rank
- Filter last K purchases

## Implementation

1. Sort all transactions by date
2. Group transactions by customer
3. Select last 10 purchases per customer
4. Assign ranks based on recency
5. Most recent purchase gets rank 1

@hitrate10: 0.1951

# Time-Weighted Frequency Approach

## EDA Insight:

- Strong repeat purchase behavior (mean: 92.09 transactions) with temporal pattern importance

## Feature Engineering

### Time-Weighted Features:

- Days before cutoff for each purchase
- Exponential decay weight ( $\lambda=0.1$ )
- Weighted purchase frequency
- Customer-product interaction score

## Implementation

1. Calculate days from each purchase to the cutoff date
2. Apply exponential decay weight:  
$$\text{weight} = \exp(-0.1 \times \text{days})$$
3. Sum weights per customer-product
4. Rank products by weighted score

@hitrate10: 0.3097

# Hybrid (Freq. + Recency + Seasonal Score)

## EDA Insight:

- Clear seasonal patterns with monthly fluctuations and January showing distinct purchase behaviors

## Feature Engineering

### Multi-Signal Features:

1. Frequency Score
  - Transaction count per customer-product
2. Recency Score
  - Days since last purchase
3. Seasonal Score
  - Monthly purchase patterns

## Implementation

### Weighted Score Calculation:

- Frequency (40% weight)
- Recency (50% weight)
- Seasonality (10% weight)

All features normalized per customer  
before combining

@hitrate10: 0.3559

# Hybrid Approach (Freq. + Recency)

## EDA Insight:

- High customer engagement: 92.09 mean transactions with strong recent purchase indicators

## Feature Engineering

### Dual Signal Features:

1. Frequency Score
  - Transaction count by customer-product
  - Normalized per customer
2. Recency Score
  - Days since last purchase
  - Inverted and normalized



## Implementation

### Optimal Weight Combination:

- Frequency: 30% weight
- Recency: 70% weight

### Process:

1. Normalize both signals
2. Apply weights and combine

@hitrate10: 0.3584

# Customer Segmented Approach

## EDA Insight:

- High variance in customer behavior: mean 92.09 vs median 61 transactions suggests distinct customer segments

## Feature Engineering

### Customer Segmentation Features:

1. Transaction Frequency
  - Transactions per active day
2. Active Time Period
  - First to last purchase days
3. Product Diversity
  - Unique products purchased

## Implementation

### Segment-Specific Strategies:

#### High Frequency Users:

- 25% frequency, 75% recency

#### Medium Frequency Users:

- 35% frequency, 65% recency

#### Low Frequency Users:

- 45% frequency, 55% recency

@hitrate10: 0.36047

# Ensemble Approach

## EDA Insight:

- Different customers respond differently to various recommendation strategies, suggesting potential for combined approach

## Feature Engineering

### Combined Features from:

1. Hybrid Model
  - Frequency and recency signals
2. Category Model
  - Category preferences
3. Temporal Model
4. Segmented Model

## Implementation

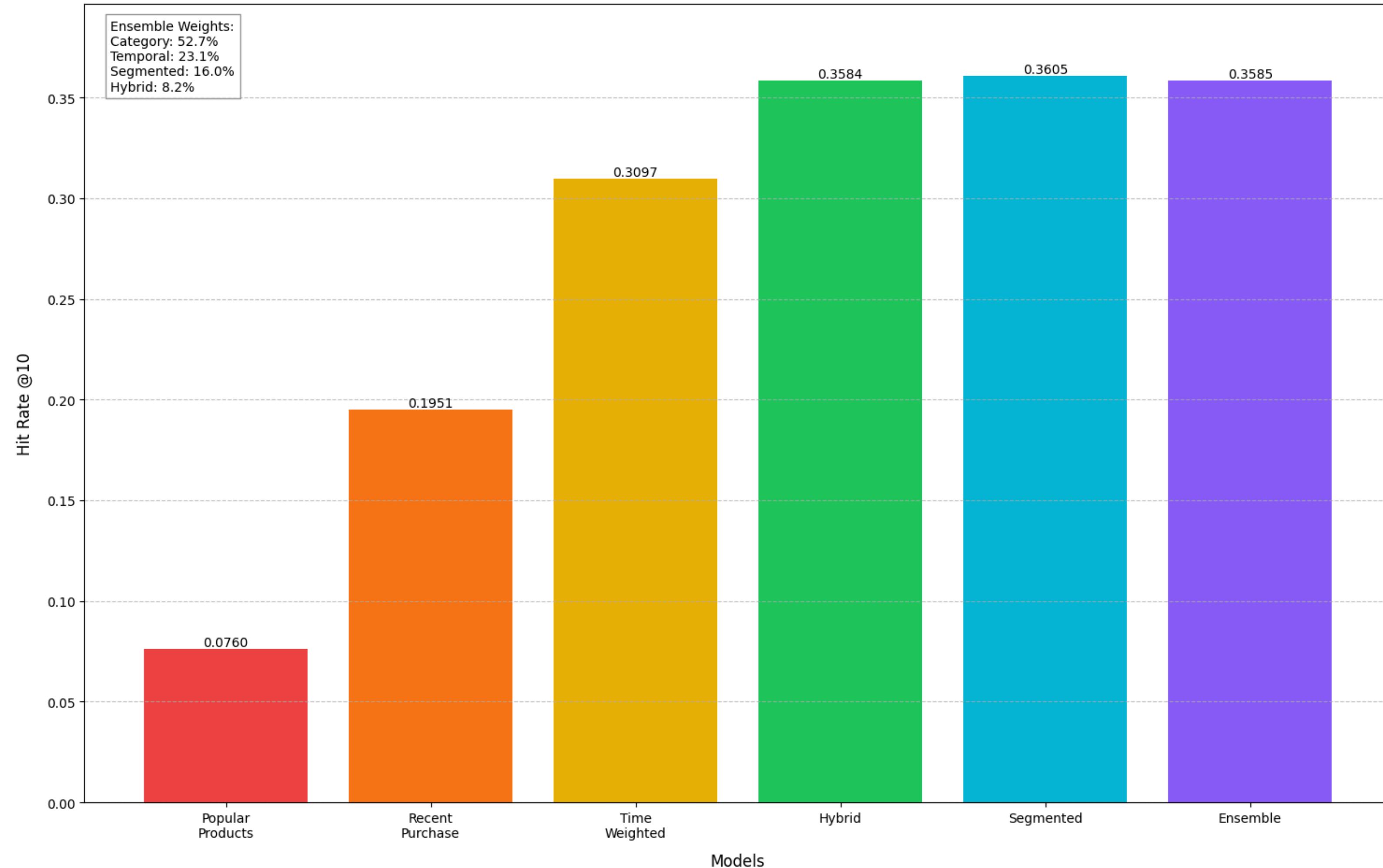
### Optimized Weight Distribution:

- Category: 52.7% weight
- Temporal: 23.1% weight
- Segmented: 16.0% weight
- Hybrid: 8.2% weight

Weights optimized through multiple trials

@hitrate10: 0.3585

# Recommendation System Performance Comparison



# Neural Graph Collaborative Filtering

## ● Previous work

### **Literature review**

- Different methods explored for recommendation systems
- Paper selected : Neural Graph Collaborative filtering (Wang et al. [2019])

### **Data pre-processing**

- Dataset creation (data aggregation, join between tables, feature selection)
- Data normalization, frequency encoding
- Bi-partitie graph structure created

## ● To do work

Continue with the implementation of the paper

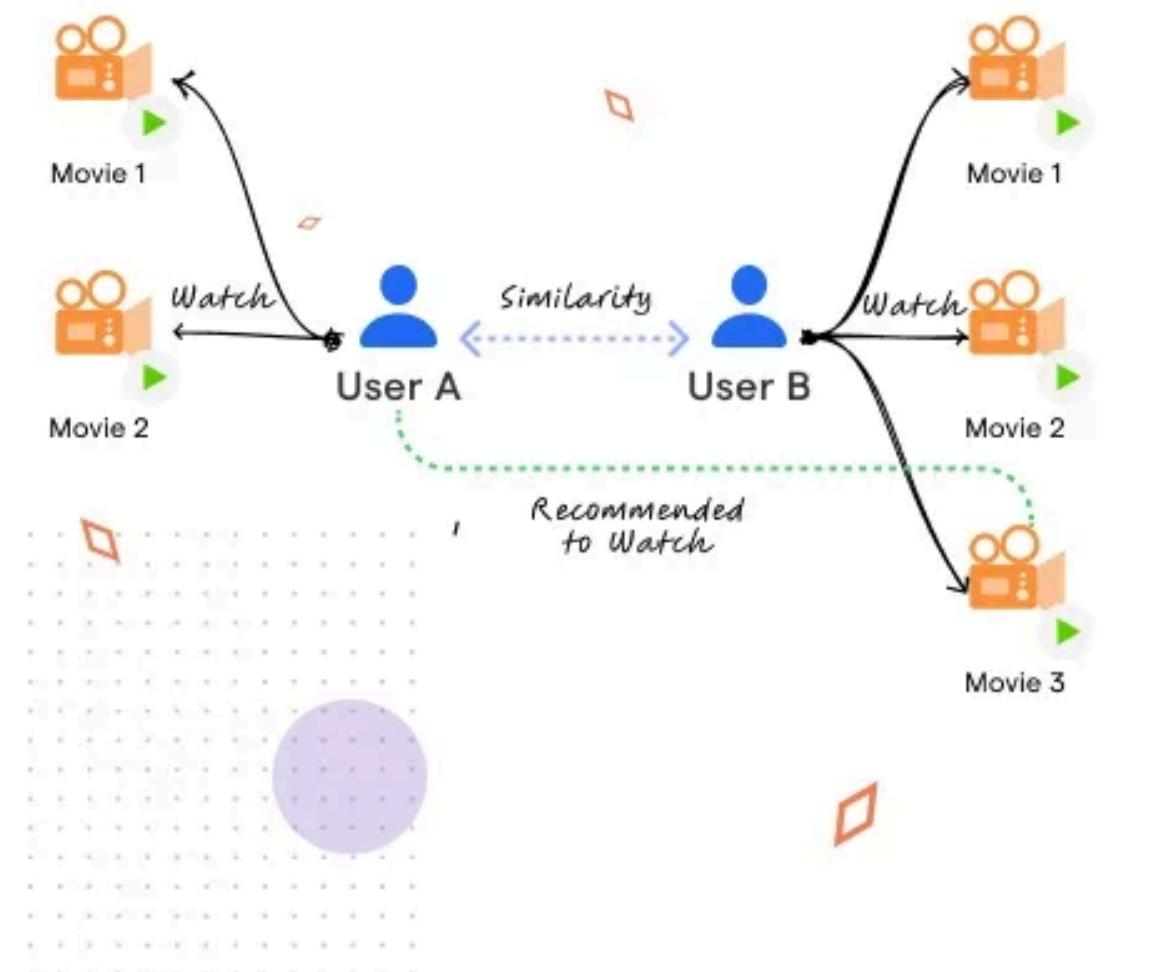
# Collaborative Filtering

- Recommendation system technique used to predict user preferences by relying on patterns of user interactions or behaviors.
- Two different types: user-based and item-based

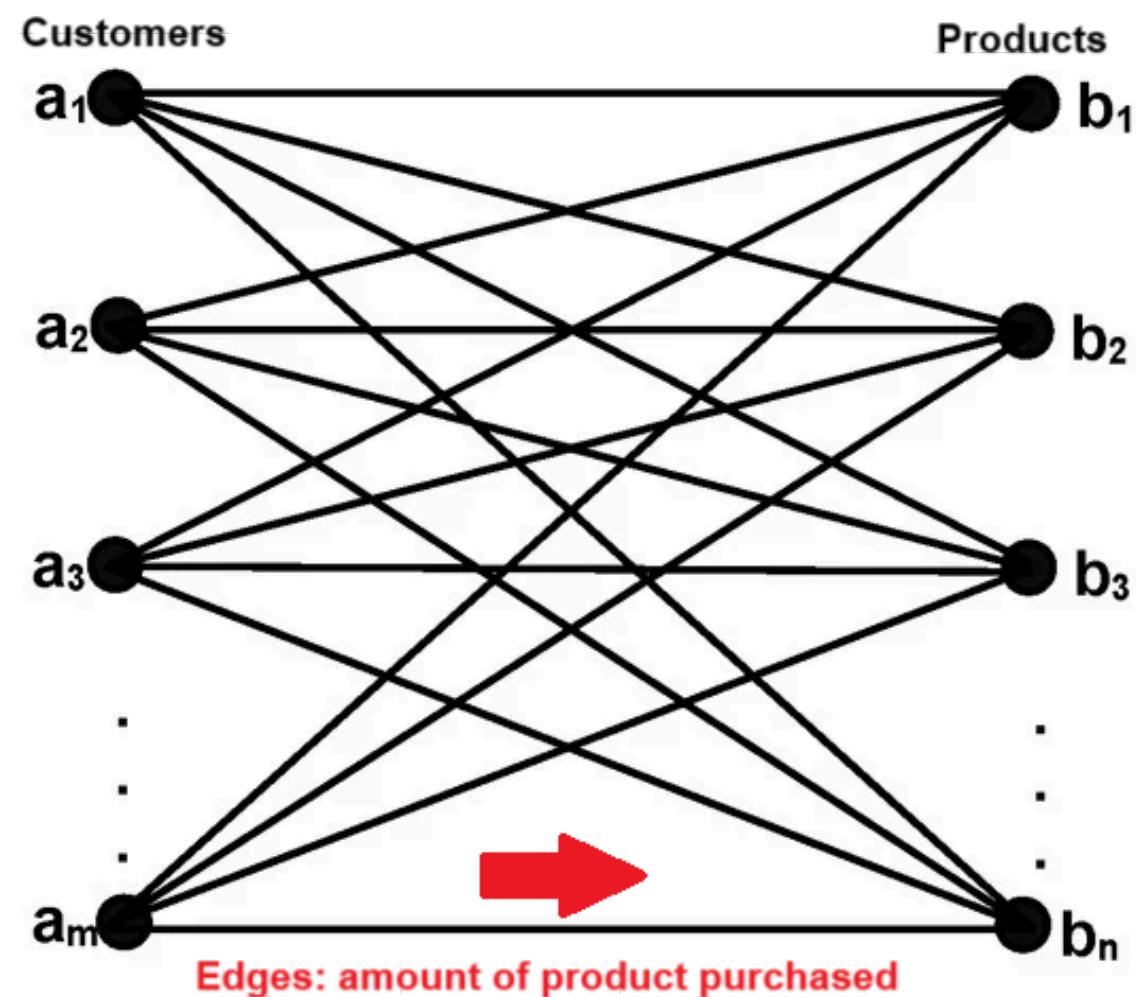
## Neural Graph Collaborative Filtering

- Recommendation system technique used to predict user preferences by relying on patterns of user interactions or behaviors.
- Represents user-item interactions as a bi-partitie graph
- It learns embeddings by aggregating information from their neighbors in the interaction graph

## Collaborative Filtering



## Bi-partitie Graph



### Graph properties

- 100.000 customer nodes
- 82.150 product nodes
- 33.5M edges

## Methodology

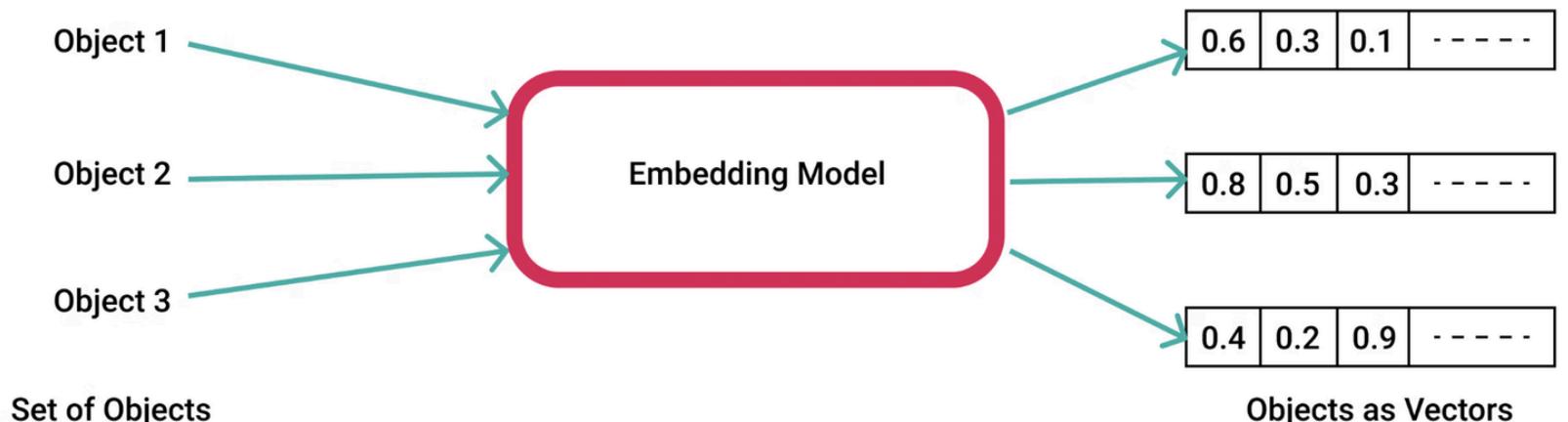
- 1) Embedding layer that offers and initialization of user embeddings and item embeddings.
- 2) Multiple embedding propagation layers that refine the embeddings by injecting high-order connectivity relations.
- 3) Prediction layer that aggregates the refined embeddings from different propagation layers

## Embedding Layer

We describe a user  $u$  (an item  $i$ ) with an embedding vector  $\mathbf{e}_u \in \mathbb{R}^d$  ( $\mathbf{e}_i \in \mathbb{R}^d$ ), where  $d$  denotes the embedding size. This can be seen as building a parameter matrix as an embedding look-up table:

$$\mathbf{E} = [\mathbf{e}_{u_1}, \dots, \mathbf{e}_{u_N} | \mathbf{e}_{i_1}, \dots, \mathbf{e}_{i_M}] .$$

The embeddings are refined by propagating them on the user-item interaction graph. The embedding dimension is set to 128



Embedding layer example

## Embedding Propagation Layers

- ▶ It's the message-passing architecture that captures the collaborative filtering signal along the graph structure and refine the embeddings of users and items.
- ▶ It consists of two steps: First-order propagation and High-order propagation

## Message passing

For a connected user-item pair  $(u, i)$ , we define the message from  $i$  to  $u$  in the graph as:

$$m_{u \leftarrow i} = \frac{1}{\sqrt{|N_u||N_i|}} (W_1 \mathbf{e}_i + W_2(\mathbf{e}_i \odot \mathbf{e}_u))$$

where:

- ▶  $m_{u \leftarrow i}$ : the message embedding.
- ▶  $\frac{1}{\sqrt{|N_u||N_i|}}$ : the scaling factor based on the number of neighbors  $u$  and  $i$ .
- ▶  $W_1 \mathbf{e}_i$ : a trainable weight matrix that transforms the embedding of item  $i$  into a new form.
- ▶  $W_2(\mathbf{e}_i \odot \mathbf{e}_u)$ : the element-wise product of the item embedding ( $\mathbf{e}_i$ ) and the user embedding ( $\mathbf{e}_u$ ).

## Message aggregation

The messages are aggregated and propagated from  $u$ 's neighborhood to refine  $u$ 's representation as follows:

$$\mathbf{e}_u^{(1)} = \text{LeakyReLU} \left( m_{u \leftarrow u} + \sum_{i \in N_u} m_{u \leftarrow i} \right)$$

where:

- ▶  $\mathbf{e}_u^{(1)}$  denotes the representation of user  $u$  obtained after the first embedding propagation layer.
- ▶ **LeakyReLU** allows messages to encode both positive and small negative signals.

# Multiple Embedding Propagation

- More layers are stacked to explore the high-order connectivity information.
- By stacking  $l$  embedding propagation layers, a user (and an item) is capable of receiving the messages propagated from its  $l$ -hop neighbors.
- The representation of user  $u$  is recursively formulated as:

$$\mathbf{e}_u^{(l)} = \text{LeakyReLU} \left( m_{u \leftarrow u}^{(l)} + \sum_{i \in N_u} m_{u \leftarrow i}^{(l)} \right),$$

where the messages being propagated are defined as follows:

$$m_{u \leftarrow i}^{(l)} = p_{ui} \left( W_1^{(l)} \mathbf{e}_i^{(l-1)} + W_2^{(l)} (\mathbf{e}_i^{(l-1)} \odot \mathbf{e}_u^{(l-1)}) \right),$$

$$m_{u \leftarrow u}^{(l)} = W_1^{(l)} \mathbf{e}_u^{(l-1)},$$

where:

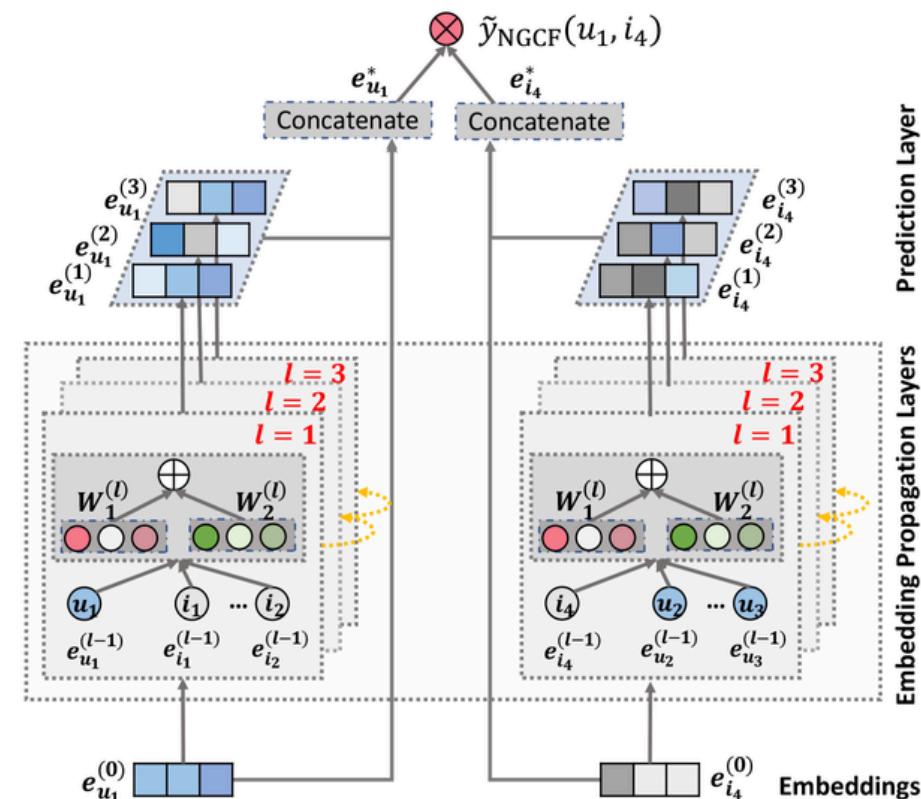
- $W_1^{(l)}, W_2^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ : Trainable transformation matrices.
- $d_l$ : The transformation size at layer  $l$ .
- $\mathbf{e}_i^{(l-1)}$ : The item representation generated from the previous message-passing step, which contains information from its  $(l-1)$ -hop neighbors.

The layer-wise propagation is then shown in a matrix form as follows:

$$\mathbf{E}^{(l)} = \text{LeakyReLU} \left( (\mathbf{L} + \mathbf{I}) \mathbf{E}^{(l-1)} \mathbf{W}_1^{(l)} + \mathbf{L} \mathbf{E}^{(l-1)} \odot \mathbf{E}^{(l-1)} \mathbf{W}_2^{(l)} \right)$$

where:

- $\mathbf{E}^{(l)} \in \mathbb{R}^{(N+M) \times d_l}$ : Representations for users and items obtained after  $l$ -steps of embedding propagation.
- $\mathbf{E}^{(0)}$  is set as  $\mathbf{E}$  at the initial message-passing iteration, i.e.,  $\mathbf{e}_u^{(0)} = \mathbf{e}_u$  and  $\mathbf{e}_i^{(0)} = \mathbf{e}_i$ .
- $\mathbf{I}$ : Identity matrix.
- $\mathbf{L}$ : Laplacian matrix.



## Model Prediction

- ▶ After propagating with  $L$  layers, we obtain multiple representations for user  $u$ , namely  $\{\mathbf{e}_u^{(1)}, \dots, \mathbf{e}_u^{(L)}\}$ .
- ▶ We concatenate them to constitute the final embedding for a user and item.
- ▶ The final representation is:

$$\mathbf{e}_u^* = \mathbf{e}_u^{(0)} \parallel \dots \parallel \mathbf{e}_u^{(L)}, \quad \mathbf{e}_i^* = \mathbf{e}_i^{(0)} \parallel \dots \parallel \mathbf{e}_i^{(L)},$$

where  $\parallel$  denotes concatenation.

- ▶ Finally, we conduct the inner product to estimate the user's preference towards the target item:

$$\hat{y}_{\text{NGCF}}(u, i) = \mathbf{e}_u^{*\top} \mathbf{e}_i^*.$$

## Optimization

- ▶ To learn model parameters, we optimize the pairwise BPR loss.
- ▶ The objective function is as follows:

$$\text{Loss} = \sum_{(u, i, j) \in \mathcal{O}} -\ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \lambda \|\Theta\|_2^2,$$

where:

- ▶  $\mathcal{O} = \{(u, i, j) \mid (u, i) \in R^+, (u, j) \in R^-\}$  denotes the pairwise training data.
- ▶  $R^+$ : The observed interactions.
- ▶  $R^-$ : The unobserved interactions.
- ▶  $\sigma(\cdot)$ : The sigmoid function.
- ▶  $\Theta = \{E, \{W_1^{(l)}, W_2^{(l)}\}_{l=1}^L\}$ : All trainable model parameters.
- ▶  $\lambda$ : Controls the  $\ell_2$ -regularization strength to prevent overfitting.
- ▶ The batch optimization uses mini-batch Adam.
- ▶ Two dropout methods *message dropout* and *node dropout*

# Neural Graph Collaborative Filtering (Wang et al. [2019])

## ● Previous work

### **Data pre-processing**

- Feature selection and pre-processing
- Bi-partitie graph structure
- Embedding initialization

### **Embedding propagation layers**

- First-order propagation layer
- High-order propagation layer
- Prediction layer

## ● To do work

- Improve embedding
- Sparse matrix
- Training
- Optimization

# Previous work

## Feature selection

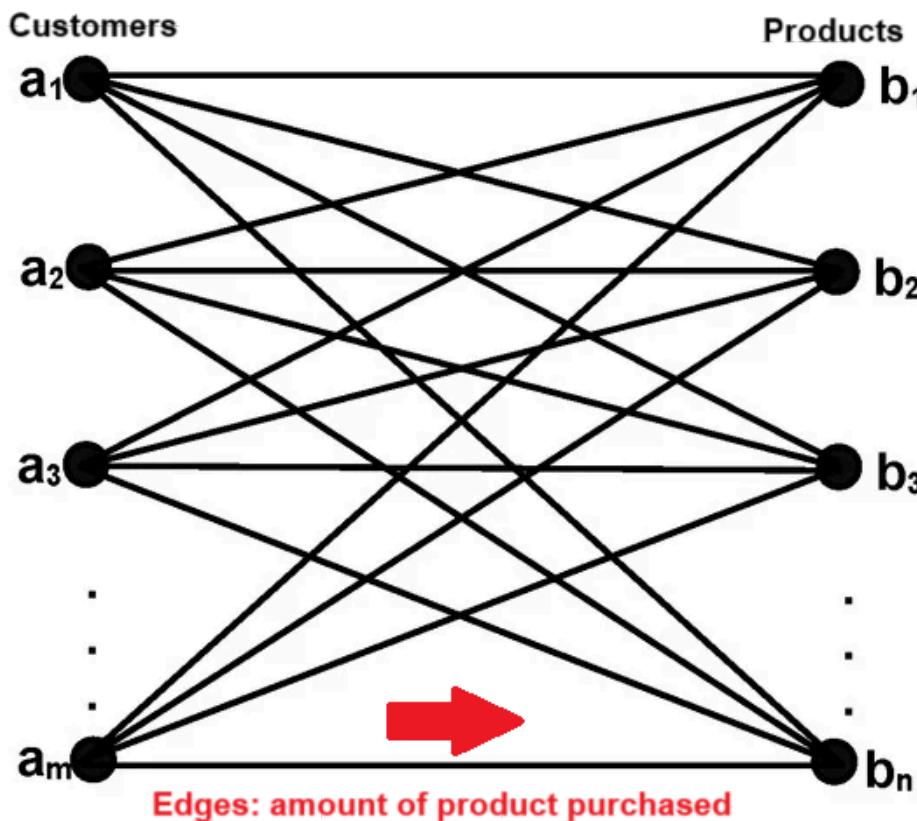
### Customer data

- Data aggregated by product and customer, shows quantity per product

### Products data

```
features_to_keep = ['product_id',  
'brand_key', 'shelf_level1', 'shelf_level2',  
'shelf_level3', 'bio', 'sugar_free',  
'gluten_free', 'halal', 'reduced_sugar',  
'vegetarian', 'vegan', 'pesticide_free',  
'no_added_sugar', 'salt_reduced',  
'no_added_salt', 'no_artificial_flavours',  
'porc', 'frozen', 'fat_free', 'reduced_fats',  
'fresh', 'alcool', 'lactose_free']
```

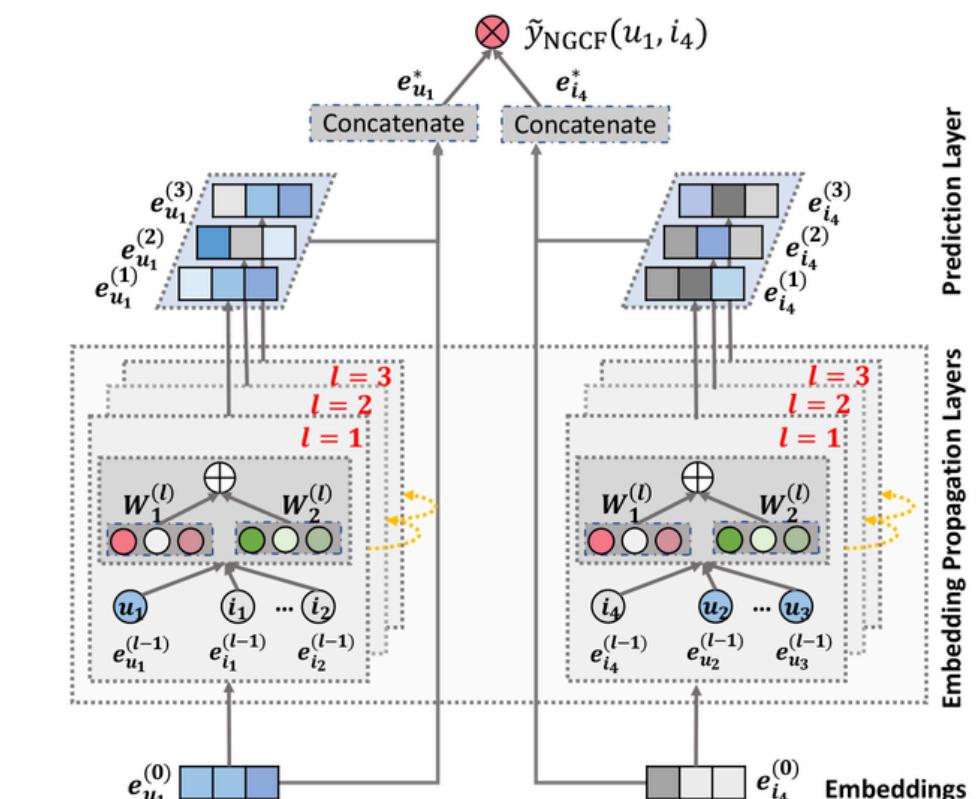
## Bi-partitie Graph



### Graph properties

- 100.000 customer nodes
- 82.150 product nodes
- 33.5M edges

## Embedding layers



### Embedding propagation layers

- First-order propagation layer
- High-order propagation layer
- Prediction layer

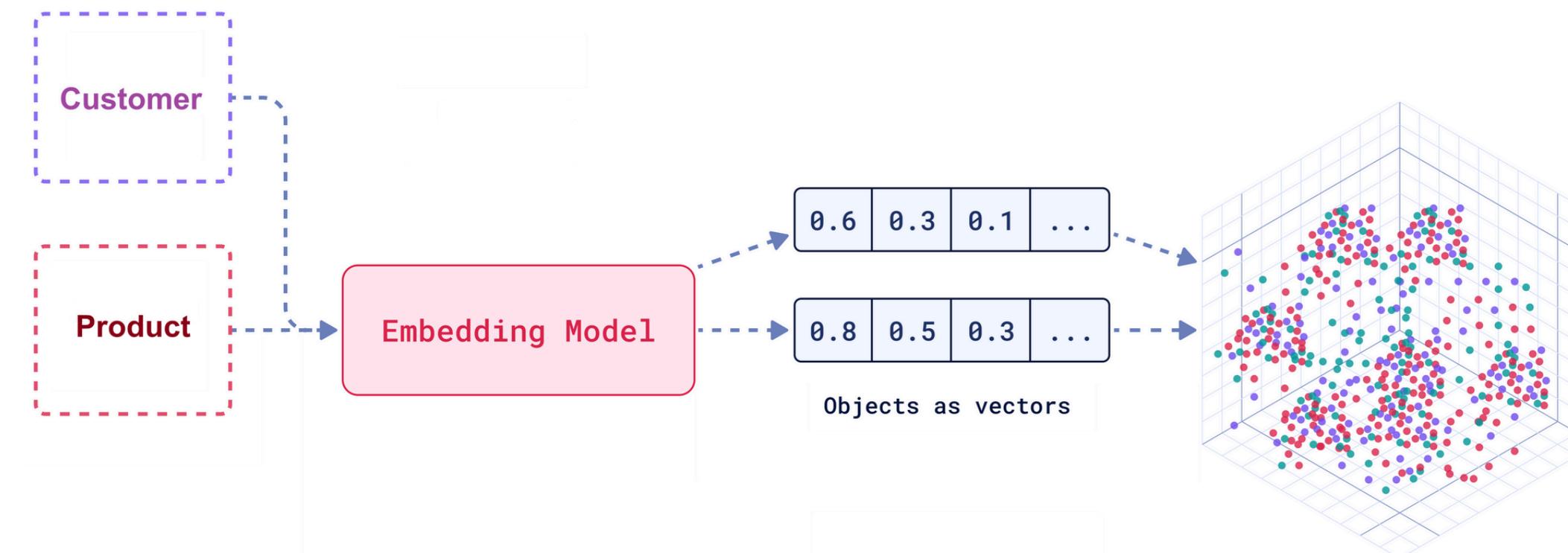
# Embedding intialization

## Before

- 128 embedding dimension
- Random initialization for customer and product embeddings

## Now

- 128 embedding dimension
- Random intialization for customer embeddings, then vectors are normalized
- Product embeddings intialized based on product features
- Both embeddings are concatenated in a single tensor
- Ensures all embeddings have the same dimension

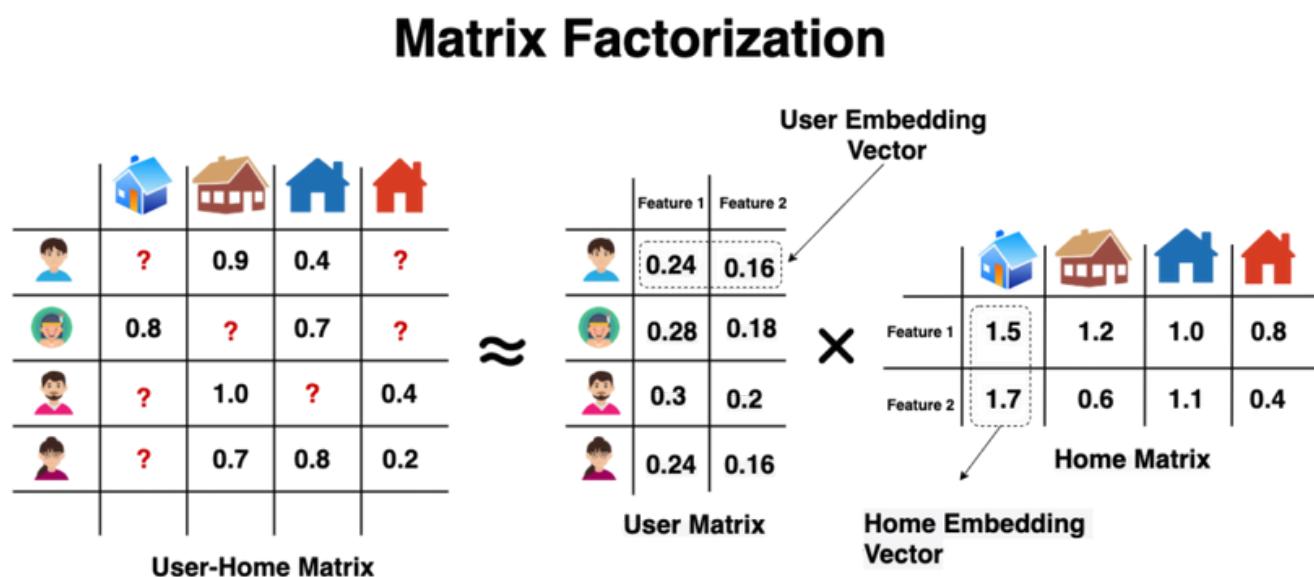


# Sparse Matrix

**Problem:** Amount of negative samples is greater than 99%

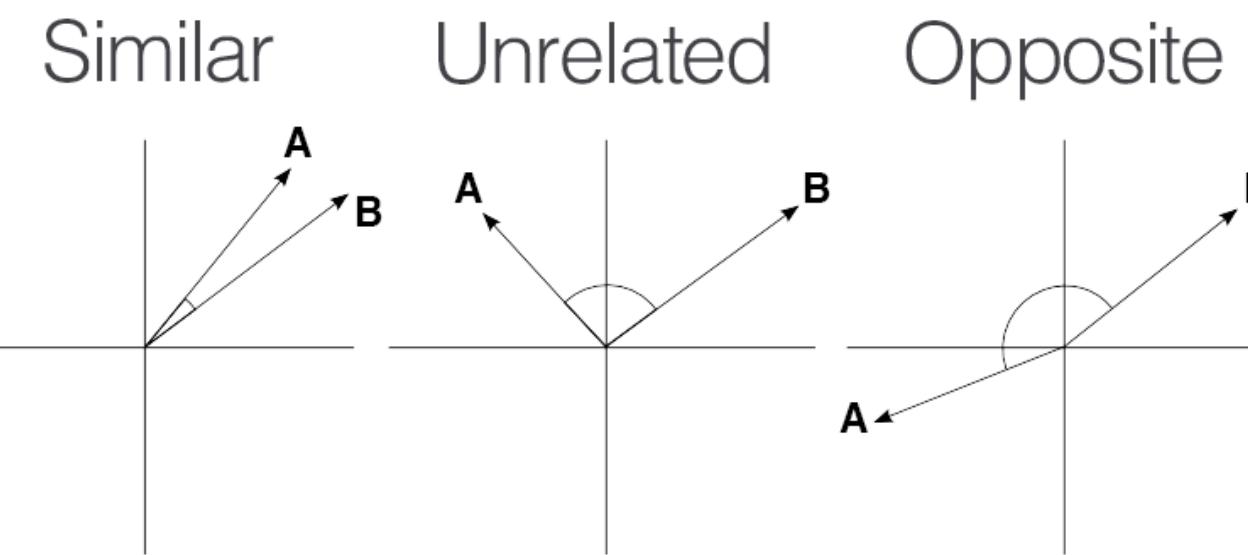
## Before

- Selecting 30 random embedding of negative products (products not purchased)



## Now

- Subset of 1000 samples randomly selected for every positive instance
- Embeddings of the sub-sets is normalized
- Cosine similarity between the positive instance and the negative ones
- The 5 most similar negative samples are selected, i.e. most challenging negative samples



# Other functions created

- **prepare\_data:** Extract user-item interactions from the data graph, apply negative sampling, return tensors for user and product indices.
- **prepare\_test\_data:** Extract user indices and their corresponding positive item indices from the test dataset.
- **train:** Data loading, forward pass, BPR loss calculation, backward pass.
- **test:** Data loading, prediction, hitrate@10 calculation,

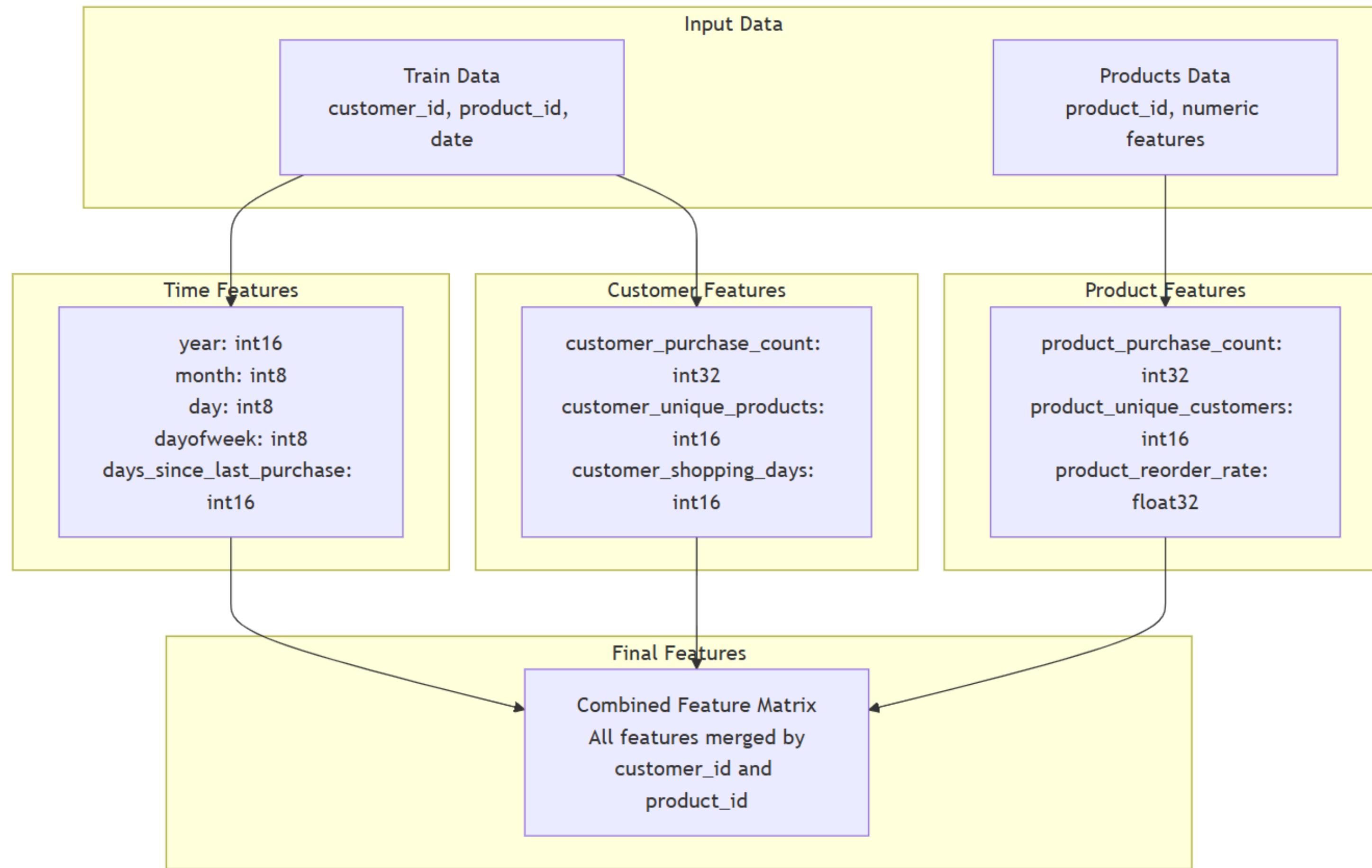
# Code optimization

- **GPU Usage:** Key computational steps, like cosine similarity, tensor operations, and loss calculations, are performed on the GPU for faster processing.
- **Parallelized Negative Sampling:** The negative sampling function (e.g., hard negative sampling) is parallelized using multiprocessing.
- **Subset Sampling for Hard Negatives:** Reduce computational overhead by sampling a smaller subset of items instead.
- **Downsampling Training Data:** A 10% random subsample of the original dataset is used during training for testing the algorithm.
- **Batch Processing:** Data is processed in batches during training and evaluation.
- **Pre-computation of Frequently Used Values:** Embeddings and indices are precomputed and cached, negative samples are computed in advance for each batch.

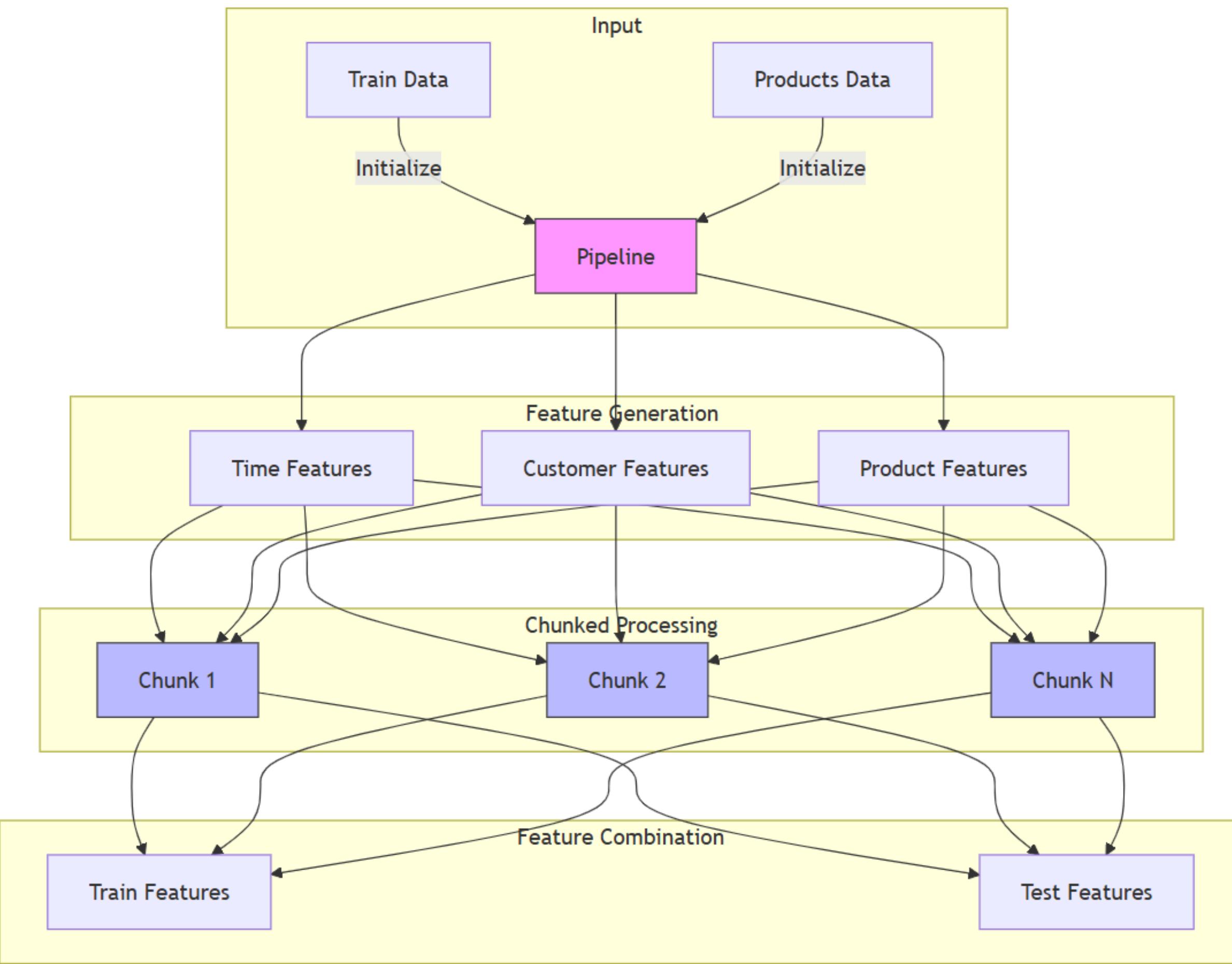
# To-do work

- Solve error about negative indices missmatch before training.
- Train the model and test it performance with the 10% sample.
- Run the code with the whole set on SLURM and predict new data.

# Feature Engineering



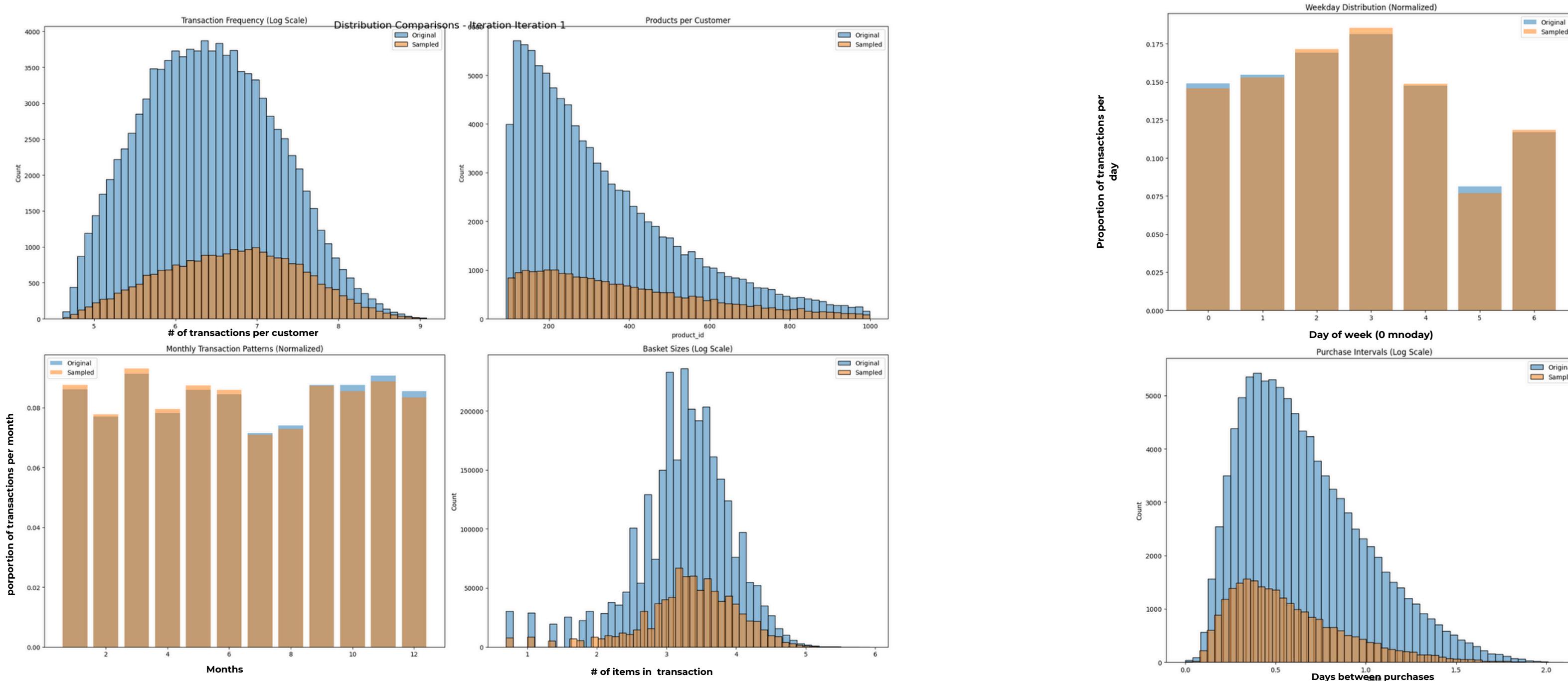
# Algorithm



# Randomized Sampling Alogirhtm

1. **Data Enrichment:** merging customer and product data for deeper analysis.
2. **Customer Weight Calculation:** Compute multidimensional weights based on different normalized features.
3. **Iterative Sampling:** Randomly sample customer using the calculated weights and compute dissimilarity scores against original data using the Kolmogorov-Smirnov test ([Smirnov, 1948](#)).
4. **Optimization:** select the sample that best preserves key patterns and relationships. (highest similarity score)
5. **Final Sample processing:** remove all the intermediate columns (weights) in step 3. keeping the original dataset features.

# Data Distribution: Original Vs Sampled



# Feature Selection

## 1. Temporal Features

- dayofweek, month, day, hour
- is\_weekend

## 2. Customer Features

- transaction\_count, store\_diversity
- product\_diversity, is\_promo, quantity
- lifetime

## 3. Product Features

- customer\_reach, purchase\_frequency
- store\_availability, promo\_frequency
- quantity patterns

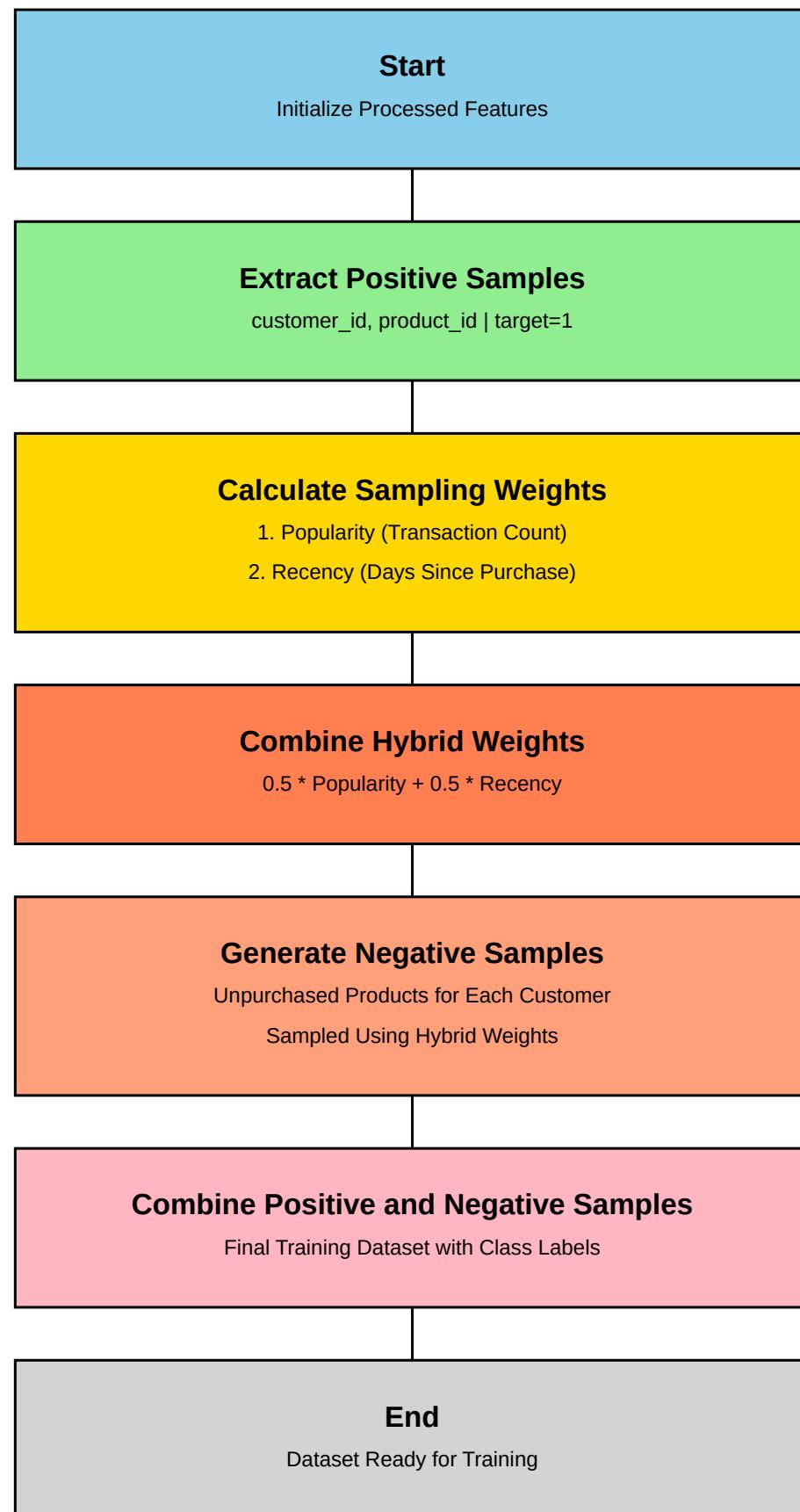
## 4. Interaction Features

- purchase frequency, promo ratio
- purchase timespan

## 5. Recency Features

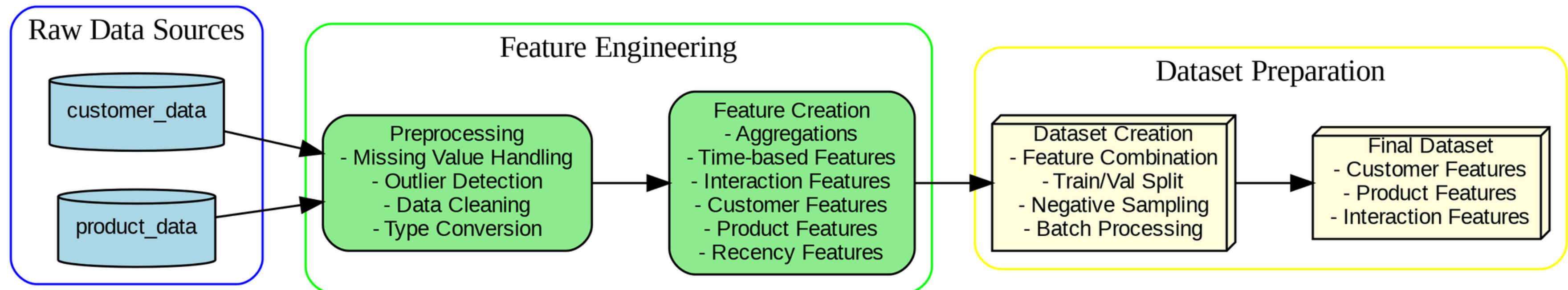
- days\_since\_purchase

# Negative Sampling

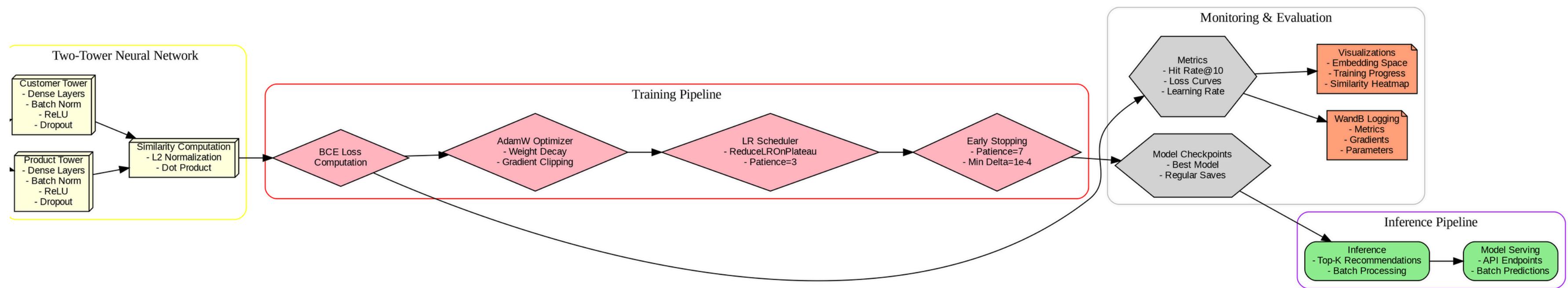


- **Combination ensures the samples are not random but targeted. (more informant).**
- **Hybrid weight ensures a balanced contribution of popularity and recency.**
- **We Probabilistically selects unpurchased products, ensuring relevance and diversity.**
- **We use a flexible negative-to- positive ratio (e.g., 2:1).**
- **This maintains a manageable class imbalance while improving model performance by exposing it to more challenging negative examples.**
- **The technique is scalable to large datasets (efficient NumPy operations).**
- **Parameters such as: weight ratios, negative sampling ratio, and sampling weights can be tuned to align with specific recommendation system needs.**

# Data Processing Pipeline



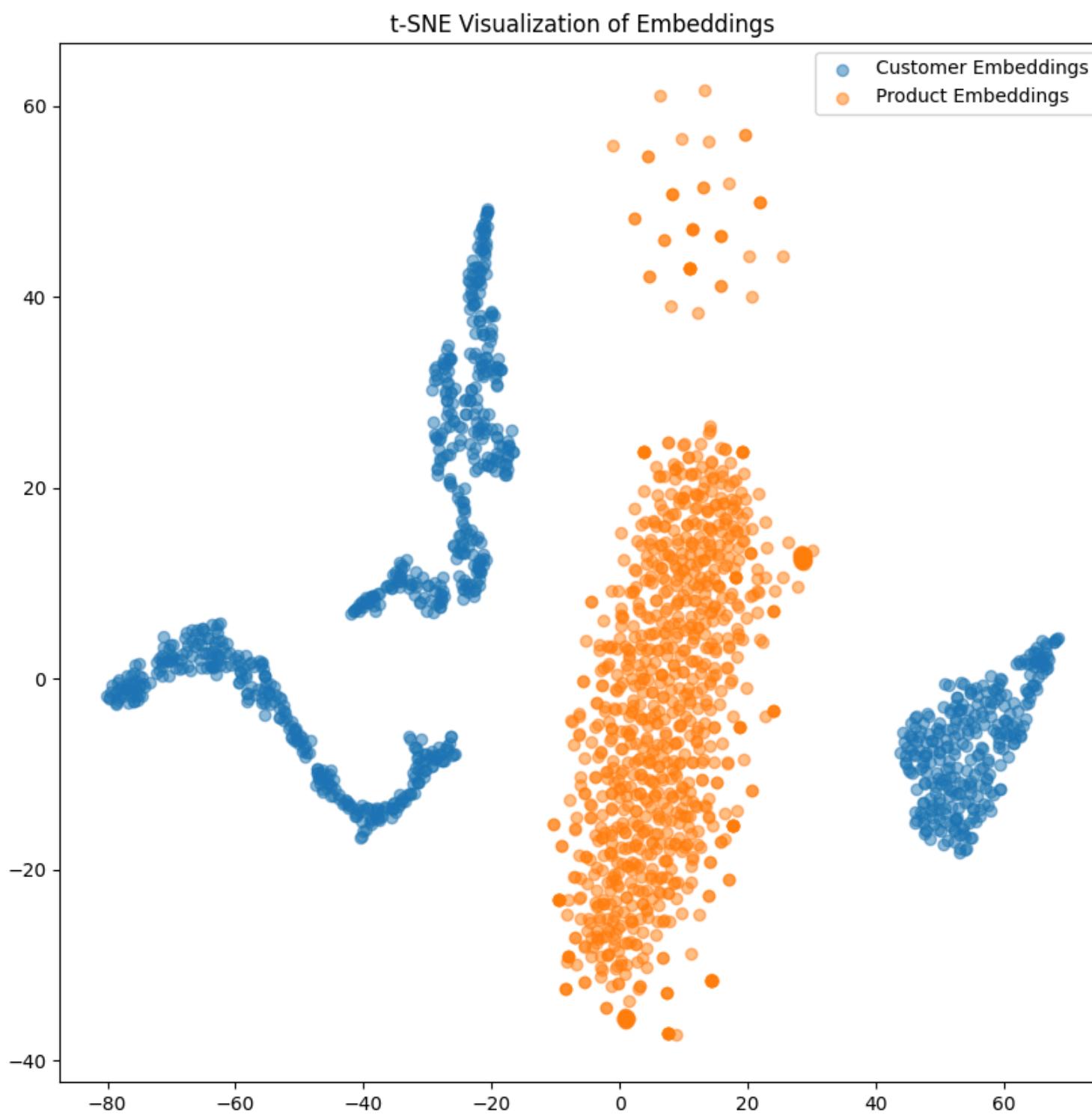
# Two Tower Architecture Diagram



# Results

```
23 Validating: 100%|██████████| 1148/1148 [03:36<00:00, 5.29it/s]
24 2024-12-16 06:12:17,099 - INFO - Epoch 6/10 - Train Loss: 0.3144, Val Loss: 0.3147, HR@10: 0.0055, LR: 0.005000
25 2024-12-16 06:12:17,102 - INFO - EarlyStopping counter: 4 out of 7
26 Training: 100%|██████████| 4591/4591 [13:57<00:00, 5.48it/s, loss=0.3137, lr=0.005000]
27 Validating: 100%|██████████| 1148/1148 [03:32<00:00, 5.41it/s]
28 2024-12-16 06:37:06,539 - INFO - Epoch 7/10 - Train Loss: 0.3137, Val Loss: 0.3133, HR@10: 0.0056, LR: 0.005000
29 2024-12-16 06:37:06,541 - INFO - EarlyStopping counter: 5 out of 7
30 Training: 100%|██████████| 4591/4591 [14:01<00:00, 5.46it/s, loss=0.3136, lr=0.005000]
31 Validating: 100%|██████████| 1148/1148 [03:34<00:00, 5.36it/s]
32 2024-12-16 07:02:04,815 - INFO - Epoch 8/10 - Train Loss: 0.3136, Val Loss: 0.3133, HR@10: 0.0055, LR: 0.005000
33 2024-12-16 07:02:04,818 - INFO - EarlyStopping counter: 6 out of 7
34 Training: 100%|██████████| 4591/4591 [14:02<00:00, 5.45it/s, loss=0.3134, lr=0.005000]
35 Validating: 100%|██████████| 1148/1148 [03:31<00:00, 5.42it/s]
36 2024-12-16 07:26:55,655 - INFO - Epoch 9/10 - Train Loss: 0.3134, Val Loss: 0.3134, HR@10: 0.0054, LR: 0.005000
37 2024-12-16 07:26:55,658 - INFO - EarlyStopping counter: 7 out of 7
38 2024-12-16 07:26:55,659 - INFO - Early stopping triggered
39 /tmp/
ipykernel_14926/2414452240.py:193: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the
default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
40 best_model_state = torch.load(os.path.join(self.config.CHECKPOINT_DIR, 'best_model.pt'))
41 Generating embeddings: 100%|██████████| 1148/1148 [03:33<00:00, 5.38it/s]
42 2024-12-16 07:30:36,772 - INFO - Performing final evaluation...
43 Validating: 100%|██████████| 1148/1148 [03:35<00:00, 5.33it/s]
44 2024-12-16 07:41:21,421 - INFO - Final HR@10: 0.0054
```

# Results Continued



## Run summary:

```
best_val_loss 0.31326
epoch 8
final_hr10 0.00537
hr10 0.00545
learning_rate 0.005
train_loss 0.31339
val_loss 0.31344
```

# Future Work

- **Improve current architecture.**
- **Use more than 10% only of the dataset.**
- **Explore more the GNN based collaborative filtering.**
- **Due to the reduction by the randomized sampling and the on the fly features embeddings, we want to experiment with GBM, XGboost, lightGBM models.**

**Thank you for your attention**

# APPENDIX

# Distribution Preserving Sampling Psuedo

---

**Algorithm 1** Distribution-Preserving Sampling Algorithm

---

**Require:** *train\_df*, *products\_df*, *sample\_fraction*, *n\_iterations*  
**Ensure:** A sampled dataset preserving key distributions

```
1: procedure DISTRIBUTIONPRESERVINGSAMPLE
2:   Step 1: Data Enrichment
3:   Merge train_df with products_df on product_id to get department_key
4:   Step 2: Calculate Customer Weights
5:   for all customer_id in train_df do
6:     Compute weights:
7:     transaction_count  $\leftarrow$  Number of transactions
8:     store_diversity  $\leftarrow$  Number of unique stores
9:     product_diversity  $\leftarrow$  Number of unique products
10:    basket_size_avg  $\leftarrow$  Average basket size
11:    dept_diversity  $\leftarrow$  Number of unique departments
12:    purchase_regularity  $\leftarrow$  Regularity of purchase intervals
13:   end for
14:   Normalize all weights and compute combined_weight
15:   Step 3: Iterative Sampling Process
16:   n_customers  $\leftarrow$  Number of unique customers
17:   n_customers_sample  $\leftarrow$  n_customers  $\times$  sample_fraction
18:   best_sample  $\leftarrow$  None, best_score  $\leftarrow$   $\infty$ 
19:   for i = 1 to n_iterations do
20:     Randomly sample n_customers_sample customers using combined_weight
21:     Extract transactions for sampled customers
22:     Compute similarity scores using Kolmogorov-Smirnov tests across:
        Transaction patterns, product diversity, store activity, temporal patterns, etc.
23:     if current similarity score < best_score then
24:       best_sample  $\leftarrow$  current sample
25:       best_score  $\leftarrow$  current similarity score
26:     end if
27:   end for
28:   Step 4: Return Final Sample
29:   Remove department_key column from best_sample
30:   return best_sample
31: end procedure
```

---

# Neural Graph Collaborative Filtering

## ● Previous work

- Created most of the part of a recommendation system, following the procedure of the paper “Neural Graph Collaborative Filtering”.

## ● To do work

- Solve error of “missmatch size in tensors” during the training phase.
- Optimize the code.
- Improve the hard sampling method to create an optimal and accurate sparse matrix

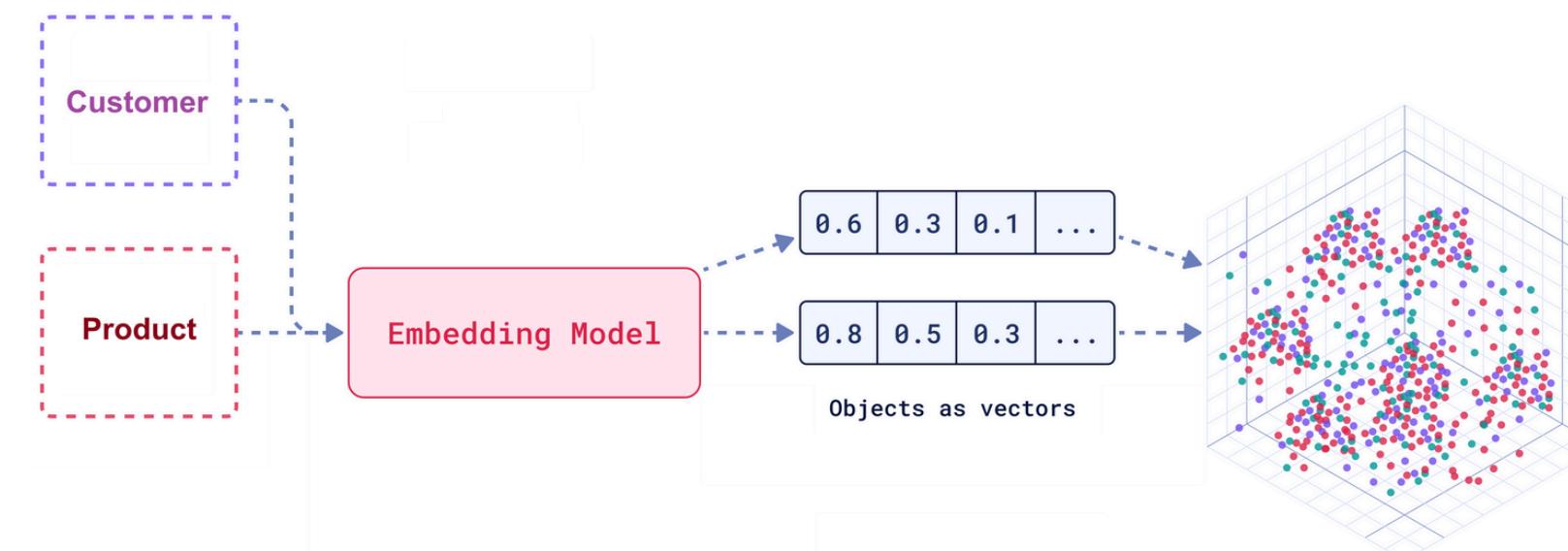
# Data Extraction

## Product dataset

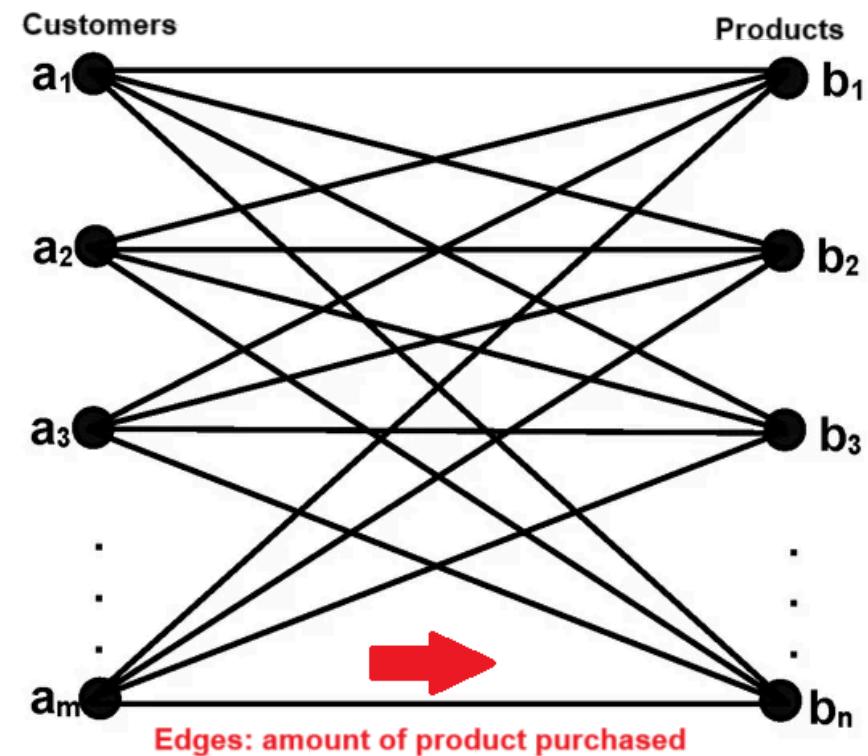
- Extracted and pre-processed more features from product dataset (went from 16 to 28) that will be taken into account in embedding initialization
- Additional features such as: 'ecoscore', 'artificial\_flavoring\_free', etc. can make a differential point for a customer to decide whether or not to purchase the item.

## Customer dataset

- Now data related to customer such as 'loyalty\_card', 'order\_channel', 'format' are taken into account.
- Customer embeddings are no longer initialized randomly.



## Bi-partitie Graph

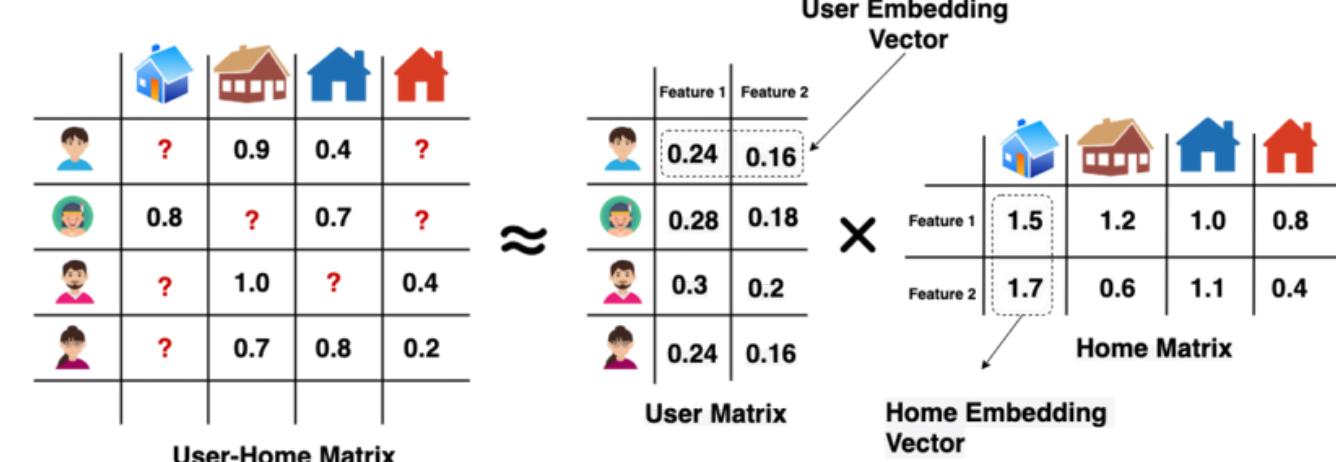


## Improvements

- Removed string attributes to speed up process and facilitate sparse matrix creation
- Indexing: 0-100k customers, 100k + products

## Sparse Matrix

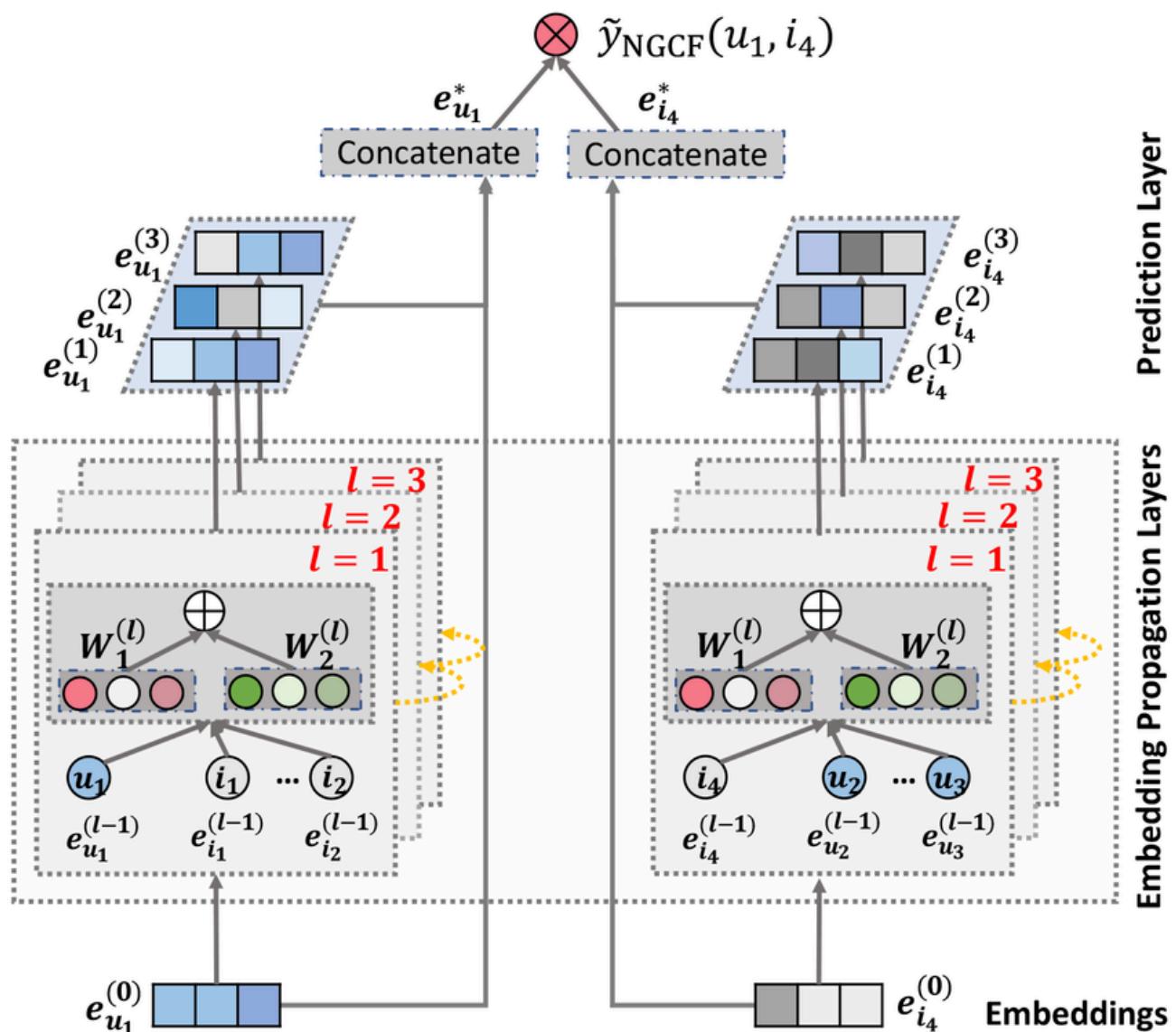
### Matrix Factorization



## Improvements

- Changed from cosine similarity to random selective sampling.
- **Hard negative sampling:** select one random negative instance from the same 'shelf\_level\_1' category
- Sparse matrix is now normalized

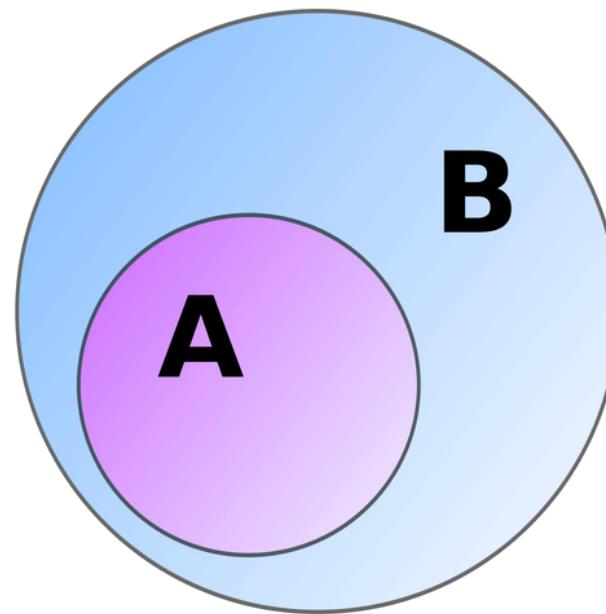
# Propagation Layers



## Improvements

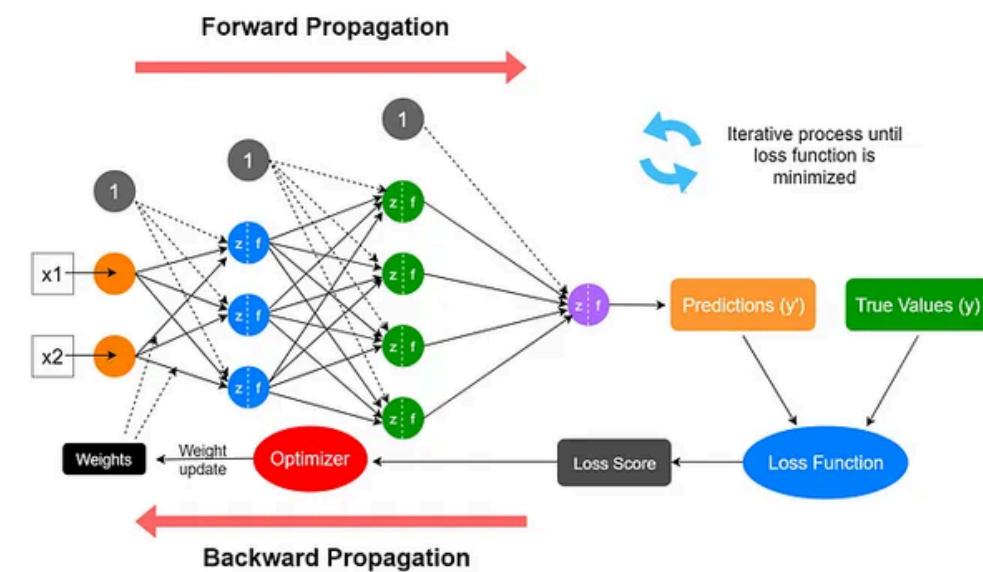
- **First-order propagation layer**
  - Added message dropout to prevent overfitting
  - Implemented LeakyReLU activation function
- **High-order propagation layers**
  - Applied node dropout to improve generalization
  - Optimized node embedding refinement by concatenating results with previous node embeddings
- **Optimization layer**
  - Applied BPR loss with regularization parameter
  - Solved problem of size tensors when calculating loss

## Subset extraction for testing



- Created a function that extracts a subset of customers
- It extracts information about customers and purchases made by the subset

## Training



- Solved inconvenience about mismatch in tensor sizes and dimensions.
- Added early stop if no improvement after 5 epochs.
- Trained for 100 epochs

# Results

- Created two different subset sizes to test model functionality
  - **First subset:** 0.1% of the original training data (100 customers)
  - **Second subset:** 1% of the original training data (1000 customers)

```
Epoch 68/100 begins...
Epoch 68: 100%|██████████| 32/32 [00:18<00:00, 1.78it/s, Loss=0.625473]
Epoch 68/100 completed. Loss: 19.744501
Epoch 68: Improvement (0.0270%) below threshold. (4/5)
Epoch 69/100 begins...
Epoch 69: 100%|██████████| 32/32 [00:17<00:00, 1.80it/s, Loss=0.624664]
Epoch 69/100 completed. Loss: 19.767059
Epoch 69: Improvement (-0.1143%) below threshold. (5/5)
Stopping early due to no improvement in loss.
```

Hit Rate @10: 0.0531

```
Epoch 51/100 begins...
Epoch 51: 100%|██████████| 32/32 [17<00:00, 1.78it/s, Loss=0.626988]
Epoch 51/100 completed. Loss: 19.901377
Epoch 51: Improvement (-0.1604%) below threshold. (4/5)
Epoch 52/100 begins...
Epoch 52: 100%|██████████| 32/32 [24<00:00, 1.32it/s, Loss=0.628172]
Epoch 52/100 completed. Loss: 19.904730
Epoch 52: Improvement (-0.1680%) below threshold. (5/5)
Stopping early due to no improvement in loss.
```

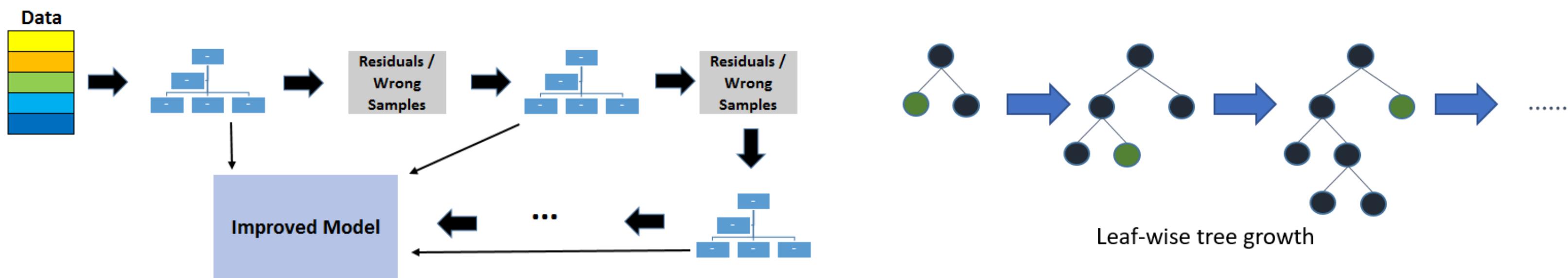
Hit Rate @10: 0.1227

## Unsuccessful results:

- Training the whole dataset. Google Colab VM disconnected after hours of running
- Tried training on a 30% subset. Google Colab still disconnects

# What is LightGBM

- **LightGBM** (Light Gradient Boosting Machine) is a high-performance gradient boosting framework designed for:
  - Speed and efficiency.
  - Handling large datasets and high-dimensional data.
  - Optimizing machine learning tasks like classification and regression.
- **Key Features:-**
  - Histogram-based learning
  - Leaf-wise tree growth
  - Built-in handling of categorical and sparse data



# Relative Negative Sampling

## Overview:

We used a negative sampling strategy to generate negative samples for training, ensuring they reflect realistic user interactions by weighting three key criteria:

### 1. Popularity:

a. Products frequently purchased by users receive higher weights.

### 2. Recency:

a. Products purchased more recently receive higher weights to reflect current trends.

### 3. Promotion Usage:

a. Products with higher promotional activity receive higher weights.

## Weighting Formula:

$$w_{\text{final}} = w_{\text{pop}} \cdot \frac{\text{popularity}}{\max(\text{popularity})} + w_{\text{rec}} \cdot \frac{1}{1 + \text{recency}} + w_{\text{promo}} \cdot \frac{\text{promo}}{\max(\text{promo})}$$

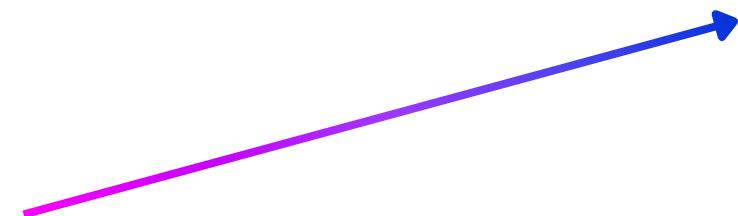
# Selected Features and Results

## Features:

- transaction\_count:
- unique\_transaction\_count:
- total\_quantity:
- avg\_quantity:
- promo\_ratio:
- days\_since\_first
- days\_since\_last
- purchase\_interval

## HitRate:

- **0.169** on test dataset
- **0.171** on kaggle



Thank you!