UNIVERSITÉ DE LYON

UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

MLDM PROJECT
PRODUCT PREDICTION

Presentated by:

Zahir AHMAD
Alejandro CARVAJAL
Amgad KHALIL

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:

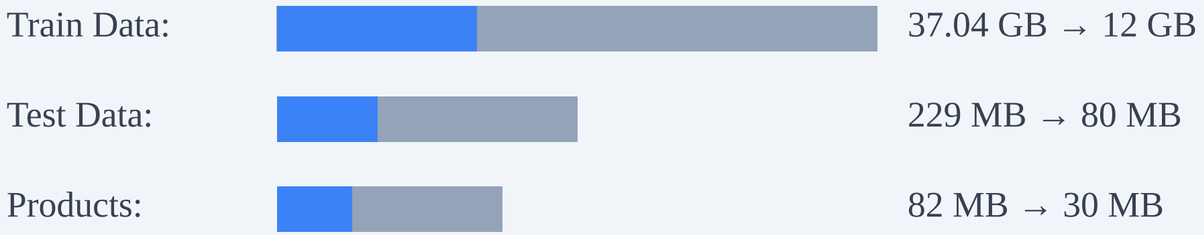
• Consistent data representation for categorical columns (no NaN)

• 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



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 (λ=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:

weight = exp(-0.1 × 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

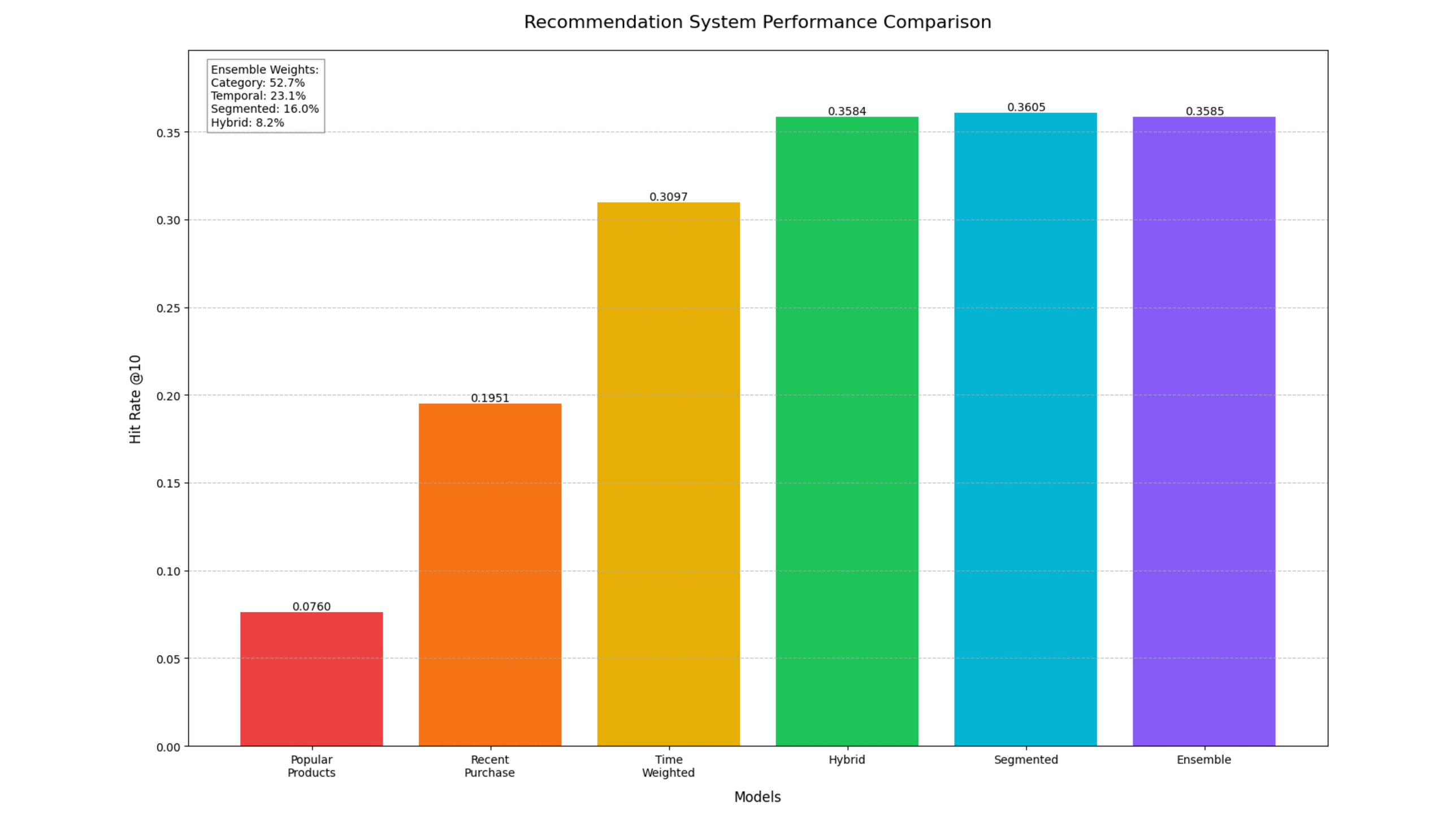
• Temporal: 23.1% weight

• Segmented: 16.0% weight

• Hybrid: 8.2% weight

Weights optimized through multiple trials

@hitrate10: 0.3585


Recommendation System Performance Comparison
Ensemble Weights:
Category: 52.7%
Temporal: 23.1%
Segmented: 16.0%
Hybrid: 8.2%
Hit Rate @10
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.0760
0.1951
0.3097
0.3584
0.3605
0.3585
Popular Products
Recent Purchase
Time Weighted
Hybrid
Segmented
Ensemble
Models

Neural Graph Collaborative Filtering

Previous work

Literature review

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

Data pre-processing

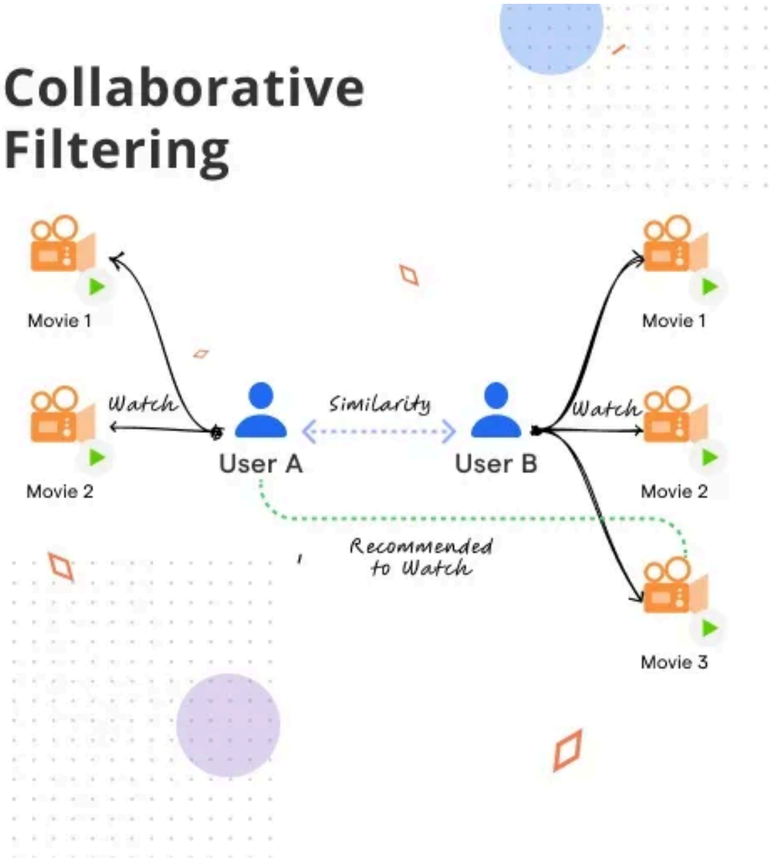
- Dataset creation (data agregation, join between tables, feature selection)
- Data normalization, frequency encoding
- Bi-partitie graph structure created

To do work

Continue with the implementation of the paper

Collaborative Filtering

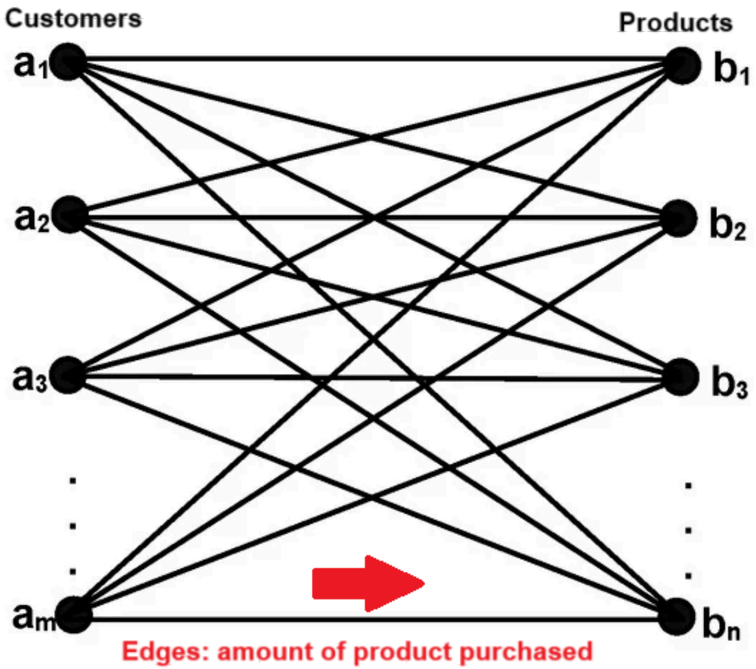
- Recommendation system technique used to predict user preferences by relying on patterns of user interactions or behaviors.
- Two different types: user-based and item-based



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

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.

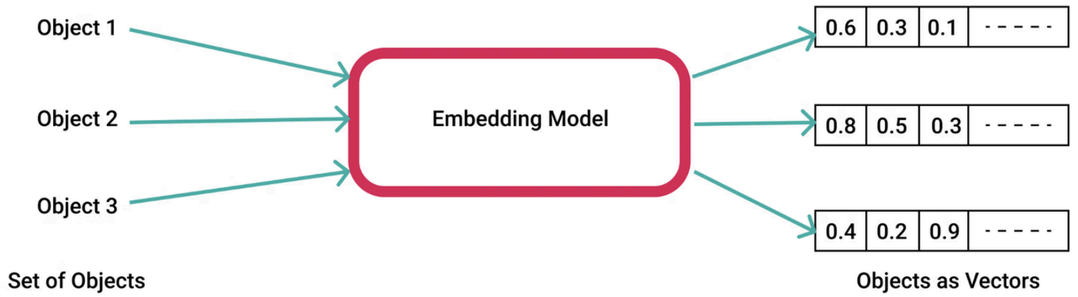
3) Prediction layer that aggregates the refined embeddings from different propagation layers

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:

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

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:

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

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 agregation

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:

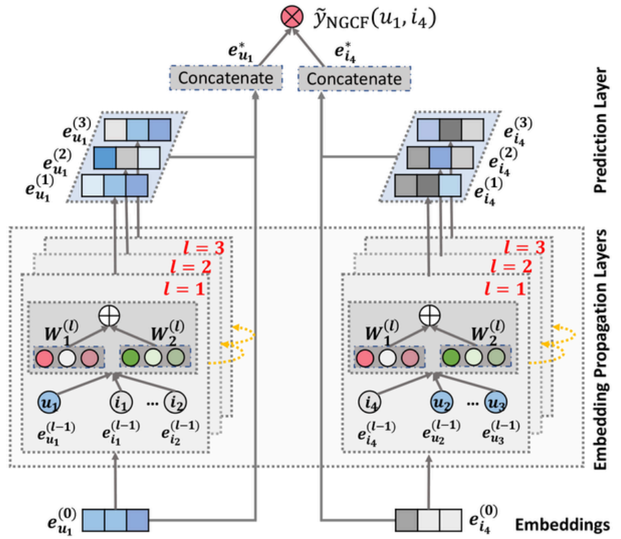
- $W_1^{(l)}, W_2^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$: Trainable transformation matrices.
- d_l: The transformation size at layer l.
- $\mathbf{e}_i^{(l-1)}$: The item representation generated from the previous message-passing step, which contains information from its $(l-1)$-hop neighbors.

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 $\{\mathbf{e}_u^{(1)}, \cdots, \mathbf{e}_u^{(L)}\}$.
- We concatenate them to constitute the final embedding for a user and item.
- The final representation is:

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

  where $\|$ 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:
  - $\mathcal{O} = \{(u, i, j) \,|\, (u, i) \in R^+, (u, j) \in R^-\}$ 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.
  - λ: Controls the ℓ_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

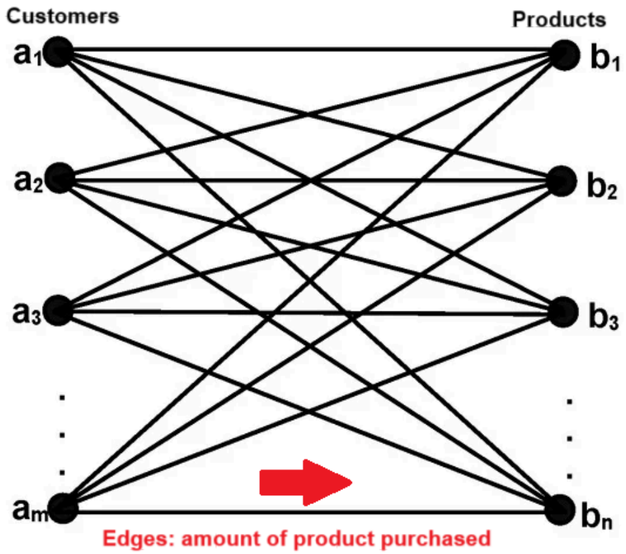
Customer data

- Data agregated 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']

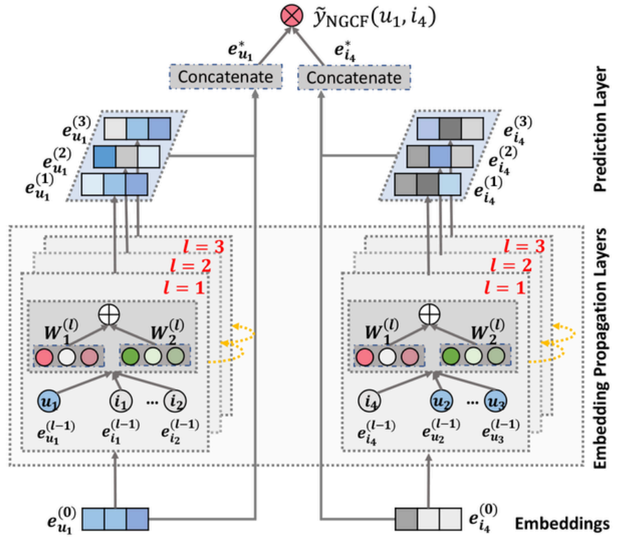
Bi-partitie Graph



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

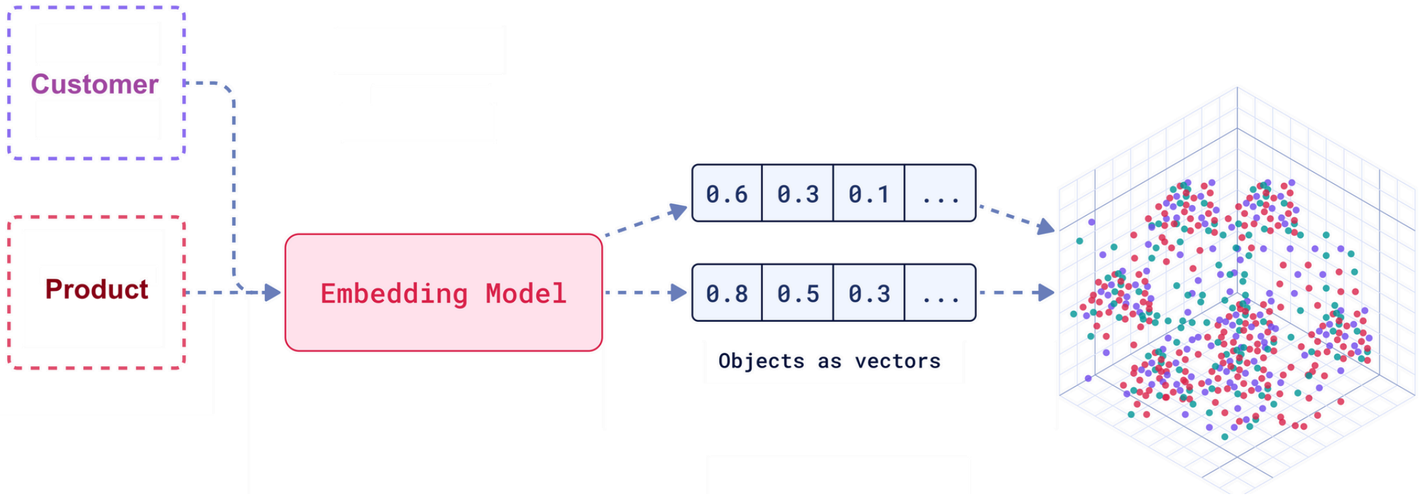
Embedding intialization

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

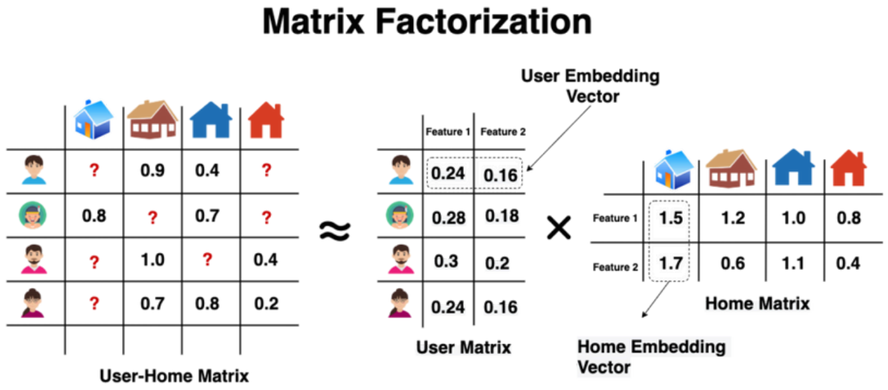


Sparse Matrix

Problem: Amount of negative samples is greater than 99%

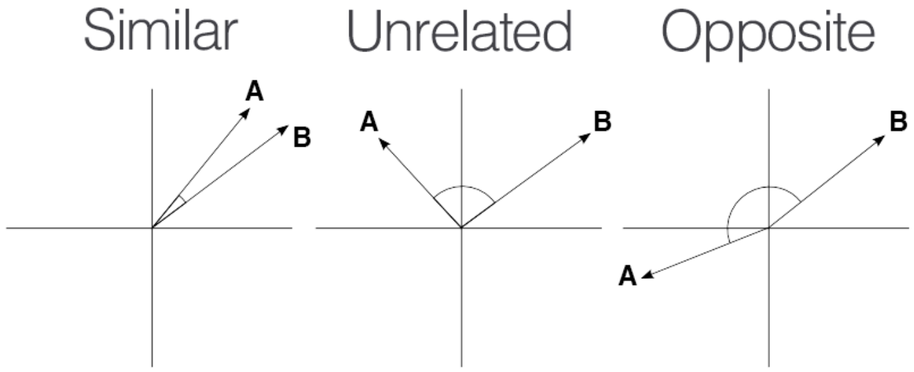
Before

- Selecting 30 random embedding of negative products (products not pruchased



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,

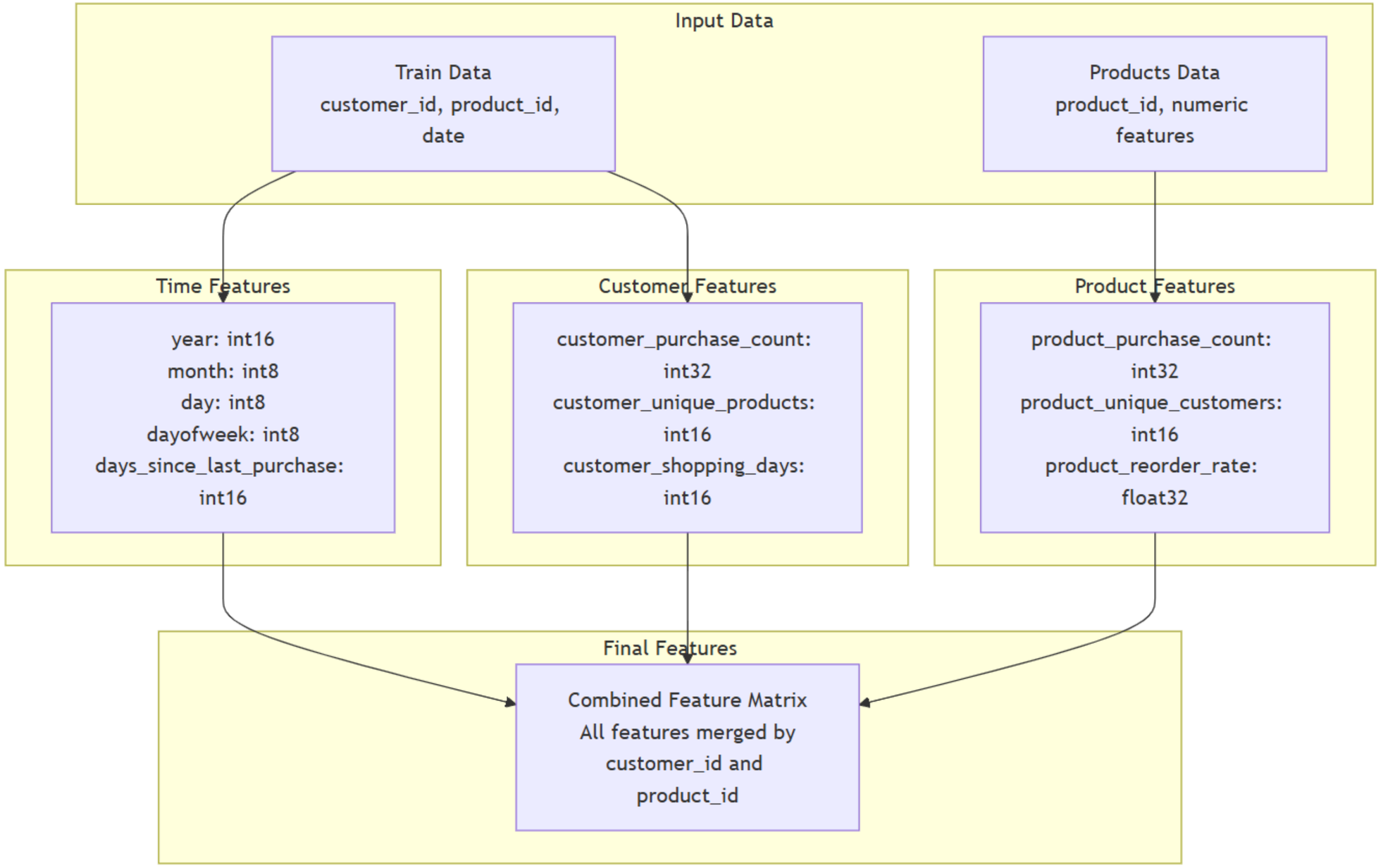
Code optimization

- **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.

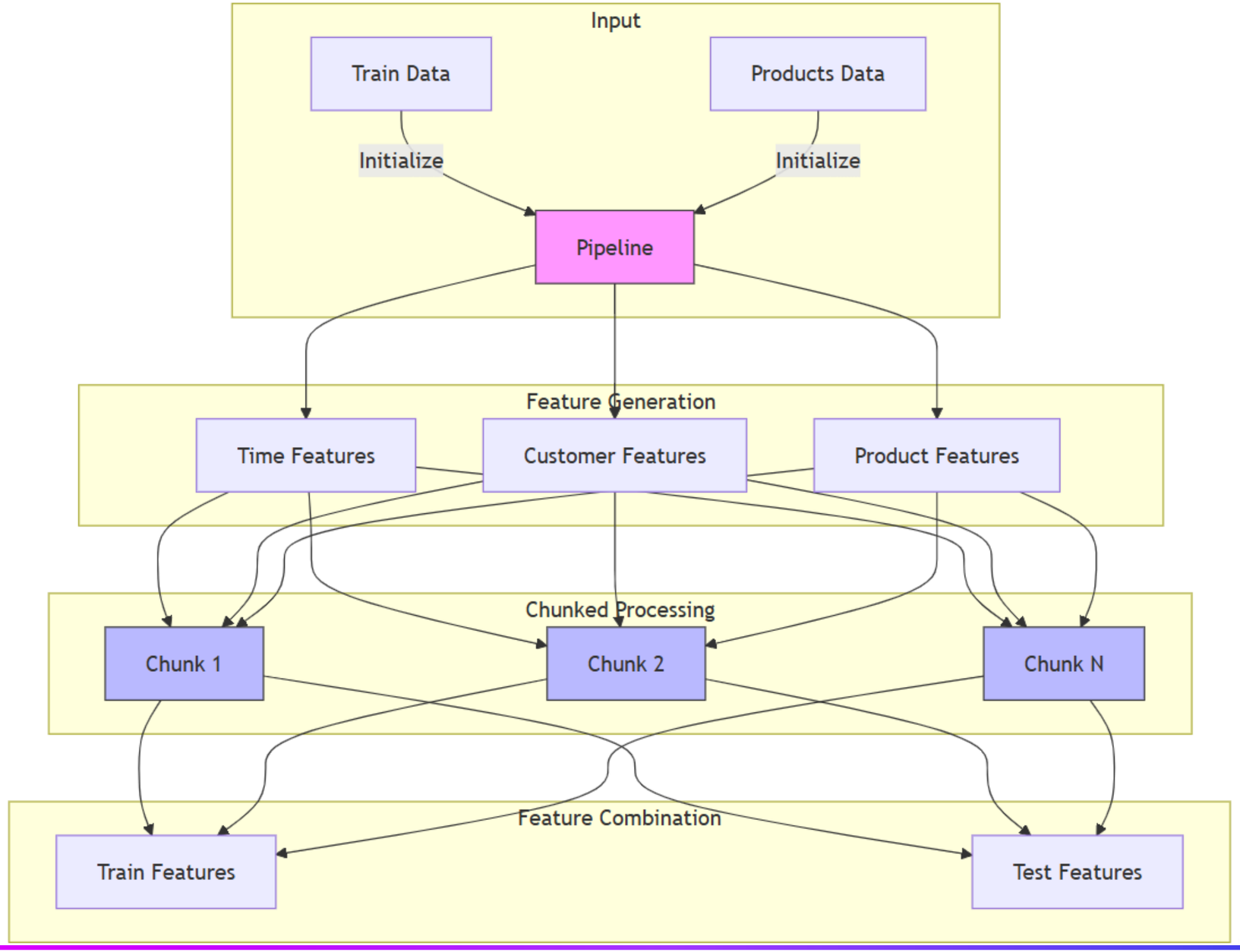
To-do work

- 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


Input
Train Data
Products Data
Initialize
Initialize
Pipeline
Feature Generation
Time Features
Customer Features
Product Features
Chunked Processing
Chunk 1
Chunk 2
Chunk N
Feature Combination
Train Features
Test Features

Neural Graph Collaborative Filtering

Previous work

Literature review

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

Data pre-processing

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

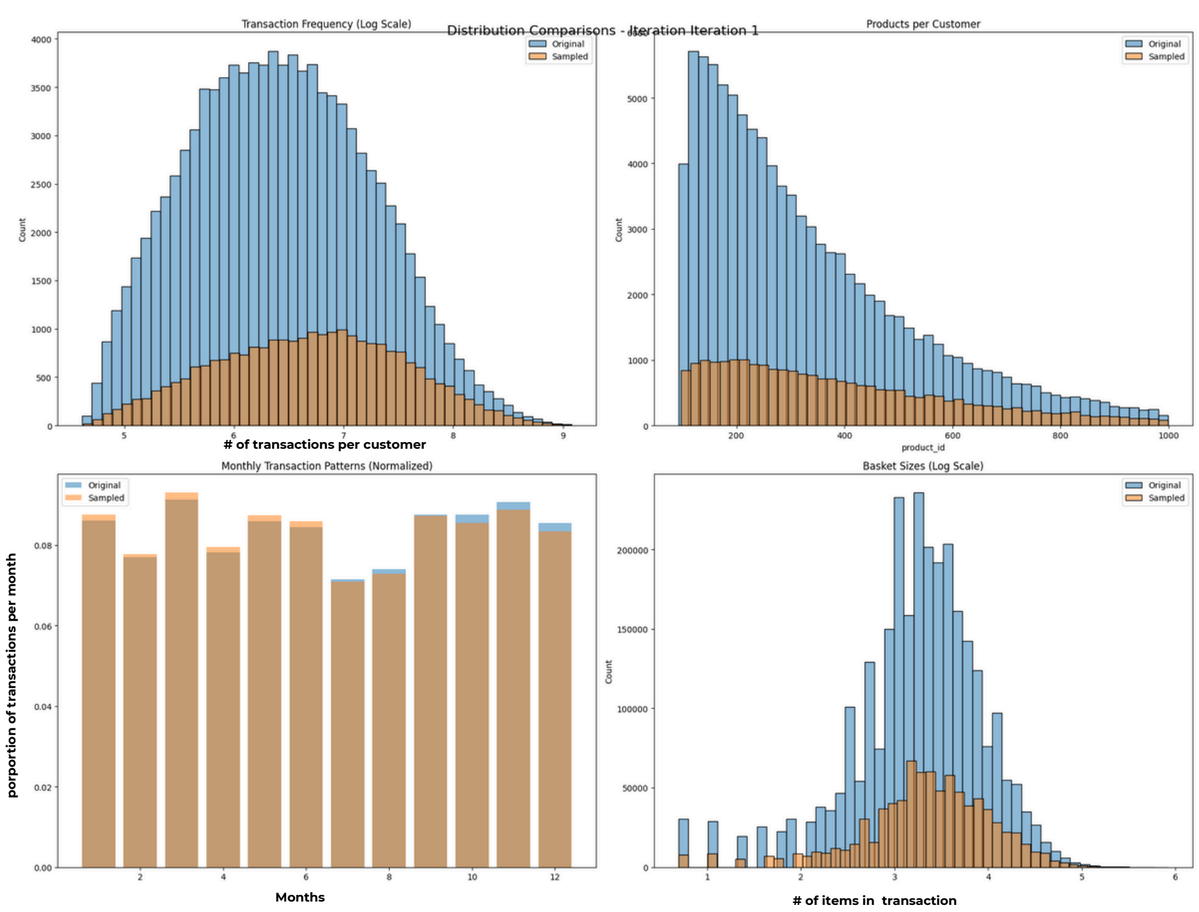
To do work

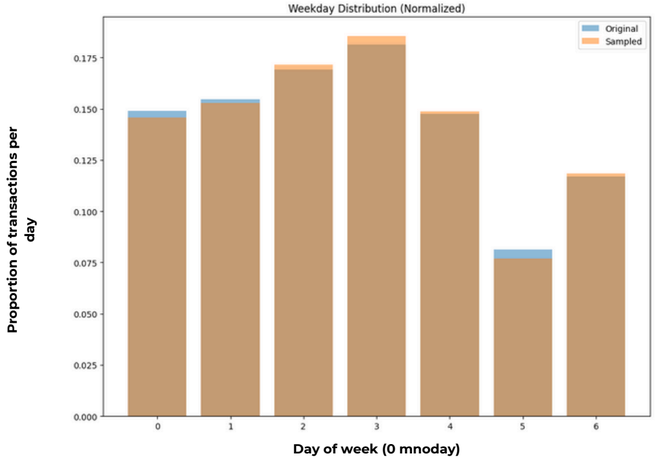
Continue with the implementation of the paper

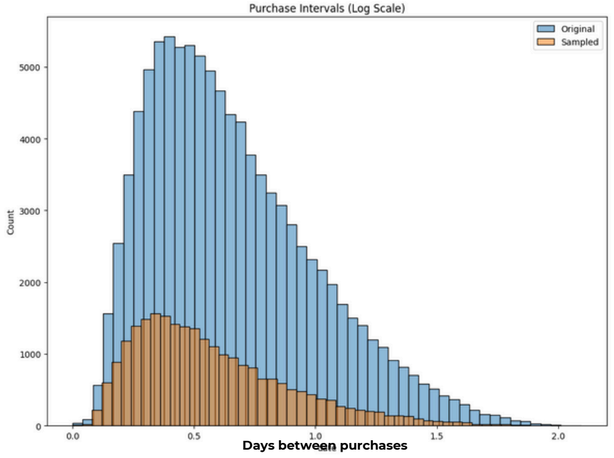
Randomized Sampling Alogirhtm

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

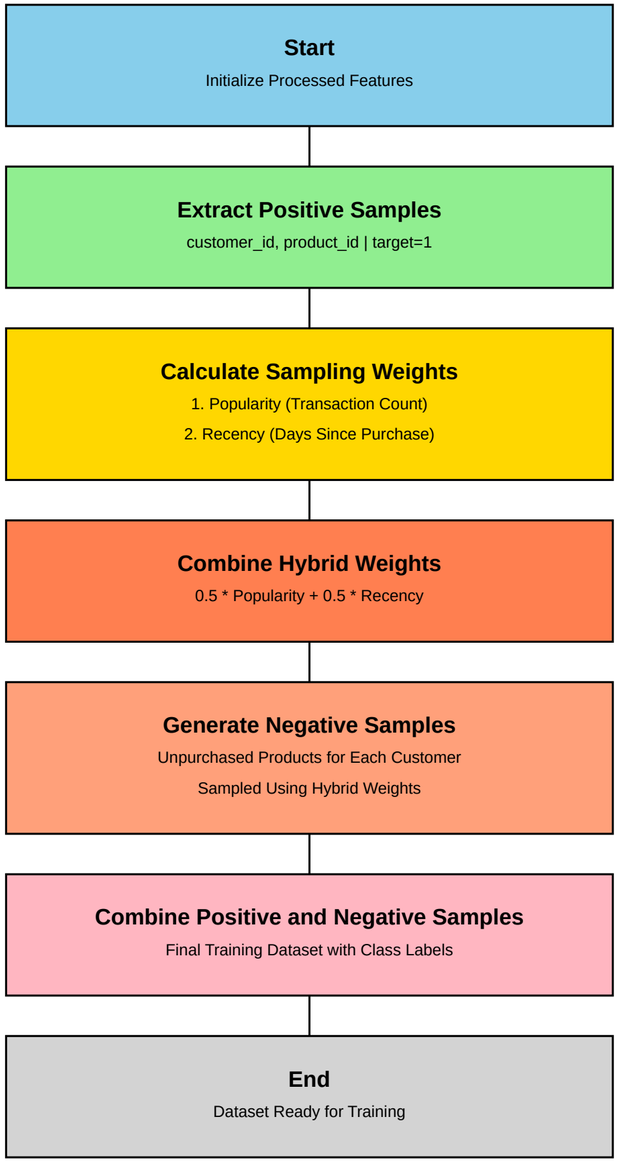
2. Customer Features
- transaction_count, store_diversity
- product_diversity, is_promo, quantity
- lifetime

3. Product Features
- customer_reach, purchase_frequency
- store_availability, promo_frequency
- quantity patterns

4. Interaction Features
- purchase frequency, promo ratio
- purchase timespan

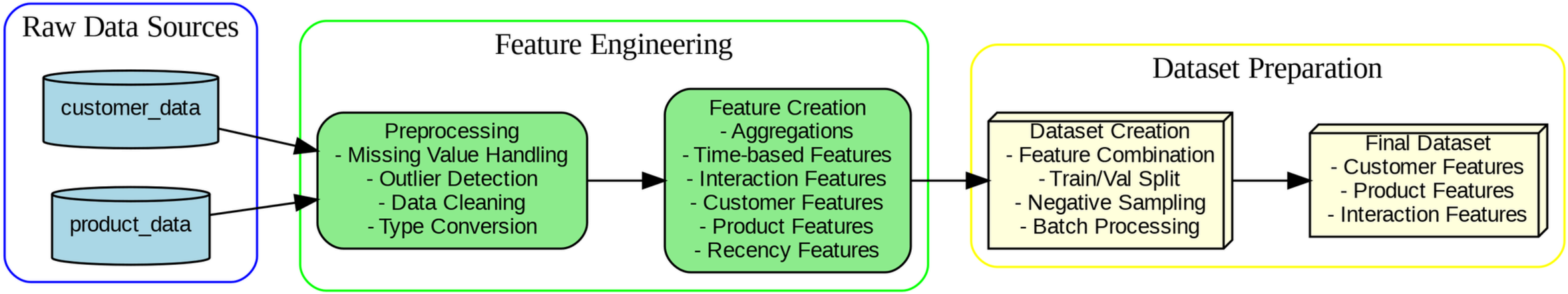
5. Recency Features
- days_since_purchase

Negative Sampling

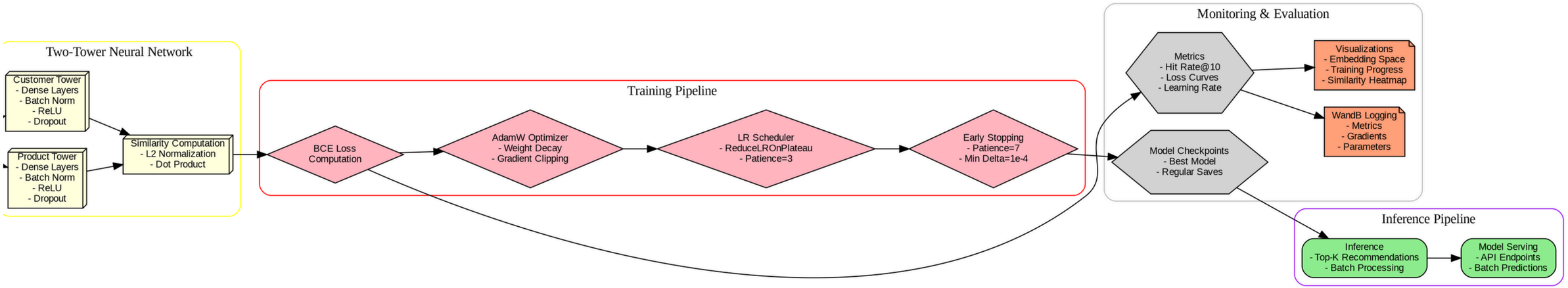


- **Combination ensures the samples are not random but targeted. (more informant).**
- **Hybrid weight ensures a balanced contribution of popularity and recency.**
- **We Probabilistically selects unpurchased products, ensuring relevance and diversity.**
- **We use a flexible negative-to- positive ratio (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



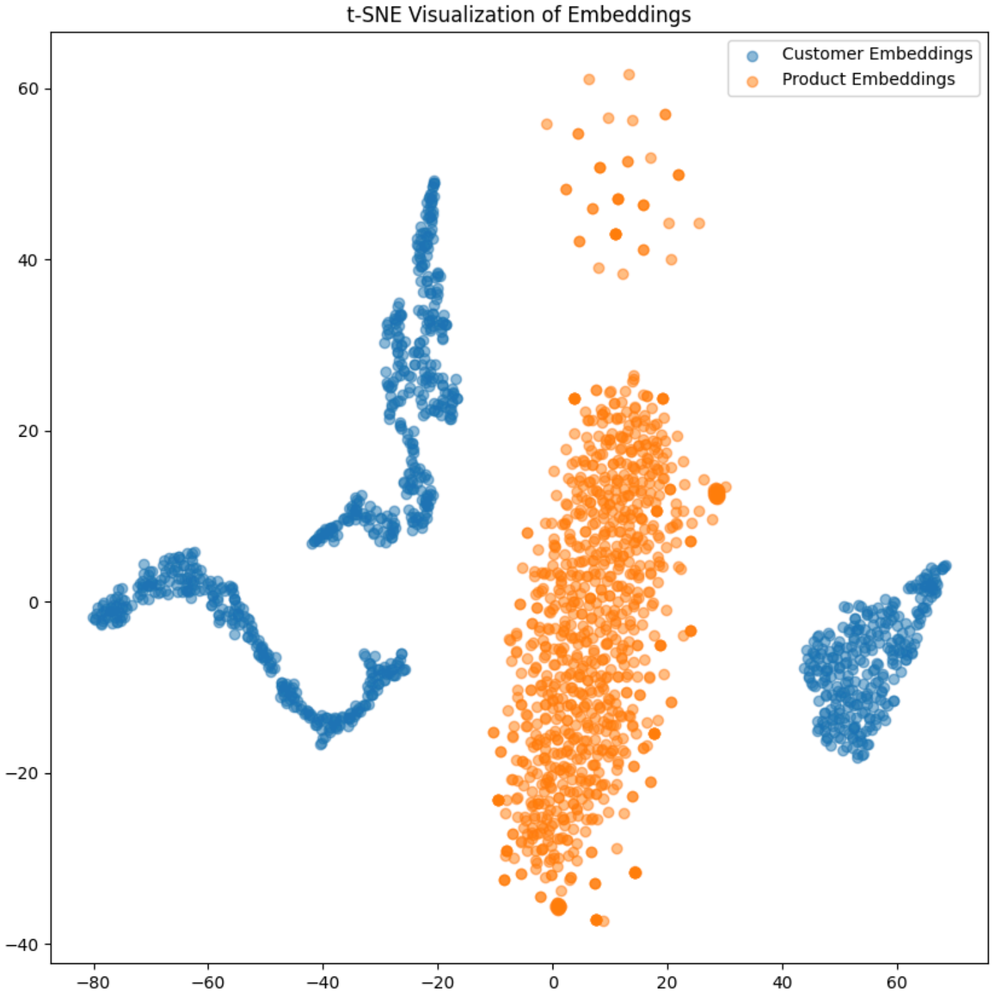
Two Tower Architecture Diagram



Results

```
Validating: 100%|██████████| 1148/1148 [03:36<00:00,  5.29it/s]
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
2024-12-16 06:12:17,102 - INFO - EarlyStopping counter: 4 out of 7
Training: 100%|██████████| 4591/4591 [13:57<00:00,  5.48it/s, loss=0.3137, lr=0.005000]
Validating: 100%|██████████| 1148/1148 [03:32<00:00,  5.41it/s]
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
2024-12-16 06:37:06,541 - INFO - EarlyStopping counter: 5 out of 7
Training: 100%|██████████| 4591/4591 [14:01<00:00,  5.46it/s, loss=0.3136, lr=0.005000]
Validating: 100%|██████████| 1148/1148 [03:34<00:00,  5.36it/s]
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
2024-12-16 07:02:04,818 - INFO - EarlyStopping counter: 6 out of 7
Training: 100%|██████████| 4591/4591 [14:02<00:00,  5.45it/s, loss=0.3134, lr=0.005000]
Validating: 100%|██████████| 1148/1148 [03:31<00:00,  5.42it/s]
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
2024-12-16 07:26:55,658 - INFO - EarlyStopping counter: 7 out of 7
2024-12-16 07:26:55,659 - INFO - Early stopping triggered
/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 http
s://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 coul
d 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.
  best_model_state = torch.load(os.path.join(self.config.CHECKPOINT_DIR, 'best_model.pt'))
Generating embeddings: 100%|██████████| 1148/1148 [03:33<00:00,  5.38it/s]
2024-12-16 07:30:36,772 - INFO - Performing final evaluation...
Validating: 100%|██████████| 1148/1148 [03:35<00:00,  5.33it/s]
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

Distibution 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

procedure DISTRIBUTIONPRESERVINGSAMPLE
    Step 1: Data Enrichment
    Merge $train\_df$ with $products\_df$ on $product\_id$ to get $department\_key$
    Step 2: Calculate Customer Weights
    for all $customer\_id$ in $train\_df$ **do**
        Compute weights:
        $transaction\_count \leftarrow$ Number of transactions
        $store\_diversity \leftarrow$ Number of unique stores
        $product\_diversity \leftarrow$ Number of unique products
        $basket\_size\_avg \leftarrow$ Average basket size
        $dept\_diversity \leftarrow$ Number of unique departments
        $purchase\_regularity \leftarrow$ Regularity of purchase intervals
    end for
    Normalize all weights and compute $combined\_weight$
    Step 3: Iterative Sampling Process
    $n\_customers \leftarrow$ Number of unique customers
    $n\_customers\_sample \leftarrow n\_customers \times sample\_fraction$
    $best\_sample \leftarrow None, best\_score \leftarrow \infty$
    for $i = 1$ to $n\_iterations$ **do**
        Randomly sample $n\_customers\_sample$ customers using $combined\_weight$
        Extract transactions for sampled customers
        Compute similarity scores using Kolmogorov-Smirnov tests across:
    Transaction patterns, product diversity, store activity, temporal patterns, etc.
        if current similarity score $<$ best_score **then**
            $best\_sample \leftarrow$ current sample
            $best\_score \leftarrow$ current similarity score
        end if
    end for
    Step 4: Return Final Sample
    Remove $department\_key$ column from $best\_sample$
    return $best\_sample$
end procedure