Kaggle Competition: Multi-Objective Recommender System

Aiden Wan Cheng-Han Wu

Master of Data Science
University of California, Irvine

1 Abstract

This work explores a multi-stage recommendation pipeline designed for the OTTO multi-objective recommender system competition. The pipeline begins by generating candidate items through session-level co-visitation signals, leveraging different weighting schemes for clicks, carts, and orders. These candidates are then re-ranked using an XGBoost model trained on session-item features such as temporal behavior, item popularity, and user-event interactions. Through five-fold cross-validation and extensive out-of-fold (OOF) predictions, the approach achieves a significantly higher recall than a simple baseline that recommends the last 20 actions. Potential future enhancements include embedding-based feature engineering (e.g., graph or sequence embeddings) and deep learning architectures (e.g., transformers, GNNs) to capture more complex patterns of user behavior.

2 Introduction

2.1 Problem Description

The competition aims to help OTTO, Germany's largest online retailer, develop a recommender system that predicts user actions, including clicks, cart additions, and orders, based on previous events within a shopping session.

A shopping session refers to the period when a customer interacts with an online store. Since customers may visit the platform multiple times, each user has their own multiple shopping sessions. Within a session, they engage with various products by clicking, adding items to their cart, or making purchases. As a result, retailers accumulate millions of session records, each capturing a sequence of interactions over a specific period.

The term "multi-objective" emphasizes the need for a unified model that predicts multiple outcomes at once. By analyzing a user's real-time behavior within a session, the model can determine which products they are most likely to click on, add to their cart, or purchase next. This capability allows retailers to deliver more relevant recommendations, ultimately enhancing the shopping experience and driving higher sales.

2.2 Exploratory Data Analysis (EDA)

2.2.1 Key Features

- 12M real-world anonymized user sessions
- 220M events, consisting of clicks, carts, and orders

2.2.2 Dataset Statistics

See the Table 1.

Dataset	#sessions	#clicks	#carts	#orders
Train	12,899,779	194,720,954	16,896,191	5,098,951
Test	1,671,803	12,340,303	1,155,698	355,292

Table 1: Dataset Statistics

2.2.3 Data Format

The sessions are stored as JSON objects containing a unique session ID and a list of events:

Listing 1: Sample Session Data

```
{
    "session_id": 12345,
    "user": "guest",
    "events": [
        {"event": "click", "item_id": 101,
            "timestamp": "2025-03-15T12
            :00:00Z"},
        {"event": "cart", "item_id": 102,
            "timestamp": "2025-03-15T12
            :05:00Z"},
        {"event": "purchase", "item_id":
            102, "timestamp": "2025-03-15
            T12:10:00Z"}
    ]
}
```

2.2.4 Descriptive statistics

- Events number per session

Statistic	Train	Test
Mean	16.80	8.29
Std	33.58	13.74
Min	2	2
50%	6	4
75%	15	8
90%	39	18
95%	68	28
Max	500	498

Table 2: Transposed Statistics of Events per Session

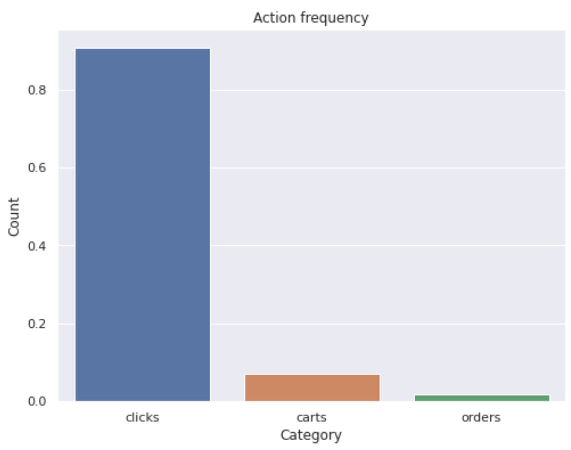


Figure 1: Action Frequency

- Action frequency
  See the Figure 1.
- Action frequency
  See the Figure 2.

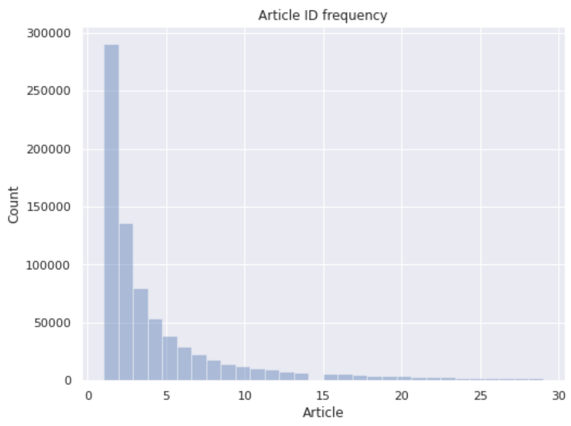


Figure 2: Action Frequency

3 Method

3.1 Two-stage Recommendation

In large-scale e-commerce scenarios, we adopt a two-stage recommendation pipeline—candidate generation followed by reranking—to balance performance and efficiency. First, we identify a relatively small subset of "likely relevant" items from a massive product catalog, using lightweight methods such as co-visitation matrices or popular-item heuristics. These candidates are then passed to a more sophisticated model that reranks them based on richer features and user-context information. This division of labor significantly reduces computational overhead, since the expensive model only processes a few hundred candidates instead of the entire product set.

3.2 Candidates Generation

3.2.1 Co-visitation Matrix

Below are the three distinct methods implemented to compute the co-visitation matrix based on session-level item co-occurrence. Each method is based on the core idea of generating item pairs from user sessions, but they differ in how they filter interactions and assign weights.

General Co-visitation Calculation For each session, up to 30 of the most recent events are retained. A self-join is performed to generate item pairs where the two items are distinct and occur within 24 hours of each other. Each pair is assigned a weight based on the event type of the second item (click, cart, or order). These weights are aggregated across sessions, and the top 20 candidates for each item are selected.

Co-visitation for Carts and Orders This method follows the same process as the general co-visitation approach but considers only cart and order events. Additionally, a 14-day time window is applied to filter item pairs. All pairs receive a uniform weight, which is summed over sessions. The top 20 candidates per item are then retained.

Time-Weighted Co-visitation for Clicks Click events are processed similarly to the general method, using a 24-hour window and up to 30 recent events per session. However, instead of a constant weight, each pair is weighted dynamically based on recency:

$$\text{wgt} = 1 + 3 \times \frac{\text{ts}_x - 1659304800}{1662328791 - 1659304800} \tag{1}$$

This assigns greater importance to more recent interactions. After aggregation, the top 25 candidates per item are selected.

3.2.2 Generate Candidates

The candidate generation method relies primarily on precomputed co-visitation matrices, with a backup suggestion mechanism derived from user history.

Click-Based Candidate Generation Candidates are retrieved using a co-visitation matrix by iterating over unique session items in reverse chronological order. If the session history contains at least 20 unique items, a re-ranking is applied using logarithmically spaced weights:

$$\text{weights} = \text{logspace}(0.1, 1, \text{len}, \text{base} = 2) - 1 \tag{2}$$

Each item's weight is adjusted by an event-type multiplier, and the top 25 items are selected. If the session history is short, the candidate list is supplemented with frequently clicked items from the test set.

Purchase-Based Candidate Generation For purchase events, two co-visitation matrices are used: one combining carts and orders, and another focusing on buy-to-buy relationships. If the purchase history is long enough, re-ranking is performed with:

$$\text{weights} = \text{logspace}(0.5, 1, \text{len}, \text{base} = 2) - 1 \tag{3}$$

Weighted scores are aggregated, and the top items are retained. If necessary, additional candidates are drawn from co-visitation matrices and top-ordered items from the test data.

Merging and Final Candidate Assembly Click-based and purchase-based candidates are merged and deduplicated. If fewer than 50 candidates remain, top-ordered items from the full training data are added to complete the list, ensuring a final selection of 50 items per session.

3.3 Re-ranking

3.3.1 Model Selection

We choose to build a XGBRanker for the re-ranking part. The reasons are as follows:

XGBoost employs Gradient Boosting, an iterative learning process that improves prediction accuracy by minimizing errors. The process begins with an initial model making a prediction, followed by calculating the residuals (errors). A decision tree is then trained to learn and correct these errors, adjusting the predictions accordingly. This cycle repeats until the model reaches optimal performance.

XGBoost's Sparse Aware Technology automatically handles missing values by optimizing information gain. Instead of requiring explicit imputation, it dynamically assigns missing values to the left or right child node based on which minimizes the loss function. This is particularly useful for features with frequent missing values, such as `session_last_click_aid` in session-level features, where a candidate product may lack click data. Traditional imputation methods, like removing samples with missing values, filling them with mean, median, or mode may introduce bias, and using algorithms like KNN or Random Forest for imputation can cause information loss, bias, or high computational costs. XGBoost's approach eliminates these issues, ensuring efficient and accurate model performance.

3.3.2 Feature Engineering

In this section, we introduce the newly engineered features used to enhance the performance of the recommender system. These features are categorized into three groups: session-level features, item-level features, and interaction features. They provide insights into user behavior, product popularity, and session-based interactions. For each session, these features are extracted specifically for candidate products that are considered for prediction, ensuring more accurate and relevant recommendations. For more detail on feature engineering, refer to the GitHub link.

4 Experiments

4.1 Environment

- **Co-visitation Calculation:** Performed on Kaggle Notebooks with two T4 GPUs.
- **Feature Engineering:** Performed on a 32GB MacBook Air M2.
- **Model Training:** Performed on Google Colab Pro using an A100 GPU.

4.2 Data Preparation

4.2.1 Generate Train and Validation Data

Our train data consists of the first 3 weeks of Kaggle train data, while our validation data is taken from the last week. The validation data is further split into validation data A and validation data B, where validation B contains the ground truths. For every session in validation data A, we generate X candidate aids. To minimize memory footprint, we divided the `ts` column by 1000.

4.2.2 Labeling

For each session, labels are assigned to candidate products to indicate whether they will be clicked, added to the cart, or purchased in subsequent interactions:

- **Clicks:** 1 if clicked next within the session, otherwise 0.
- **Carts:** 1 if added to the cart next within the session, otherwise 0.
- **Orders:** 1 if purchased next within the session, otherwise 0.

4.3 Model Training

4.3.1 Setup and Parameter Initialization

```
xgb_parms = {
    'objective':'rank:pairwise',
    'eval_metric':'map',
    'tree_method':'hist',
    'device': 'cuda',
    'learning_rate':0.1,
    'max_depth':4,
    'subsample':0.7,
    'colsample_bytree':0.5,
    'random_state': 42
}
```

4.3.2 Data Loading and Preprocessing

The full dataset is stored in a single parquet file and read using Dask to efficiently handle large data volumes. Then infinity values in features (e.g., those derived from rates) are replaced with NaN values because XGBoost cannot process infinity.

4.3.3 Cross-Validation Setup

A specific fold is chosen manually (e.g., fold 0) from a predefined list of folds. Each fold contains two sets: one for training sessions and one for validation sessions. The Dask DataFrame is filtered using the session identifiers for the current fold. The filtered training and validation data are then computed into Pandas DataFrames.

4.3.4 Constructing the XGBoost DMatrix

To construct the XGBoost DMatrix for ranking tasks, we first compute group sizes based on the number of events per session, which XGBoost requires for ranking objectives. We then create both training and validation matrices from the chosen features and target labels, and assign the computed group information to each DMatrix so that XGBoost can properly handle per-session ranking.

4.3.5 Model Training

The model is trained using `xgb.train` with a high number of boosting rounds (10,000) but with early stopping after 200 rounds if no improvement is observed on the validation set. Verbose output is provided every 100 rounds. Once training is complete, the model is saved to disk with a filename indicating the current fold, target type, and version number.

4.3.6 Validation Inference

Due to potentially large validation data, inference is performed in batches (2,000,000 rows per batch). Predictions are generated on each batch and stored into a local out-of-fold (OOF) array. The local OOF predictions are mapped back into the global OOF array using the original indices of the validation set. These predictions are saved for later evaluation.

4.4 Local Validation

4.4.1 Aggregating OOF Predictions

A global prediction array (`final_oof`) is initialized with a fixed size (90,062,550 rows). For each of the 5 folds, the validation indices and corresponding local OOF predictions (saved during model training) are loaded and placed into their corresponding positions in the global OOF array. This ensures that every instance in the dataset receives an OOF prediction.

4.4.2 Candidate Ranking and Submission Assembly

The global OOF predictions are merged with the session-item DataFrame (converted from the original Dask DataFrame) by assigning the prediction score to each row. The DataFrame is then sorted by session and prediction score (in descending order) to prioritize items with higher predicted relevance. For each session, items are ranked, and only the top 20 candidates are retained. This is accomplished by grouping by session and computing a cumulative count per group, then filtering out items beyond the 20th rank.

4.4.3 Evaluation Metric Computation

The submission is merged with a preprocessed validation labels dataset (loaded from a parquet file) based on `session` and `type`. Sessions without predictions receive an empty candidate list. For each session, the number of correct predictions ("hits") is determined by computing the intersection between the predicted candidate list and the ground truth. The ground truth count is clipped to a maximum of 20.

Recall per type is calculated by summing the hits across sessions and dividing by the total ground truth count. The final score is computed as a weighted sum of the recalls for different types, using weights (clicks: 0.10, carts: 0.30, orders: 0.60) according to the requirement.

4.5 Validation Results

Below are the validation results compared against the commonly referenced baseline approach:

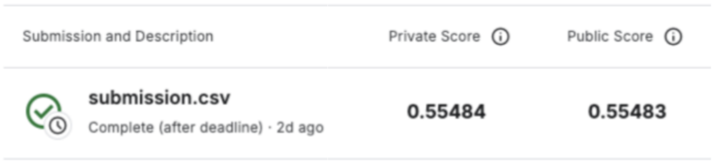
- **Per-Type Recall:**
  - Orders: 0.607055
  - Clicks: 0.498113
  - Carts: 0.38514
- **Weighted Final Score:**

$$0.607 \times 0.60 + 0.38 \times 0.30 + 0.498 \times 0.10 = 0.53 \tag{4}$$

Comparison to Baseline: A commonly cited baseline in the OTTO competition (as discussed by Radek Osmulski) achieves a score of approximately 0.432 by simply taking the user's last 20 actions as the prediction. The rationale for this baseline is that users tend to revisit recently interacted items (e.g., repeated clicks or purchases). In comparison, the approach outlined here—incorporating co-visitation signals, user-history weighting, and an XGBoost reranker—improves upon the baseline by nearly 0.10 in absolute score, reflecting a substantial gain in recall performance.

4.6 Inference and Final Score

For inference, we create a new candidate dataframe (using our technique to generate candidates before) but this time from Kaggle's test data. Then we make item features from all 4 weeks of Kaggle train plus 1 week of Kaggle test. And we make user features from Kaggle test. We merge the features to our candidates. Then we use our saved models to infer predictions. Lastly, we select 20 by sorting the predictions and choosing the top 20. The final leaderboard score is **0.55483**.



Submission and Description	Private Score	Public Score
submission.csv Complete (after deadline) · 2d ago	0.55484	0.55483

5 Future Work and Potential Improvements

Despite the effectiveness of the current system, there are several avenues for enhancing both the feature engineering and modeling aspects. However, due to time and computational constraints, the following advanced methods were not fully explored.

5.1 Advanced Feature Engineering

Embedding-Based Features: Instead of relying solely on manual feature engineering, we can learn low-dimensional representations (embeddings) of items or sessions. This can be achieved via Word2Vec-like training on item sequences. Learned embeddings can capture latent item-item relationships more effectively than manual co-occurrence metrics.

Temporal and Contextual Signals: Incorporating more sophisticated time-based features (e.g., session velocity, day-of-week patterns) or contextual cues (like device type or user geography) can add valuable signals. These features could be combined with session embeddings to reflect short-term user intent more accurately.

5.2 Deep Learning Models

Sequence Modeling Architectures: Recurrent Neural Networks (RNNs), GRUs, or LSTM-based models can explicitly model the sequential nature of user interactions within a session. This is especially useful when predicting the next item or purchase event.

6 Reference

- `https://www.kaggle.com/competitions/otto-recommender-system`
- `https://www.kaggle.com/competitions/otto-recommender-system/discussion/364991`
- `https://www.kaggle.com/competitions/otto-recommender-system/discussion/370210`
- `https://www.kaggle.com/competitions/otto-recommender-system/discussion/364721`

7 Source Code

For the execution code, please refer to the GitHub link.

8 Statement of Collaboration

This report is the result of a comprehensive investigation and discussion conducted by Aiden Wan and Cheng-Han Wu on the entire project. In terms of implementation, Aiden Wan was responsible for Candidates Generation, Model Training, and Local Validation. Meanwhile, Cheng-Han Wu was in charge of Exploratory Data Analysis (EDA), Feature Engineering, and Data Labeling.