Ali Mobile Recommendation Algorithm: A Comprehensive MySQL Implementation for Purchase Prediction

This report details a MySQL-based approach to the Ali Mobile Recommendation Algorithm challenge, focusing on predicting user purchases. It covers data understanding, database setup, preprocessing, feature engineering, prediction generation, and evaluation considerations, all within the MySQL environment. The methodology adheres to the project descriptions and dataset structures provided for the Tianchi competition.[1]

I. Project Overview and Data Understanding

A. Introduction to the Ali Mobile Recommendation Challenge

The Ali Mobile Recommendation Challenge tasks participants with developing algorithms to predict user behavior on Alibaba's mobile e-commerce platforms.[1] Specifically, the core objective for the period in question was to predict the purchase behavior of users on **December 19th**, given a dataset of user behavior records spanning from **November 18th to December 18th**.[2] This one-month historical data serves as the foundation for learning user preferences and item characteristics to forecast future transactions.

The competition is set within the M-Commerce (mobile commerce) landscape, which introduces unique data characteristics compared to traditional desktop e-commerce. Mobile platforms provide richer contextual information, such as user location (though often sparse or requiring careful handling) and distinct patterns in access times, which can influence recommendation strategies.[1] For Season 1 of the competition, participants were provided with a "small subset" of the overall data, comprising approximately 10,000 users' behavior data and information on one million products.[2] This dataset was downloadable, allowing for local processing and model development [2], making a comprehensive MySQL-based solution, as explored in this report, a feasible approach.

B. Detailed Data Schema and Definitions

The competition provides two primary data files in CSV format: tianchi_mobile_recommend_train_user.csv containing user behavioral data, and tianchi_mobile_recommend_train_item.csv containing item-specific information.[2] Understanding the structure and meaning of each field within these files is paramount for effective data processing and feature engineering.

1. tianchi_mobile_recommend_train_user.csv (User Behaviors)

This file logs user interactions with items over time. The schema is as follows 2:

- user_id: An identifier for the user.
- item_id: An identifier for the item.
- behavior_type: A numerical code representing the type of interaction:
  - 1: Browse (user viewed the item)
  - 2: Collect (user added the item to a wishlist or marked it as a favorite)
  - 3: Add to cart (user placed the item in their shopping cart)
  - 4: Purchase (user completed a transaction for the item) – This is the target behavior for prediction.
- user_geohash: A geohash string representing the user's geographical location at the time of the action. This field can be NULL.
- item_category: An identifier for the category to which the item belongs.
- time: A timestamp indicating when the action occurred. The format typically captures both date and hour (e.g., 'YYYY-MM-DD HH').

2. tianchi_mobile_recommend_train_item.csv (Item Information)
This file provides supplementary details about the items 2:

- item_id: An identifier for the item, corresponding to item_id in the user behavior data.
- item_geohash: A geohash string representing the item's geographical location (e.g., warehouse or seller location). This field can be NULL.
- item_category: An identifier for the category of the item, corresponding to item_category in the user behavior data.

An important observation is the presence of item_category in both tables. The item_category in tianchi_mobile_recommend_train_user.csv reflects the category associated with the item at the specific moment of user interaction. Conversely, the item_category in tianchi_mobile_recommend_train_item.csv likely represents the item's current or master category assignment. While items might theoretically change categories over time, for a one-month observation window, these are often assumed to be consistent. If discrepancies exist, the category recorded at the time of behavior in train_user_behaviors is generally more relevant for analyzing that specific event. The train_item_info table serves primarily as a canonical reference for item properties and a complete list of items. For the SQL-based approach in this report, the item_category from the user behavior table will be prioritized for event-specific analysis, while the item table will be used for item-centric features.

The following table summarizes the proposed MySQL schema for these datasets:

Table 1: Data Schema and Definitions

Table Name	Column Name	Proposed MySQL Data Type	Description
train_user_behaviors	user_id	INT UNSIGNED	User identity
train_user_behaviors	item_id	INT UNSIGNED	Item identity
train_user_behaviors	behavior_type	TINYINT UNSIGNED	Behavior: 1 (Browse), 2 (Collect), 3 (Add to cart), 4 (Purchase)
train_user_behaviors	user_geohash	VARCHAR(16)	User location (can be NULL)
train_user_behaviors	item_category	INT UNSIGNED	Category of the commodity (at time of interaction)
train_user_behaviors	time_str	VARCHAR(19)	Action time string (e.g., 'YYYY-MM-DD HH') - to be parsed
train_user_behaviors	event_time	DATETIME	Parsed action time
train_item_info	item_id	INT UNSIGNED	Item identity (Primary Key)
train_item_info	item_geohash	VARCHAR(16)	Item location (can be NULL)
train_item_info	item_category	INT UNSIGNED	Canonical category of the commodity

II. MySQL Database Setup and Data Ingestion

Setting up the database correctly and efficiently loading the data are foundational steps. This involves creating the database schema and then ingesting the voluminous CSV data.

A. Database and Table Creation

First, a dedicated database for the project should be created. Within this database, two primary tables will store the user behavior data and item information, respectively.

SQL

```
CREATE DATABASE IF NOT EXISTS ali_recsys CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE ali_recsys;

-- Table to store user behavior data
CREATE TABLE IF NOT EXISTS train_user_behaviors (
    user_id INT UNSIGNED NOT NULL,
    item_id INT UNSIGNED NOT NULL,
    behavior_type TINYINT UNSIGNED NOT NULL COMMENT '1:Browse, 2:Collect, 3:Add to cart, 4:Purchase',
    user_geohash VARCHAR(16) DEFAULT NULL,
    item_category INT UNSIGNED NOT NULL,
    time_str VARCHAR(19) NOT NULL COMMENT 'Original time string e.g., YYYY-MM-DD HH',
    event_date DATE DEFAULT NULL,
    event_hour TINYINT UNSIGNED DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

-- Table to store item information
CREATE TABLE IF NOT EXISTS train_item_info (
    item_id INT UNSIGNED NOT NULL,
    item_geohash VARCHAR(16) DEFAULT NULL,
    item_category INT UNSIGNED NOT NULL,
    PRIMARY KEY (item_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

The train_user_behaviors table includes time_str to hold the original time string, and event_date and event_hour columns which will be populated by parsing time_str. This separation facilitates easier date-based and hour-based operations.

B. Data Loading Strategies for CSV Files

MySQL's LOAD DATA INFILE command is generally the most efficient method for importing large CSV files directly into tables. It is significantly faster than row-by-row inserts, especially for datasets of the scale encountered in this competition (even the "small subset" can be millions of rows).[4]

The time column in tianchi_mobile_recommend_train_user.csv is a string combining date and hour. It's beneficial to parse this into separate DATE and TINYINT (for hour) columns during or immediately after loading for easier querying.

SQL

```sql
-- Load data into train_user_behaviors table
-- Ensure the CSV file path is correct and MySQL server has permissions to read it.
-- If using LOAD DATA LOCAL INFILE, ensure 'local_infile' is enabled on both client and server.
LOAD DATA LOCAL INFILE '/path/to/your/tianchi_mobile_recommend_train_user.csv'
INTO TABLE train_user_behaviors
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n' -- or '\r\n' if using Windows line endings
IGNORE 1 ROWS -- Skip header row
(user_id, item_id, behavior_type, user_geohash, item_category, time_str);

-- Populate event_date and event_hour from time_str
-- This step assumes time_str is in 'YYYY-MM-DD HH' format.
-- Adjust STR_TO_DATE format string if necessary.
UPDATE train_user_behaviors
SET event_date = STR_TO_DATE(SUBSTRING_INDEX(time_str, ' ', 1), '%Y-%m-%d'),
    event_hour = CAST(SUBSTRING_INDEX(time_str, ' ', -1) AS UNSIGNED);

-- Load data into train_item_info table
LOAD DATA LOCAL INFILE '/path/to/your/tianchi_mobile_recommend_train_item.csv'
INTO TABLE train_item_info
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n' -- or '\r\n'
IGNORE 1 ROWS -- Skip header row
(item_id, item_geohash, item_category);
```

C. Indexing for Performance

Given the multi-million row scale of train_user_behaviors, appropriate indexing is critical for query performance during preprocessing, feature engineering, and prediction. Without indexes, queries involving filtering, joining, and aggregation would necessitate full table scans, leading to prohibitively long execution times.

The time dimension, represented by event_date and event_hour, is central to this recommendation task. The prediction target is for a specific future date (December 19th), and training data spans a defined period (November 18th to December 18th).[2] Many preprocessing steps and feature engineering calculations will involve filtering or aggregating data based on event_date. For example, identifying recent user activities or calculating feature values over specific time windows (e.g., "last 7 days") directly relies on efficient date-based lookups.

Therefore, indexes involving event_date are paramount. Composite indexes that include event_date along with other frequently queried columns like user_id, item_id, and behavior_type can significantly accelerate common query patterns.

Initial indexes to consider:

SQL

```
-- Indexes for train_user_behaviors
ALTER TABLE train_user_behaviors ADD INDEX idx_user_date (user_id, event_date);
ALTER TABLE train_user_behaviors ADD INDEX idx_item_date (item_id, event_date);
ALTER TABLE train_user_behaviors ADD INDEX idx_date_behavior (event_date, behavior_type);
ALTER TABLE train_user_behaviors ADD INDEX idx_user_item_date (user_id, item_id, event_date);
ALTER TABLE train_user_behaviors ADD INDEX idx_category_date (item_category, event_date);

-- Indexes for train_item_info (Primary Key on item_id already exists)
ALTER TABLE train_item_info ADD INDEX idx_item_cat (item_category);
```

These indexes are chosen to support common access patterns:

- idx_user_date: Queries related to a specific user's activity over time.

- idx_item_date: Queries related to an item's activity over time.
- idx_date_behavior: Queries filtering by date and behavior type (e.g., all purchases on a given day).
- idx_user_item_date: Queries focusing on specific user-item interactions over time, crucial for user-item features.
- idx_category_date: Queries analyzing behavior within specific categories over time.
- idx_item_cat (on train_item_info): Queries involving item categories.

Further indexing strategies might emerge as specific feature engineering queries are developed.

III. Data Preprocessing and Cleaning in MySQL

After loading the data, several preprocessing steps are necessary to prepare it for feature engineering. This includes ensuring correct data types, applying competition-specific filtering rules, and handling missing values.

A. Date and Time Transformation

The event_date and event_hour columns were populated from the time_str during the data ingestion phase. This step ensures that temporal information is readily usable in a structured format for date-based calculations and range queries. The training data spans from November 18th to December 18th, and the prediction target is December 19th.[2] All date-related logic will use these boundaries.

B. Implementing Competition-Specific Sample Filtering

Some research associated with the Tianchi competition describes a sample filtering strategy to refine the dataset, focusing on interactions deemed more likely to be relevant for prediction.[2] This filtering aims to reduce computational complexity and potentially improve model performance by removing "invaluable samples." The rules are based on the time interval between an interaction and a "sample date" (which, in our prediction context, is December 19th, or the end of the observation period, December 18th, when generating features).

The specific filtering rules outlined are [2]:

- For **browse samples** (behavior_type = 1), only those with a date interval of **1 day** were considered.
- For **collection (behavior_type = 2) and add-to-cart (behavior_type = 3) samples**, only those with a date interval of **less than 7 days** were used.

These rules reflect the intuition that very old, non-committal interactions (like a browse from weeks ago) are less predictive of an imminent purchase than recent, higher-intent actions (like adding to cart yesterday). Purchase behaviors (behavior_type = 4) within the November 18th - December 18th window are not subjected to this specific recency filtering; historical purchases are strong signals and are retained for feature engineering.

For a pure SQL-based prediction approach, this "sample filtering" translates into creating a working set of recent, high-signal interactions. Instead of training an external ML model on a reduced dataset, these filters help define the pool of user-item activities from which features predictive of a December 19th purchase will be derived. This focuses analytical efforts on the most promising interactions.

The following table summarizes this logic:

Table 2: Sample Filtering Logic Summary

Behavior Type	behavior_type ID	Filtering Time Window (Relative to Prediction Day - Dec 19th)	Rationale
Browse	1	Interactions on Dec 18th (1 day prior)	Recall decreases dramatically after 1 day; high volume.
Collect	2	Interactions from Dec 12th to Dec 18th (< 7 days prior)	Seven-day cycle influence; recall very small beyond 7 days.
Add to Cart	3	Interactions from Dec 12th to Dec 18th (< 7 days prior)	Seven-day cycle influence; recall very small beyond 7 days.
Purchase	4	All historical purchases (Nov 18 - Dec 18) retained for features	Not explicitly filtered by [2]'s time-interval rules; crucial for target prediction.

A new table, filtered_user_behaviors, can be created to store these relevant

interactions.

```
SQL
```

```sql
CREATE TABLE IF NOT EXISTS filtered_user_behaviors AS
SELECT *
FROM train_user_behaviors
WHERE
    -- Keep all purchases within the observation window
    (behavior_type = 4 AND event_date BETWEEN '2014-11-18' AND '2014-12-18')
    OR
    -- Keep browse actions only from Dec 18th
    (behavior_type = 1 AND event_date = '2014-12-18')
    OR
    -- Keep collect and cart actions from Dec 12th to Dec 18th (within 7 days of Dec 19th)
    (behavior_type IN (2, 3) AND event_date BETWEEN '2014-12-12' AND '2014-12-18');

-- Add necessary indexes to the filtered table
ALTER TABLE filtered_user_behaviors ADD INDEX idx_fub_user_date (user_id, event_date);
ALTER TABLE filtered_user_behaviors ADD INDEX idx_fub_item_date (item_id, event_date);
ALTER TABLE filtered_user_behaviors ADD INDEX idx_fub_date_behavior (event_date, behavior_type);
ALTER TABLE filtered_user_behaviors ADD INDEX idx_fub_user_item_date (user_id, item_id, event_date);
```

This filtered_user_behaviors table, along with all historical purchases from train_user_behaviors (if not entirely included by the filter, e.g., purchases before Dec 12th), will form the basis for feature engineering. For simplicity, the above query includes all purchases from Nov 18 - Dec 18.

C. Handling Missing Data

The user_geohash in train_user_behaviors and item_geohash in train_item_info can contain NULL values.[2] For a pure SQL approach without sophisticated geospatial libraries or functions, utilizing geohash information effectively is challenging. If these features were to be used, NULLs could be treated as a distinct category (e.g.,

'UNKNOWN_GEOHASH'). However, given the potential sparsity and complexity, this report will primarily focus on non-geospatial features. Queries using these columns should be mindful of NULLs (e.g., using COALESCE(user_geohash, 'UNKNOWN') or filtering them out).

D. Data Integrity Checks (Optional but Recommended)

While competition datasets are typically pre-cleaned, in real-world scenarios, data integrity checks are vital. This could include:

- Checking for item_ids in train_user_behaviors that do not exist in train_item_info (orphaned records).
- Verifying that behavior_type values are within the expected range (1-4).
- Ensuring event_date values fall within the competition's specified timeframe.

For this project, it is assumed the provided data is largely consistent.

IV. Feature Engineering for Purchase Prediction using MySQL

Feature engineering is the process of creating informative variables from the raw data to improve the predictive power of a model. The goal is to extract signals from user behavior data up to and including December 18th that are indicative of a purchase on December 19th.

A. Defining the Prediction Target and Candidate Pairs

The ultimate target is to predict (user_id, item_id) pairs that will result in a purchase (behavior_type = 4) on December 19th. A crucial aspect of many recommendation approaches, including those discussed for this competition, is to focus on "interactive pairs".[2] An interactive pair is a (user, item) combination where the user has had at least one interaction (browse, collect, add-to-cart, or even a prior purchase) with the item before the prediction date. This strategy is adopted because predicting purchases for user-item pairs with no prior interaction history ("non-interactive pairs") is significantly more challenging. While non-interactive pairs might constitute a large portion of actual purchases, focusing on interactive pairs allows for leveraging existing user-item signals, effectively pruning the vast space of all possible user-item combinations to a more manageable and signal-rich subset.

Therefore, the candidate (user_id, item_id) pairs for feature engineering will be those that have at least one interaction recorded in the train_user_behaviors table (or, more specifically, the filtered_user_behaviors table plus all historical purchase interactions) during the November 18th - December 18th period.

A base table of unique user-item pairs that have interacted can be created:

SQL

```
CREATE TABLE IF NOT EXISTS candidate_user_item_pairs AS
SELECT DISTINCT user_id, item_id
FROM train_user_behaviors -- Using all interactions to define candidates
WHERE event_date BETWEEN '2014-11-18' AND '2014-12-18';

ALTER TABLE candidate_user_item_pairs ADD PRIMARY KEY (user_id, item_id);
```

Features will then be generated for these candidate pairs.

B. User-Based Features

These features describe the general behavior and preferences of a user, aggregated over the observation period (Nov 18 - Dec 18).

- u_total_actions: Total number of actions performed by the user.
- u_distinct_items_interacted: Number of unique items the user interacted with.
- u_purchase_count: Total number of purchases made by the user.
- u_purchase_ratio: Ratio of purchases to total actions.
- u_days_active: Number of distinct days the user was active.
- u_avg_actions_per_active_day: Average actions per active day.
- u_last_action_recency: Days since the user's last action (from Dec 18th).
- u_last_purchase_recency: Days since the user's last purchase (from Dec 18th).
- u_distinct_categories_interacted: Number of unique categories the user interacted with.

Example SQL to generate some user features (to be stored in a user features table or joined later):

SQL

```
CREATE TABLE IF NOT EXISTS user_features AS
SELECT
```

```
    user_id,
    COUNT(*) AS u_total_actions,
    COUNT(DISTINCT item_id) AS u_distinct_items_interacted,
    SUM(CASE WHEN behavior_type = 4 THEN 1 ELSE 0 END) AS u_purchase_count,
    SUM(CASE WHEN behavior_type = 4 THEN 1 ELSE 0 END) / COUNT(*) AS u_purchase_ratio,
    COUNT(DISTINCT event_date) AS u_days_active,
    DATEDIFF('2014-12-18', MAX(event_date)) AS u_last_action_recency,
    DATEDIFF('2014-12-18', MAX(CASE WHEN behavior_type = 4 THEN event_date ELSE NULL
END)) AS u_last_purchase_recency,
    COUNT(DISTINCT item_category) AS u_distinct_categories_interacted
FROM train_user_behaviors -- or filtered_user_behaviors for certain features
WHERE event_date BETWEEN '2014-11-18' AND '2014-12-18'
GROUP BY user_id;

ALTER TABLE user_features ADD PRIMARY KEY (user_id);
```

C. Item-Based Features

These features describe the characteristics and popularity of an item.

- i_total_interactions: Total number of interactions the item received.
- i_distinct_users_interacted: Number of unique users who interacted with the item.
- i_purchase_count: Total number of times the item was purchased.
- i_purchase_ratio: Ratio of purchases to total interactions for the item.
- i_conversion_rate: (Purchases / Browses) or (Purchases / (Browses + Carts + Collects)).
- i_last_interaction_recency: Days since the item's last interaction.
- i_last_purchase_recency: Days since the item's last purchase.

Example SQL for item features:

```
SQL


CREATE TABLE IF NOT EXISTS item_features AS
SELECT
    item_id,
    COUNT(*) AS i_total_interactions,
    COUNT(DISTINCT user_id) AS i_distinct_users_interacted,
```

```
    SUM(CASE WHEN behavior_type = 4 THEN 1 ELSE 0 END) AS i_purchase_count,
    SUM(CASE WHEN behavior_type = 4 THEN 1 ELSE 0 END) / COUNT(*) AS i_purchase_ratio,
    DATEDIFF('2014-12-18', MAX(event_date)) AS i_last_interaction_recency,
    DATEDIFF('2014-12-18', MAX(CASE WHEN behavior_type = 4 THEN event_date ELSE NULL
END)) AS i_last_purchase_recency
FROM train_user_behaviors -- or filtered_user_behaviors
WHERE event_date BETWEEN '2014-11-18' AND '2014-12-18'
GROUP BY item_id;

ALTER TABLE item_features ADD PRIMARY KEY (item_id);
```

D. User-Item Interaction Features

These are often the most powerful features, describing the specific history between a user and an item.

- ui_browse_count: Number of times the user browsed this item.
- ui_collect_count: Number of times the user collected this item.
- ui_cart_count: Number of times the user added this item to cart.
- ui_purchase_count_hist: Number of times the user purchased this item historically.
- ui_last_browse_recency: Days since user last browsed this item.
- ui_last_collect_recency: Days since user last collected this item.
- ui_last_cart_recency: Days since user last carted this item.
- ui_last_any_interaction_recency: Days since user last interacted (any type) with this item.
- ui_has_interacted_type_X: Boolean flags for each interaction type.

Example SQL for user-item features:

SQL

```
CREATE TABLE IF NOT EXISTS user_item_interaction_features AS
SELECT
    user_id,
    item_id,
    SUM(CASE WHEN behavior_type = 1 THEN 1 ELSE 0 END) AS ui_browse_count,
    SUM(CASE WHEN behavior_type = 2 THEN 1 ELSE 0 END) AS ui_collect_count,
```

```
    SUM(CASE WHEN behavior_type = 3 THEN 1 ELSE 0 END) AS ui_cart_count,
    SUM(CASE WHEN behavior_type = 4 THEN 1 ELSE 0 END) AS ui_purchase_count_hist,
    DATEDIFF('2014-12-18', MAX(CASE WHEN behavior_type = 1 THEN event_date ELSE NULL
END)) AS ui_last_browse_recency,
    DATEDIFF('2014-12-18', MAX(CASE WHEN behavior_type = 2 THEN event_date ELSE NULL
END)) AS ui_last_collect_recency,
    DATEDIFF('2014-12-18', MAX(CASE WHEN behavior_type = 3 THEN event_date ELSE NULL
END)) AS ui_last_cart_recency,
    DATEDIFF('2014-12-18', MAX(event_date)) AS ui_last_any_interaction_recency
FROM train_user_behaviors -- or filtered_user_behaviors for recency on specific recent actions
WHERE event_date BETWEEN '2014-11-18' AND '2014-12-18'
GROUP BY user_id, item_id;

ALTER TABLE user_item_interaction_features ADD PRIMARY KEY (user_id, item_id);
```

E. Time-Window Features

As suggested by research [2], features calculated over different recent time windows (e.g., last 1, 3, 7 days before the prediction day) can capture trends and recent intent.

- ui_actions_last_1_day: User-item interactions on Dec 18th.
- ui_actions_last_3_days: User-item interactions from Dec 16th-18th.
- ui_actions_last_7_days: User-item interactions from Dec 12th-18th. These can be counts of specific behaviors (e.g., ui_cart_count_last_3_days).

Example for ui_cart_count_last_7_days:

SQL

```
-- This can be added as a column to user_item_interaction_features
-- or calculated in a combined feature table.
-- For instance, when creating a final feature table:
-- (SELECT COUNT(*) FROM train_user_behaviors sub
--  WHERE sub.user_id = main.user_id AND sub.item_id = main.item_id
--  AND sub.behavior_type = 3 AND sub.event_date BETWEEN '2014-12-12' AND '2014-12-18') AS
ui_cart_count_7d
```

F. Category-Based Features

Features involving item categories can capture broader user interests and item

context.

- user_category_purchase_count: Number of times user purchased items from this item's category.
- user_category_browse_ratio: User's browse ratio within this item's category.
- item_rank_in_category_by_purchase: Rank of the item within its category based on purchase counts.

Example for item_rank_in_category_by_purchase:

```
SQL
```

```
-- This requires joining with train_item_info to get the canonical category
-- and then using window functions.
-- (SELECT RANK() OVER (PARTITION BY ti.item_category ORDER BY ifeat.i_purchase_count DESC)
--  FROM train_item_info ti JOIN item_features ifeat ON ti.item_id = ifeat.item_id
--  WHERE ti.item_id = main.item_id) AS item_cat_purchase_rank
```

G. Feature Table Creation

A final consolidated feature table, user_item_features_dec18, should be created by joining candidate_user_item_pairs with the various feature tables (user, item, user-item interaction) and including calculated time-window and category-based features.

```
SQL
```

```
CREATE TABLE IF NOT EXISTS user_item_features_dec18 AS
SELECT
    cp.user_id,
    cp.item_id,
    -- User features
    uf.u_total_actions, uf.u_distinct_items_interacted, uf.u_purchase_count AS
u_total_purchase_count,
    uf.u_purchase_ratio, uf.u_days_active, uf.u_last_action_recency AS
u_g_last_action_recency,
    uf.u_last_purchase_recency AS u_g_last_purchase_recency,
```

```
uf.u_distinct_categories_interacted,
    -- Item features (joined with train_item_info for canonical category)
    itf.i_total_interactions, itf.i_distinct_users_interacted, itf.i_purchase_count AS
i_total_purchase_count,
    itf.i_purchase_ratio, itf.i_last_interaction_recency AS i_g_last_interaction_recency,
    itf.i_last_purchase_recency AS i_g_last_purchase_recency, ti.item_category,
    -- User-item interaction features
    uif.ui_browse_count, uif.ui_collect_count, uif.ui_cart_count,
uif.ui_purchase_count_hist,
    uif.ui_last_browse_recency, uif.ui_last_collect_recency, uif.ui_last_cart_recency,
    uif.ui_last_any_interaction_recency,
    -- Time-window features (examples)
    (SELECT COUNT(*) FROM train_user_behaviors tb
     WHERE tb.user_id = cp.user_id AND tb.item_id = cp.item_id AND tb.behavior_type = 3
     AND tb.event_date BETWEEN '2014-12-12' AND '2014-12-18') AS ui_cart_count_7d,
    (SELECT COUNT(*) FROM train_user_behaviors tb
     WHERE tb.user_id = cp.user_id AND tb.item_id = cp.item_id AND tb.behavior_type = 1
     AND tb.event_date = '2014-12-18') AS ui_browse_count_1d,
    -- User-Category interaction features (example: user's purchase count in this item's category)
    (SELECT COUNT(*) FROM train_user_behaviors ucb
     WHERE ucb.user_id = cp.user_id AND ucb.item_category = ti.item_category AND
ucb.behavior_type = 4
     AND ucb.event_date BETWEEN '2014-11-18' AND '2014-12-18') AS
user_cat_purchase_count
FROM
    candidate_user_item_pairs cp
LEFT JOIN
    user_features uf ON cp.user_id = uf.user_id
LEFT JOIN
    item_features itf ON cp.item_id = itf.item_id
LEFT JOIN
    train_item_info ti ON cp.item_id = ti.item_id -- For canonical item_category
LEFT JOIN
    user_item_interaction_features uif ON cp.user_id = uif.user_id AND cp.item_id =
uif.item_id;

ALTER TABLE user_item_features_dec18 ADD PRIMARY KEY (user_id, item_id);
-- Add further indexes as needed for scoring performance
ALTER TABLE user_item_features_dec18 ADD INDEX idx_cat_ipur_cnt (item_category,
```

i_total_purchase_count);

Missing values in this feature table (e.g., if a user has no purchase history, u_last_purchase_recency would be NULL) should be handled, typically by imputing with a sensible default (e.g., a large number for recency, zero for counts) before scoring.

The following table provides an overview of some key engineered features that would populate user_item_features_dec18:

Table 3: Overview of Key Engineered Features

Feature Name	Description/Purpose	Key SQL Logic/Components Used (Conceptual)	Feature Type
u_total_purchase_count	Total historical purchases by the user.	SUM(CASE WHEN behavior_type=4 THEN 1 ELSE 0 END) GROUP BY user_id	User
i_total_purchase_count	Total times item was purchased historically.	SUM(CASE WHEN behavior_type=4 THEN 1 ELSE 0 END) GROUP BY item_id	Item
ui_cart_count_7d	Number of times user added this item to cart in last 7 days (before Dec 19).	COUNT(CASE WHEN behavior_type=3 AND event_date >= '2014-12-12' THEN 1 END) GROUP BY user_id, item_id	User-Item, Time
ui_last_any_interaction_recency	Days since user last interacted (any type) with this item (as of Dec 18).	DATEDIFF('2014-12-18', MAX(event_date)) GROUP BY user_id, item_id	User-Item, Time
user_cat_purchase_count	User's total purchases in the item's specific category.	User purchases in category / User interactions in category	User-Category

ui_purchase_count_hist	Number of times user purchased this specific item historically.	SUM(CASE WHEN behavior_type=4 THEN 1 ELSE 0 END) GROUP BY user_id, item_id	User-Item
i_cat_purchase_rank (Conceptual)	Rank of item's purchases within its category.	RANK() OVER (PARTITION BY item_category ORDER BY i_total_purchase_count DESC)	Item-Category

V. Generating Purchase Predictions for December 19th

With the user_item_features_dec18 table populated, the next step is to use these features to score each candidate (user_id, item_id) pair for its likelihood of being purchased on December 19th. For a pure SQL approach, this typically involves a heuristic scoring model.

A. Scoring User-Item Pairs

While advanced solutions for this competition often employ machine learning models like Logistic Regression or Gradient Boosting Decision Trees [2], a pure SQL solution relies on a simpler, rule-based or weighted-sum scoring mechanism. The features engineered in the previous step serve as inputs to this scoring logic.

A weighted sum approach assigns weights to different features based on their presumed importance:
$$Score = w1 \times feature1 + w2 \times feature2 + \cdots + wn \times featuren$$
For example, higher weights might be given to:

- ui_purchase_count_hist (user previously bought this item)
- ui_cart_count_7d (user recently carted this item)
- Low values for ui_last_cart_recency (very recent cart addition)
- i_total_purchase_count (item is generally popular)

Handling NULLs in features is important here. COALESCE(feature_column, default_value) can be used. Recency features (lower is better) might need to be transformed (e.g., 1/(1+recency)) or handled carefully in the weighting.

Example scoring logic within a SQL query:

```SQL
-- This query calculates a score and ranks pairs.
-- The actual weights (w1, w2, etc.) need to be determined heuristically or via offline tuning if a validation set is used.
SELECT
    user_id,
    item_id,
    (
        (COALESCE(ui_purchase_count_hist, 0) * 50.0) +
        (COALESCE(ui_cart_count_7d, 0) * 20.0) +
        (CASE WHEN COALESCE(ui_last_cart_recency, 999) < 3 THEN 15.0 ELSE 0 END) + -- Bonus for very recent cart
        (COALESCE(ui_collect_count, 0) * 10.0) +
        (COALESCE(i_total_purchase_count, 0) * 0.1) + -- Item popularity
        (COALESCE(u_total_purchase_count, 0) * 0.05) - -- User general purchase tendency
        (COALESCE(ui_last_any_interaction_recency, 999) * 0.5) -- Penalty for old interactions
        -- Add more weighted features as needed
    ) AS prediction_score
FROM
    user_item_features_dec18
ORDER BY
    prediction_score DESC;
```

This scoring is heuristic. More sophisticated rules could be implemented using CASE statements to create a simplified decision-tree-like logic.

B. Selecting Top-N Predictions

The evaluation for the competition is based on the F1-score, which considers the precision and recall of a submitted *set* of predicted purchases against a reference set of actual purchases.[2] This means the final output should be a list of (user_id, item_id) pairs, not their scores. The scoring mechanism is an intermediate step to rank pairs and decide which ones to include in the final prediction set.

A threshold can be applied to prediction_score, or a fixed number (Top-N) of highest-scoring pairs can be selected. The choice of N (or the threshold) directly impacts the size of the "prediction set" and thus affects precision and recall.

Competition rules sometimes give guidance on the expected volume of predictions, such as "Part 1 answers shall include the purchase data of 50% of all users who have purchase behavior on the very day of observation".[2] This suggests that the evaluation might be sensitive to the number of predictions submitted relative to the actual number of purchases. Without knowing the actual number of purchases on Dec 19th beforehand, selecting N requires some estimation or is set based on capacity/rules.

For instance, to select the top 100,000 predictions:

SQL

```sql
CREATE TABLE IF NOT EXISTS final_predictions AS
SELECT user_id, item_id --, prediction_score -- Score can be kept for analysis but not for submission
FROM (
    SELECT
        user_id,
        item_id,
        (
            (COALESCE(ui_purchase_count_hist, 0) * 50.0) +
            (COALESCE(ui_cart_count_7d, 0) * 20.0) +
            (CASE WHEN COALESCE(ui_last_cart_recency, 999) < 3 THEN 15.0 ELSE 0 END) +
            (COALESCE(ui_collect_count, 0) * 10.0) +
            (COALESCE(i_total_purchase_count, 0) * 0.1) +
            (COALESCE(u_total_purchase_count, 0) * 0.05) -
            (COALESCE(ui_last_any_interaction_recency, 999) * 0.5)
        ) AS prediction_score,
        ROW_NUMBER() OVER (ORDER BY
            (
                (COALESCE(ui_purchase_count_hist, 0) * 50.0) +
                (COALESCE(ui_cart_count_7d, 0) * 20.0) +
                (CASE WHEN COALESCE(ui_last_cart_recency, 999) < 3 THEN 15.0 ELSE 0 END) +
                (COALESCE(ui_collect_count, 0) * 10.0) +
                (COALESCE(i_total_purchase_count, 0) * 0.1) +
                (COALESCE(u_total_purchase_count, 0) * 0.05) -
                (COALESCE(ui_last_any_interaction_recency, 999) * 0.5)
            ) DESC
```

```
    ) as rn -- Rank based on score
  FROM
    user_item_features_dec18
  -- Optional: Add a WHERE clause here to filter out pairs with very low scores
  -- WHERE ( (COALESCE(ui_purchase_count_hist, 0) * 50.0) +... ) > some_threshold
) ranked_predictions
WHERE rn <= 100000; -- Example: Select Top 100,000 predictions
```

VI. Formatting Output and Conceptual Evaluation

The final step involves generating the prediction file in the required format and understanding how it would be evaluated.

A. Generating the Final Prediction File

The competition typically requires a submission file containing only the user_id and item_id of the predicted purchases. The final_predictions table created above already contains this. The output should be a distinct list of these pairs.

Table 4: Prediction Output Format

Column Name	MySQL Data Type	Description
user_id	INT UNSIGNED	The ID of the user predicted to make a purchase
item_id	INT UNSIGNED	The ID of the item predicted to be purchased

A SQL query to generate the content for the submission file would be:

```
SQL
```

```
SELECT DISTINCT user_id, item_id FROM final_predictions;
```

This result set would then be exported to a CSV file, typically named tianchi_mobile_recommendation_predict.csv or similar, as specified by competition

rules.

B. Conceptual Evaluation using F1-Score

The primary evaluation metric for this competition is the F1-score, which is the harmonic mean of precision and recall.[2] These metrics are defined as:

- Precision=|prediction set||prediction set∩reference set| Precision measures the proportion of predicted purchases that were actual purchases.
- Recall=|reference set||prediction set∩reference set| Recall measures the proportion of actual purchases that were correctly predicted.
- F1−score=Precision+Recall2×Precision×Recall The F1-score balances precision and recall.

Here, the "prediction set" is the list of (user_id, item_id) pairs submitted by the participant, and the "reference set" is the ground truth list of actual purchases that occurred on December 19th. General discussions of these metrics in recommender systems can also be found.[6]

If a ground truth table ground_truth_dec19_purchases (user_id, item_id) were available locally for testing, the F1-score could be calculated using SQL:

SQL

```
-- Assume 'final_predictions (user_id, item_id)' and
-- 'ground_truth_dec19_purchases (user_id, item_id)' tables exist.

-- Number of true positives (intersection)
SET @true_positives = (
    SELECT COUNT(*)
    FROM final_predictions fp
    JOIN ground_truth_dec19_purchases gt
    ON fp.user_id = gt.user_id AND fp.item_id = gt.item_id
);

-- Number of predicted positives
SET @predicted_positives = (SELECT COUNT(*) FROM final_predictions);

-- Number of actual positives (reference set size)
```

```
SET @actual_positives = (SELECT COUNT(*) FROM ground_truth_dec19_purchases);

-- Calculate Precision and Recall (handle division by zero)
SET @precision = IF(@predicted_positives > 0, @true_positives / @predicted_positives, 0);
SET @recall = IF(@actual_positives > 0, @true_positives / @actual_positives, 0);

-- Calculate F1-Score (handle division by zero)
SET @f1_score = IF((@precision + @recall) > 0, (2 * @precision * @recall) / (@precision + @recall), 0);

SELECT @true_positives, @predicted_positives, @actual_positives, @precision, @recall, @f1_score;
```

VII. Complete Runnable MySQL Script and Execution Notes

This section would ideally consolidate all SQL DDL and DML statements into a single, runnable script or a sequence of scripts.

A. Consolidated MySQL Script

A full script would include:

1. CREATE DATABASE
2. USE database
3. CREATE TABLE train_user_behaviors
4. CREATE TABLE train_item_info
5. LOAD DATA INFILE for train_user_behaviors
6. UPDATE train_user_behaviors to parse time_str into event_date, event_hour
7. LOAD DATA INFILE for train_item_info
8. ALTER TABLE statements to add indexes to raw data tables.
9. CREATE TABLE filtered_user_behaviors AS SELECT... (Applying sample filtering)
10. ALTER TABLE statements to add indexes to filtered_user_behaviors.
11. CREATE TABLE candidate_user_item_pairs AS SELECT...
12. ALTER TABLE to add primary key to candidate_user_item_pairs.
13. CREATE TABLE user_features AS SELECT...
14. ALTER TABLE to add primary key to user_features.
15. CREATE TABLE item_features AS SELECT...
16. ALTER TABLE to add primary key to item_features.
17. CREATE TABLE user_item_interaction_features AS SELECT...
18. ALTER TABLE to add primary key to user_item_interaction_features.
19. CREATE TABLE user_item_features_dec18 AS SELECT... (Joining all features)

20. ALTER TABLE to add primary key and indexes to user_item_features_dec18.
21. CREATE TABLE final_predictions AS SELECT... (Scoring and Top-N selection)
22. SELECT user_id, item_id FROM final_predictions (For output generation)

Each step should be commented to explain its purpose.

B. Execution Order and Dependencies

The scripts must be run in the order listed above, as later steps depend on tables and data created by earlier ones. For example, feature engineering tables cannot be built before raw data is loaded and preprocessed.

C. Dataset Size Considerations and MySQL Configuration

The approach described is intended for the "small dataset" provided in Season 1 of the competition.[2] Even so, train_user_behaviors can contain tens of millions of rows.

- **MySQL Configuration:** Ensure MySQL server settings like innodb_buffer_pool_size are adequately configured for the available system memory to optimize performance. tmp_table_size and max_heap_table_size might also be relevant if complex queries generate large temporary tables in memory.
- **Disk Space:** Sufficient disk space is required for the database and any temporary tables MySQL might create during query execution.
- **Execution Time:** Feature engineering steps, especially those involving joins across large tables and subqueries for time-window features, can be time-consuming. Execution times will vary significantly based on hardware (CPU, RAM, disk speed).

VIII. Conclusion and Potential Next Steps

A. Summary of the SQL-Based Recommendation Approach

This report has outlined a comprehensive methodology for tackling the Ali Mobile Recommendation challenge using solely MySQL. The approach encompasses:

1. **Data Ingestion and Structuring:** Loading raw CSV data into a relational schema with appropriate data types and indexes.
2. **Preprocessing:** Applying competition-specific filtering rules to focus on recent and high-intent user behaviors.
3. **Feature Engineering:** Systematically creating a rich set of features capturing user preferences, item characteristics, user-item interactions, time-window effects, and category influences.
4. **Heuristic Prediction:** Employing a weighted scoring mechanism within SQL to

rank candidate user-item pairs and generate a final set of purchase predictions for the target date.
5. **Output Generation:** Formatting predictions according to typical competition requirements.

This pure SQL approach provides a strong baseline and demonstrates how far one can go with relational database tools for a complex recommendation task.

B. Limitations of a Pure SQL Approach

While powerful for data manipulation and aggregation, a pure SQL approach has inherent limitations compared to solutions incorporating dedicated machine learning libraries:

- **Model Complexity:** SQL is not designed for implementing complex iterative algorithms like gradient boosting, neural networks, or even logistic regression directly. The heuristic scoring model is a simplification.
- **Feature Interactions:** Discovering and modeling complex, non-linear interactions between features is challenging in SQL. ML models often excel at this.
- **Hyperparameter Tuning:** Optimizing weights in the heuristic scoring model or thresholds for rules is a manual or trial-and-error process in SQL, unlike the systematic hyperparameter tuning available in ML frameworks.
- **Scalability for Advanced Models:** While SQL scales well for data storage and retrieval, training very large ML models is typically done in specialized distributed computing environments.

C. Suggestions for Further Enhancements

The SQL-based work performed here lays a solid foundation for more advanced techniques.

- **Advanced Feature Engineering:**
  - More sophisticated sequence analysis (e.g., time decay on interaction recency, specific sequences like browse -> collect -> cart -> purchase within defined time lags).
  - If user_geohash data quality and density permit, features based on user-item proximity or user location preferences could be explored (though this often requires specialized geospatial functions).
  - Incorporate item embedding similarities if pre-computed (e.g., from content analysis or collaborative filtering methods outside of this SQL scope).
- **Transition to Machine Learning Models:** The most significant enhancement would be to export the user_item_features_dec18 table (or a similar feature set) to

a CSV file. This file can then serve as the input for training sophisticated machine learning models using Python (with libraries like scikit-learn, XGBoost, LightGBM, TensorFlow/Keras) or other ML platforms. This aligns with the strategies often employed in such competitions, where SQL is used for data preprocessing and feature extraction, and ML tools are used for model training and prediction.[1] The detailed feature engineering performed in SQL provides a rich input for these models, potentially leading to higher predictive accuracy. The work detailed in this report, therefore, serves as a critical and valuable initial phase in a more comprehensive recommendation system pipeline.

- **Validation Strategy:** To tune the heuristic scoring model's weights or to evaluate any model, a robust validation strategy is needed. This would involve splitting the Nov 18 - Dec 18 data, for example, using Nov 18 - Dec 11 for feature generation and Dec 12 for validation (predicting purchases on Dec 12). This allows for offline evaluation and tuning before generating final predictions for Dec 19th.
- **Ensemble Methods:** If multiple scoring models or prediction strategies are developed, their outputs could be combined (ensembled) to potentially improve robustness and accuracy.

By systematically building upon the data processing and feature engineering capabilities of MySQL, and then potentially leveraging external machine learning tools, participants can construct powerful and effective recommendation systems for challenges like the Ali Mobile Recommendation Algorithm.

引用的著作

1. Ali Mobile Recommendation Algorithm_ALGORITHEM_Tianchi competition-Alibabacloud Tianchi introduction, 访问时间为 五月 14, 2025, https://tianchi.aliyun.com/competition/entrance/1/introduction?lang=en-us/1000
2. The Hierarchical Model to Ali Mobile Recommendation Competition - OpenReview, 访问时间为 五月 14, 2025, https://openreview.net/attachment?id=yshN1FEc6S&name=pdf
3. GitHub - hzhaoaf/tianchi_contest: The code for tianchi recommenda ..., 访问时间为 五月 14, 2025, https://github.com/hzhaoaf/tianchi_contest
4. pandas 运用-读取量级数据存入mysql中- 赏金猎人小熊- 博客园, 访问时间为 五月 14, 2025, https://www.cnblogs.com/wengzx/p/18846675
5. A mobile recommendation system based on logistic regression and Gradient Boosting Decision Trees - ResearchGate, 访问时间为 五月 14, 2025, https://www.researchgate.net/publication/309778358_A_mobile_recommendation_system_based_on_logistic_regression_and_Gradient_Boosting_Decision_Trees
6. Recommendation Systems: Algorithms, Challenges, Metrics, and Business Opportunities, 访问时间为 五月 14, 2025, https://www.mdpi.com/2076-3417/10/21/7748