Manual

Contents

HybridSVD Recommender System: User Manual

1. Project Overview

This project implements **HybridSVD**, a recommendation algorithm that enhances standard Matrix Factorization (PureSVD) by integrating side information (auxiliary data).

The code currently supports two datasets:

- **DC Core 20:** A Google Local Reviews dataset of restaurants in Washington D.C. (Side info: *Categories, Description, Geolocation*).
- **MovieLens 100k:** A standard movie recommendation benchmark (Side info: *Genres*)

2. Environment Setup

The project relies on Python 3.9+ and the following libraries:

Here are the concise descriptions for your project's dependencies:

- **NumPy:** Handles dense arrays and vectorized math operations for scoring recommendations.
- **Pandas:** Loads raw data (JSON/CSV) and formats results for analysis.
- **SciPy:** Manages sparse matrices and computes the core Truncated SVD (svds).
- **Matplotlib:** Generates static plots for data distribution (Long Tail) and model metrics.
- **Folium:** Creates interactive maps to visualize restaurant locations in Washington D.C.
- **Optuna:** Automates Bayesian optimization to find the best model hyperparameters.
- **Sksparse (cholmod):** Performs efficient Cholesky decomposition on sparse matrices for the HybridSVD equation.
- Streamlit

3. Project Structure

File	Role	Description
Main.ipynb	**Controller**	**Start here.** This notebook runs the entire pipeline: loading data, training, optimizing, and visualizing.
HybridSVD.py	**Model**	Contains the core HybridSVD() algorithm and the scoring function Recommend_topN().
Bayesian_...py	**Optimizer**	Uses **Optuna** to find the best hyperparameters (k, alpha, d) instead of slow grid search.
Data.py	**Data Loader**	Handles loading raw JSON/CSV files, creating sparse matrices, and popularity scaling.
Benchmark.py	**Metrics**	Calculates **HitRate**, **NDCG**, and **Coverage**.
Other_Baselines.py	**Comparison**	Runs standard baselines (Random, MostPopular, PureSVD) to compare against HybridSVD.
Visualization.py	**Plotting**	Generates the bar charts and exports the final LaTeX results table.

4. How to Run an Experiment

Open **Main.ipynb** and follow these steps. You do not need to touch the .py files unless you are changing the core logic.

Step 1: Configure the Experiment (Cell 2)

Set the variables to choose which data and features to test:

```
# Select dataset and similarity matrix
Dataset = "ml-100k" # Either DC Core 20 or ml-100k
Similarity = "Genre"  # item-item similarity type
# Options for Washington DC Core 20: "Category", "Description", "Geolocation"
# Options for Movielens 100k: "Genre"

# Hyperparameters tuning
n_trials=150 # number of trials for bayesian hyperparameter optimization
seed=15 # seed for bayesian optimization

# Generating top-N recommendations
N = 10

# Output directory
output_dir=f"outputs/{Dataset} {Similarity}"
[11]
```

Step 2: Run the Pipeline

Run the cells sequentially. The notebook will:

1. **Load Data:** Checks dimensions and sparsity.
2. **Analyze:** Plots the "Long Tail" distribution graph.
3. **Optimize:** Runs Optuna. **(This step takes the longest and can be skipped)**.
4. **Baselines:** Calculates scores for Random, MostPopular, and PureSVD.
5. **Visualize:** Produces comparison charts and saves them to the outputs/ folder.

Step 3: View Results

Check the “**outputs**” folder. You will find:

- **.csv / .tex:** The final metrics table.
- **.jpg:** Plots for NDCG, HitRate, and Coverage.
- **.pkl:** Saved hyperparameters (so you don't have to re-run optimization next time).

5. Input Data Format (For New Datasets)

If you want to use a new dataset, you must provide pre-computed matrices in the Datasets/ folder.

The code expects **SciPy Sparse Matrices (.npz)**:

1. **Interaction Matrices (R):**
    - R train: Training (for hyperparameter tuning) .
    - R val: Validation set (for hyperparameter tuning).
    - R train1: Final retraining set.
    - R test: Final test set.
2. **Similarity Matrix (Z):**
    - An Item-Item similarity matrix where $\mathbf{Z_{ij}}$ represents how similar item **i** is to item **j**.

Crucial: These matrices must be sparse.

6. Troubleshooting Common Errors

1. Optuna Memory Error

- **Cause:** The dataset or similarity matrix is too large for RAM.
- **Fix:** Ensure your Z matrix in Data.py is a scipy.sparse matrix, NOT a dense numpy array.

2. "Dimension Mismatch"

- **Cause:** The Z matrix (items x items) does not match the shape of the R matrix (users x items).
- **Fix:** Ensure the number of columns in R exactly matches the number of rows/columns in Z.

WebServer instructions

This is a large file because it contains the ~10,000 photos we used in the app. You can also download the file without the photos.

The Following explains how to use the WebServer application for this project.

Environment Setup

Python 3.13.10 was used here. This project will run on Python 3.x.
The following libraries are used.

- Streamlit: Constructs the interactive web application, manages the user interface, and handles the front-end logic.
- Folium: Generates interactive Leaflet maps to visualize geospatial data and locations.
- streamlit-folium: Enables the rendering of Folium maps directly within the Streamlit application.
- Geopy: Manages geocoding services (via Nominatim) to convert addresses to coordinates and calculates geodesic distances between locations.
- Standard Python libraries (json, os, random): Handle file system navigation, JSON data parsing, and random number generation.
- NumPy, SciPy, Pandas

Inputs

- The WebServer takes the training data from the model in the form of **vt_tilde.npy** and **Ls.npz**, this contains the training data from the model.
- To be able to reconstruct which index refers to which item in the dataset there are two map files also required. These are **restaurant_item_map_rest_20.json** and **restaurant_user_map_rest_20.json.**
    - These map from the index in the training data (0-x) to the corresponding gmap/userID of these locations or users
- The **meta-District_of_Columbia.json** and **item_metadata.csv** correspond to the metadata from the original dataset. **Item_metadata.csv** is not used.
- **Categorized_restaurants.csv** refers to the restaurant categories used in the WebServer's cold start logic. This creates a list of restaurants that fit into these bigger categories instead of the smaller ones so you can choose just one of 12 categories. This is also how the persona_vectors_lift determines which restaurants belong to a certain category.
- **Persona_vectors_lift.npy** refers to the latent vector representation of the preferences of top users of a certain category.
- **R_retrain_rest_20** is the dataset that does not have the most recent interaction. Ls and vt_tilde come from this.
- **Resolved_restaurants** maps from gmap ids to photos of restaurants.

- Blocked restaurants and categories exist because these are items that should not be in the dataset or are confusing.
- **Photos** this folder contains the photos that we downloaded using the google API. It contains the top 3 photos for each location.

How to use:

- If you have a different dataset or training you just add the new ls.npz and vt_tilde.npy. The item_map and user_map must be updated to to make sure the index corresponds to the actual restaurants.
- Then run lift_vectors_debias.py
    - This outputs the required vectors
- To run the webserver go into the python terminal and write **streamlit run app31.py**

Google Image Retriever

This script (gmapID_retriever_GoogleAPI.py) is designed to enrich your dataset by fetching official Google Maps data for a specific list of restaurants. It performs the following actions:

1. **Filters** your raw dataset to only include relevant restaurants.
2. **Identifies** the unique Google Place ID for each restaurant.
3. **Fetches Details:** GPS coordinates, official address, and editorial summaries.
4. **Downloads Photos:** Saves up to 3 high-quality images per restaurant.
5. **Budget Protection:** Monitors spending in real-time to prevent exceeding Google Cloud billing limits.

2. Google Cloud API Key

You must have a valid Google Cloud API Key with the **Places API (New or Legacy)** enabled.

- **Cost Warning:** Google Places API is not free. This script uses a budget limit, but you must have a billing account attached to your API key. With a student account you can get up to ~$200 of credits.

4. Input Files Structure

Ensure these files are in the same directory as the script (or update the paths in the config):

1. **item_metadata.csv**: Must contain at least the columns gmap_id, name, and address.
2. **restaurant_item_map_rest_20.json**: A dictionary used as a filter. The script extracts values from this JSON to decide which gmap_ids are relevant.

5. How to Run

Run the script from your terminal:

Bash

```
python gmapID_retriever_GoogleAPI.py
```

6. Outputs

A. JSON Data (resolved_restaurants.json)

A structured file containing the enriched data. Example entry:

JSON

"gmap_id_123": {

"place_id": "ChIJ...",

"name": "Le Diplomate",

"address": "1601 14th St NW, Washington, DC...",

"description": "Bustling French brasserie...",

"location": { "lat": 38.91, "lng": -77.03 },

"photos": ["photos/ChIJ..._photo_0.jpg"]

}

B. Photo Directory (photos/)

Contains JPG images renamed by Place ID to avoid conflicts.

- [PlaceID]_photo_0.jpg
- [PlaceID]_photo_1.jpg