Menu Engineering Supercharged

MDS Capstone Final Report

Hankun Xiao, Yasmin Hassan, Jessie Zhang, Zhiwei Zhang

Table of contents

1 Executive Summary

This project supports our partner, Heymate, in delivering data-driven menu insights to their restaurant clients. Leveraging over 3 million popularity records and the power of large language models, we developed a structured data cleaning pipeline and generated a weighted scoring mechanisma to identify top-performing dishes. The final product includes a scalable recommendation system that suggests the most popular menu items based on restaurant type, enabling merchants to optimize their offerings with minimal technical effort.

2 Introduction

Our capstone partner, Heymate, provides an all-in-one business management system, with the majority of its clients being restaurants. However, these restaurant owners constantly face the challenge of designing menus that align with customer preferences and evolving market trends due to the lack of access to broader market data. Our project aims to bridge that gap by transforming publicly available menu data into structured insights that can support data-informed menu design, enabling the partner to offer greater value to their clients through market-driven recommendations.

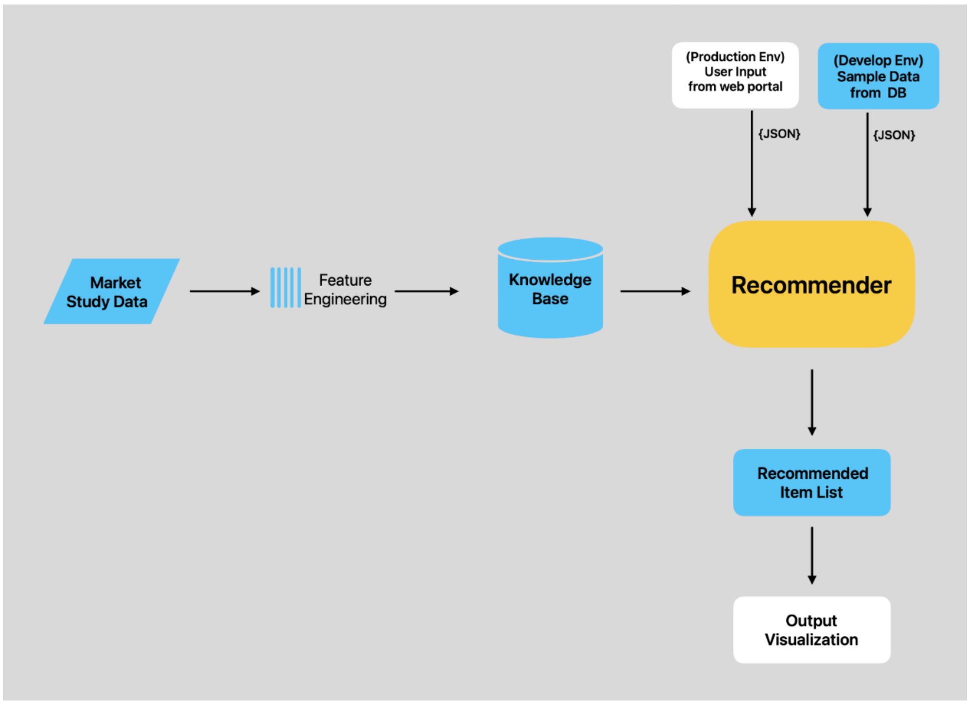
Our initial resource was internal database provided by Heymate, which contained thousands of raw menu records from their clients. However, the internal data alone was insufficient to support robust recommendation modeling due to limited scale of menu and trending information. To address this, we explored various open-source datasets, and found a menu dataset containing millions of menu records along with associated popularity metrics on Uber Eats (Sakib (2023)), making it a great data source in terms of scale, quality, and coverage.

However, messy data, large-scale datasets, and customized menu item names posed significant challenges to the construction of our recommendation system. Therefore, our scientific objective was to design an end-to-end pipeline that leverages diverse data science techniques to build a robust and scalable recommendation system, powered by a popularity-based knowledge base and capable of handling multi-type inputs.

3 Data Pipeline Framework

Based on the project objective, we developed the following modular data pipeline framework: (Figure 1)

Figure 1: Data Pipeline Framework - The core recommender supported by two pipelines: one for modeling from market data, and another for generating recommendations based on input



3.1 Model Input Pipeline (Horizontal Workflow)

1. Data Ingestion: After researching across multiple open-source datasets, we found a Uber Eats dataset from Kaggle covers a wide range of restaurant types, menu items, and their corresponding descriptions.
2. Feature Engineering: As the raw dataset was quite messy, it needed to undergo a series of preprocessing and standardization process so that we can use them for further popularity score calculation.
3. Knowledge Base Construction: Under suggestion and domain knowledge from the project partner,we buily a weighted scoring knowledge base on various popularity metrics. This serves as the foundation for capturing market trends and powering recommendations.

3.2 Testing and Production Pipeline (Vertical Workflow)

1. Recommender Input: The recommender system can process JSON inputs from two sources, including client restaurant type from Heymate internal database, and user input of restaurant types via the web portal for real-time recommendations.
2. Recommendation Module: Based on the specified restaurant type, the engine searches the knowledge base, ranks menu items using scoring logic, and generates a tailored Recommended Item List.

3. Output Visualization: Final recommendations are displayed through a user-friendly dashboard, enabling clear interpretation and easy action on the results.

4 Transition from Framework to Data Product

Having established our framework, we shifted our focus to building a scalable data product. At its core is a recommendation engine that suggests top-performing dishes by leveraging aggregated popularity patterns from similar restaurant types.

4.1 Data Wrangling Module

4.1.1 Exploratory Data Analysis

To support the development of a robust recommendation engine, we first conducted exploratory data analysis (EDA) to assess data quality and structure. The insights suggest the need to perform wrangling tasks for a clean, consistent, and high-quality model input dataset.

1. **Data Quality and Inconsistencies:** The raw Uber Eats data contains more than 5 million menu records, but it presented significant quality issues (shown in Table 1), including missing critical metrics, formatting inconsistencies, posing a major challenge in popularity score calculation. Therefore, we implemented preprocessing logic to remove all these incomplete, duplicated, or other invalid menu records, which results in around 3.2 million records for next step.

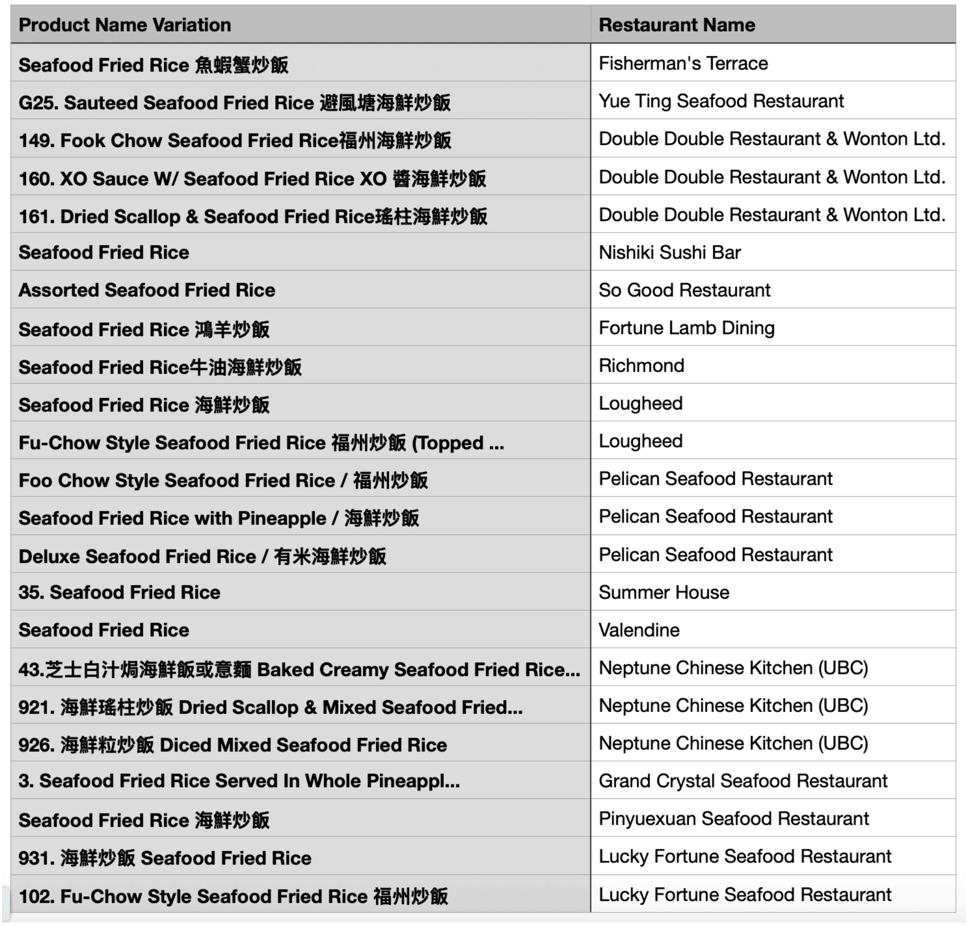
Table 1: Null Entries in Uber Eats dataset (Model Input Data)

column_name	null_count
id	0
restaurant_name	0
score	1958476
ratings	1958476
restaurant_type	2499
full_address	33745
menu_category	160
menu_name	164
menu_item_description	1452305
price	160

2. **Large-Scale Data Processing:** Even after preprocessing, the dataset still contained over **3 million** records, making full in-memory processing impractical due to runtime and memory constraints. To ensure scalability, we implemented batch processing by retrieving indexed chunks and deployed it via a Flask framework with multithreaded API calls for concurrent execution.

3. **Menu Naming Variability:** The biggest challenge was the inconsistent naming of similar dishes, for example, "seafood fried rice" appeared in various forms and languages (Figure 2). This hindered similarity comparisons and recommendation accuracy. To address this, we used OpenAI's GPT API for semantic standardization, enhancing consistency and model performance.

Figure 2: Seafood Fried Rice Spelling Variation

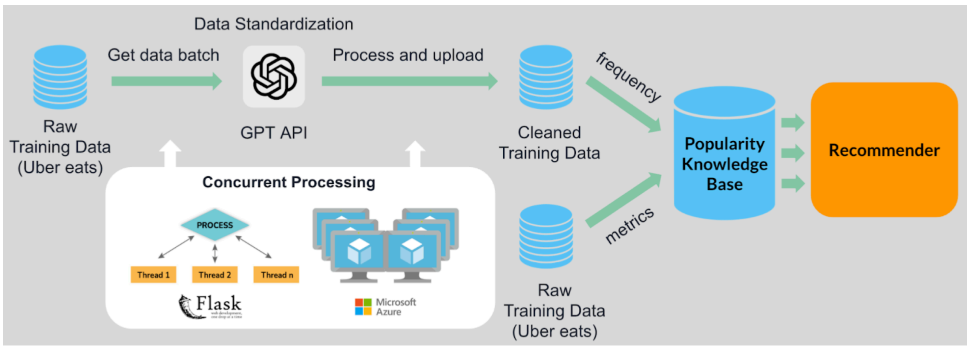


Product Name Variation	Restaurant Name
Seafood Fried Rice 魚蝦蟹炒飯	Fisherman's Terrace
G25. Sauteed Seafood Fried Rice 避風塘海鮮炒飯	Yue Ting Seafood Restaurant
149. Fook Chow Seafood Fried Rice福州海鮮炒飯	Double Double Restaurant & Wonton Ltd.
160. XO Sauce W/ Seafood Fried Rice XO 醬海鮮炒飯	Double Double Restaurant & Wonton Ltd.
161. Dried Scallop & Seafood Fried Rice瑤柱海鮮炒飯	Double Double Restaurant & Wonton Ltd.
Seafood Fried Rice	Nishiki Sushi Bar
Assorted Seafood Fried Rice	So Good Restaurant
Seafood Fried Rice 鴻羊炒飯	Fortune Lamb Dining
Seafood Fried Rice牛油海鮮炒飯	Richmond
Seafood Fried Rice 海鮮炒飯	Lougheed
Fu-Chow Style Seafood Fried Rice 福州炒飯 (Topped ...	Lougheed
Foo Chow Style Seafood Fried Rice / 福州炒飯	Pelican Seafood Restaurant
Seafood Fried Rice with Pineapple / 海鮮炒飯	Pelican Seafood Restaurant
Deluxe Seafood Fried Rice / 有米海鮮炒飯	Pelican Seafood Restaurant
35. Seafood Fried Rice	Summer House
Seafood Fried Rice	Valendine
43.芝士白汁焗海鮮飯或意麵 Baked Creamy Seafood Fried Rice...	Neptune Chinese Kitchen (UBC)
921. 海鮮瑤柱炒飯 Dried Scallop & Mixed Seafood Fried...	Neptune Chinese Kitchen (UBC)
926. 海鮮粒炒飯 Diced Mixed Seafood Fried Rice	Neptune Chinese Kitchen (UBC)
3. Seafood Fried Rice Served In Whole Pineappl...	Grand Crystal Seafood Restaurant
Seafood Fried Rice 海鮮炒飯	Pinyuexuan Seafood Restaurant
931. 海鮮炒飯 Seafood Fried Rice	Lucky Fortune Seafood Restaurant
102. Fu-Chow Style Seafood Fried Rice 福州炒飯	Lucky Fortune Seafood Restaurant

4.1.2 Solution Pipeline

All wrangling tasks were encapsulated in modular Python scripts and integrated into a unified data pipeline (Figure 3). This pipeline includes:

Figure 3: Solution Pipeline (Model Training)



1. Data Cleaning: Due to inconsistencies in the raw menu data from Uber Eats, we implemented preprocessing procedures. Since dishes were named differently across restaurants, we developed a semantic standardization pipeline powered by the GPT API to normalize dish names and group similar items under unified labels.
2. Cleaning Deployment: This standardization process was deployed using a Flask-based batch-processing mechanism to enable concurrent processing. The Heymate engineering team will explore Azure as a potential option for faster and automated deployment. Each batch of raw data was processed and uploaded to generate a cleaned dataset.
3. Scoring and Knowledge Base: Once the data was standardized, we joined each menu item with key popularity metrics. This aggregated table will be stored and used to calculate popularity scores, which will be discussed in detail in a later section.

For the model testing dataset (Figure 5), the similar logic is applied here as Heymate's internal database also exbihits some extent of quality issues (shown in Figure 4). The standardized input will pass through the recommender system to generate top-rated dish recommendations.

Figure 4: Null Entries in internal dataset (Model Testing Data)

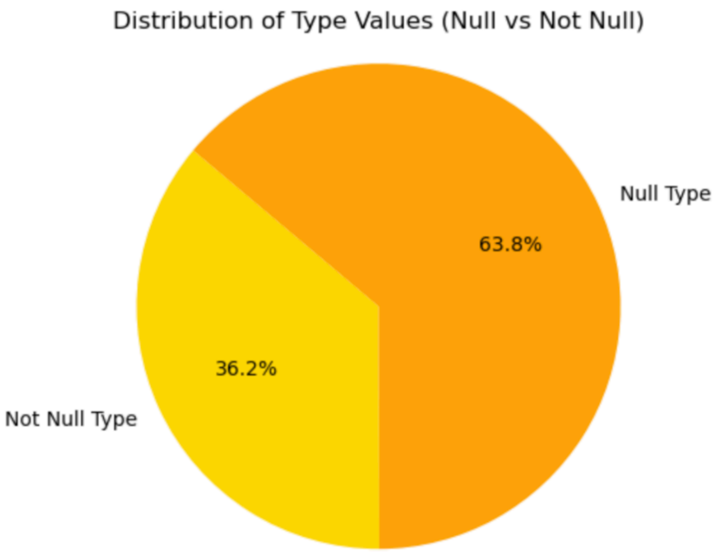
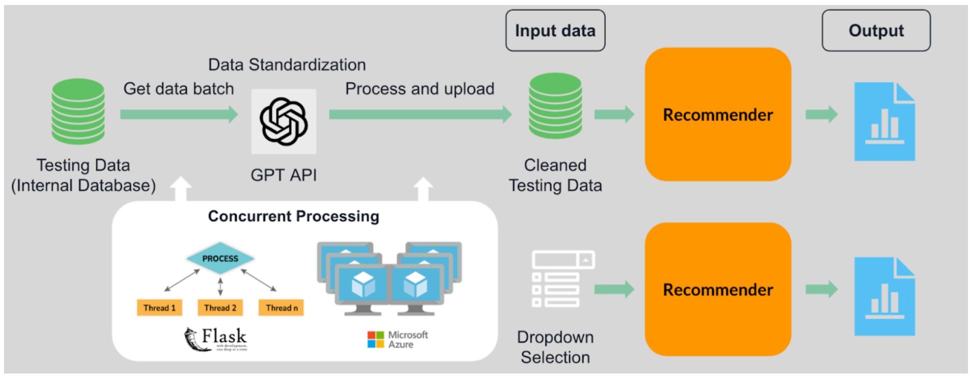


Figure 5: Solution Pipeline (Model Testing)

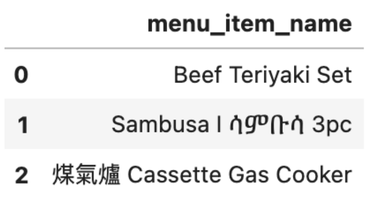


4.2 Data Cleaning Module

After reviewing the overall pipeline, we now take a closer look at the data cleaning component, which plays a key role in addressing inconsistencies in menu dish names and enhancing the quality of downstream recommendations. Specifically, we want to address following issues:

- Mixed languages: Many item names contain both English and non-English text (Figure 6).
- Inconsistent restaurant types: Broad categories (e.g. "American") are mixed with specific terms (e.g. "wings", "sandwich"), making aggregation difficult.
- Combo indicators: Expressions such as "Combo", "Set of Two", or "Family Meal" are used inconsistently, complicating identification.
- Quantity terms: Words like "6 Pc" add additional noise and make parsing less accurate.

Figure 6: Cleaning Challenge Example



	menu_item_name
0	Beef Teriyaki Set
1	Sambusa l ሳምቡሳ 3pc
2	煤氣爐 Cassette Gas Cooker

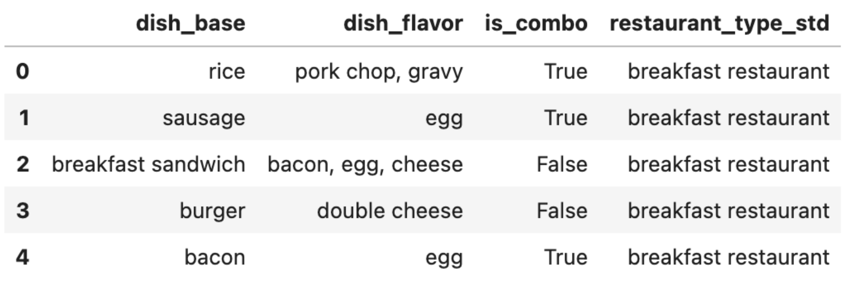
Without resolving these issues, it would have been impossible to run any structured analysis across restaurants at scale. Since our project timeline are relatively tight, we decided to utilize GPT API as it can achieve semantic inference, format alignment, as well as language processing in a more efficient manner while maintaining relativeky great quality. Therefore, we developed a prompt-engineering pipeline that includes two main parts:

1. System prompt: Defines the input and output schema as well as cleaning rules, such as how to identify the core dish name, extract up to five descriptors, determine whether an item is a combo, and standardize the restaurant type.
2. User prompt: Feeds batches of raw menu rows into the model in a list of dictionary format.

The GPT model returns structured output for each menu item, including (Figure 7):

- `dish_base`: core identity (e.g. "fried rice").
- `dish_flavor`: tags like cooking method or toppings (e.g. ["chicken"]).
- `is_combo`: boolean indicating if the item is a combo.
- `restaurant_type_std`: standardized restaurant type aligned with Google Maps Food and Drinks category.

Figure 7: Cleaned Menu Iten Example



	dish_base	dish_flavor	is_combo	restaurant_type_std
0	rice	pork chop, gravy	True	breakfast restaurant
1	sausage	egg	True	breakfast restaurant
2	breakfast sandwich	bacon, egg, cheese	False	breakfast restaurant
3	burger	double cheese	False	breakfast restaurant
4	bacon	egg	True	breakfast restaurant

To ensure extraction quality and reliability, we iteratively refined our prompt engineering strategy:

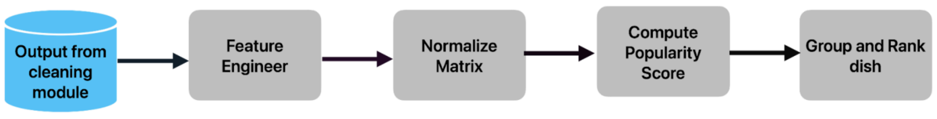
- Required vs. optional fields: We clearly specified which fields (e.g. item name) must be present, and which are optional (e.g. menu description). When optional fields were missing, the model was instructed to infer based on other inputs.

- Formatting rules: We enforced strict formatting in the output, including lowercase, singular, American English spellings, to ensure consistency.
- Controlled restaurant type output: We constrained the model to select from a fixed list of restaurant types aligned with Google Maps Food and Drinks categories.
- Combo identification: We embedded recognition logic for different combo indicators, such as "set of", "combo", or "family meal".
- Prompt length optimization: Through iterative testing, we shortened prompt size while maintaining output quality, helping reduce API costs.
- Row indexing: Each row in the batch was assigned a unique index to link the model's structured output back to other features like rating or score.

With this module, we were able to clean inconsistent menu data into a structured format suitable for downstream analysis and recommendation.

4.3 Recommendations Algorithm

Figure 8: Recommendations Workflow



4.3.1 Feature Integration

Once the LLM produces cleaned menu data (in table `cleaned_menu_mds`), we proceeded to feature integration, enriching this data with relevant popularity metrics. At this stage, we:

- Joined the cleaned Uber Eats menu data (text data) with popularity metrics in the original Uber Eats metadata table `Restaurants_mds`, which contains rating counts and rating scores.
- This step allowed us to bring back three key popularity indicators that were temporarily removed during cleaning:
  - Number of Ratings: how many users rated the dish
  - Average Rating: the restaurant average rating score given by users
  - Item Frequency: how often a dish appears, counting occurrences grouped by dish base, dish flavor, and restaurant type

- We also removed entries where `is_combo == True` to avoid noise in popularity estimates.

The resulting dataset forms the **core knowledge base**, a structured, popularity-aware version of Uber Eats data, on which all recommendations are based.

4.4 Normalize Matrix

For popularity score calculation, as the raw values of frequency or rating counts can be heavily skewed or on different scales. We applied **MinMaxScaler** to rescale them to the [0, 1] range to ensure that the three indicators (rating counts, rating scores, frequency) are comparable and balanced.

Normalized columns include:

- `freq_scaled`: normalized frequency of the dish
- `rating_scaled`: normalized count of user ratings
- `score_scaled`: normalized average user rating

This step creates a uniform basis for computing the final popularity score.

4.5 Compute Popularity Score

We calculate a weighted popularity score for each dish as follows:

```
popularity_score = (0.2 * freq_scaled + 0.6 * rating_scaled + 0.2 * score_scale)
```

These weights were selected based on exploratory testing and domain intuition from the partner, prioritizing the number of reviews as a strong signal of customer engagement, an approach supported by industry practices/research in similar recommendation systems (Chitalia (2023) Al-Rubaye and Sukthankar (2020))

- **60% rating count: (`rating_scaled`)** Measures how many people have reviewed; we viewed it as the strongest proxy of broad customer engagement.
- **20% average score: (`score_scaled`)** captures perceived quality but is often biased by low volume.
- **20% frequency: (`freq_scaled`)** reflects widespread presence on menus but doesn't always indicate desirability.
  All computed scores are stored in a SQL table: `cleaned_menu_with_popularity` This becomes the knowledge base table used for filtering and recommendations.

4.6 Group and Rank Dishes (Filtering & Output Logic)

At recommendation time, the system:

1. Accepts input from a restaurant partner (up to 3 restaurant types)
2. Filters the knowledge base (`cleaned_menu_with_popularity`) using:
   - Exact matches on restaurant_type_std
   - Filters out duplicate dishes (e.g. same base/flavor combo for the same type)
3. For multi-type requests, the popularity scores are averaged across the selected types
4. Dishes are then ranked based on their average popularity score

Finally, the top N dishes (configurable) are returned as output.

4.6.1 Clarification on Filtering Scope

Currently, our system supports **structured filtering only** specifically, by one to three restaurant_type_std values (e.g. "pizza restaurant", "sandwich shop"). These types are extracted during LLM cleaning and matched exactly during the filtering process. This was inspired by best practices in popularity-based recommenders, which focus on transparency and operational simplicity in early-stage deployment Sreekala (2020). At this stage, we do **not** support:

- Keyword-based filtering (e. searching for "pepperoni")

- Semantic/embedding-based matching (e.g., interpreting “comfort food” or “something spicy”)
- GPT-powered fuzzy search at recommendation time

This was a deliberate choice to prioritize **transparency, control,** and **execution speed** in our MVP. Additionally, our dataset, though enriched and cleaned, lacked consistent free-text queries or labelled user intent, which made implementing semantic search or GPT-based query interpretation less feasible at this stage.

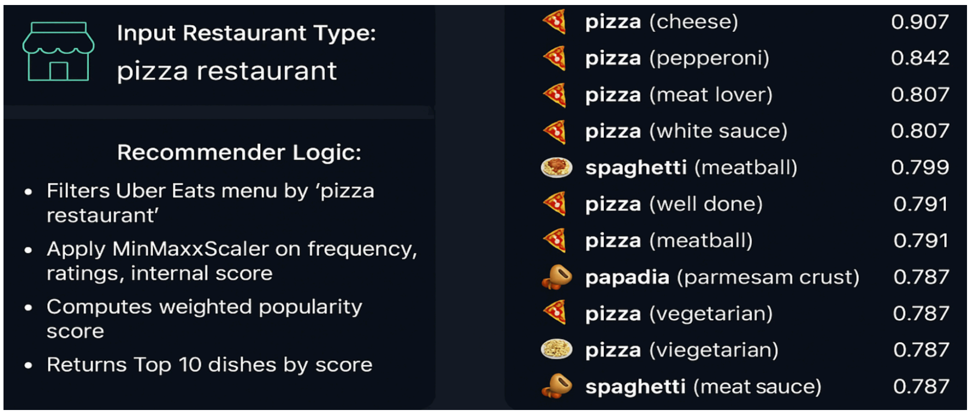
5 Example Use Case: Recommending for a Pizza Restaurant

To demonstrate how the recommendation engine works in practice, we include a use case where a partner restaurant is identified as a **“pizza restaurant”**. The system filters all dishes from our cleaned Uber Eats knowledge base that match this restaurant type and scores each based on its normalized rating score, frequency, and rating count. The weighted score is then used to rank the items, returning the top 10 most relevant dishes.

As shown in the figure below Figure 9, the top-ranked items for a pizza restaurant include popular classics such as:

- Pizza (cheese) and Pizza (pepperoni) — widely rated and highly frequent across menus.
- Pizza (meat lover) and Spaghetti (meatball) — dishes with consistent performance.
- Less expected but still relevant items like Papadia (parmesan crust) also appear due to strong metric combinations.

Figure 9: Recommendations For a Pizza Restaurant”



Input Restaurant Type:
pizza restaurant

Recommender Logic:

- Filters Uber Eats menu by ‘pizza restaurant’
- Apply MinMaxxScaler on frequency, ratings, internal score
- Computes weighted popularity score
- Returns Top 10 dishes by score

pizza (cheese)	0.907
pizza (pepperoni)	0.842
pizza (meat lover)	0.807
pizza (white sauce)	0.807
spaghetti (meatball)	0.799
pizza (well done)	0.791
pizza (meatball)	0.791
papadia (parmesam crust)	0.787
pizza (vegetarian)	0.787
pizza (viegetarian)	0.787
spaghetti (meat sauce)	0.787

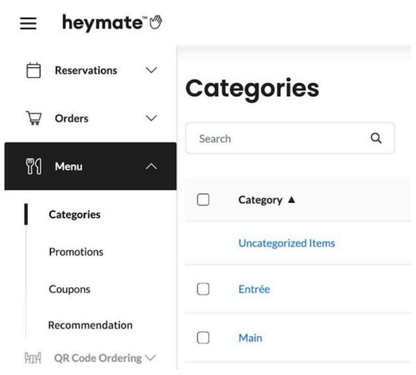
5.1 Visualization Demo: Surfacing Actionable Recommendations

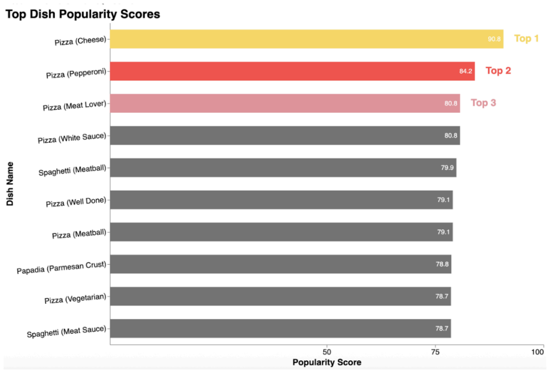
To make our recommendation results interpretable and actionable, we built a two-part visualization:

5.1.1 CRM System Integration Mockup

On the left of Figure 10, we show a conceptual UI for how this recommendation module could be embedded in the Heymate CRM by the Heymate engineering team. This allows restaurant managers to easily access insights from their dashboard, specifically under the “Recommendation” tab, alongside other operational menus.

Figure 10: Visualization Demo”





5.1.2 Recommendation Output Visualization

On the right of Figure 10 is an Altair-based bar chart showcasing the top dishes sorted by computed popularity score. Key design features include:

- Top 3 highlights using Heymate brand colours (yellow, red, rose).
- Pop score bars are annotated directly with percentage labels.
- Hover tooltips for dish name and score for ease of exploration.
- Combined label (dish base + flavor) for intuitive comprehension.

This visualization not only validates that our scoring pipeline works but also provides a clear and professional way for stakeholders to compare and act on dish performance. It is also exportable as a standalone HTML component and ready for dashboard integration.

6 How to Use the Data Product

We designed this data product to support Heymate’s internal team and clients in making data-informed menu decisions.

6.1 Intended Usage

Heymate can utilize this tool in two primary ways:

- Internal Use Mode:
  - Data Update & Validation: When a new data source becomes available, the internal team can upload and process it through the system to evaluate performance with updated inputs.
  - Testing & Iteration: By inputting internal client restaurant types, internal users can assess the quality of dish name cleaning and observe how the system generates standardized output. This enables iterative refinement and ensures reliability before onboarding new clients.
- Client-Facing Deployment: On the frontend, a client can simply select their restaurant type (e.g. “pizza restaurant”), and the system returns a list of top recommended dishes based on aggregated popularity metrics from similar restaurants. This empowers merchants to make data-driven menu decisions aligned with current market trends.

Overall speaking, to enable a scalable, end-to-end pipeline that supports both internal testing and client-facing deployment, we applied several key data science techniques. Below, we highlight three methods that were critical to addressing real-world data challenges.

7 Data Science Methods

We applied many data science techniques in our capstone project and are highlighting 3 here:

- LLM Integration
- Distributed Deployment
- Materialized View

7.1 LLM Integration

We leveraged large language models (LLMs) to clean and standardize the menu data. The raw data was not available for direct use in our recommendation system due to inconsistencies in formatting, spelling variations, and the presence of multiple languages. By integrating LLMs, we were able to extract key information of dish bases and dish flavors, from item names, categories, and descriptions with high accuracy. Even in the case when the description is missing, the LLMs can still infer the information from the item name and category, as well as the context of the restaurant name.

7.1.1 Limitation

This approach requires payment for API usage. After evaluating trade-offs between computational power and cost, we chose to use the **ChatGPT-4o mini** model.

7.1.2 Alternative Methods Considered

- Regular Expressions: Effective for extracting structured patterns, but not feasible here due to inconsistent formatting and multilingual input.
- Custom Deep Learning Model: While potentially powerful, this would require labelled training data and significant time and computational resources. We don't have such resources within our project scope.
- Locally Deployed LLM: This could reduce long-term costs, but setup and maintenance would bring practical challenges within our capstone timeline.

7.2 Distributed Deployment

Our project involved processing a large dataset, over 3 million rows from the Uber Eats dataset. Cleaning this data sequentially would have taken approximately 5,000 hours, which was not feasible within our timeline. To solve this, we implemented a distributed deployment infrastructure to significantly speed up processing. We designed and deployed an HTTP-triggered function to process data in batches. Each instance is triggered via a web request, with the batch range passed as parameters. The system logs the task status at both the start and end of execution. To monitor tasks internally, we also built a dashboard in Looker Studio. Initially, we planned to deploy using Azure Functions on our partner's cloud infrastructure. However, due to security configuration challenges, we were unable to proceed with this plan. As a fallback, we deployed the system locally using the Flask framework and ran up to 20 worker instances concurrently, achieving a 20x speedup.

7.2.1 Limitation

The number of concurrent worker instances is limited by the ChatGPT API rate limit.

7.2.2 Alternative Approaches Considered

We evaluated other cloud deployment solutions, such as EC2 from Amazon Web Services and Google Cloud Functions from the Google Cloud Platform. However, our partner uses Microsoft Azure, and we prioritized consistency within that ecosystem. In the future, our partner's engineering team plans to migrate our local deployment to Azure Functions, once security configurations are in place.

7.3 Materialized View

A Materialized View (also known as a Persistent Derived Table) is a database optimization technique that stores the result of a query as a physical table. This allows for much faster data retrieval by avoiding repeated computation over large datasets. Initially, it took around 6 minutes to generate a restaurant recommendation due to the volume of data. Such delays are unacceptable in production, especially since the recommendation module will eventually be integrated into the partner's CRM system. To optimize query performance, we implemented a materialized view to cache pre-computed results. This optimization reduced runtime from 6 minutes to just 3 seconds.

7.3.1 Limitations

- Additional Storage Cost: The materialized view aggregates data from the original 3-million-row table, incurring some storage cost. However, this is minimal relative to the base data and is not a major concern.
- Maintenance and Updates: When new data is ingested into the model input pipeline, the materialized view must be refreshed. To handle this, we established an automated workflow that updates the materialized view upon the successful completion of each data ingestion task.

8 Justification Over Other Products/Interfaces

There are existing solutions that rely entirely on LLMs to build AI agents for restaurant recommendations. In these systems, user inputs are translated into prompts and sent to an LLM, which generates recommendations based on its internal knowledge.

However, this approach has several clear limitations:

1. LLMs are transformer-based models and are not well-suited for handling structured logic or computations involving large-scale tabular data.
2. Interpretability is low: these systems often function as black boxes, making it difficult to understand or explain how the recommendations are generated.

3. Lack of real market data: These models typically do not incorporate up-to-date or domain-specific datasets. In contrast, our system is built on a dataset of over 3 million real menu records, providing a much more grounded and data-driven foundation for recommendations.

9 Conclusions and Recommendations

9.1 What Problem Were We Solving?

Heymate! seeks to empower its restaurant partners by offering data-driven menu recommendations that encourage customer return visits. However, partners often lack clear insights into which menu items perform well across the market and why. Our project aimed to close this gap by designing a popularity-based recommendation system that scores dishes based on the number of ratings, average rating scores, and frequency across menus, offering partners a transparent foundation for data-backed decision-making.

9.2 How Does Our Solution Address It?

We developed a full-stack pipeline that:

- Cleans and standardizes restaurant menu data using LLMs,
- Joins the cleaned internal menu with Uber Eats data to enrich it with restaurant-level popularity signals,
- Computes a weighted popularity score using MinMaxScaler across three metrics,
- Filters and ranks menu items based on restaurant type(s),
- Visualizes the results in an interactive Altair chart for use in Heymate's CRM.

This product functions well as a minimum viable recommendation engine, particularly for restaurant partners without access to historical purchase data. Its transparent logic and simple interface make it accessible for immediate use and experimentation.

9.3 Limitations

While our data product offers a practical solution for transforming raw menu data into structured insights, several limitations remain:

- **Popularity Bias:** Our current popularity scoring system relies on static metrics, such as item frequency or appearance across menus, due to the lack of real transaction data. This makes it a proxy measure and may not fully reflect true customer preferences. The score could be further improved by incorporating additional quantitative features like price, review volume, or order frequency if available in the future.
- **Static Reference:** Our analysis is currently based on static snapshots of menu data. Without timestamped records, we are unable to capture how item popularity or menu composition evolves. This limits our ability to detect trends such as seasonal specials or emerging bestsellers.
- **Hard Filtering:** Restaurant type filtering is an exact match only (e. "pizza restaurant"), limiting support for broader or fuzzy queries like "comfort food" or "date night meals". A more flexible filtering mechanism could improve the versatility of the tool.

- **Evaluation Challenge:** Due to limited deployment and lack of real-time feedback from users, our validation process is based on illustrative case studies (e.g. the pizza restaurant example). While useful for demonstration, A/B testing or real user feedback would be necessary to confirm the real-world effectiveness of the tool.
- **Scalability Constraint:** Although our local pipeline performs well, we haven't deployed it to Azure yet due to resource and security constraints. As a result, the system is not yet scalable enough for real-time or production use cases.
- **Lack of Granular Personalization:** While our system standardizes restaurant types (e.g. "Chinese restaurant"), it does not yet support finer-grained classification at the sub-type level (e.g. "Szechuan," "Cantonese"). Additionally, the recommender ignores important factors such as local popularity, pricing, and user-specific preferences. These limitations reduce the system's ability to deliver tailored insights for diverse audiences.

9.4 Recommendations for Heymate

To evolve this prototype into a production-ready recommender, we recommend:

1. Temporal Tracking: Incorporate timestamped data to uncover seasonal patterns and trends.
2. Restaurant Feedback Loop: Integrate feedback from partner restaurants to refine recommendations over time.
3. Semantic Filtering: Support flexible, natural-language queries using embeddings or GPT-powered matching.
4. Cloud Deployment: Deploy to Azure to support scalability and integrate directly with Heymate's CRM system.
5. Subtype Extraction: Enhance LLM outputs with more detailed tags (e.g. "spicy," "gluten-free," "vegan") for deeper filtering.
6. Evaluation Framework: Design structured evaluations using click data, client interviews, or business KPIs to measure impact.

Our product lays a solid foundation for **scalable, transparent**, and **explainable menu recommendations**. With further iteration, it can evolve into a dynamic and adaptive system that supports Heymate's long-term vision for partner success.

References

Al-Rubaye, S., and G. Sukthankar. 2020. “Popularity-Based Ranking of GitHub Repositories.” *IEEE*. https://ieeexplore.ieee.org/document/9458206.

Chitalia, A. 2023. “Yelp Popularity Score Calculator.” https://scholarworks.sjsu.edu/etd_projects/1261.

Sakib, Ahmed Shahriar. 2023. “Uber Eats USA Restaurants and Menus.” https://www.kaggle.com/datasets/ahmedshahriarsakib/uber-eats-usa-restaurants-menus.

Sreekala, Keshetti. 2020. “Popularity-Based Recommendation System.” https://www.researchgate.net/publication/355773477.