Institute of Systems Science, National University of Singapore

Master of Technology
Project Report

Contactless Menu Recommender system
Project Report

Team Members

Mu Aohua - A0121924M

Lee Joon Hui Jeremy - A0048174A

Executive Summary

A 2021 Forbes article (*Morgan*, 2021) about “What Will Restaurants Look Like After Covid?” highlighted that restaurants will increasingly use technology such as digital menus to facilitate food ordering and maintain social distancing in order to reduce human interaction. Such drastic changes force restaurant owners to adopt technology that might not be optimised for a restaurant.

One such technology that the team has identified as an opportunity to improve upon is the digital menu. Using the lessons taught through the course, the team will be using tools such as Python, Neo4j, as well as techniques like TF-IDF and Expert Systems to build a menu item recommender based on web technology to help restaurateurs increase revenue.

Through this project, the team discovers a new understanding of creating a restaurant menu as well as the impact that recommendation brings. Not only that, the team also learned that the software developed here can also be easily applied to other businesses that provide a menu (such as retail) to drive up their business income.

Problem Description

Overview

The Covid-19 pandemic has changed how dine-in restaurant businesses operate. With social distancing in-place as well as having the expectation of minimising human to human contact, more restaurants are implementing digital menus. However, these menus could not replicate the warm hospitality that waiters and waitresses bring because they are often just a replication of the static physical counterparts.

This creates a missed revenue opportunity for the restaurants because with the absence of human interaction with the restaurant staff, the outlets could no longer recommend appropriate dishes to the diners, hence reducing the potential of the customers spending more per order.

SWOT Analysis

To address this missed opportunity, we use SWOT analysis to identify the solution to the problem. Here are what we have gathered from our analysis:

Strength

Restaurateurs are the food expert recommenders because they know the characteristics of their dishes and they are able to match them with the taste profile of their customers. For instance, a waitress would offer other similar food items based on the general food category that you like (e.g. seafood) and would recommend complementary dishes to go with it (e.g. white wine).

Weakness

However, with the reduction of human to human interaction within restaurants, such recommendations are no longer possible and they translate to an opportunity cost of allowing the customers to spend more per order.

Opportunity

We have seen the effectiveness of recommender systems used by e-commerce businesses like Amazon and YouTube to drive up revenue, and these technologies can be easily transferred to the digital menu that the restaurants are using today.

Threat

However there are multiple solutions today to fill in the digital menu trend and here are some of the competitors as well as their pros and cons:

Competitors	**Pros**	**Cons**
PDF on Google Drive	Cheap and simple to start using.	Hard for consumers to navigate the menu. Still requires a server to take order, hence violating safe-distance. No recommendation feature.
Emenu companies like TabSquare and iMakan	Designed for digital devices hence better usability.	Could be pricey to adopt (up to hundreds of dollars). Complicated to set up. No recommendation feature.

In addition to the limitations mentioned in the table, these solutions also suffer from having the need for stable internet connection and the need to wait for the menu to load.

Solution

Based on the SWOT analysis, our team decided to create a hybrid recommender system that could recommend dishes based on similar and complementary items that live within the digital menu. We believe that this approach drives more revenue for our customers as compared to other competitors, hence creating a competitive edge. Our implementation also does not suffer from the need of having a stable internet connection while still providing instant responses because we have implemented on-device recommendation (see Solution section), which is crucial for restaurants that see huge ordering demand.

We will be using Putien restaurant as our project example, however our solution can be applied to any restaurant. We have also implemented our software using Progressive Web Application (PWA) thus removing any device dependency while still providing a native-like experience.

Knowledge Modeling

Taking reference from the knowledge model construction technique created by University of Amsterdam(*Knowledge Model Construction*, 2001), we will define our approach of capturing the knowledge through identification, specification and refinement.

Knowledge Identification

We have identified 4 data sources that allow us to build the recommendation model. Of the 4 sources, we discovered that having the dish's ingredients enabled us to create a pretty accurate recommender system. This breakthrough came when we realised that restaurant staff implicitly suggest a dish based on our dietary preference which often ties back to ingredients. For example a waiter would usually ask for the kind of protein that we prefer or the spiciness level that we can accept for the dishes.

Such information is largely omitted from existing menus as they are often not useful to the consumer, therefore we look at using external data sources like recipe websites to supplement the ingredient information.

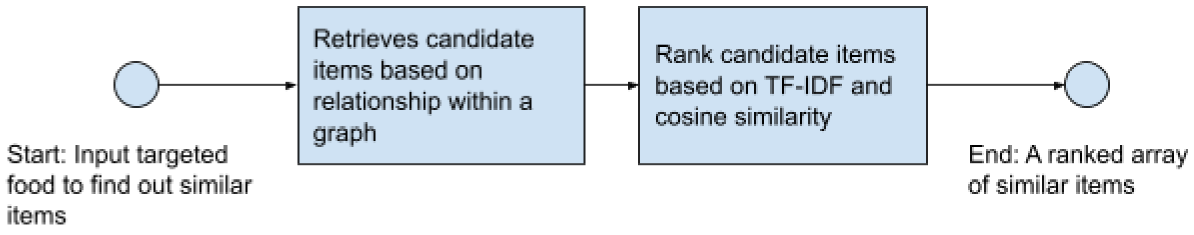
Type of Information	**Source of Information**	**Reason to acquire Information**
Restaurant menu	Restaurant website (we are using Putien's site for this project)	Menu provides essential information of the items that we are using for recommendation. They include: - Category - Dish name - Price - Image
Ingredients of each dish	Various recipe websites	Ingredients allow us to link multiple dishes together so that we can query specific relationships among them.
Order history	Orders created within the application	Order history enhances the relationship among the foods that are not directly related to ingredients. This in turn allows us to provide recommendation based on complementary relationship (cooccurrence rule) and popular items (occurrence rule)
Expert knowledge	Chefs and cooks who know how how to create the dishes	Their knowledge allows us to fill in missing gaps that the first 2 sources have. Some examples of gaps include missing key ingredients as well as the proportion of the ingredients.

Knowledge Specification

We will be employing 3 types of recommendation, namely similar items recommendation, complementary items recommendation as well as popular items recommendation. They are deployed at 3 specific use cases of the application which we will cover more in the **solution** section.

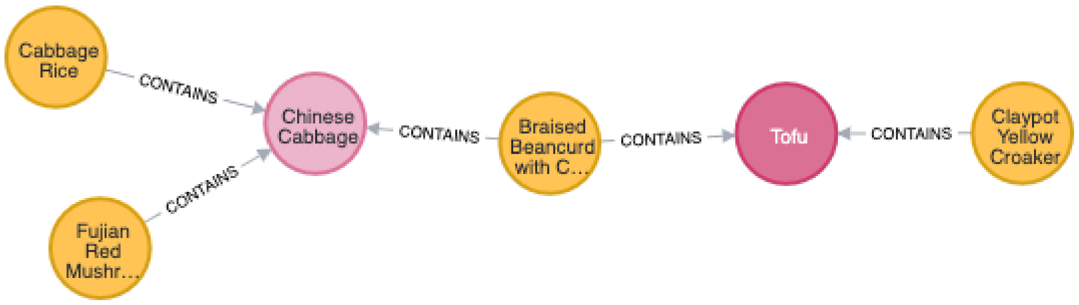
Similarity Recommendation

Our approach to recommending similar items requires 2 steps; the first step is to find out the a list of similar candidates with respect to the target dish, whereas the second step is to perform ranking within the candidates:


Start: Input targeted food to find out similar items
Retrieves candidate items based on relationship within a graph
Rank candidate items based on TF-IDF and cosine similarity
End: A ranked array of similar items

Step 1: Candidate selection

We represent each dish in the menu as a node within a graph and then create relationships among the dishes through the use of ingredients. For example, a typical Chinese dish “Braised Beancurd with Chinese Cabbage” would have yielded 3 other similar dishes based on its key ingredients with just 1 degree of freedom (see graph below).



The results gathered from the relationship are then further ranked based on the second step specified in the next section.

Step 2: Candidate ranking using Term Frequency - Inverse Document Frequency and Cosine Similarity

Through the use of TF-IDF, we rank all the ingredient importance within a dish with respect to other dishes. We can then drop unimportant ones (e.g. salt) and use those with high significance (e.g. Croaker fish) to perform a Cosine Similarity in order to find other similar items.

To illustrate, we will be comparing the following 3 dishes:

D1

D2

D3

The table below tabulates the TF-IDF value of all the ingredients:

Ingredients	Claypot Yellow Croaker Soup(**d1**)	Yellow Croaker Fish with Garlic and Tofu(**d2**)	Cold Pork Trotters Jelly(**d3**)	D/df_i	IDF_i	Claypot Yellow Croaker Soup(**d1**)	Yellow Croaker Fish with Garlic and Tofu(**d2**)	Cold Pork Trotters Jelly(**d3**)
Enoki Mushroom	200g			3/1 = 3	0.4771	200 * 0.4771 = 95.42	0	0
Tofu	200g	160g		3/2 = 1.5	0.1761	200 * 0.1761 = 35.22	160 * 0.1761 = 28.176	0
Salt	5g	10g	5g	3/3 = 1	0	5 * 0 = 0	10 * 0 = 0	5 * 0 = 0
Cooking Wine	10g			3/1 = 3	0.4771	10 * 0.4771 = 4.771	0	0
Ginger	10g		50g	3/2 = 1.5	0.1761	10 * 0.1761 = 1.761	0	50 * 0.1761 = 8.805
Green Onion	10g			3/1 = 3	0.4771	10 * 0.4771	0	0

						= 4.771		
Pepper	5g			3/1 = 3	0.4771	5 * 0.4771 = 2.3855	0	0
Yellow Croaker	450g	300g		3/2 = 1.5	0.1761	450 * 0.1761 = 79.245	300 * 0.1761 = 52.83	0
Coriander		10g	20g	3/2 = 1.5	0.1761	0	10 * 0.1761 = 1.761	20 * 0.1761 = 3.522
Chicken stock		100g		3/1 = 3	0.4771	0	100 * 0.4771 = 47.71	0
Ground bean sauce		10g	10g	3/2 = 1.5	0.1761	0	10 * 0.1761 = 1.761	10 * 0.1761 = 1.761
Garlic		20g	20g	3/2 = 1.5	0.1761	0	20 * 0.1761 = 3.522	20 * 0.1761 = 3.522
Sugar			10g	3/1 = 3	0.4771	0	0	10 * 0.4771 = 4.771
Pig Trotter			1000g	3/1 = 3	0.4771	0	0	1000 * 0.4771 = 477.1
Chives			20g	3/1 = 3	0.4771	0	0	30 * 0.4771 = 14.313
Vinegar			10g	3/1 = 3	0.4771	0	0	30 * 0.4771 = 14.313
Chicken Essence			100g	3/1 = 3	0.4771	0	0	30 * 0.4771 = 14.313
Sesame Oil			20g	3/1 = 3	0.4771	0	0	30 * 0.4771 = 14.313

We then compare similarity score among the different permutation of items:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

|D1| = $\sqrt{95.42^2 + 35.22^2 + 4.771^2 + 1.761^2 + 4.771^2 + 2.3855^2 + 79.245^2}$ = 129.1492

|D2| = $\sqrt{28.176^2 + 52.83^2 + 1.761^2 + 47.71^2 + 1.761^2 + 3.522^2}$ = 76.6795

|D3| =

$\sqrt{8.805^2 + 3.522^2 + 1.761^2 + 3.522^2 + 4.771^2 + 477.1^2 + 14.313^2 + 14.313^2 + 14.313^2 + 14.313^2}$
= 478.0921

Sim(D1, D2) = (35.22*28.176 + 79.245*52.83)/(129.1492*76.6795)=0.5230
Sim(D1, D3) = (1.761*8.805)/(76.6795*478.0921)=0.0004

As a result, we can conclude that item D2 is more similar to item D1 as compared to item D3. You can find the detailed implementation here.

Complementary Item Recommendation

We are using **incremental** cooccurrence counting of each item within all the orders in the complementary items recommendation. As cooccurrence counting is not viable for large datasets, we will keep a total count between the relationship of 2 items in our item-item graph. We will be making the assumption that the items within an order are ranked. With this assumption, we will only increase the count for the next immediate purchased dish. Here is an example:

We have 3 orders with 3 items each:

1. A → B → C
2. B → C → D
3. B → D → E

After the first order is made, we will have an updated graph that looks like this:
A -1-> B -1-> C where the link between A,B and B,C now has a count.

With the 2nd order, our updated graph will look this:
A -1-> B -2-> C -1-> D

Finally the final graph looks like this after accounting for the 3rd order:
A -1-> B -2-> C -1-> D -1-> E
B -1-> D

When we want to retrieve the complementary items for say item B, We can easily get the answer C and D. Moreover, because the dishes are weighted, C will have a higher ranking than D. Note that our recommendation can dynamically change over time as more orders are added, hence providing the best options based on the wisdom of the crowd.

Despite the simplicity of the algorithm, we recognise that such an approach has its drawback, namely:

1. It might end up with a **complete graph** with enough orders, which makes the recommendation meaningless as the complementary items for a dish is no different from the complementary items of another. However, we can overcome this by maintaining the count based on a **sliding window** of when orders are made. In this case, the range of the sliding window forms our hyper parameter for tuning.
2. There is a likelihood that our recommendation may not be very accurate when most customers are not adding the items in sequence. That said, we observed that people usually add things accordingly, for example, ordering drinks and desserts after main dishes.

In order to support on-device recommendation, we store all the items' complementary items that we generated from the backend within the device. We would then update the graph locally when an order is made.

Popular Item Recommendation

This is the simplest implementation of the 3 recommendations. We essentially maintain an occurrence counter within each dish and increment it when orders are made. We will then sort the counter in descending order and display the results on the frontend:

dish_name	order_occurrence
100-Second Stewed Yellow Croaker	51
Deep Fried Pork Trotter with Salt and Pepper	35
Stir-fried Yam	34
PUTIEN Ca Fen	28
Starters Platter	25
Braised Bamboo Shoot	24

Knowledge Refinement

The gist of the knowledge refinement is to fill up gaps that our experts identified. Based on their experience and a few rounds of trial and error, have made 2 significant improvements to the way we store our data in a graph. They are mainly Graph Labeling and Additional Information between Nodes:

Graph labeling

Labeling helps us to query for information more efficiently. There are 2 types of nodes represented in our graph, namely the **main dishes** and their corresponding **ingredients**. The structure of the labels are as follow:

Dish Node

```
(dish_a:Dish:[category of dish]:[additional information])
```

For example:
`(claypot_croaker:Dish:Soup:NonFried)`

Ingredient Node

```
(ingredient_a:Ingredient:[category of ingredient]:[additional
information])
```
For example:
`(rice:Ingredient:Carbohydrate:Main)`

We can then write efficient queries that take advantage of the labels for our recommendations. For example, in our candidate selection process for similar items, we run the following query to retrieve all the neighboring dishes related to the same ingredients:

```
MATCH
(n:Dish:Meat)-[:CONTAINS]->(m:Ingredient:Main)<-[:CONTAINS]-(o:Dish)
WHERE o <> n return n, m, o
```

Additional Information between Node Relationships

In our ingredient-item relationship, we store the proportion of ingredients with respect to each item based on their weight in grams. This weightage is derived from either the recipe websites or through domain experts. Doing so allows us to determine the importance of the ingredients. For example, we are able to identify pork as the main ingredient for the dish “Sweet and sour pork with lychee”. Here is a representation of the ingredient-item relationship:

```
CREATE
(pork_lychee)-[:CONTAINS {quantity:500, unit:'gram'}]->(pork),
(pork_lychee)-[:CONTAINS {quantity:100, unit:'gram'}]->(garlic),
(pork_lychee)-[:CONTAINS {quantity:5, unit:'gram'}]->(salt),
(pork_lychee)-[:CONTAINS {quantity:30, unit:'gram'}]->(egg),
(pork_lychee)-[:CONTAINS {quantity:5, unit:'gram'}]->(sugar),
(pork_lychee)-[:CONTAINS
{quantity:10,unit:'gram'}]->(chicken_essence)
```

Moreover, in our item-item relationship, we also store the order count of each item so that we can use it to promote popular items within the application. We update the count with the query below:

```
MATCH (n),(m)
WHERE ID(n) = 123 and ID(m) = 456
MERGE (n)-[o:COMPLEMENTS]->(m)
ON CREATE SET o.count = 1
ON MATCH SET o.count = o.count + 1;
```

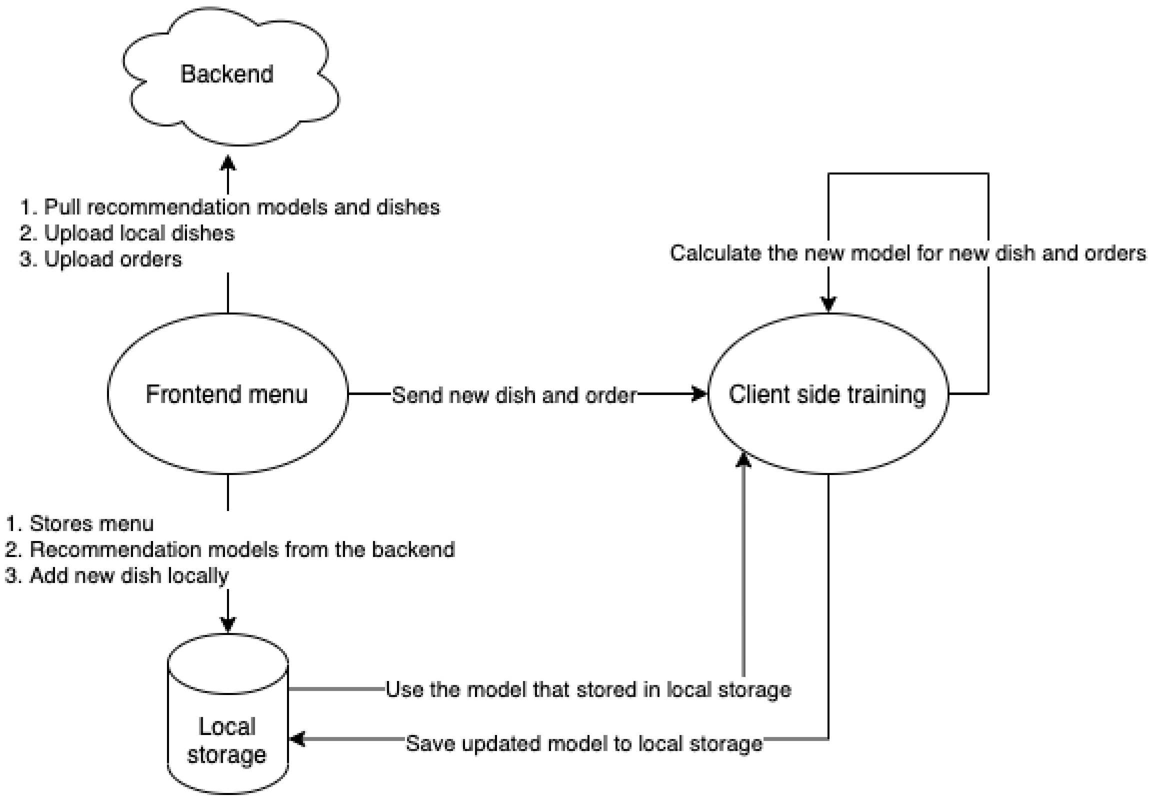
Solution

System Architecture

We intentionally design our architecture to support on-device and offline recommendation. This approach allows restaurants to provide a consistent user experience while removing the need to rely on the backend servers. Our web application is based on a 3-tier architecture that consists of a web frontend layer, a backend layer and a database layer.

We develop the frontend using React and Progressive Web Application technologies to support offline usage. We also use Flask (Python) as the base framework for our backend to provide RESTful APIs for the frontend to consume. The backend is connected to both MongoDB and Neo4j to store existing orders and to store items and ingredients relationships respectively.

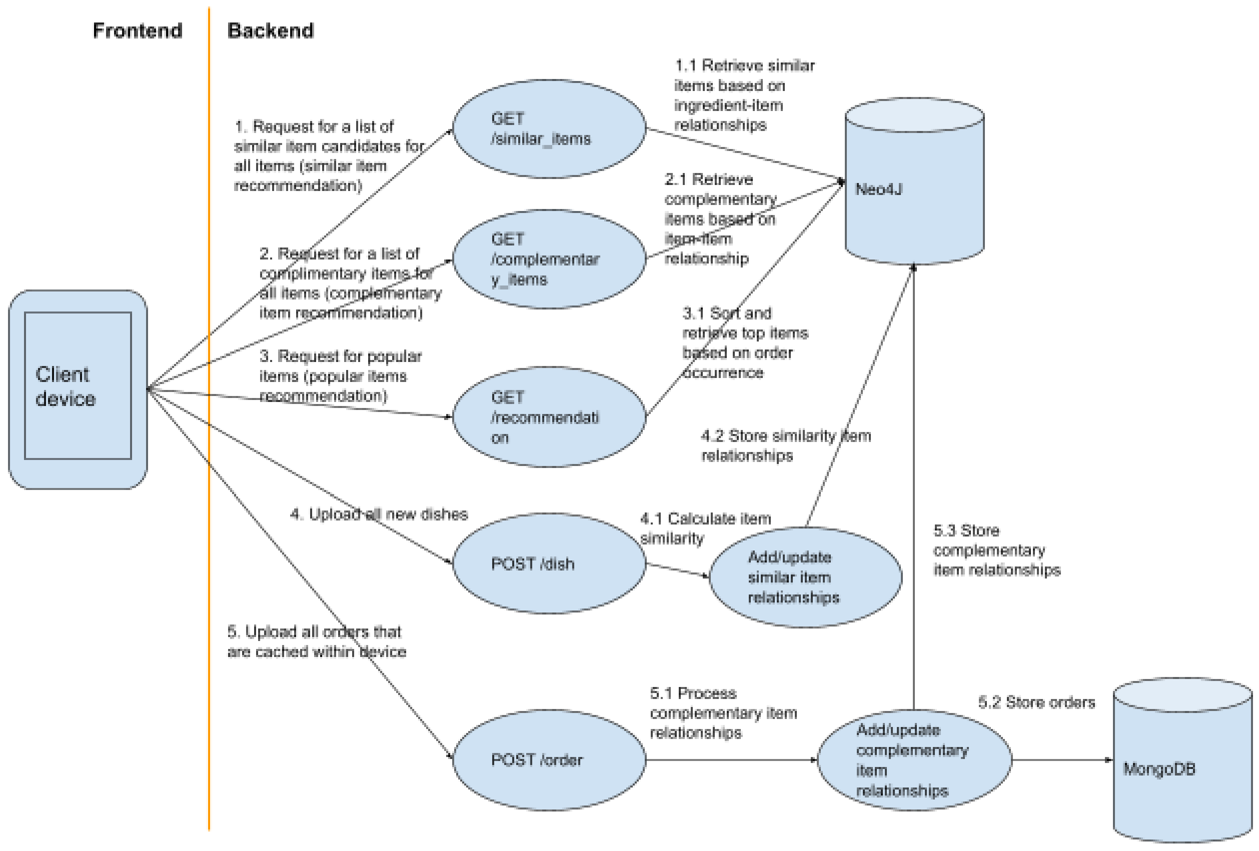
Frontend


Backend
1. Pull recommendation models and dishes
2. Upload local dishes
3. Upload orders
Calculate the new model for new dish and orders
Frontend menu
Send new dish and order
Client side training
1. Stores menu
2. Recommendation models from the backend
3. Add new dish locally
Local storage
Use the model that stored in local storage
Save updated model to local storage

Backend

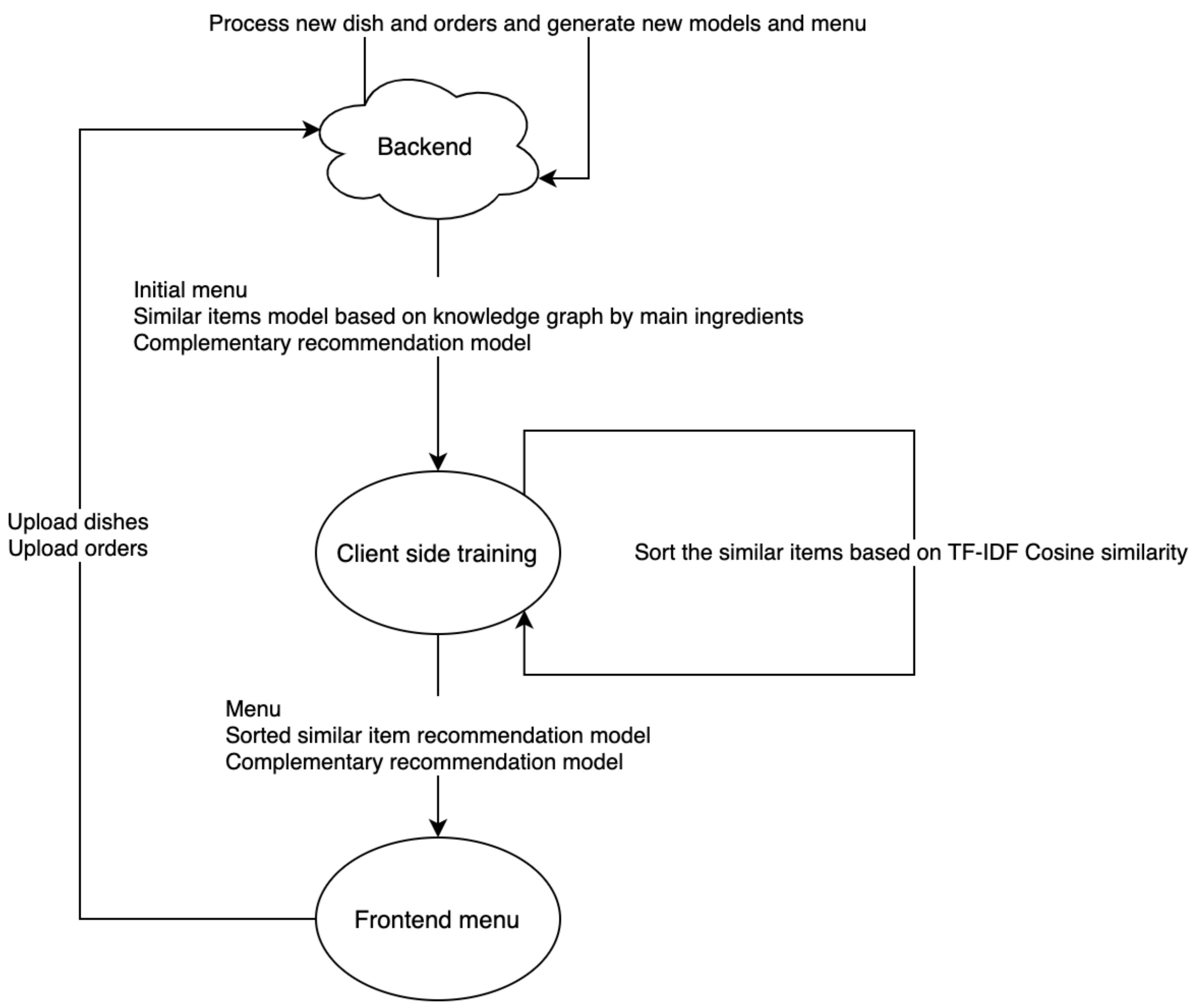
We design our API to support client side offline recommendations. From the diagram below, our flow 1 to 3 provide GET requests that return the entire menus as well as their related recommended items instead of just a single result. Although such setup would incur an initial higher performance penalty (both download size and speed as compared to on-demand requests), subsequent recommendations do not need further server requests and that in turn provide a superior performance boost.

Similarly, we also allow orders and new dishes cached on the frontend to perform a batch upload to the backend (Flow 4 and 5) at desired cadence instead of on-demand push. The backend will then refresh its recommendation models then publish them back to the frontend.


Frontend
Backend
Client device
1. Request for a list of similar item candidates for all items (similar item recommendation)
2. Request for a list of complimentary items for all items (complementary item recommendation)
3. Request for popular items (popular items recommendation)
4. Upload all new dishes
5. Upload all orders that are cached within device
GET /similar_items
GET /complementary_items
GET /recommendation
POST /dish
POST /order
1.1 Retrieve similar items based on ingredient-item relationships
2.1 Retrieve complementary items based on item-item relationship
3.1 Sort and retrieve top items based on order occurrence
4.1 Calculate item similarity
4.2 Store similarity item relationships
Add/update similar item relationships
5.1 Process complementary item relationships
Add/update complementary item relationships
5.2 Store orders
5.3 Store complementary item relationships
Neo4J
MongoDB

Hybrid model

We deploy similar recommendation strategies on both backend and frontend in order to support on-device recommendation:


Process new dish and orders and generate new models and menu
Backend
Initial menu
Similar items model based on knowledge graph by main ingredients
Complementary recommendation model
Upload dishes
Upload orders
Client side training
Sort the similar items based on TF-IDF Cosine similarity
Menu
Sorted similar item recommendation model
Complementary recommendation model
Frontend menu

In the process above, the end device client can refine its own recommendation models locally based on the initial models gathered from the backend. New dishes and orders from various clients can then sync back up to the backend in order to generate improved recommendation models. Maintaining the ability to update models based on new data from clients ensures that we provide the most timely recommendation without establishing a constant backend connection.

Moreover, having the ability to record and update all new information gathered from the frontend, the backend can refine existing models and push them back to the client in order to prevent staleness.

Project Scope

Due to time constraint, we wanted to start small by building a workable prototype. We identified Putien's menu to be our use case because their dishes are general enough to find recipes online. However, our implementation can be extended to other restaurants and even other retail businesses that use a menu.

Assumptions

Syncing of Data Among Devices and Servers

We assume that restaurants would frequently upload their on-device data back to the servers in order to maintain consistent models across clients. If syncing does not happen enough, recommendation models might differ from devices to devices, hence creating an inconsistent user experience.

Knowledge Acquisition

As the ingredient plays a crucial role in our recommendation, we assume that the ingredients of the dishes are easily obtainable from any recipe website or domain experts from the restaurants.

System Features

Our key features that separates us from other solutions include:

1. Supports offline and on-device recommendations so that customers get to view and order their food regardless of internet connection.
2. Provides 3 types of models within software that help drive appropriate recommendations.
3. Works with any businesses that rely on menus to sell their products.

Limitations

Our software comes with its own limitations, they include:

1. The risk of bad-actors tempering with on-device data as local storage can be easily accessed. This will cause all other devices and backend servers to generate inaccurate data once data is synchronised.
2. The risk of complementary items becoming irrelevant if a complete graph is formed. A sliding window to build the relationship of these dishes from orders will reduce this risk.
3. The risk of providing wrong recommendations. In the case of similar items recommendations, this can occur if incorrect ingredients are used. In the case of complementary items recommendation, dishes are not added in sequence.

Conclusion and References

Improvements

With more time to enhance our application, we would implement the following features:

User-item Recommendation

Our existing recommendations are based on the information of the dishes and orders. We could make our recommendation more personalised by using the orders to generate a user-item collaborative filtering model.

Expanding to Other Cases

Our current implementation is applied only to a single band. However, we can test out our system ability on multiple brands such as an online ordering platform to have a greater impact on the recommendation.

Reference

Morgan, B. (2021, February 18). *What Will Restaurants Look Like After Covid?* Forbes. https://www.forbes.com/sites/blakemorgan/2021/01/07/what-will-restaurants-look-like-after-covid/?sh=6e341f5b1acb

Knowledge Model Construction. (2001). Knowledge Modeling. http://ksi.cpsc.ucalgary.ca/KAW/KAW98/schreiber/

Appendix

Project Proposal

Refer to this link:
https://github.com/aohua/KG-food/blob/main/ProjectReport/Project%20Proposal.pdf

Mapped System Functionalities

Similar Items Recommendation

1. Knowledge Graph
2. TF-IDF Vectorization
3. Cosine Similarity

Complementary Items Recommendation

1. Knowledge Graph
2. Cooccurrence Counting

Popular Items Recommendation

1. Knowledge Graph
2. Occurrence Counting

Installation and User Guide

Refer to this link:
https://github.com/aohua/KG-food/blob/main/ProjectReport/Installation%20and%20User%20Guide.pdf

Individual Project Report

For Mu Aohua's report, refer to this link:
https://github.com/aohua/KG-food/blob/main/ProjectReport/MU%20AOHUA%20-%20A0121924M%20-%20%20Individual%20Project%20Report.pdf

For Lee Joon Hui Jeremy's report, refer to this link:
https://github.com/aohua/KG-food/blob/main/ProjectReport/LEE%20JOON%20HUI%20JEREMY%20-%20A0048174A%20-%20%20Individual%20Project%20Report.pdf