Cookr - The Future of Food Tech Hackathon
Anu Ramya R | Samyuktha V

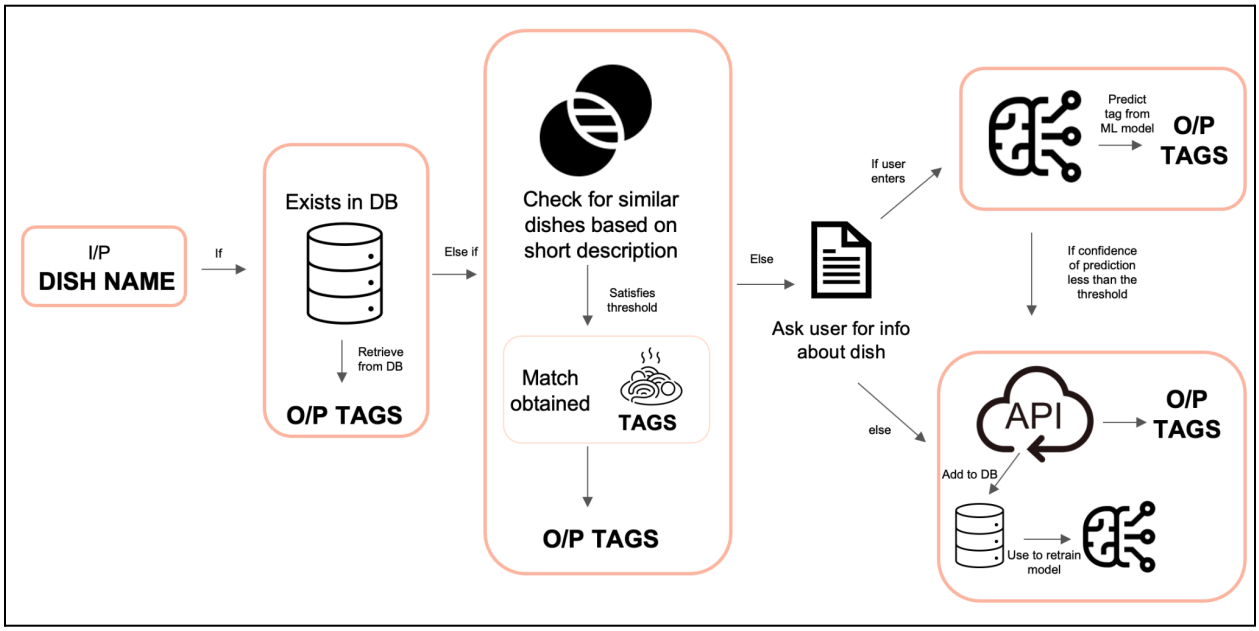
Problem Statement:

1. Item Categorization

Description:

Develop a model or conduct research to outline the essential procedures for categorizing items automatically. Upon a cook adding an item to their kitchen, the model should assign the item to multiple categories automatically.

Design Flow



Workflow:

1. Data Collection
    - Generate a list of dishes (predominantly Indian and also other cuisines) to create the database.
    - Utilizes the OpenAI API, specifically the GPT-3.5 model, to collect comprehensive information for each dish based on a predefined prompt. The prompt specifies input features such as main ingredients, nutritional content, region of popularity, and output classifications including cuisine type, meal type, and health considerations.
    - After receiving responses from the API, the code stores data about each dish as a JSON file.

About the Dataset

Features:

- Dish name
- Main Ingredients (list)

  Nutritional value (per serving):
- Calories
- Protein in g
- Sugar in g
- Saturated Fat in g
- Glycemic index
- Region where it's famous
- Cooking method
- 100-word description
- Preparation time
- Spice Level
- Contains allergens

Output Categories:

- Cuisine
- Meal Type (Breakfast/Lunch/Dinner/Snack/etc)
- Dietary preference(Veg/non-veg/vegan)
- Protein-rich
- Diabetic friendly
- Pregnancy friendly

Sample JSON file obtained for a dish:

{} Aloo Gobi.json > ...


```json
{
  "InputCols": {
    "List of Main Ingredients": "Potatoes, Cauliflower, Spices",
    "Nutritional Value per Serving": {
      "Calories": 170,
      "Saturated Fat in daily %": 5,
      "Sugar in g": 5,
      "Protein in g": 5,
      "Glycemic Index": 20
    },
    "Region where Dish is Popular": "India",
    "Description": "Aloo Gobi is a classic Indian vegetarian dish
    "Spice Level": 6,
    "Cooking Method": "Stir-Frying",
    "Preparation Time": 35,
    "Contains Allergens": false
  },
  "OutputCols": {
    "Cuisine Type": "Indian",
    "Meal Type": "Lunch, Dinner",
    "Dietary Preference": "Veg",
    "Protein Rich": false,
    "Pregnancy Safe": true,
    "Diabetic Friendly": true
  }
}
```

2. Preprocessing for ML Model
    - Load JSON files into a pandas Dataframe.
    - Categorical variables are visualized using count plots.
    - Check for distribution of data among different classes within a label and combine classes with low frequency.

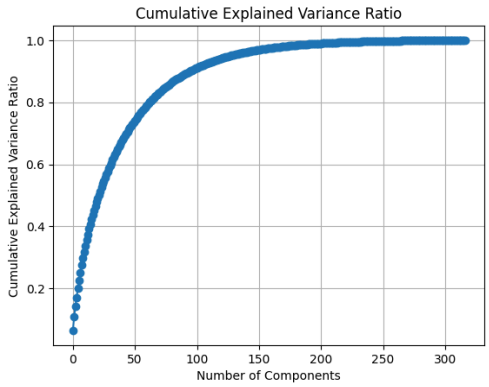
Column transformations:

Numeric columns:

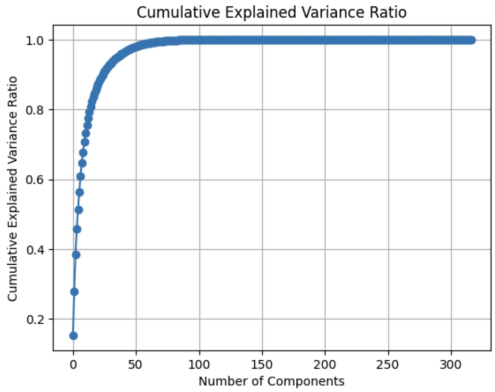
- Standard scaler applied to numeric columns like nutritional value, prep time, etc.

Text based columns (Sentence transformers followed by PCA):

- Utilized Sentence Transformer model to generate embeddings for text columns, such as 'Description' and 'CookingMethod' to capture the semantic information and relationships between words.
- Applied PCA to reduce dimensionality of the sentence embeddings.
- Identified optimal no.of reduced dimensions using elbow plot of cumulative variance explained by principal components



Description



Cooking Method

Categorical Columns (Word2Vec Transformations):

- Trained Word2Vec models for other text based categorical columns like Dish Name, Region and list of Ingredients to generate word embeddings.
- Computed correlation matrices for relationship among the vector dimensions to identify optimal size of vectors, and visualized embeddings correlation using heatmaps.
- Transformed text data into embeddings and flattened them into multiple columns.

Predictor Variables (One-hot Encoding)

- Performed one-hot encoding for categorical variables like Cusine, Meal Type and Dietary Preference.

Overall, the preprocessing pipeline involves scaling numeric data, one-hot encoding of categorical features, transforming text data into numerical embeddings using both Sentence Transformer and Word2Vec models, reducing dimensionality, and preparing the data for model training.

3. Model Building
    - Training and Evaluation: Models are trained and evaluated using training and testing datasets. Metrics include accuracy scores and classification reports.
    - Individual Binary Classification: XGBoost classifiers predict individual binary labels (Protein-Rich, Diabetic-Friendly, Pregnancy-Safe).
    - Multi-label Classification: MultiOutputClassifier with XGBoost is used for multi-label prediction (DietaryPreference, MealType, Cuisine).

    Model evaluation:

Category	Test Accuracy
Protein-Rich	92.5%
Pregnancy-Safe	93.75%
Diabetic Friendly	83.75%
Meal Type	63.75 %
Dietary Preference	81.25%
Cuisine	46.25%

4. Tag Generation

    - If a dish already exists in the DB, generate concise-tags by retrieving appropriate information and infer additional tags from the data.

    - If the dish is not available in the database, search for similar dish using cosine similarity of sentence embeddings from description of the dish. If it satisfies a threshold, return the tags of match dish as output.

    - If no similar dish is found, as the user if they are willing to give information about the dish's ingredients, nutritional content, cooking method, etc.

If they provide the information, generate prediction from ML model:

Predicts attributes like protein-rich, diabetic-friendly, and pregnancy-safe labels, as well as dietary preferences, cuisine type, and meal types.

- Infers additional tags based on feature data:
    - Seafood
    - Chicken
    - Eggs
    - Mutton
    - Millet-based
    - Contains Allergens
    - Spicy

- Interprets dietary preferences, cuisine type, and meal types from the model's output.
- Provides a concise summary of predicted attributes and tags, aiding in understanding the dish's nutritional value and dietary compatibility.

If user doesn't provide dish information or confidence of ML model prediction doesn't satisfy the threshold:

Get the necessary information about the dish using the OpenAI API and infer tags from the response.

Sample output:

Dish Exists in DB:

```
Paniyaram  :  ['Pregnancy-Safe', 'Diabetic-Friendly', 'Veg', 'South Indian', 'Snack']
```

If dish does not exist in DB, find a similar dish:

```
dish=input("Enter name of dish you would like to add: ")
tags=generate_tags(dish)
print(dish," : ",tags)
```

✓ 22.0s

```
Dish not available in database.

Found a similar dish in database:  Egg Curry
Egg Gravy  :  ['Eggs', 'Spicy', 'May Contain Allergens', 'protein-Rich', 'Pregnancy-Safe', 'Diabetic-Friendly', 'Veg', 'South Indian', 'Lunch']
```

Finding using model prediction:

```
Dish not available in database.


Tags generated using model predictions:

Brownie  :  ['Pregnancy-Safe', 'Non-Veg', 'American', 'Dessert']
```

Find tags using API:


```
Fetching data using API...

{
  "InputCols": {
    "List of Main Ingredients": "Ragi (Finger Millet), Buttermilk, Green chilies, Ginger, Curry leaves",
    "Nutritional Value per Serving": {
      "Calories": 150,
      "Saturated Fat in daily %": 5,
      "Sugar in g": 3,
      "Protein in g": 8,
      "Glycemic Index": 45
    },
    "Region where Dish is Popular": "South India",
    "Description": "Ragi Kool is a traditional South Indian dish made from finger millet (ragi) and buttermilk. It is seasoned
    "Spice Level": 4,
    "Cooking Method": "Boiling and Tempering",
    "Preparation Time": 30,
    "Contains Allergens": false
  },
  "OutputCols": {
    "Cuisine Type": "South Indian",
    "Meal Type": "Breakfast, Snack",
    "Dietary Preference": "Veg",
    "Protein Rich": true,
    "Pregnancy Safe": true,
    "Diabetic Friendly": true
  }
}
Ragi Kool  :  ['protein-Rich', 'Pregnancy-Safe', 'Diabetic-Friendly', 'Veg', 'South Indian', 'Breakfast/Snack']
```


2. Last Mile Delivery Batching

In the rapidly evolving last-mile delivery ecosystem of an e-commerce marketplace, the goal is to optimize delivery speed and cost by intelligently batching orders for delivery by the same rider under a set of rules. The task is to design an algorithm that determines the optimal way to batch orders according to the specified rules, ensuring that the total delivery time and cost are minimized. This problem is simplied to be solved geometrically in a cartesian co-ordinate system where lat and long mark x,y on a 2D cartesian plane.

Incorporating All Rules

1. **Same Kitchen, Same Customer**: Batch orders from the same kitchen to the same customer if they are ready at the same time (within a 10-minute window) to the same rider.
2. **Different Kitchens, Same Customer**: If orders for the same customer come from two kitchens 1 km apart and are ready at the same time (within a 10-minute window), assign them to the same rider.
3. **Same Kitchen, Different Customers**: Orders from the same kitchen to two different customers (1 km apart) ready at the same time (within a 10-minute window) should be batched to the same rider.
4. **Orders En Route**: For two orders going to the same customer, where the second kitchen's pick-up is on the way to the customer and ready by the time the rider reaches it (considering a 10-minute preparation window), assign both to the same rider.

5. **Intersecting Delivery Paths**: For two orders where the second customer's drop-off is en route to the first customer's location, and the second kitchen's pick-up is also on the way, batch them if both are ready at the same time (or within a 10-minute window by the time the rider reaches the kitchen).

Input:

1. num_orders: The number of orders.
2. num_drivers: The number of available drivers for delivery.
3. Kitchens: A dictionary of kitchens where key is **kitchen id** and value is **(lat, long) tuple**
4. Drivers: A dictionary of drivers where key is **driver id** and value is current **(lat, long) tuple**
5. Customers: A dictionary of customers where key is **customer id** and value is **customer delivery location (lat, long) tuple.**
6. orders: A dictionary of orders, where the key is the order ID, and the value is a list with **[Kitchen ID, Customer ID, readyTime in minutes].**
3. distances: A matrix representing the distances between any two points in the delivery network (including kitchens and customers).

For ease of computation, assume all drivers travel at same speed and treat lat long as cartesian co-ordinates, and solve geometrically where two lat long points can be connected by straight line for min distance between them, ignore routes and obstacles.

Output:

Return a tuple consisting of two elements:
1. The minimum total delivery time required to deliver all orders following the batching rules.
2. A dictionary with driver IDs as keys and the list of order IDs they are responsible for delivering as values.

Implementation

1. Helper Functions:
- **calculate_distance(point1, point2)**: Calculates the Euclidean distance between two points in a 2D Cartesian plane using the distance formula.
- **find_nearest_order(driver_location, available_orders, Kitchens, Customers)**: Finds the nearest order to a given driver location based on the kitchen's location. It iterates through

available orders and computes the distance between the driver's location and the kitchen associated with each order.

2. Main Simulation Function: `simulate_delivery`

- Inputs:

- orders: A dictionary containing order details including kitchen ID, customer ID, and readiness time.
- Kitchens: A dictionary mapping kitchen IDs to their respective locations.
- Customers: A dictionary mapping customer IDs to their respective delivery locations.
- Drivers: A dictionary containing driver IDs and their initial locations.
- driver_start_point: Initial starting point for drivers.

- Logic:

- Sort orders by readiness time to process them in chronological order.
- Initialize dictionaries to track driver locations, assigned orders, and time elapsed for each driver.
- Iterate through sorted orders:
  - For each order, find the nearest available driver based on current locations and assign the order to that driver.
  - Calculate travel and waiting times for the assigned driver.
  - Update driver's information including assigned orders, elapsed time, and location.

- Compute the overall completion time as the maximum time elapsed among all drivers.

Outputs

- Returns the overall completion time and a dictionary mapping driver IDs to the list of order IDs they are responsible for delivering.

3. Main Execution Block: `main`

- Initializes simplified example inputs including kitchens, customers, orders, drivers, and the initial driver start point.
- Calls the `simulate_delivery` function to obtain the optimized delivery assignment.
- Prints the overall completion time and the list of orders assigned to each driver.
- Optionally, visualizes the delivery process using a visualization function (`visualize_delivery_process`).

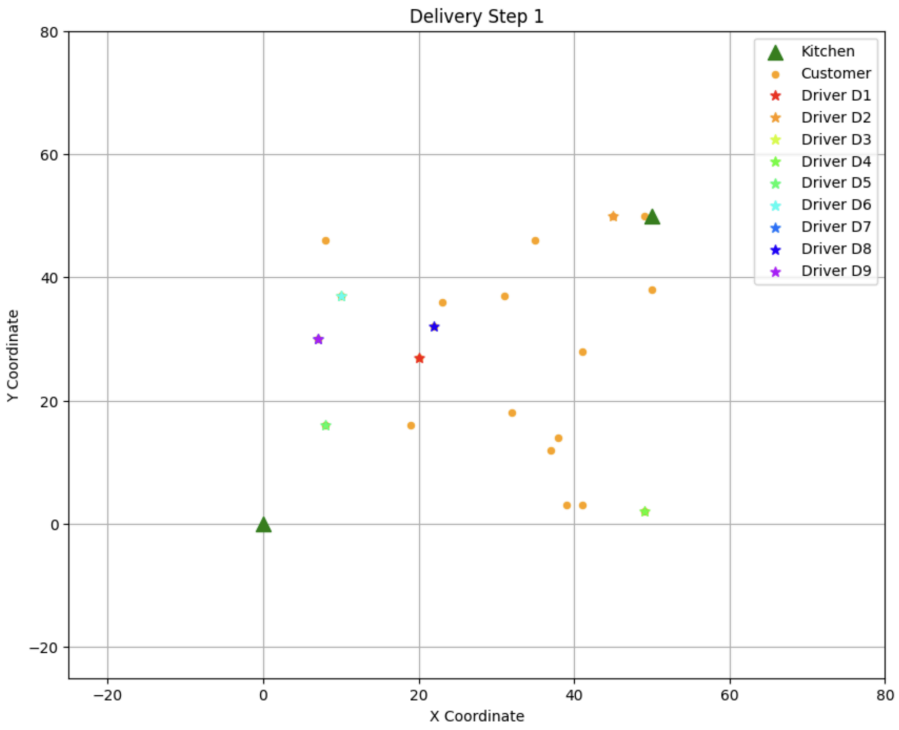
Conclusion:

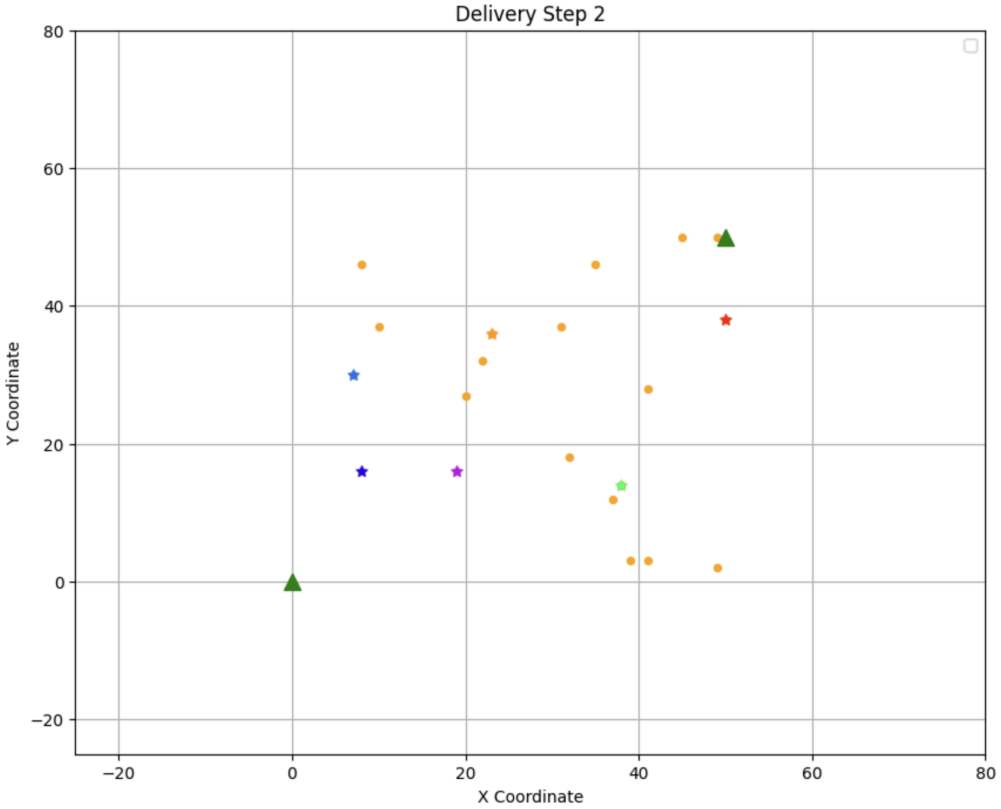
The solution effectively optimizes the last-mile delivery process by intelligently assigning orders to drivers based on their locations and readiness times. It utilizes basic geometric calculations and iteration to achieve the desired optimization outcome, ensuring efficient and timely delivery of orders in the e-commerce marketplace.

Sample Assignments

```
Overall Completion Time: 418.5246293316233
D1: ['O6', 'O3', 'O18', 'O15', 'O13', 'O12', 'O8', 'O14']
D2: ['O22', 'O24', 'O0']
D3: ['O21']
D4: ['O16']
D5: ['O4', 'O5']
D6: ['O11']
D7: ['O9', 'O10', 'O1', 'O19', 'O2']
D8: ['O7', 'O23']
D9: ['O17', 'O20']
D10: []
```

Visualization of delivery steps:




Delivery Step 2
Y Coordinate
X Coordinate
80
60
40
20
0
−20
−20
0
20
40
60
80