**Integration Plan for Product Recommendation and Price Prediction Models**

**1. Introduction**

This document outlines the integration of two advanced machine learning models into a grocery application, enhancing its functionality with both product recommendations and price prediction capabilities. The models used are as follows:

1. **Product Recommendation Model**: Utilizing the Apriori algorithm, this model recommends products based on association rules derived from user purchase patterns.
2. **Price Prediction Model**: A decision tree regression model that predicts product prices based on several input features.

We will delve into the backend integration of these models, real-time basket update processes, and their interaction with the frontend components, creating a seamless user experience.

**2. Backend Integration**

**2.1 Model Loading**

We employ **Flask** to build a robust backend API, which facilitates model interaction. Pre-trained models are loaded directly from disk for both recommendation and price prediction functionalities. Below is the implementation snippet for loading the models:

from flask import Flask, request, jsonify

import joblib

import pandas as pd

from mlxtend.frequent\_patterns import apriori, association\_rules

app = Flask(\_\_name\_\_)

# Load models

product\_model = joblib.load('product\_recommendation\_model.pkl')

price\_model = joblib.load('price\_model.pkl')

* **joblib.load()** is used for efficient loading of models stored in .pkl files.
* The **product\_model** is the core recommendation engine.
* The **price\_model** is the decision tree regression model for predicting product prices.

**2.2 Preparing Data for Recommendations**

To generate product recommendations, we rely on both transaction and product data. The product information and transaction details are stored in **Pandas DataFrames** as shown below:

# Load product data and transaction data

products = pd.DataFrame({

'product\_id': [21137, 13176, 12130, 11859, 21938],

'product\_name': ['Milk', 'Bread', 'Butter', 'Eggs', 'Apples']

})

basket = pd.DataFrame([

[1, 1, 0, 0, 1],

[1, 1, 1, 0, 0],

[0, 1, 0, 1, 1],

[1, 0, 1, 1, 1],

], columns=[21137, 13176, 12130, 11859, 21938])

* The **products** DataFrame contains essential details like product IDs and names.
* The **basket** DataFrame captures transactional data where each row signifies a separate transaction, and each column corresponds to a product.

**2.3 Generating Recommendations**

To produce relevant product recommendations, we employ frequent itemset mining and association rule generation:

# Generate frequent itemsets and association rules

frequent\_itemsets = apriori(basket, min\_support=0.07, use\_colnames=True, verbose=1)

frequent\_itemsets.sort\_values("support", ascending=False, inplace=True)

rules = association\_rules(frequent\_itemsets, metric="support", min\_threshold=0.02)

rules.sort\_values(by="lift", inplace=True)

* **apriori()** is utilized to find frequent itemsets within the transactional data.
* **association\_rules()** generates association rules from the identified itemsets, which are then sorted based on their **lift** values, prioritizing stronger relationships between products.

**2.4 API Endpoints**

We create several API endpoints to handle requests related to reorder predictions, price predictions, and product recommendations:

1. **Predict Reorder Probability:**

@app.route("/predict\_reorder", methods=["POST"])

def predict\_reorder():

features = request.json.get('features')

prediction = product\_model.predict([features])

return jsonify({"reorder\_probability": prediction[0]})

This endpoint receives product-related features and returns the probability that a user will reorder a specific item.

* **/predict\_reorder** endpoint receives features and returns the probability of reordering a product.
* **features** is a JSON object with product-related data.

1. **Predict Price**:

@app.route("/predict\_price", methods=["POST"])

def predict\_price():

features = request.json.get('features')

predicted\_price = price\_model.predict([features])

return jsonify({"predicted\_price": predicted\_price[0]})

* This endpoint forecasts the price of a product based on input features such as category, demand, or seasonal trends.
* **/predict\_price** endpoint receives product features and returns the predicted price.

1. **Get Recommendations**:

@app.route("/get-recommendations", methods=["POST"])

def get\_recommendations():

product\_id = request.json.get('product\_id')

rec\_count = request.json.get('rec\_count', 5)

recommendations = names\_of\_products(rules, product\_id, rec\_count)

return jsonify(recommendations.to\_dict(orient='records'))

* This endpoint delivers a list of product recommendations based on the product ID provided by the user.
* **get-recommendations** endpoint provides product recommendations based on the product\_id.

2.5 **Helper Functions for Recommendations**

These functions assist in generating product recommendations by leveraging association rules:

def arl\_recommender(rules\_df, id, rec=1):

sorted\_rules = rules\_df.sort\_values("lift", ascending=False)

recommendation\_list = []

for i, k in enumerate(sorted\_rules["antecedents"]):

if id in k:

recommendation\_list.extend(list(sorted\_rules.iloc[i]["consequents"]))

return list(set(recommendation\_list))[:rec]

def names\_of\_products(rules\_df, bought, recommend=5):

rec = arl\_recommender(rules\_df, bought, recommend)

name\_of\_rec = {i: products[products["product\_id"] == i]["product\_name"].iloc[0] for i in rec}

return pd.DataFrame(name\_of\_rec.items(), columns=["product\_id", "product\_name"])

* **arl\_recommender()** identifies products frequently bought together with the specified product ID.
* **names\_of\_products()** retrieves and maps product IDs to their respective names.

**3. Real-Time Basket Update**

**3.1 Capturing Purchase Events**

Whenever a user completes a transaction, the system captures the event and its associated details:

{

"transaction\_id": "txn123",

"product\_ids": [21137, 13176, 21938],

"user\_id": "user789"

}

3.2 **Updating Basket Data**

Update the basket DataFrame to include the new transaction:

import pandas as pd

# Example existing basket DataFrame

basket = pd.DataFrame([

[1, 1, 0, 0, 1],

[1, 1, 1, 0, 0],

], columns=[21137, 13176, 12130, 11859, 21938])

# Function to update basket with a new transaction

def update\_basket(transaction\_id, product\_ids):

global basket

new\_transaction = [1 if product\_id in product\_ids else 0 for product\_id in basket.columns]

basket.loc[transaction\_id] = new\_transaction

**3.3 Persisting Changes**

We use **SQLAlchemy** to persist the updated basket data in a SQLite database:

# Persisting Updated Basket Data

import sqlalchemy

engine = sqlalchemy.create\_engine('sqlite:///basket.db')

def persist\_basket():

basket.to\_sql('basket', engine, if\_exists='replace', index=False)

3.4 **Reflecting Updates in Models**

After updating the basket, we reload the data and recalculate frequent itemsets and association rules:

def reload\_basket():

global basket

basket = pd.read\_sql('basket', engine)

# Recalculate frequent itemsets and association rules

frequent\_itemsets = apriori(basket, min\_support=0.07, use\_colnames=True)

rules = association\_rules(frequent\_itemsets, metric="support", min\_threshold=0.02)

**4. Frontend Integration**

**4.1 Fetching Recommendations**

The frontend makes a POST request to retrieve product recommendations from the backend:

function fetchRecommendations(productId, recCount) {

fetch('/get-recommendations', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({ product\_id: productId, rec\_count: recCount }),

})

.then(response => response.json())

.then(data => {

console.log('Recommendations:', data);

})

.catch(error => console.error('Error fetching recommendations:', error));

}

**4.2 Fetching Price Predictions**

A similar POST request is made to predict product prices:

function fetchPricePrediction(features) {

fetch('/predict\_price', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({ features: features }),

})

.then(response => response.json())

.then(data => {

console.log('Predicted Price:', data.predicted\_price);

})

.catch(error => console.error('Error fetching price prediction:', error));

}

**Conclusion**

This document outlines a thorough approach to integrating machine learning models into the Discount Mate Application. From backend services using Flask to frontend integration and real-time basket updates, this solution provides a robust framework for personalized product recommendations and price prediction, ensuring an enhanced user experience.