PROGRAMMING ASSIGNMENT-02

Intelligent system

Recipe recommendation FastAPI Application

ITBIN-2110-0094 - Lakshitha Naveen

Contents

Introduction

Overview

This project provides a recipe recommendation system that takes a list of ingredients as input and returns a list of recommended recipes. The recommendation system uses a TF-IDF vectorizer for feature extraction and cosine similarity for matching.

Application link- [FastAPI - Swagger UI (railway.app)]()

Github link - https://github.com/LakshithaNaveenRathnasiri/FastAPI-application.git

Model Training

Data Collection

Dataset: The dataset is a JSON file containing recipe data with ingredients and cuisine information. It is loaded from /content/recipe/train.json.

- Used Kaggle dataset.

Model Architecture

TF-IDF Vectorizer: Used for transforming ingredient texts into numerical feature vectors.

Cosine Similarity: Measures the similarity between the user's input ingredients and the recipes' ingredients.

Training Process

Data Preprocessing-

Ingredients are cleaned and tokenized.

Stopwords are removed and words are lemmatized.

Vectorization: The cleaned ingredients are vectorized using a TF-IDF Vectorizer with n-grams.

Recommendation Function: Calculates similarity scores and returns top recommendations based on user input.

code - import pandas as pd

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import train_test_split

# Load the dataset
train_data_path = '/content/recipe/train.json'
train_df = pd.read_json(train_data_path)

# Clean and preprocess data
def preprocess_ingredients(ingredients):
    lemmatizer = WordNetLemmatizer()
    custom_stopwords = set(stopwords.words('english')).union({'fresh',
'chopped', 'sliced', 'diced', 'large', 'small', 'medium', 'extra', 'extra-
virgin', 'virgin'})

    cleaned_ingredients = []
    for ingredient in ingredients:
        ingredient = ingredient.lower()
        ingredient = re.sub(r'[^a-z\s]', '', ingredient)
        tokens = [lemmatizer.lemmatize(word) for word in ingredient.split() if
word not in custom_stopwords]
        cleaned_ingredients.append(' '.join(tokens))

    return ' '.join(cleaned_ingredients)

train_df['cleaned_ingredients'] =
train_df['ingredients'].apply(preprocess_ingredients)

# Vectorization with n-grams
vectorizer = TfidfVectorizer(stop_words='english', ngram_range=(1, 2))
ingredient_matrix = vectorizer.fit_transform(train_df['cleaned_ingredients'])

# Recommendation function
def recommend_recipes(user_input, num_recommendations=5):
    user_input_str = ' '.join(user_input)
    user_input_vector = vectorizer.transform([user_input_str])
    similarity_scores = cosine_similarity(user_input_vector,
ingredient_matrix)
    similarity_scores = similarity_scores.flatten()
    top_indices = similarity_scores.argsort()[-num_recommendations:][::-1]
    return train_df.iloc[top_indices]

# Example usage
user_input = ['chicken', 'tomato', 'rice']
recommendations = recommend_recipes(user_input)
```

```
print("Recommended Recipes:")
print(recommendations[['cuisine', 'ingredients']])




# Evaluate precision and relevance
def evaluate_precision_at_n(user_input, relevant_cuisine, n=5):
    recommendations = recommend_recipes(user_input, num_recommendations=n)
    relevant_recommendations = recommendations[recommendations['cuisine'] ==
relevant_cuisine]
    precision_at_n = len(relevant_recommendations) / n
    return precision_at_n

def evaluate_most_relevant_cuisine(user_input, n=5):
    recommendations = recommend_recipes(user_input, num_recommendations=n)
    cuisine_counts = recommendations['cuisine'].value_counts()
    most_relevant_cuisine = cuisine_counts.idxmax()
    precision_at_n = cuisine_counts.max() / n
    return most_relevant_cuisine, precision_at_n

most_relevant_cuisine, precision = evaluate_most_relevant_cuisine(user_input,
n=5)
print(f'Most Relevant Cuisine: {most_relevant_cuisine}')
print(f'Precision@5 for {most_relevant_cuisine}: {precision:.2f}')
```

Evaluation

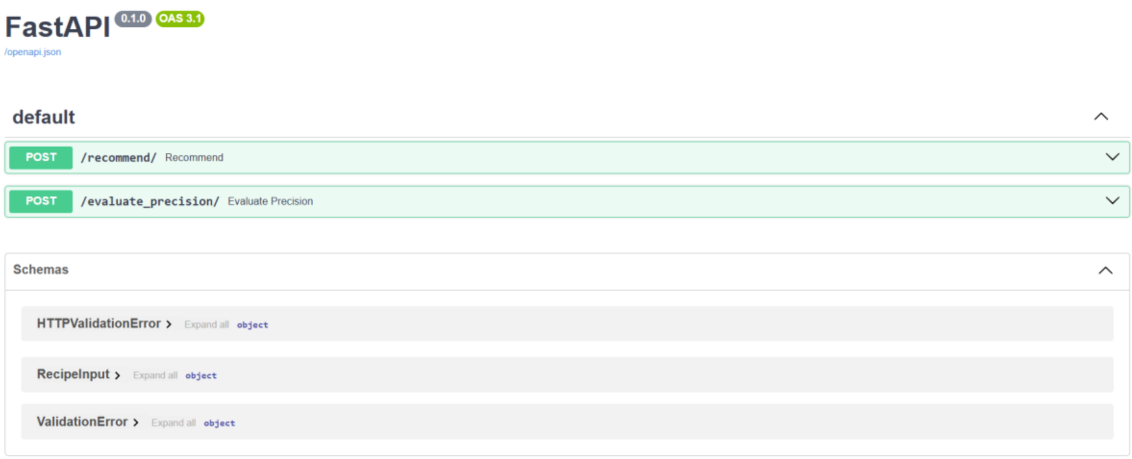
Precision at N: Measures how many of the top-N recommendations match the relevant cuisine.

Most Relevant Cuisine: Identifies the most frequently recommended cuisine among the top-N recommendations.

FastAPI Application

Overview

The FastAPI application serves the trained model to provide real-time recipe recommendations based on user input ingredients.



Application Structure

app.py: Contains the FastAPI application and API endpoints.

Dependencies: pandas, gdown, joblib, fastapi, pydantic, scikit-learn, uvicorn.

API Endpoints

POST /recommend/:

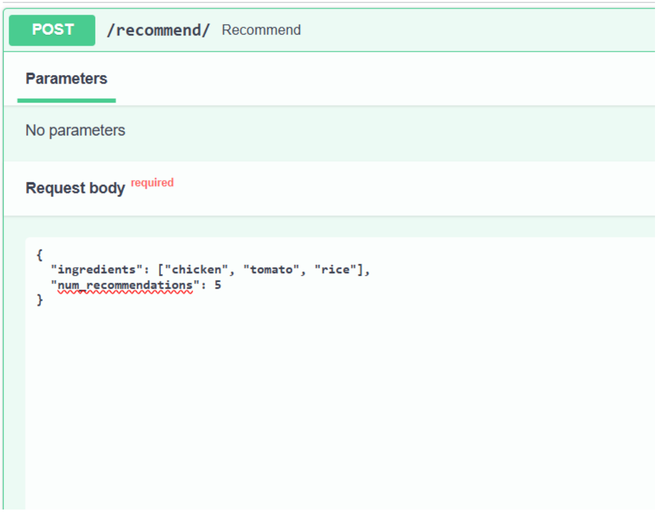
Description: Provides recipe recommendations based on user input ingredients.

Request:

json

Copy code

```
{
 "ingredients": ["chicken", "tomato", "rice"],
 "num_recommendations": 5
}
```



Response: List of recommended recipes with cuisine and ingredients.

POST /evaluate_precision/:

Description: Evaluates the precision of recommendations for a given cuisine.

Request:

json

```
{
  "user_input": ["chicken", "tomato", "rice"],
  "relevant_cuisine": "Italian",
  "n": 5
}
```

Response: Precision at N for the relevant cuisine.

App code –

```
import pandas as pd
import gdown
import joblib
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List
from sklearn.metrics.pairwise import cosine_similarity

# Initialize FastAPI app
app = FastAPI()

# Google Drive file IDs (Make sure these files are accessible to 'Anyone with the link')
vectorizer_file_id = '1rcmhfOhxKvlDBjTmCKRnW1FtJiGAaoHP'
dataset_file_id = '1XbiaBkKHmyX5P4eRaO5LxYcsSqZfGAF5'

# Download URLs for Google Drive files
vectorizer_download_url = f'https://drive.google.com/uc?id={vectorizer_file_id}'
dataset_download_url = f'https://drive.google.com/uc?id={dataset_file_id}'

# Paths to save downloaded files
vectorizer_path = 'tfidf_vectorizer.joblib'
dataset_path = 'recipe_dataset.csv'

# Function to download files with error handling
def download_file(url, output_path):
    try:
        gdown.download(url, output_path, quiet=False)
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to download {output_path}: {str(e)}")

# Download the necessary files
download_file(vectorizer_download_url, vectorizer_path)
download_file(dataset_download_url, dataset_path)

# Load the pre-trained TF-IDF vectorizer model
try:
    vectorizer = joblib.load(vectorizer_path)
except Exception as e:
    raise HTTPException(status_code=500, detail=f"Failed to load vectorizer: {str(e)}")

# Load the dataset
try:
    train_df = pd.read_csv(dataset_path)
```

```
except Exception as e:
    raise HTTPException(status_code=500, detail=f"Failed to load dataset:
{str(e)}")

# Ensure the 'cleaned_ingredients' column exists
if 'cleaned_ingredients' not in train_df.columns:
    raise HTTPException(status_code=500, detail="Column 'cleaned_ingredients'
not found in dataset.")

# Vectorize the ingredients using the loaded vectorizer
ingredient_matrix = vectorizer.transform(train_df['cleaned_ingredients'])

# Define input model for FastAPI
class RecipeInput(BaseModel):
    ingredients: List[str]
    num_recommendations: int = 5

# Recommendation function
def recommend_recipes(user_input, num_recommendations=5):
    user_input_str = ' '.join(user_input)
    user_input_vector = vectorizer.transform([user_input_str])
    similarity_scores = cosine_similarity(user_input_vector,
ingredient_matrix)
    similarity_scores = similarity_scores.flatten()
    top_indices = similarity_scores.argsort()[-num_recommendations:][::-1]
    return train_df.iloc[top_indices]

# API Endpoint for recommending recipes
@app.post("/recommend/")
def recommend(data: RecipeInput):
    try:
        recommendations = recommend_recipes(data.ingredients,
data.num_recommendations)
        return recommendations[['cuisine',
'ingredients']].to_dict(orient='records')
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error during
recommendation: {str(e)}")

# Precision evaluation function
@app.post("/evaluate_precision/")
def evaluate_precision(user_input: List[str], relevant_cuisine: str, n: int =
5):
    try:
        recommendations = recommend_recipes(user_input, num_recommendations=n)
        relevant_recommendations = recommendations[recommendations['cuisine']
== relevant_cuisine]
        precision_at_n = len(relevant_recommendations) / n
```

```python
        return {"precision_at_n": precision_at_n}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error during precision evaluation: {str(e)}")

# Run FastAPI application
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Running the Application Locally

Install Dependencies: Ensure all required libraries are installed using pip install -r requirements.txt.

Run FastAPI App: Start the FastAPI server with uvicorn app:app --reload.

Access Application: Open http://127.0.0.1:8000 in your browser to access the application.

```
Microsoft Windows [Version 10.0.22631.4112]
(c) Microsoft Corporation. All rights reserved.

F:\Horizon lectures\Year 4 semester 1 Lecture notes\Intelligent system\recipe>uvicorn app:app --reload
INFO:     Will watch for changes in these directories: ['F:\\Horizon lectures\\Year 4 semester 1 Lecture notes\\Intellig
ent system\\recipe']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [6732] using StatReload
Downloading...
From: https://drive.google.com/uc?id=1rcmhfOhxKvlDBjTmCKRnW1FtJiGAaoHP
To: F:\Horizon lectures\Year 4 semester 1 Lecture notes\Intelligent system\recipe\tfidf_vectorizer.joblib
100%|██████████████████████████████████████████████████| 2.66M/2.66M [00:01<00:00, 1.90MB/s]
Downloading...
From: https://drive.google.com/uc?id=1XbiaBkKHmyX5P4eRaO5LxYcsSqZfGAF5
To: F:\Horizon lectures\Year 4 semester 1 Lecture notes\Intelligent system\recipe\recipe_dataset.csv
100%|██████████████████████████████████████████████████| 12.5M/12.5M [00:02<00:00, 4.24MB/s]
C:\Users\Acer\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\base.py:348: InconsistentVersionWarning:
 Trying to unpickle estimator TfidfTransformer from version 1.3.2 when using version 1.3.1. This might lead to breaking
code or invalid results. Use at your own risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
C:\Users\Acer\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\base.py:348: InconsistentVersionWarning:
 Trying to unpickle estimator TfidfVectorizer from version 1.3.2 when using version 1.3.1. This might lead to breaking c
ode or invalid results. Use at your own risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
INFO:     Started server process [3000]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Deployment Setup

Railway Deployment - FastAPI - Swagger UI (railway.app)

Preparation:

Ensure the following files are present in the project directory:

- Procfile
- requirements.txt
- app.py

Procfile:

web: uvicorn app:app --host 0.0.0.0 --port $PORT

requirements.txt:

fastapi

uvicorn

scikit-learn

pandas

gdown

joblib

Deployment Steps

- upload locally tested files to git hub.
- connect railway acc to GitHub
- choose repository that we need to deploy
- deploy that application from railway
- open the application

CI/CD Pipeline Documentation

Overview

The CI/CD pipeline automates the testing, building, and deployment process for the FastAPI application. This setup ensures that code changes are automatically tested and deployed to Railway without manual intervention.

CI/CD Pipeline Setup

GitHub Actions Workflow

Purpose: This GitHub Actions workflow file defines the steps for testing, building, and deploying the application to Railway.

Contents-

yaml

```
name: Deploy FastAPI App

on:
  push:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2
        with:
          # Checkout the code from the repository


      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.9'
```

```
    # Set up Python environment version 3.9

  - name: Install dependencies
    run: |
      pip install -r requirements.txt
      # Install the dependencies specified in requirements.txt

  - name: Run tests
    run: |
      # Add commands to run tests if applicable
      echo "No tests found."
      # Placeholder for test execution; customize as needed

  - name: Deploy to Railway
    env:
      RAILWAY_API_KEY: ${{ secrets.RAILWAY_API_KEY }}
    run: |
      railway login --token $RAILWAY_API_KEY
      railway up
      # Log in to Railway CLI and deploy the application
```

Explanation of Steps

Checkout Code: Retrieves the latest code from the GitHub repository.

Set up Python: Sets up the Python environment with the specified version (3.9).

Install Dependencies: Installs the Python packages listed in requirements.txt.

Run Tests: Executes tests to ensure the application works as expected. If no tests are present, a placeholder message is displayed.

Deploy to Railway: Logs in to Railway CLI using the API key stored in GitHub secrets and deploys the application.

2. Environment Variables

RAILWAY_API_KEY: This secret is used to authenticate and deploy the application to Railway. It is stored securely in GitHub secrets.

To Add the Secret:

Navigate to your GitHub repository.

Go to Settings > Secrets and variables > Actions.

Click New repository secret.

Add RAILWAY_API_KEY as the name and your Railway API key as the value.

Deployment Process

Push Changes: When changes are pushed to the master branch, the workflow is triggered.

Checkout Code: The code is checked out from the repository.

Set up Python Environment: The appropriate Python version is set up.

Install Dependencies: Required Python packages are installed.

Run Tests: Any tests are executed (if defined).

Deploy Application: The application is deployed to Railway using the Railway CLI.

Online Testing

After deployment, Railway provides a URL to access the application. You can find the URL in the Railway dashboard under your project details. The URL typically follows the format:

https://web-production-a2847.up.railway.app/docs#

Conclusion

Summary

This documentation covers the model training process, FastAPI application setup, deployment to railway, and CI/CD pipeline configuration for the recipe recommendation system.