Meetlife Tour Itinerary Planner

A Mini-Project Report Submitted in the
Partial Fulfillment of the Requirements
for the Award of the Degree of

BACHELOR OF TECHNOLOGY

IN

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Submitted by

A.Narotham Reddy	**21881A7304**
G.Madhusudhan	**21881A7316**
K.Hyman Reddy	**21881A7330**
Md.Irfan Ahmed	**21881A7335**

SUPERVISOR

Ch.Manish chabbra

Assistant professor

Department of Artificial Intelligence And Machine Learning

VARDHAMAN COLLEGE OF ENGINEERING

(AUTONOMOUS)

Affiliated to JNTUH, Approved by AICTE, Accredited by NAAC with A++ Grade, ISO 9001:2015 Certified

Kacharam, Shamshabad, Hyderabad – 501218, Telangana, India

JULY, 2024

VARDHAMAN COLLEGE OF ENGINEERING
(AUTONOMOUS)
Affiliated to JNTUH, Approved by AICTE, Accredited by NAAC with A++ Grade, ISO 9001:2015 Certified
Kacharam, Shamshabad, Hyderabad – 501218, Telangana, India

Department of Artificial Intelligence And Machine Learning

CERTIFICATE

This is to certify that the project titled **Meetlife Tour Itinerary Planner** is carried out by

A.Narotham Reddy	**21881A7304**
G.Madhusudhan	**21881A7316**
K.Hyman Reddy	**21881A7330**
Md.Irfan Ahmed	**21881A7335**

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING** during the year 2023-24.

Signature of the Supervisor
Ch.Manish chabbra
Assistant professor

Signature of the HOD
Dr. Gagandeep Arora
Professor and Head, AIML

Acknowledgement

The satisfaction that accompanies the successful completion of the task would be put incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crown all the efforts with success.

We wish to express our deep sense of gratitude to **Ch.Manish chabbra**, Assistant professor and Project Supervisor, Department of ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING, Vardhaman College of Engineering, for his able guidance and useful suggestions, which helped us in completing the project in time.

We are particularly thankful to **Dr. Gagandeep Arora**, the Head of the Department, Department of ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING, his guidance, intense support and encouragement, which helped us to mould our project into a successful one.

We show gratitude to our honorable Principal **Dr. J.V.R. Ravindra**, for providing all facilities and support.

We avail this opportunity to express our deep sense of gratitude and heartful thanks to **Dr. Teegala Vijender Reddy**, Chairman and **Sri Teegala Upender Reddy**, Secretary of VCE, for providing a congenial atmosphere to complete this project successfully.

We also thank all the staff members of Electronics and Communication Engineering department for their valuable support and generous advice. Finally thanks to all our friends and family members for their continuous support and enthusiastic help.

A.Narotham Reddy

G.Madhusudhan

K.Hyman Reddy

Md.Irfan Ahmed

Abstract

Abstract—Exploring different places to enjoy the vibe of nature is a dream for everyone. This paper proposes an approach for an effective tour itinerary using K-Means Clustering. Places Dataset was prepared for Machine Learning. Sub-categories for each place were assigned through NLP of Place Description based on the type of place. Subcategories were further merged into a common category. Location Coordinates, Ratings, and votes for each place were retrieved using web scraping. These places were clustered based on the weight of each place according to user-preferred category, city, number of days, and number of places for a day. The itinerary is specified for each day and geospatially presented on a website. A hotel is recommended in the city based on a user's preferences. Tourism is a significant sector contributing to global economies, including a substantial share in GDP, like in India where it contributes 9.2 percentageand is projected to grow annually by 7.8 percentage by 2032. Efficient tour planning involves selecting optimal destinations, accommodations, and daily itineraries tailored to user preferences, which can be complex and time-consuming. This project introduces a Tour Recommendation System leveraging machine learning and data mining techniques to automate and enhance the tour planning process. By integrating web scraping, natural language processing (NLP), and K-Means clustering with Google Maps data, the system provides personalized recommendations based on location, user preferences, and real-time reviews. The system aims to streamline tour planning, improve decision-making for travelers, and contribute to the tourism sector's growth through innovative technology applications.

Keywords:K-Means Clustering, Google Maps API, Categorization, NLP , Flask , Web Scrapping

Table of Contents

Abbreviations

Abbreviation	Description
NLP	Natural Language Processing
API	Application Programming Interface
AI	Artificial Intelligence
CSS	Comma-Separated Values
JS	JavaScript
CSV	Comma Separated Values

CHAPTER 1

Introduction

An agent of an online Platform or Agency often recommends tour Planning. Tour planning involves finding the best places, the types of places he wants to visit, the hotel to stay at, and the itinerary for each day based on user preferences. These things may be complex task for individual tasks and involve an effective understanding of each place. The faster emergence of technology has led to automation in each sector through Machine Learning. The Tourism Sector is ranked eighth for GDP contribution and 9.2This approach proposes a day-wise itinerary recommendation based on user preferences. K-Means Clustering was used to cluster places based on their locations and the categories preferred by the user. It will revolutionalize the already existing manual processes. They tend to provide more information about a place. Throughout this research, we applied different data mining techniques to prepare a dataset for applying Machine Learning algorithms. We used techniques like web scraping and NLP for extracting text and useful information from the web and datasets. This helped in advancing the recommendations of the proposed system. With this proposed system we aim to provide user-friendly recommendations based on ratings of places based on Google Maps Platform.

CHAPTER 2

Literature Survey

2.1 A Collaborative Location Based Travel Recommendation System through Enhanced Rating Prediction for the Group of Users

[1] Recommender systems recommend based on user interests over various domains. However these recommender systems face numerous challenges to slow down their effectiveness. Recent researches show that incorporating social networks data may increase the accuracy and prediction capabilities of these recommender systems. This paper analysis social network data-based recommender systems and also covering different recommendation algorithms, system functionalities, interfaces, filtering techniques, and AI methods. And also it analyze existing models, objectives, methodologies, and data sources, offering insights for mastering travel recommendation systems. In this it introduces a location recommendation system based on Social Pertinent Trust Walker (SPTW) model, it compares with the existing random walk models, and it will provides enhancements for group user recommendations along with experimental results.

2.2 Long-Term Preference Mining With Temporal and Spatial Fusion for Point-of-Interest Recommendation

[2] To provide a customized travel suggestion based on user preference and specific needs it requires to develop a travel recommendation system that uses social media activity like twitter data, as well as friends and family

data of recent travel history to understand their interest. A machine learning classifier classifies the tweets that are relevant to travel. The travel relevant tweets are used to obtain personalized travel recommendations. Instead of personalized travel recommendations, this model considers user's most recent interest by incorporating time-sensitive recency weight into the model. This model outperformed with the existing personalized recommendation model, with accuracy of 75.23

2.3 Long-Term Preference Mining With Temporal and Spatial Fusion for Point-of-Interest Recommendation

[3] To enhance the experience of journey, transportation, weather, accommodation, travel-time we look for help in planning a trip. Today, the total information is present on internet but searching for a suitable package or service may be more time consuming. A recommender system can help to find in various trip related queries. In this report, they provide the review on various factors such as hotels, restaurants, attractions, tourism package and planning and also they assist recommendations on requirements such as transportation, food, outfits. They had divide the travel based recommendation system and also introduced selection criteria, features, and technical aspects with datasets, methods, and result.

2.4 Personalized tourist route recommendation model with a trajectory understanding via neural networks

[4] The excess use of digital cameras and sharing of photos using social media platforms such as Flickr have resulted in large volume of geotagged photos available on the interent. By using these geotagged photos, they proposed a method for recommending tourists places to users' preference. By

using users' travel history in one city they can recommend tourist places in another city. This is developed by using publicly available Flickr dataset featuring photos from various cities. This context-aware personalized approach predicts more accurately tourists places in unfamiliar places and provide major preferences compared to other recommendation methods.

2.5 Thematic Travel Recommendation System Using an Augmented Big Data Analytical Model

[5] Trip planning become complex and time consuming due to large amount of available information on internet. And also for individual travellers unique interests and preference. To exaggerate trip planning, thematic travel planning come out using text-based data mining to give more personalized tourism services. They proposed an augmented model that integrates analytics of various big data sources—static and dynamic—including destination images, tourist activity reviews, weather forecasts, and social media events, to offer user-centric and location-based thematic recommendations. The implementation of this recommendation model leverages multimodal ranking of user preferences. The effectiveness of this model is evaluated by using various statistical and machine learning techniques, including Spark models, naïve Bayes classifier, trigonometric functions, deep learning convolutional neural networks (CNN), time series analysis, and natural language processing (NLP) with sentiment analysis using AFINN. This hybrid recommendation learns user preferences and ranks them, using explicit information to give personalized, augmented recommendations.

2.6 JESSY: An Intelligence Travel Assistant

[6] Classical recommender system provides recommendations with a list of recommendations with only single item. It describes the requirement of a recommender system capable of suggesting packages of items, useful in contexts like travel planning where recommendations involve numerous components.

Travel planning requires composite recommendations by specific time and budget. The CompRec-Trip, addresses this need by generating top-k package recommendations that uses users' preferences. It also has graphical user interface enabling users to customize the recommendations and mix external local information into their planning. It uses rating information from already existed recommender systems and allows flexible package configuration.

CHAPTER 3

Existing system and its disadvantages

3.1 Online Travel Booking Platforms

Travelers use online platforms like Expedia, Booking.com, TripAdvisor, and Airbnb to book flights, accommodations, and activities. These platforms aggregate options from various providers and offer user reviews and ratings to assist in decision-making.

3.2 Manual Travel Planning by Individual Travelers

In this approach, travelers independently research and plan their trips using various online resources such as travel blogs, review sites, maps, and social media. They manually compile information about destinations, accommodations, and activities, and create itineraries based on their findings.

3.3 Travel Agency Services

Travelers hire professional travel agents to plan their trips. Travel agents use their expertise and industry connections to create customized travel itineraries and handle bookings on behalf of their clients.

3.4 Disadvantages

- Time-Consuming: Travelers need to spend a significant amount of time researching destinations, accommodations, and activities, often resulting in hours or days of planning.
- Inconsistent Information: Information gathered from different sources

can be inconsistent or outdated, leading to potential inaccuracies and confusion.

- Overwhelming Choices: The vast number of options available for destinations, hotels, and activities can overwhelm travelers, making it difficult to make informed decisions.

- Fragmented Information: Online platforms often provide separate services for flights, hotels, and activities, requiring travelers to manually piece together their itinerary.

- Lack of Real-Time Updates: While online platforms provide booking capabilities, they may not offer real-time updates on weather, traffic, or availability, leading to potential disruptions.

- Limited Integration: These platforms may not integrate user preferences and reviews comprehensively, resulting in less personalized recommendations.

CHAPTER 4

Proposed system and its advantages

4.1 Description

The proposed system leverages machine learning and data mining techniques to automate and enhance the tour planning process. It uses a dataset compiled from reliable sources such as Google Maps Platform reviews to provide personalized, optimized itineraries based on user preferences. The system integrates various features to offer a comprehensive and user-friendly travel planning experience, including real-time updates, clustering algorithms for place categorization, and interactive maps for visualization.

4.2 Advantages

- Time Efficiency: The automated system significantly reduces the time required for trip planning by quickly processing large amounts of data and generating recommendations within minutes.
- Personalization: The system provides personalized recommendations based on user preferences, such as desired activities, budget, and travel dates. This ensures a tailored travel experience for each user.
- Real-Time Updates: The system can integrate real-time updates on weather, traffic, and availability, allowing travelers to adjust their plans accordingly and avoid disruptions.
- Accurate and Consistent Information: By leveraging data from reliable sources like Google Maps, the system ensures that travelers receive accurate and up-to-date information.
- Simplified Decision-Making: The system narrows down options based on user preferences, making decision-making easier and less overwhelming.

- Cost-Effective: The automated system reduces the need for expensive travel agent services by providing comprehensive planning tools directly to the user.
- Comprehensive Reviews: The system aggregates reviews from multiple sources, providing travelers with a holistic view of their options based on user feedback.
- Efficient Itinerary Planning: The system automates the creation of day-wise itineraries, optimizing schedules and ensuring efficient use of time.
- Enhanced Group Coordination: The system can accommodate preferences from multiple users, making it easier to plan trips for groups and ensuring everyone's interests are considered.
- Sustainable Travel Options: The system can suggest eco-friendly travel options, promoting sustainable tourism practices.

CHAPTER 5

System Requirements

5.1 Software Requirements

- Operating System: Windows 10, macOS, or Linux, depending on user preference and compatibility with other software.
- Python: Programming language used for implementing machine learning models and data processing.
- Anaconda: Distribution of Python and R for scientific computing, including packages like NumPy, pandas, and scikit-learn.
- Jupyter Notebook: An interactive computing environment for developing and testing code.
- Selenium: Tool for web scraping to extract data from online sources.
- BeautifulSoup: Library for parsing HTML and XML documents for web scraping.
- Leaflet.js: JavaScript library for interactive maps, used for visualizing itineraries.
- Flask/Django: Web frameworks for building the backend of the application.
- Bootstrap: Frontend framework for designing responsive web pages.
- Google Maps API: For accessing detailed information about places and integrating maps into the application.

5.2 Hardware Requirements

- Computer/Server: A robust computer or server to handle data processing and machine learning tasks.
- High-Speed Internet Connection: Essential for web scraping, real-time updates, and accessing online resources.
- Storage: Sufficient storage capacity for large datasets and user data.
- Processor: A multi-core processor (Intel i5 or higher) to ensure efficient data processing.
- RAM: At least 8GB of RAM to handle multiple processes and large datasets simultaneously.
- Graphics Card: A dedicated graphics card (NVIDIA GTX series or equivalent) for machine learning model training.
- Backup Storage: External storage devices for data backup and recovery.
- Display: High-resolution monitor for better visualization of data and maps.
- Peripheral Devices: Keyboard, mouse, and other necessary input devices.
- Power Supply: Uninterruptible power supply (UPS) to prevent data loss during power outages.

CHAPTER 6

System Design

6.1 Architecture

The architecture of the proposed system is designed to integrate various components for efficient data processing, user interaction, and recommendation generation. The system follows a modular approach to ensure scalability, maintainability, and ease of development. Here is an overview of the architecture:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam consectetur libero dui, sit amet rutrum lectus mollis ac. Phasellus mattis augue quis auctor ullamcorper. Sed congue rutrum turpis, sit amet tincidunt erat laoreet ac. Morbi in feugiat erat, sit amet placerat lorem. Quisque cursus gravida nulla, nec pulvinar justo rutrum eget. Aenean et dolor vitae enim congue maximus. Praesent consequat finibus imperdiet. Sed ipsum erat, efficitur vitae urna a, convallis ornare ipsum. Quisque fringilla risus enim, ut elementum dui consectetur eu.

6.1.1 Data Layer

- Data Sources: Google Maps reviews, user input, and other relevant travel databases.
- Data Collection: Web scraping scripts, API calls to Google Maps, and user input forms.
- Data Storage: A relational database (e.g., MySQL) to store structured data, such as places, categories, ratings, and user preferences.

6.1.2 Processing Layer

- Data Preprocessing: Scripts for cleaning, normalizing, and transforming raw data into a usable format.

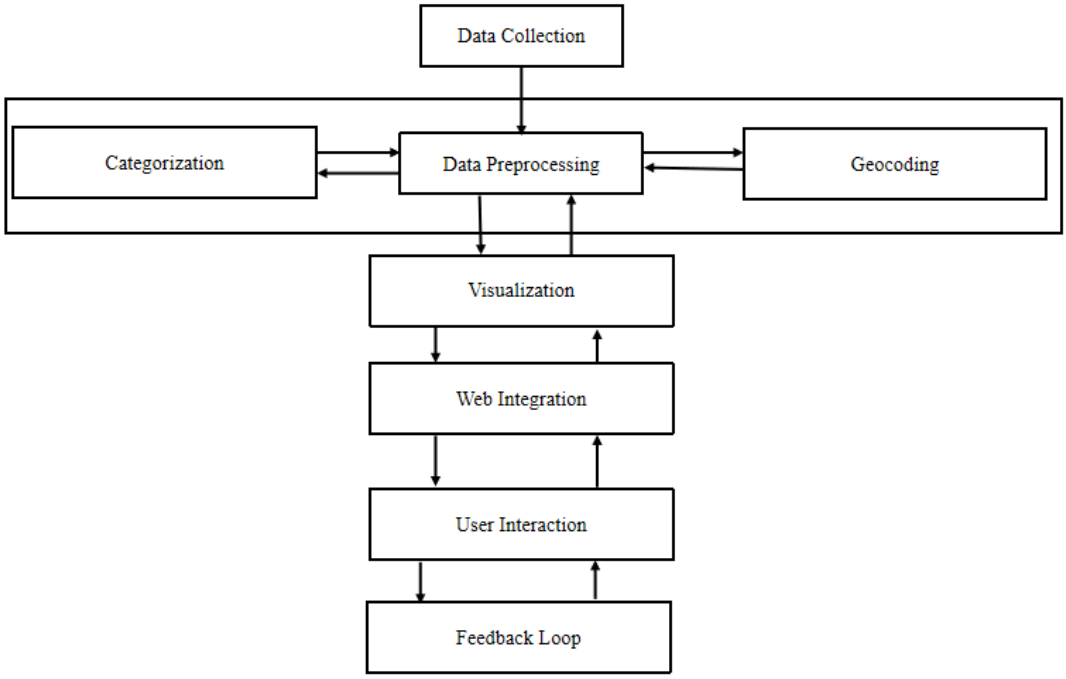

Figure 6.1: Architecture

- Machine Learning Models: Algorithms for clustering, rating calculation, and recommendation generation.
- Real-Time Data Integration: APIs to fetch real-time data (e.g., weather, traffic conditions).

6.1.3 Application Layer

- Backend Server: A web server (e.g., Flask or Django) to handle requests, run algorithms, and communicate with the database.
- Business Logic: Core logic for itinerary generation, user preference matching, and clustering .
- Frontend Interface: A web application (using HTML, CSS, JavaScript) for user interaction and visualization.

6.1.4 User Interface

- Interactive Forms: For inputting travel preferences and displaying recommendations.

- Maps Integration: Leaflet.js for interactive maps showing recommended places and itineraries.
- Visualization: Dynamic charts and graphs to visualize travel plans and clusters.

6.2 Working Model

The proposed system starts with users entering their travel preferences, including the city they plan to visit, the types of places they are interested in, the number of places they wish to visit per day, and the duration of their trip. This input is collected through a user-friendly web interface. The backend processes this data, using web scraping techniques and APIs to gather detailed information about potential destinations. This data, including place names, coordinates, ratings, reviews, and categories, is stored in a structured database. Using the collected data, the system employs the K-Means clustering algorithm to group similar places based on their geographic locations and categories. This clustering helps in organizing the itinerary to minimize travel distance and time. The system then calculates a weighted score for each place based on ratings and the number of reviews, normalizing these scores for consistency. The frontend dynamically updates to display available categories for the selected city and allows users to refine their preferences. The system generates an optimal itinerary, recommending the best places to visit each day, ensuring diversity by selecting from different clusters. The final itinerary is displayed on an interactive map, providing a visual representation of the route. Users can view detailed information about each place, including its ratings, categories, and descriptions, making the trip planning process efficient and tailored to their preferences.

6.2.1 User Input

- Users enter their travel preferences, including the city, categories of interest, number of places per day, and duration of the trip.
- The frontend captures this input through interactive forms.

6.2.2 Data Collection

- The system gathers data from Google Maps using web scraping and API calls.
- Reviews, ratings, coordinates, and other relevant information are collected and stored in the database.

6.2.3 Data Preprocessing

- Collected data is cleaned, normalized, and structured for use in machine learning models.
- Missing values are handled, and weights are calculated based on ratings and votes.

6.2.4 Recommendation Generation

- The K-Means clustering algorithm is applied to group places based on their geographical coordinates and user-specified categories.
- The system calculates scores for each place using a weighted average of ratings and votes, normalized to a 2.5 to 3.5 scale.

6.2.5 Itinerary Planning

- The system generates day-wise itineraries by selecting top-rated places from each cluster.
- It ensures a balanced itinerary with diverse activities each day.

6.2.6 Real-Time Data Integration

- The system fetches real-time data such as weather updates and traffic conditions to refine the itinerary.

6.2.7 User Interaction and Visualization

- The frontend displays the recommended itinerary on an interactive map using Leaflet.js.
- Users can view details of each place, including ratings, reviews, and descriptions.

6.2.8 Feedback and Adjustment

- Users can provide feedback on the recommendations, and the system can adjust future itineraries accordingly.
- Real-time adjustments can be made based on changing conditions or user preferences.

6.3 Algorithms Used

6.3.1 Web Scraping

- BeautifulSoup and Selenium: Used for extracting data from Google Maps and other sources.

6.3.2 Natural Language Processing (NLP)

- Text Cleaning and Tokenization: Used to preprocess and analyze reviews and descriptions.

6.3.3 Clustering

- K-Means Clustering: Groups places based on geographical coordinates and user-specified categories to ensure convenient travel plans.

6.3.4 Weight Calculation

- Weighted Average: Calculates the weight of each place using ratings and votes, normalized to a 2.5 to 3.5 scale for consistency.

6.3.5 Itinerary Optimization

- Heuristic Algorithms: Used to optimize daily travel plans, ensuring a balanced and efficient itinerary.

6.3.6 Real-Time Data Integration

- API Calls: Fetches real-time data such as weather and traffic to adjust plans dynamically.

CHAPTER 7

Implementation

7.1 module description

7.1.1 Data Collection and Preprocessing Module

This module utilizes web scraping techniques and APIs to gather comprehensive data about tourist destinations, including their names, coordinates, ratings, reviews, and categories. The collected data undergoes preprocessing to clean and structure it for further analysis.

7.1.2 Recommendation Engine Module

The heart of the system, this module employs machine learning techniques such as K-Means clustering to group similar destinations based on geographical proximity and user-defined categories. It calculates a weighted score for each destination, combining ratings and reviews, to prioritize the most appealing places.

7.1.3 User Interface Module

The frontend interface enables users to input their travel preferences interactively. It dynamically updates to display available categories for the selected city and allows users to refine their choices. Users can view recommended itineraries, explore detailed information about each destination, and interact with an interactive map displaying the proposed route.

7.1.4 Database Management Module

Responsible for storing and managing the structured data collected during the data collection phase. It ensures efficient retrieval and storage of destination details, supporting real-time updates and queries to enhance user experience.

7.2 Technologies used

7.2.1 Backend

Python (Flask), pandas, scikit-learn, Selenium, BeautifulSoup.

7.2.2 Frontend

HTML, CSS, JavaScript (AJAX, Leaflet.js), Axios for HTTP requests.

7.2.3 Database

Utilizes CSV files for storing and processing data related to travel destinations and user preferences.

7.3 sample code

```
import pandas as pd
from geopy.distance import geodesic

# Load the dataset containing restaurant locations
restaurant_data = pd.read_csv('zomatotest.csv', encoding='latin-1')

# Drop rows with missing or invalid latitude and longitude values
restaurant_data = restaurant_data.dropna(subset=['latitude', 'longitude'])
restaurant_data = restaurant_data[(restaurant_data['latitude'] >= -90)
& (restaurant_data['latitude'] <= 90)]
restaurant_data = restaurant_data[(restaurant_data['longitude'] >= -180)
& (restaurant_data['longitude'] <= 180)]

# Ask for the user's location
user_latitude = float(input("Enter your latitude: "))
user_longitude = float(input("Enter your longitude: "))
```

```
user_location = (user_latitude, user_longitude)

# Ask for user preferences
preferred_cuisine = input("Enter your preferred cuisine: ")
max_cost = float(input("Enter your maximum budget (cost): "))
min_rating = float(input("Enter your minimum rating: "))

# Function to calculate distance between two coordinates
def calculate_distance(restaurant_location):
    return geodesic(user_location, restaurant_location).kilometers

# Calculate distance for each restaurant
restaurant_data['Distance'] = restaurant_data.apply
(lambda row: calculate_distance((row['latitude'], row['longitude'])), axis=1)

# Filter restaurants within 10 km radius
restaurants_within_radius = restaurant_data[restaurant_data
['Distance'] <= 50]
restaurants_within_radius1 = restaurant_data[restaurant_data
['Distance'] <= 10]

# Filter restaurants based on user preferences
filtered_restaurants = restaurants_within_radius1[
    (restaurants_within_radius1['cuisines'].str.contains
    (preferred_cuisine, case=False)) &
   (restaurants_within_radius1['average_cost_for_two'] <= max_cost) &
    (restaurants_within_radius1['aggregate_rating'] >= min_rating)
]

# Sort filtered restaurants based on rating and cost
sorted_restaurants = filtered_restaurants.sort_values(by=
['aggregate_rating', 'average_cost_for_two'], ascending=[False, True])
```

```
if sorted_restaurants.empty:
    print("No restaurants match your criteria.
    Here are the top 10 nearest restaurants:")

    # Sort restaurants by distance
    nearest_restaurants = restaurants_within_radius.sort_values
    (by='Distance').head(10)

    for index, restaurant in nearest_restaurants.iterrows():
        print(f"{restaurant['name']} - Cuisine:
        {restaurant['cuisines']}, Cost:
        {restaurant['average_cost_for_two']}, Rating:
        {restaurant['aggregate_rating']}, Distance:
        {restaurant['Distance']} km")
else:
    # Get the top suggestion
    top_suggestion = sorted_restaurants.iloc[0]  #
    Assuming the first restaurant is the best suggestion

    # Print the details of the top suggestion
    print("Top Suggestion:")
    print("Name:", top_suggestion['name'])
    print("Cuisine:", top_suggestion['cuisines'])
    print("Cost:", top_suggestion['average_cost_for_two'])
    print("Rating:", top_suggestion['aggregate_rating'])
    print("Distance:", top_suggestion['Distance'], "km")
import pandas as pd
from nltk.corpus import wordnet
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import MultiLabelBinarizer
```

```
# Load your dataset
data = pd.read_csv('Places.csv')

# Preprocess the data
data.dropna(subset=['Place', 'Place_desc'], inplace=True)
# Remove rows with missing data
data['Place'] = data['Place'].str.lower()
# Convert place names to lowercase
data['Place_desc'] = data['Place_desc'].str.lower()
# Convert place descriptions to lowercase

# Define the list of keywords
keywords = [
    "adventure",
    "scenic",
    "wildlife",
    "cultural",
    "religious",
    "urban",
    "rural",
    "coastal",
    "historic",
    "archaeological",
    "natural",
    "ecological",
    "botanical",
    "geological",
    "artistic",
    "architectural",
    "food",
    "shopping",
```

```
"entertainment",
"nightlife",
"hiking",
"camping",
"trekking",
"safari",
"water sports",
"skiing",
"cycling",
"birdwatching",
"botanical gardens",
"national parks",
"museums",
"monuments",
"temples",
"mosques",
"churches",
"palaces",
"forts",
"ruins",
"lakes",
"rivers",'Mountains','Mountain roads, Tunnels' 'Hill stations',
'Tea gardens', 'Spice plantations', 'Beaches',
'Backwaters', 'Waterfalls', 'Deserts', 'Plains',
'Islands', 'Jungles', 'Sanctuaries', 'Reserves',
'Festivals', 'Fairs', 'Carnivals', 'Dance forms',
'Music', 'Cuisine', 'Handicrafts', 'Textiles',
'Pottery', 'Sculptures', 'Paintings', 'Folklore',
'Legends', 'Mythology', 'Traditions', 'Customs',
'Rituals', 'Yoga', 'Meditation', 'Ayurveda',
'Wellness', 'Spiritual', 'Religious ceremonies',
'Pilgrimages', 'Ashrams', 'Gurudwaras', 'Synagogues',
```

'Citadel', 'Bastions', 'Ramparts', 'Gateways', 'Courtyards', 'Havelis', 'Mansions', 'Bungalows', 'Villas', 'Residences', 'Homestays', 'Guesthouses', 'Inns', 'Lodges', 'Accommodations','Scenic drives', 'Road trips', 'Motorcycling', 'Rock climbing', 'Mountaineering', 'Paragliding', 'Mountain biking', 'Zip-lining', 'Cable car rides', 'Sightseeing', 'Photography', 'Stargazing', 'Picnicking', 'Exploring', 'Nature walks', 'Wildlife spotting', 'Fishing', 'Whitewater rafting', 'Kayaking', 'Canoeing', 'Off-roading', 'Adventure tours', 'Horseback riding', 'Hot air ballooning', 'Helicopter tours', 'Sledging', 'Ice climbing', 'Snowshoeing', 'Snowboarding', 'Cross-country skiing', 'Glacier walking', 'Mushing', 'Dog sledding', 'Heli-skiing', 'Snowmobiling', 'Avalanche safety training', 'Ski jumping', 'Snow tubing', 'Ice skating', 'Ice fishing', 'Ice diving', 'Ice climbing', 'Ice sailing', 'Ice golfing', 'Ice kayaking', 'Ice swimming', 'Ice boating', 'Snow sculpture festivals', 'Winter carnivals', 'Christmas markets', 'Fireworks displays', 'Winter sports competitions', 'Après-ski parties', 'Snowman building contests', 'Winter picnics', 'Winter solstice celebrations', 'Ice hotels', 'Winter festivals''Scenic drives', 'Road trips', 'Motorcycling', 'Rock climbing', 'Mountaineering', 'Paragliding', 'Mountain biking', 'Zip-lining', 'Cable car rides', 'Sightseeing', 'Photography', 'Stargazing', 'Picnicking', 'Exploring', 'Nature walks', 'Wildlife spotting', 'Fishing', 'Whitewater rafting', 'Kayaking', 'Canoeing', 'Off-roading', 'Adventure tours', 'Horseback riding', 'Hot air

```
    ballooning', 'Helicopter tours', 'Sledging', 'Ice
    climbing', 'Snowshoeing', 'Snowboarding', 'Cross-
    country skiing', 'Glacier walking', 'Mushing', 'Dog
    sledding', 'Heli-skiing', 'Snowmobiling', 'Avalanche
    safety training', 'Ski jumping', 'Snow tubing', 'Ice
    skating', 'Ice fishing', 'Ice diving', 'Ice climbing',
    'Ice sailing', 'Ice golfing', 'Ice kayaking', 'Ice
    swimming', 'Ice boating', 'Snow sculpture festivals',
    'Winter carnivals', 'Christmas markets', 'Fireworks
    displays', 'Winter sports competitions', 'Après-ski
    parties', 'Snowman building contests', 'Winter
    picnics', 'Winter solstice celebrations', 'Ice
    hotels', 'Winter festivals'
]

# Function to find synonyms of a word
def get_synonyms(word):
    synonyms = set()
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonyms.add(lemma.name().lower())
    return synonyms

# Generate a set of all synonyms of the keywords
all_keywords = set()
for keyword in keywords:
    all_keywords.add(keyword)
    all_keywords |= get_synonyms(keyword)

# Initialize a list to store predicted types for each row
predicted_types = []
```

```
# Prioritize classification based on keywords in place names
for index, row in data.iterrows():
    row_predicted_types = []
    for keyword in all_keywords:
        if keyword in row['Place'] or keyword in row['Place_desc']:
            row_predicted_types.append(keyword)
    predicted_types.append(row_predicted_types)

# Convert list of lists to a single list of unique types
all_predicted_types = set([item for sublist in
predicted_types for item in sublist])

# Add a new column for the predicted types in the dataset
data['Predicted_Types'] = [', '.join(types) for types in predicted_types]

# Save the updated dataset
data.to_csv('New_Places.csv', index=False)
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Download NLTK resources
nltk.download('stopwords')
nltk.download('wordnet')

# Load the dataset
df = pd.read_csv('testcat.csv', encoding = 'latin-1')
```

```
# Define categories and their corresponding keywords or descriptions
categories = {
    "Adventure": "Hiking Camping Trekking Safari Water
    sports Skiing Cycling Birdwatching Botanical gardens
    National parks Scenic drives Road trips Motorcycling
    Rock climbing Mountaineering Paragliding Mountain
    biking Zip-lining Cable car rides Sightseeing
    Photography Stargazing Picnicking Exploring Nature
    walks Wildlife spotting Fishing Whitewater rafting
    Kayaking Canoeing Off-roading Adventure tours
    Horseback riding Hot air ballooning Helicopter tours
    Sledging Ice climbing Snowshoeing Snowboarding Cross-
    country skiing Glacier walking Mushing Dog sledding
    Heli-skiing Snowmobiling Avalanche safety training Ski
    jumping Snow tubing Ice skating Ice fishing Ice diving
    Ice sailing Ice golfing Ice kayaking Ice swimming Ice
    boating Snow sculpture festivals Winter carnivals
    Christmas markets Fireworks displays Winter sports
    competitions Après-ski parties Snowman building
    contests Winter picnics Winter solstice celebrations
    Ice hotels Winter festivals",

    "Scenic": "Scenic drives Road trips Sightseeing
    Photography Stargazing Picnicking Exploring Nature
    walks Wildlife spotting Lakes Rivers Mountains
    Mountain roads Tunnels Hill stations Tea gardens
    Spice plantations Beaches Backwaters Waterfalls
    Deserts Plains Islands Jungles Sanctuaries Reserves",

    "Cultural": "cultural historical archaeological
    artistic religion temples mosques churches palaces
```

```
forts ruins museums monuments folklore legends
mythology traditions customs rituals festivals fairs
carnivals dance forms music",

"Food": "food cuisine",

"Shopping": "shopping",

"Entertainment": "entertainment nightlife",

"Water Sports": "water sports fishing whitewater
rafting kayaking canoeing",

"Wellness": "yoga meditation ayurveda wellness
spiritual religious ceremonies pilgrimages ashrams
gurudwaras synagogues",

"Architecture": "architectural gateways courtyards
havelis mansions bungalows villas residences
homestays guesthouses inns lodges accommodations
citadel bastions ramparts",

"Beaches": "beaches coastal islands",

"Deserts": "deserts",

"Forests": "forests jungles sanctuaries reserves",

"Historic Sites": "historical sites archaeological
sites monuments ruins",

"Art and Culture": "art culture museums galleries
```

```
performing arts",

"Religious Sites": "temples mosques churches shrines",

"Adventure Sports": "adventure sports extreme sports
water sports snow sports",

"Family-Friendly": "family-friendly kids-friendly
amusement parks zoo aquarium",

"Offbeat": "offbeat lesser-known hidden gems
unexplored",

"Spiritual Journeys": "spiritual religious ceremonies
pilgrimages ashrams gurudwaras yoga meditation
ayurveda",

"Festivals and Events": "festivals fairs carnivals",

"Historical and Architectural Marvels": "historical
archaeological architectural temples mosques churches
palaces forts ruins",
"Nature and Wildlife": "natural ecological botanical
geological national parks lakes rivers backwaters
waterfalls deserts plains islands jungles sanctuaries
reserves",

"Adventure and Trekking": "adventure trekking camping
safari hiking mountaineering rock climbing
paragliding zip-lining whitewater rafting kayaking
canoeing off-roading horseback riding hot air
ballooning helicopter tours sledging ice climbing
```

```
    snowboarding cross-country skiing glacier walking
    mushing dog sledding heli-skiing snowmobiling ski
    jumping snow tubing ice skating ice diving ice
    sailing ice golfing ice kayaking ice swimming ice
    boating snow sculpture festivals winter carnivals
    fireworks displays winter sports competitions après-
    ski parties snowman building contests winter picnics
    winter solstice celebrations ice hotels winter
    festivals",

    "Cultural Immersion": "cultural artistic folklore
    legends mythology traditions customs rituals dance
    forms music",

    "Culinary Experiences": "food cuisine",

    "Luxury and Wellness Retreats": "wellness spa health
    retreats",

    "Offbeat and Eco-Tourism": "offbeat lesser-known
    hidden gems unexplored",

    "Beach Holidays": "beaches coastal islands"
}


# Text preprocessing
def preprocess_text(text):
    if pd.isna(text):
        return ""
    text = text.lower()  # Lowercase text
    text = re.sub(r'\d+', '', text)  # Remove numbers
```

```
    text = re.sub(r'\s+', ' ', text)  # Remove extra spaces
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    text = text.strip()  # Remove leading and trailing spaces
    return text

# Apply preprocessing to categories
categories = {k: preprocess_text(v) for k, v in categories.items()}

# Extract and preprocess predicted types
df['Predicted_Types'] = df['Predicted_Types'].apply(preprocess_text)

# Convert categories to list for vectorization
category_names = list(categories.keys())
category_descriptions = list(categories.values())

# Vectorize the categories
vectorizer = TfidfVectorizer(stop_words=stopwords.words('english'))
category_vectors = vectorizer.fit_transform(category_descriptions)

# Classify the predicted types into top N categories
N = 3  # Number of top categories to select

def classify_predicted_types(predicted_type):
    if not predicted_type:
        return ["Other"]
    predicted_vector = vectorizer.transform([predicted_type])
    cosine_similarities = cosine_similarity(predicted_vector,
    category_vectors).flatten()
    top_indices = cosine_similarities.argsort()[-N:][::-1]
    top_categories = [category_names[i] for i in top_indices]
    return top_categories
```

```
df['Categories'] = df['Predicted_Types'].apply(classify_predicted_types)

# Convert list of categories to string for easier CSV storage
df['Categories'] = df['Categories'].apply(lambda x: ', '.join(x))

# Save the updated dataset
df.to_csv('Updatedf_Places.csv', index=False, encoding = 'latin-1')

print("Classification completed and saved to 'Updated_Places.csv'")
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Download NLTK resources
nltk.download('stopwords')
nltk.download('wordnet')

# Load the dataset
df = pd.read_csv('testcat.csv', encoding = 'latin-1')

# Define categories and their corresponding keywords or descriptions
categories = {
    "Adventure": "Hiking Camping Trekking Safari Water
    sports Skiing Cycling Birdwatching Botanical gardens
    National parks Scenic drives Road trips Motorcycling
    Rock climbing Mountaineering Paragliding Mountain
    biking Zip-lining Cable car rides Sightseeing
```

```
Photography Stargazing Picnicking Exploring Nature
walks Wildlife spotting Fishing Whitewater rafting
Kayaking Canoeing Off-roading Adventure tours
Horseback riding Hot air ballooning Helicopter tours
Sledging Ice climbing Snowshoeing Snowboarding Cross-
country skiing Glacier walking Mushing Dog sledding
Heli-skiing Snowmobiling Avalanche safety training
Ski jumping Snow tubing Ice skating Ice fishing Ice
diving Ice sailing Ice golfing Ice kayaking Ice
swimming Ice boating Snow sculpture festivals Winter
carnivals Christmas markets Fireworks displays Winter
sports competitions Après-ski parties Snowman
building contests Winter picnics Winter solstice
celebrations Ice hotels Winter festivals",

"Scenic": "Scenic drives Road trips Sightseeing
Photography Stargazing Picnicking Exploring Nature
walks Wildlife spotting Lakes Rivers Mountains
Mountain roads Tunnels Hill stations Tea gardens
Spice plantations Beaches Backwaters Waterfalls
Deserts Plains Islands Jungles Sanctuaries Reserves",

"Cultural": "cultural historical archaeological
artistic religion temples mosques churches palaces
forts ruins museums monuments folklore legends
mythology traditions customs rituals festivals fairs
carnivals dance forms music",

"Food": "food cuisine",

"Shopping": "shopping",
```

```
    "Entertainment": "entertainment nightlife",

    "Water Sports": "water sports fishing whitewater
    rafting kayaking canoeing",

    "Wellness": "yoga meditation ayurveda wellness
    spiritual religious ceremonies pilgrimages ashrams
    gurudwaras synagogues",

    "Architecture": "architectural gateways courtyards
    havelis mansions bungalows villas residences
    homestays guesthouses inns lodges accommodations
    citadel bastions ramparts",

    "Beaches": "beaches coastal islands",

    "Deserts": "deserts",

    "Forests": "forests jungles sanctuaries reserves",

    "Historic Sites": "historical sites archaeological
    sites monuments ruins",

    "Art and Culture": "art culture museums galleries
    performing arts",

    "Religious Sites": "temples mosques churches shrines",

  "Adventure Sports": "adventure sports extreme sports water sports snow sports",

    "Family-Friendly": "family-friendly kids-friendly
    amusement parks zoo aquarium",
```

```
"Offbeat": "offbeat lesser-known hidden gems
unexplored",

"Spiritual Journeys": "spiritual religious ceremonies
pilgrimages ashrams gurudwaras yoga meditation
ayurveda",

"Festivals and Events": "festivals fairs carnivals",
"Historical and Architectural Marvels": "historical
archaeological architectural temples mosques churches
palaces forts ruins",

"Nature and Wildlife": "natural ecological botanical
geological national parks lakes rivers backwaters
waterfalls deserts plains islands jungles sanctuaries
reserves",

"Adventure and Trekking": "adventure trekking camping
safari hiking mountaineering rock climbing
paragliding zip-lining whitewater rafting kayaking
canoeing off-roading horseback riding hot air
ballooning helicopter tours sledging ice climbing
snowboarding cross-country skiing glacier walking
mushing dog sledding heli-skiing snowmobiling ski
jumping snow tubing ice skating ice diving ice
sailing ice golfing ice kayaking ice swimming ice
boating snow sculpture festivals winter carnivals
fireworks displays winter sports competitions après-
ski parties snowman building contests winter picnics
winter solstice celebrations ice hotels winter
festivals",
```

```
    "Cultural Immersion": "cultural artistic folklore
    legends mythology traditions customs rituals dance
    forms music",

    "Culinary Experiences": "food cuisine",

    "Luxury and Wellness Retreats": "wellness spa health
    retreats",

    "Offbeat and Eco-Tourism": "offbeat lesser-known
    hidden gems unexplored",

    "Beach Holidays": "beaches coastal islands"
}


# Text preprocessing
def preprocess_text(text):
    if pd.isna(text):
        return ""
    text = text.lower()  # Lowercase text
    text = re.sub(r'\d+', '', text)  # Remove numbers
    text = re.sub(r'\s+', ' ', text)  # Remove extra spaces
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    text = text.strip()  # Remove leading and trailing spaces
    return text

# Apply preprocessing to categories
categories = {k: preprocess_text(v) for k, v in categories.items()}

# Extract and preprocess predicted types
```

```
df['Predicted_Types'] = df['Predicted_Types'].apply(preprocess_text)

# Convert categories to list for vectorization
category_names = list(categories.keys())
category_descriptions = list(categories.values())

# Vectorize the categories
vectorizer = TfidfVectorizer(stop_words=stopwords.words('english'))
category_vectors = vectorizer.fit_transform(category_descriptions)

# Classify the predicted types into top N categories
N = 3  # Number of top categories to select

def classify_predicted_types(predicted_type):
    if not predicted_type:
        return ["Other"]
    predicted_vector = vectorizer.transform([predicted_type])
    cosine_similarities = cosine_similarity
    (predicted_vector, category_vectors).flatten()
    top_indices = cosine_similarities.argsort()[-N:][::-1]
    top_categories = [category_names[i] for i in top_indices]
    return top_categories

df['Categories'] = df['Predicted_Types'].apply(classify_predicted_types)

# Convert list of categories to string for easier CSV storage
df['Categories'] = df['Categories'].apply(lambda x: ', '.join(x))

# Save the updated dataset
df.to_csv('Updatedf_Places.csv', index=False, encoding = 'latin-1')

print("Classification completed and saved to 'Updated_Places.csv'")
```

```
import pandas as pd
import time
import random
from selenium import webdriver
from selenium.webdriver.common.by import By
from bs4 import BeautifulSoup
import os

# Load your dataset with correct encoding
df = pd.read_csv('help.csv', encoding = 'latin-1')

# Initialize a new column for votes if not present
if 'votes' not in df.columns:
    df['votes'] = None

# Set up Selenium WebDriver
driver = webdriver.Chrome()  # Ensure that chromedriver is in your PATH

def get_votes_from_google_maps(place_name, city):
    search_query = f"{place_name}, {city}"
   search_url = f"https://www.google.com/maps/search/{search_query}/"

    driver.get(search_url)
    time.sleep(4)  # Wait for the page to load

    # Check if a CAPTCHA is present
    if "sorry" in driver.current_url.lower():
        print("CAPTCHA encountered. Please solve it manually.")
        while "sorry" in driver.current_url.lower():
         time.sleep(15)  # Wait and allow the user to solve the CAPTCHA

    try:
```

```
        # Use BeautifulSoup to parse the page
        soup = BeautifulSoup(driver.page_source, 'html.parser')
        votes_element = soup.find('span', {'aria-label':
        lambda x: x and 'reviews' in x})
        if votes_element:
          votes_str = votes_element.text.strip("()").replace(",", "")
            # Extract the numeric part
            votes = ''.join(filter(str.isdigit, votes_str))
            return votes
        else:
            return None
    except Exception as e:
        return None

# Start index and limit for requests
start_index = 0  # Set this to your starting index
max_requests = 2991  # Limit for the number of requests

# Set the file paths
progress_file = 'loccordinates-Copy1.csv'
final_file = 'loccordinates-Copy2.csv'

# Ensure the files are writable
if os.path.exists(progress_file):
    os.remove(progress_file)

if os.path.exists(final_file):
    os.remove(final_file)

# Iterate through the dataset and update votes
try:
   for index in range(start_index, min(start_index + max_requests, len(df))):
```

```
        row = df.iloc[index]
        if pd.isna(row['votes']):
            place_name = row['Place_Name']
            # Adjust this column name as per your dataset
        city = row['City']  # Adjust this column name as per your dataset
           print(f"Processing {place_name}, {city} at index {index}")

            votes = get_votes_from_google_maps(place_name, city)
            if votes:
                df.at[index, 'votes'] = votes
           # Save progress after each request to avoid losing data in
            case of interruptions
           df.to_csv(progress_file, index=False, encoding='latin-1')

        # Sleep for a random duration between 1 and 2 seconds to comply with
            rate limiting
            sleep_duration = random.uniform(2, 3)
            time.sleep(sleep_duration)
finally:
    # Save the updated dataset in case of any interruption
    df.to_csv(final_file, index=False, encoding='latin-1')
    # Close the WebDriver
    driver.quit()
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Load your dataset
df = pd.read_csv('Placestestf.csv', encoding='latin-1')

# Ensure your dataset has the necessary columns
required_columns = ['City', 'Place_Name', 'latitude', 'longitude',
```

```
'Ratings', 'votes', 'Categories', 'Place_desc']
for col in required_columns:
    if col not in df.columns:
        raise ValueError(f"Missing required column: {col}")

# Drop rows with missing coordinates
df = df.dropna(subset=['latitude', 'longitude'])

# Handle missing values in Ratings and votes by filling with a default value
df['Ratings'].fillna(0, inplace=True)
df['votes'].fillna(0, inplace=True)

# Function to calculate the weight based on ratings and votes
def calculate_weight(row):
    if row['Ratings'] > 0 and row['votes'] > 0:
        return (row['Ratings'] + (row['votes'] / 1000)) / 2
        # Aggregate by averaging
    elif row['Ratings'] > 0:
        return row['Ratings']
    elif row['votes'] > 0:
        return row['votes'] / 1000  # Scale votes down
    else:
        return 0  # Default value if both are missing

df['Weight'] = df.apply(calculate_weight, axis=1)

# Min-max normalization to scale weights between 2.5 and 3.5
min_weight = df['Weight'].min()
max_weight = df['Weight'].max()

df['Score'] = 2.5 + (df['Weight'] - min_weight) * (3.5 - 2.5) /
(max_weight - min_weight)
```

```
# Function to match categories
def match_categories(categories_string, selected_categories):
    if pd.isna(categories_string):
        return False
    categories_list = categories_string.split(',')
    for category in categories_list:
        if any(cat.strip().lower() in category.strip().lower()
        for cat in selected_categories):
            return True
    return False

# Function to recommend places
def recommend_places(city, categories, num_days, places_per_day):
    # Filter places for the specified city
    city_places = df[df['City'] == city]
   city_places = city_places.sort_values(by=['Score'], ascending=False)

    # Filter places based on user-specified categories
    primary_places = city_places[city_places['Categories'].apply
    (lambda x: match_categories(str(x), categories))]
    secondary_places = city_places[~city_places['Categories'].apply
    (lambda x: match_categories(str(x), categories))]

    # Combine primary and secondary places
    combined_places = pd.concat([primary_places, secondary_places]).
    drop_duplicates(subset=['Place_Name'])

    # Apply K-Means clustering
    k = 5  # Adjust based on your requirements
    kmeans = KMeans(n_clusters=k, random_state=42)
    combined_places['Cluster'] = kmeans.fit_predict
```

```
 (combined_places[['latitude', 'longitude']])

 # Plot the clusters (optional)
 plt.figure(figsize=(12, 8))
plt.scatter(combined_places['longitude'], combined_places['latitude'],
 c=combined_places['Cluster'], cmap='viridis', marker='o')
 plt.colorbar(label='Cluster')
 plt.xlabel('Longitude')
 plt.ylabel('Latitude')
 plt.title(f'K-Means Clustering of Places in {city}')
 plt.show()

 # Group by clusters to ensure diversity
 clusters = combined_places.groupby('Cluster')

 # Create itinerary
 itinerary = []
 selected_indices = set()  # To keep track of selected places to
 avoid duplicates
 for day in range(num_days):
     daily_itinerary = []
     for cluster_id, cluster_data in clusters:
     cluster_places = cluster_data[~cluster_data['Place_Name'].isin
         (selected_indices)].head(places_per_day)
        daily_itinerary.extend(cluster_places.to_dict('records'))
         selected_indices.update(cluster_places['Place_Name'])
         if len(daily_itinerary) >= places_per_day:
             break
     itinerary.append(daily_itinerary[:places_per_day])
     # Ensure we only take the desired number of places

   # Drop the selected places to avoid duplicates in subsequent days
```

```
      combined_places = combined_places[~combined_places['Place_Name']
         .isin(selected_indices)]

         # Update clusters after dropping the selected places
         clusters = combined_places.groupby('Cluster')

    return itinerary

# Take user inputs
city = input("Enter the city name: ")

# Display available categories in the city
available_categories = df[df['City'] == city]['Categories'].dropna().unique()
available_categories = [cat.strip() for sublist in
available_categories for cat in sublist.split(',')]
available_categories = list(set(available_categories)) # Get unique categories
print("Available categories in the city:", ', '.join(available_categories))

# Ensure user selects valid categories
selected_categories = []
while not selected_categories:
   categories = input("Enter categories (comma-separated) from the above list: ")
    .split(',')
   selected_categories = [cat.strip() for cat in categories if cat.strip()
    in available_categories]
    if not selected_categories:
      print("Please select valid categories from the list provided.")

places_per_day = int(input("Enter the number of places per day: "))

# Calculate the maximum number of days
total_places = df[(df['City'] == city) & (df['Categories'].apply
```

```
(lambda x: match_categories(str(x), selected_categories)))].shape[0]
max_days = -(-total_places // places_per_day)  # Ceiling division
print(f"Based on {places_per_day} places per day, you can
cover the places in a maximum of {max_days} days.")

num_days = int(input(f"Enter the number of days (max {max_days}): "))
if num_days > max_days:
    print(f"You requested more days than the recommended
    maximum of {max_days} days.")
    print("You will be provided with the top places based
    on weight for the additional days.")

    # Ask user for additional categories only once
   additional_categories = input("Enter additional categories (comma-separated)
    from the available list or press Enter to skip: ").split(',')
    additional_categories = [cat.strip() for cat in
    additional_categories if cat.strip() in available_categories]

    if additional_categories:
        selected_categories.extend(additional_categories)
        total_places = df[(df['City'] == city) & (df['Categories'].
     apply(lambda x: match_categories(str(x), selected_categories)))].shape[0]
     max_days = -(-total_places // places_per_day)  # Ceiling division
        print(f"Based on {places_per_day} places per day,
      you can now cover the places in a maximum of {max_days} days.")

   # Adjust num_days if additional categories increased the max_days
   num_days = int(input(f"Enter the number of days (max {max_days}): "))

# Generate the initial itinerary
itinerary = recommend_places(city, selected_categories,
min(num_days, max_days), places_per_day)
```

```
# If the user requested more days, add top weighted places for the extra days
if num_days > max_days:
    additional_days = num_days - max_days
    top_places = df[df['City'] == city].sort_values(by='Weight',
    ascending=False)
    .drop_duplicates(subset=['Place_Name'])
    additional_itinerary = []
    selected_indices = set([place['Place_Name']
   for day in itinerary for place in day]) # Update with already selected places
    for day in range(additional_days):
        daily_itinerary = top_places[~top_places['Place_Name'].
    isin(selected_indices)].head(places_per_day).to_dict('records')
        additional_itinerary.append(daily_itinerary)
        selected_indices.update([place['Place_Name'] for
        place in daily_itinerary])
    itinerary.extend(additional_itinerary)

# Display the itinerary
for day, daily_itinerary in enumerate(itinerary, start=1):
    print(f"Day {day}:")
    for place in daily_itinerary:
     print(f" {place['Place_Name']} - {place['Categories']} - Rating:
        {place['Ratings']} - Votes: {place['votes']} -
        Score: {place['Score']} - Latitude:
        {place['latitude']} - Longitude:
        {place['longitude']} - Description:
        {place['Place_desc']}")
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

```
# Load the dataset
file_path = 'hotel_with_restaurantf.csv'
df = pd.read_csv(file_path)

# Fill missing values with 2 for rating columns and ensure all values are numeric
rating_columns = ['Service Quality', 'Amenities', 'Food and Drinks',
'Value for Money', 'Location', 'Cleanliness', 'multi_cuisine_restaurant']
df[rating_columns] = df[rating_columns].apply(pd.to_numeric, errors='coerce')
.fillna(2)

# Calculate composite score
df['Score'] = df[rating_columns].mean(axis=1)

# Define features and target
X = df[rating_columns]
y = df['Score']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the model
model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Calculate mean squared error
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

```
def get_property_types(city, df):
    # Get unique property types in the selected city
   property_types = df[df['city'] == city]['property_type'].unique()
    return property_types

def recommend_hotels(city, room_count, property_type, df, model):
    # Filter by city
    city_df = df[df['city'] == city]

    # Filter by room count
    city_df = city_df[city_df['room_count'] >= room_count]

    # If property type is specified, filter by property type
    if property_type:
        city_df = city_df[city_df['property_type'] == property_type]

  # If no hotels match the filters, return the highest score hotels in the city
    if city_df.empty:
        city_df = df[df['city'] == city]

    # Predict scores for filtered hotels
    features = city_df[rating_columns]
    city_df['Predicted_Score'] = model.predict(features)

    # Get the top 5 hotels with the highest predicted score
    top_5_hotels = city_df.nlargest(5, 'Predicted_Score')

    # Select only the required columns
  top_5_hotels = top_5_hotels[['latitude', 'longitude', 'property_name',
    'address', 'city']]
```

```
    return top_5_hotels

# Example usage
city = input("Enter city: ")
property_types = get_property_types(city, df)
print(f"Available property types in {city}: {property_types}")

property_type = input("Select property type from above (or press Enter to skip): ")
room_count = int(input("Enter required number of rooms: "))

top_5_hotels = recommend_hotels(city, room_count, property_type if
property_type else None, df, model)
print(f"The top 5 hotels are:\n{top_5_hotels}")
function fetchCategories() {
    const city = document.getElementById('city').value;
    if (city) {
        axios.post('/get_categories', { city: city })
            .then(response => {
            const categoriesDiv = document.getElementById('available-
                categories');
           categoriesDiv.innerHTML = '<label>Available categories in the city:
                </label><br>';
                response.data.forEach(category => {
                    const checkbox = document.createElement('input');
                    checkbox.type = 'checkbox';
                    checkbox.name = 'categories';
                    checkbox.value = category;
                    checkbox.id = category;
                   checkbox.onchange = () => { calculateMaxDays(); };

                    const label = document.createElement('label');
                    label.htmlFor = category;
```

```
                label.appendChild(document.createTextNode(category));

                    const br = document.createElement('br');

                    categoriesDiv.appendChild(checkbox);
                    categoriesDiv.appendChild(label);
                    categoriesDiv.appendChild(br);
                });
            })
            .catch(error => console.error('Error:', error));
    }
}

// Function to calculate maximum days based on selected
categories and places per day
function calculateMaxDays() {
    const city = document.getElementById('city').value;
   const placesPerDay = parseInt(document.getElementById('places_per_day')
    .value);
   const selectedCategories = Array.from(document.querySelectorAll('input[name=
    "categories"]:checked')).map(el => el.value);

    if (city && placesPerDay && selectedCategories.
    length > 0) {
        axios.post('/calculate_max_days', { city:
        city, places_per_day: placesPerDay, selected_categories:
        selectedCategories })
            .then(response => {
              const maxDaysDiv = document.getElementById('max-days');
           maxDaysDiv.textContent = Based on ${placesPerDay} places per day, you
           can cover the places in a maximum of ${response.data.max_days} days.;
```

```
                const numDays = parseInt(document.getElementById
                ('num_days').value);
                if (numDays > response.data.max_days)
                {
                    showAdditionalCategories();
                } else {
             document.getElementById('additional-categories').style.display =
                    'none';
                }
            })
            .catch(error => console.error('Error:', error));
    }
}

// Function to show additional categories dynamically
function showAdditionalCategories() {
    const city = document.getElementById('city').value;

    axios.post('/get_categories', { city: city })
        .then(response => {
            const additionalCategoriesDiv = document.getElementById
            ('additional-categories');
            additionalCategoriesDiv
            .style.display = 'block';
            const additionalCategoriesList = document.getElementById
            ('additional-categories-list');
            additionalCategoriesList.innerHTML = '';

        const selectedCategories = Array.from(document.querySelectorAll
          ('input[name="categories"]:checked')).map(el => el.value);

            response.data.forEach(category => {
```

```
                if (!selectedCategories.includes(category)) {
                    const checkbox = document.createElement('input');
                    checkbox.type = 'checkbox';
                    checkbox.name = 'additional_categories';
                    checkbox.value = category;
                    checkbox.id = 'additional_' + category;

                    const label = document.createElement('label');
                    label.htmlFor = 'additional_' + category;
                label.appendChild(document.createTextNode(category));

                    const br = document.createElement('br');

                    additionalCategoriesList.appendChild(checkbox);
                    additionalCategoriesList.appendChild(label);
                    additionalCategoriesList.appendChild(br);
                }
            });
        })
        .catch(error => console.error('Error:', error));
}
from flask import Flask, request, render_template, jsonify
import pandas as pd
from sklearn.cluster import KMeans
import folium

app = Flask(_name_)

# Load dataset
df = pd.read_csv('Placestestf.csv', encoding='latin-1')

# Ensure dataset has necessary columns
```

```
required_columns = ['City', 'Place_Name', 'latitude', 'longitude',
'Ratings', 'votes', 'Categories', 'Place_desc']
for col in required_columns:
    if col not in df.columns:
        raise ValueError(f"Missing required column: {col}")

# Drop rows with missing coordinates
df = df.dropna(subset=['latitude', 'longitude'])

# Handle missing values in Ratings and votes
df['Ratings'] = df['Ratings'].fillna(0)
df['votes'] = df['votes'].fillna(0)

# Function to calculate weight
def calculate_weight(row):
    if row['Ratings'] > 0 and row['votes'] > 0:
        return (row['Ratings'] + (row['votes'] / 1000)) / 2
    elif row['Ratings'] > 0:
        return row['Ratings']
    elif row['votes'] > 0:
        return row['votes'] / 1000
    else:
        return 0

df['Weight'] = df.apply(calculate_weight, axis=1)

# Min-max normalization
min_weight = df['Weight'].min()
max_weight = df['Weight'].max()

df['Score'] = 2.5 + (df['Weight'] - min_weight) * (3.5 - 2.5) /
(max_weight - min_weight)
```

```
# Function to match categories
def match_categories(categories_string, selected_categories):
    if pd.isna(categories_string):
        return False
    categories_list = categories_string.split(',')
    for category in categories_list:
        if any(cat.strip().lower() in category.strip().lower() for
        cat in selected_categories):
            return True
    return False

# Function to recommend places
def recommend_places(city, categories, num_days, places_per_day):
    city_places = df[df['City'] == city].sort_values(by=['Score'],
    ascending=False)
    primary_places = city_places[city_places['Categories'].apply
    (lambda x: match_categories(str(x), categories))]
    secondary_places = city_places[~city_places['Categories'].apply
    (lambda x: match_categories(str(x), categories))]
    combined_places = pd.concat([primary_places, secondary_places]).
    drop_duplicates(subset=['Place_Name'])

    k = 5
    kmeans = KMeans(n_clusters=k, random_state=42)
   combined_places['Cluster'] = kmeans.fit_predict(combined_places[['latitude',
    'longitude']])

    clusters = combined_places.groupby('Cluster')
    itinerary = []
    selected_indices = set()
    maps = []
```

```
    for day in range(num_days):
        daily_itinerary = []
     day_map = folium.Map(location=[df['latitude'].mean(), df['longitude']
        .mean()], zoom_start=10)
        for cluster_id, cluster_data in clusters:
            cluster_places = cluster_data[~cluster_data['Place_Name']
             .isin(selected_indices)].head(places_per_day)
             for place in cluster_places.to_dict('records'):
                 folium.Marker(
                     location=[place['latitude'], place['longitude']],
                  popup=f"<b>{place['Place_Name']}</b><br>Categories:
              {place['Categories']}<br>Description: {place['Place_desc']}",
                       icon=folium.Icon(color='red')
                 ).add_to(day_map)
            daily_itinerary.extend(cluster_places.to_dict('records'))
             selected_indices.update(cluster_places['Place_Name'])
             if len(daily_itinerary) >= places_per_day:
                 break
        if daily_itinerary:  # Only append non-empty days
             itinerary.append(daily_itinerary[:places_per_day])
             maps.append(day_map)
      combined_places = combined_places[~combined_places['Place_Name']
        .isin(selected_indices)]
        clusters = combined_places.groupby('Cluster')

    return itinerary, len(itinerary), maps

@app.route('/', methods=['GET', 'POST'])
def index():
    available_cities = df['City'].unique()
    if request.method == 'POST':
        city = request.form['city']
```

```
        selected_categories = request.form.getlist('categories')
      additional_categories = request.form.getlist('additional_categories')
        places_per_day = int(request.form['places_per_day'])
        num_days = int(request.form['num_days'])

        if additional_categories:
            selected_categories.extend(additional_categories)

        itinerary, available_days, maps = recommend_places(city,
        selected_categories, num_days, places_per_day)

      return render_template('index.html', available_cities=available_cities,
        itinerary=itinerary,
      available_days=available_days, selected_categories=selected_categories,
        maps=maps)

   return render_template('index.html', available_cities=available_cities)

@app.route('/get_categories', methods=['POST'])
def get_categories():
    data = request.get_json()
    city = data['city']
   available_categories = df[df['City'] == city]['Categories'].dropna().unique()
    available_categories = [cat.strip() for sublist
    in available_categories for cat in sublist
    .split(',')]
    available_categories = list(set(available_categories))
    return jsonify(available_categories)

@app.route('/calculate_max_days', methods=['POST'])
def calculate_max_days():
    data = request.get_json()
```

```
    city = data['city']
    selected_categories = data['selected_categories']
    places_per_day = data['places_per_day']

    total_places = df[(df['City'] == city) & (df['Categories'].apply
   (lambda x: match_categories(str(x), selected_categories)))].shape[0]
    max_days = -(-total_places // places_per_day)  # Ceiling division
    return jsonify(max_days=max_days)

@app.route('/map/<int:day>')
def map(day):
    global maps
    return render_template('map.html', map_html=maps[day-1]
    .get_root().render())

if _name_ == '_main_':
    app.run(debug=True)
```

CHAPTER 8

Testing

8.1 The system underwent rigorous testing to ensure :

8.1.1 Functionality

Correctness in data retrieval, processing, and visualization of travel destinations.

8.1.2 Performance

Efficiency in recommending places based on user preferences and real-time updates of ratings and votes.

8.1.3 User Interface

Responsiveness and interactivity of the web application, including dynamic updates and error handling.

CHAPTER 9

Result

9.1 Output

- User interface showing input fields for city selection, category checkboxes, and places per day.
- Dynamic updates of available categories based on selected city and calculation of maximum travel days.
- Visual representation of recommended travel destinations on an interactive map using Leaflet.js.
- The model creates a day-wise itinerary, ensuring a balanced distribution of places to visit each day.
- The recommended places are visualized on a map using Leaflet.js, with different colors representing different clusters.
- The interface allows users to input their preferences and see the itinerary along with the map visualization.

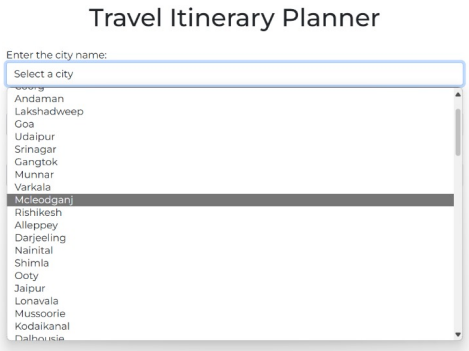

Figure 9.1

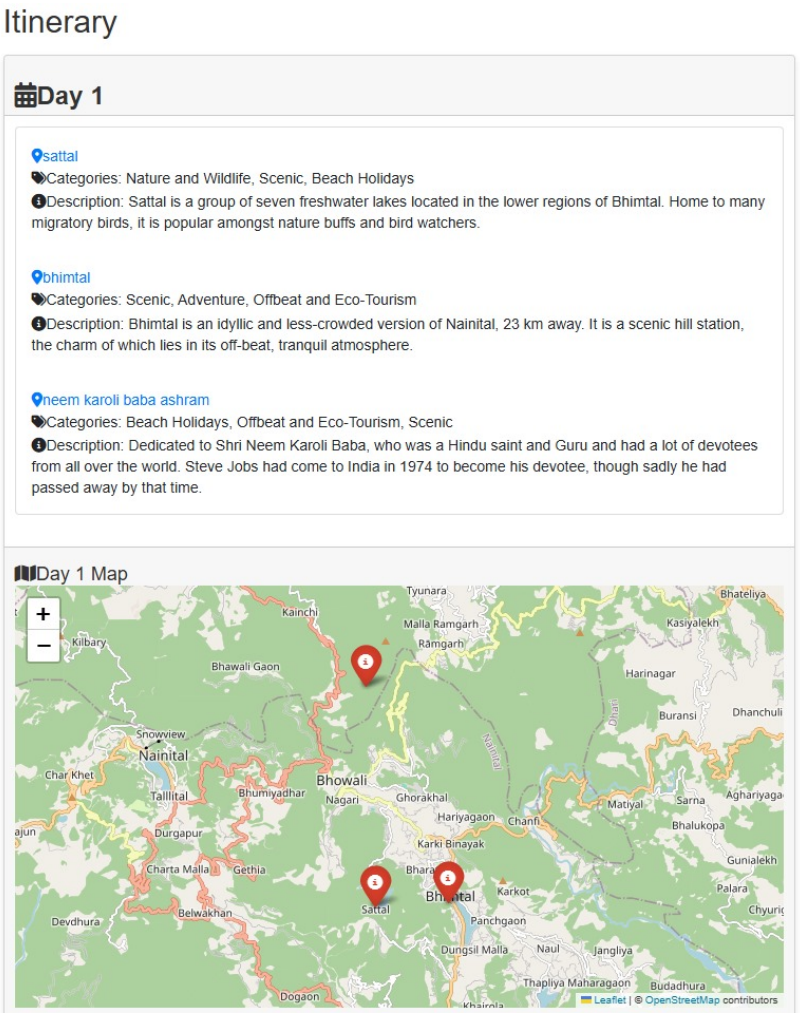

Figure 9.2

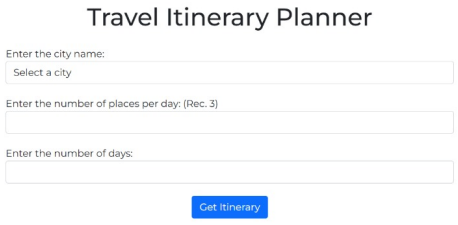

Figure 9.3

CHAPTER 10

Conclusion

In conclusion, the proposed Tour Recommendation System leverages modern technology and data-driven approaches to enhance the process of tour planning and itinerary generation. By integrating machine learning algorithms, web scraping techniques, and user-friendly interfaces, the system addresses several challenges present in traditional manual planning methods.

This system offers several advantages over existing approaches. It automates the collection of place data from Google Maps, ensuring up-to-date and accurate information about each location's ratings and reviews. The use of K-Means clustering enables the grouping of similar places geographically, enhancing diversity in recommended itineraries. Moreover, by dynamically fetching categories based on user-selected cities, the system provides personalized recommendations tailored to individual preferences.

The implementation of this system demonstrates its potential to revolutionize the tourism sector by simplifying the complex task of tour planning. It offers users a streamlined experience, suggesting the best places to visit, accommodations, and daily itineraries based on comprehensive data analysis. This not only saves time but also enhances the overall travel experience by ensuring optimal use of time and resources.

Looking forward, future enhancements could include integrating more advanced machine learning models for personalized recommendation systems, expanding data sources beyond Google Maps, and improving the user interface for enhanced interactivity and usability. With continuous advancements in technology and data analytics, the Tour Recommendation System is poised to become an indispensable tool for travelers and tour operators alike, contributing to the growth and efficiency of the tourism industry.

CHAPTER 11

Future Enhancement

11.1 Future enhancements may include :

11.1.1 Enhanced Recommendation Algorithms

Integration of advanced machine learning models for better place recommendations based on user feedback and preferences.

11.1.2 Real-Time Data Integration

Incorporation of APIs for real-time updates on ratings, reviews, and travel advisories.

11.1.3 Mobile Application Development

Expansion to mobile platforms for on-the-go travel planning and enhanced user accessibility.