

**DTI5125: Data science Applications**

**Tourism Recommender System | Final Report**

**Professor: Dr. Arya Rahgozar**

**Group 12**

**Pouria Bahri: 300352271**

**Minoo Zarghami: 300370956**

**Shideh Moshref Razavi: 300340506**

**Summer 2024**

Table of Contents

[1 Introduction 3](#_Toc172462401)

[2 Model Training 3](#_Toc172462402)

[2.1 Dataset Description 3](#_Toc172462403)

[2.2 Data Preprocessing 3](#_Toc172462404)

[2.3 Model Training and Evaluation 3](#_Toc172462405)

[2.3.1 Clustering Analysis 3](#_Toc172462406)

[2.3.2 Classification: 5](#_Toc172462407)

[2.3.3 Confusion matrices 6](#_Toc172462408)

[3 Recommender Dataset 7](#_Toc172462409)

[3.1 Data Cleansing 7](#_Toc172462410)

[3.2 Preprocessing 7](#_Toc172462411)

[4 User Entry Data 7](#_Toc172462412)

[4.1 Step 1: Initial User Input 7](#_Toc172462413)

[4.2 Step 2: Text Mining 8](#_Toc172462414)

[4.3 Step 3: Combining Dataframes 9](#_Toc172462415)

[5 Applying Weights 10](#_Toc172462416)

[6 Recommending Cities 10](#_Toc172462417)

[7 Ngrok Setup and Integration with Flask and Dialogflow 11](#_Toc172462418)

[7.1 Ngrok Setup 11](#_Toc172462419)

[7.2 Downloading Ngrok: 11](#_Toc172462420)

[7.3 Installation: 11](#_Toc172462421)

[7.4 Account Setup and Authentication: 11](#_Toc172462422)

[7.4.1 Flask Application Setup: 11](#_Toc172462423)

[7.4.2 Starting the Flask Server 11](#_Toc172462424)

[7.4.3 Ngrok Dashboard 12](#_Toc172462425)

[7.5 Configuring Dialogflow Webhook 13](#_Toc172462426)

[7.6 Testing and Debugging 13](#_Toc172462427)

[7.7 Dialogflow Setup for Tourist Recommender Chatbot 13](#_Toc172462428)

[7.7.1 Agent Creation 13](#_Toc172462429)

[7.8 Recommender system architecture 14](#_Toc172462430)

[7.9 Intents and Entities Configuration 14](#_Toc172462431)

[7.9.1 Intents 14](#_Toc172462432)

[7.9.2 Entities: 14](#_Toc172462433)

[7.9.3 Training Phrases: 15](#_Toc172462434)

[7.9.4 Parameters and Prompts: 15](#_Toc172462435)

[7.10 Fulfillment and Webhook Configuration 16](#_Toc172462436)

[7.11 Integration and Testing 17](#_Toc172462437)

[7.12 Final Deployment 17](#_Toc172462438)

# Introduction

The objective of this project is to develop a chatbot recommender system for tourists planning to visit India. The recommender system leverages machine learning models to provide personalized recommendations based on various travel-related attributes such as region, season, accommodation, destination, and budget.

# Model Training

We initially had to train text classifiers to use them when we’re interacting with the user in chatbot. For doing so, we had a dedicated dataset. Additionally, by using that and incorporating machine learning methods we trained the classifiers.

## Dataset Description

The dataset contains multiple columns that describe different attributes related to travel in India, including Region, Season, Accommodation, Destination, and Budget. Each column has labels used for classification tasks to aid in building the recommendation system.

## Data Preprocessing

We had to preprocess the model training dataset to be able to utilize it.

* **Lowercasing:** Converted all text data to lowercase to ensure uniformity.
* **Splitting Data:** Split the data into training and testing sets for each classification task.
* **Vectorization:** Used TF-IDF (Term Frequency-Inverse Document Frequency) to convert text data into numerical features.

## Model Training and Evaluation

### Clustering Analysis

* **K-Means Clustering:** We applied the K-Means clustering algorithm to partition the data into distinct groups. K-Means works by initializing a set number of centroids and iteratively adjusting them to minimize the distance between the data points and their respective centroids.
* **Elbow Method:** To determine the optimal number of clusters, we used the Elbow method. This method involves plotting the sum of squared distances from each point to its assigned centroid for different numbers of clusters. The 'elbow point' on the plot indicates the optimal number of clusters, which is 3.

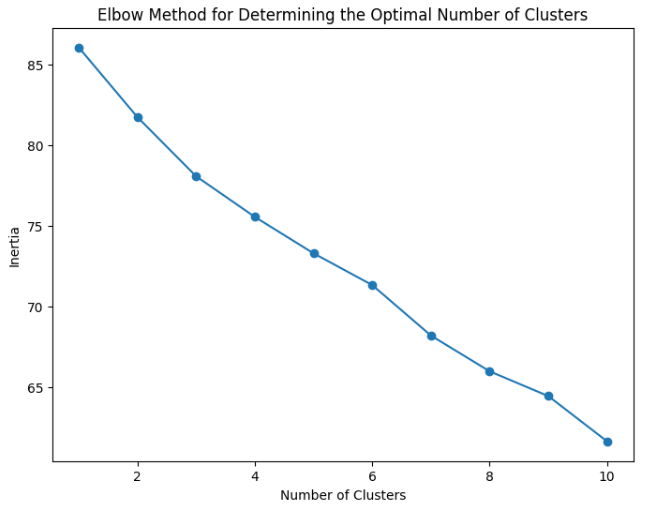


Figure 1. Elbow Method

* **Silhouette Score**: To evaluate the quality of the clusters, we calculated the silhouette score. The silhouette score measures how similar a data point is to its own cluster compared to other clusters. A high silhouette score indicates that the data points are well matched to their own cluster and poorly matched to neighboring clusters, signifying well-defined clusters.

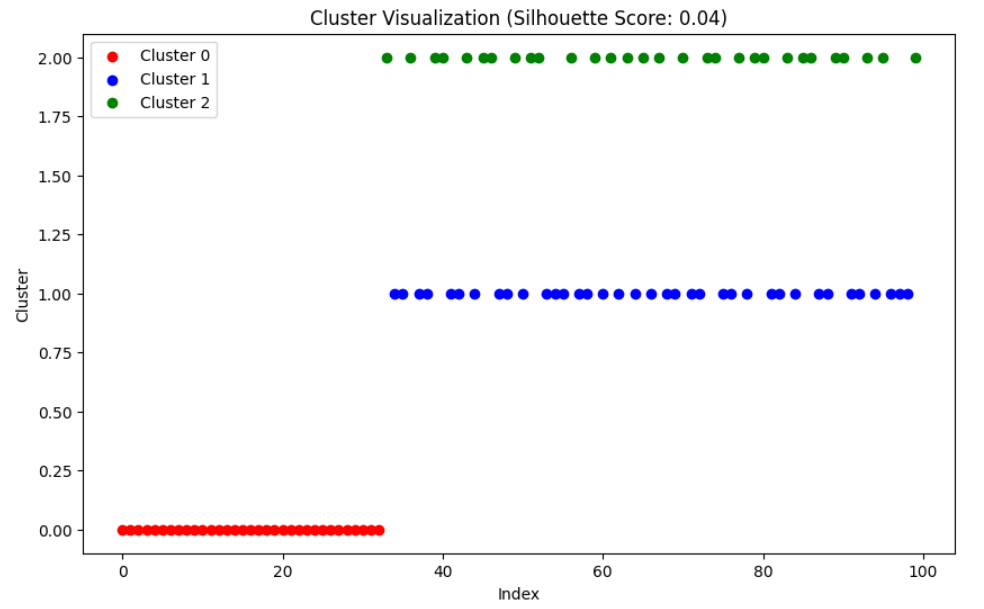


Figure 2. Silhouette score

Due to the nature of our dataset, there were inherent challenges in effectively using clustering algorithms. Despite attempts to utilize these algorithms, the silhouette scores obtained were quite low, indicating poorly defined clusters. As a result, the decision was made to pivot to classification algorithms instead. These models provided better performance and more accurate predictions, making them more suitable for our recommendation system. Figure 2 shows that while the first cluster (Cluster 0) was identified correctly, the identification for the other two clusters was subpar. Given that the training dataset had labels in chronological order, a step-like graph was expected, which the clustering algorithms were unable to produce.

### Classification:

After training various models, we could achieve the below models with at least 80% accuracy.

* Multinomial Naive Bayes (Region)
* Support Vector Machine (Season)
* Random Forest (Accommodation)
* Random Forest (Destination)
* Gradient Boosting (Budget)

The graph below shows the accurate number of these metrics.

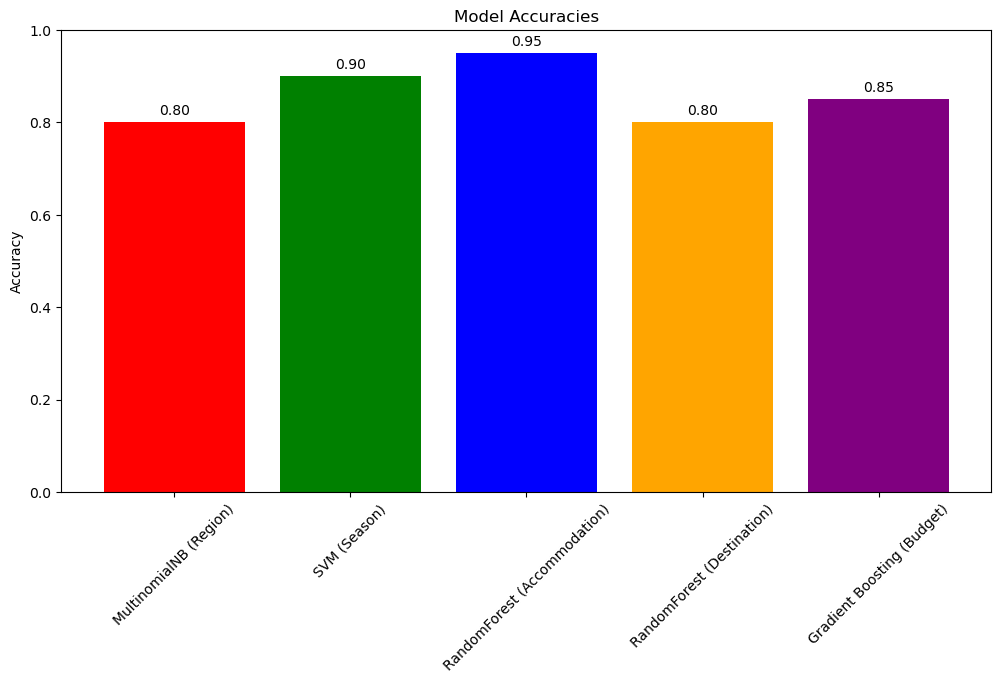


Figure 3. Accuracy scores

### Confusion matrices

To effectively evaluate the performance of our classification models, we used various performance metrics. The primary metric utilized was accuracy, which measures the proportion of correctly predicted instances out of the total instances. Additionally, we used confusion matrices for a more detailed analysis. A confusion matrix provides insight into the types of errors being made by the model by displaying the number of true positives, true negatives, false positives, and false negatives. This allows us to identify specific areas where the model may be underperforming. these matrixes are presented below.

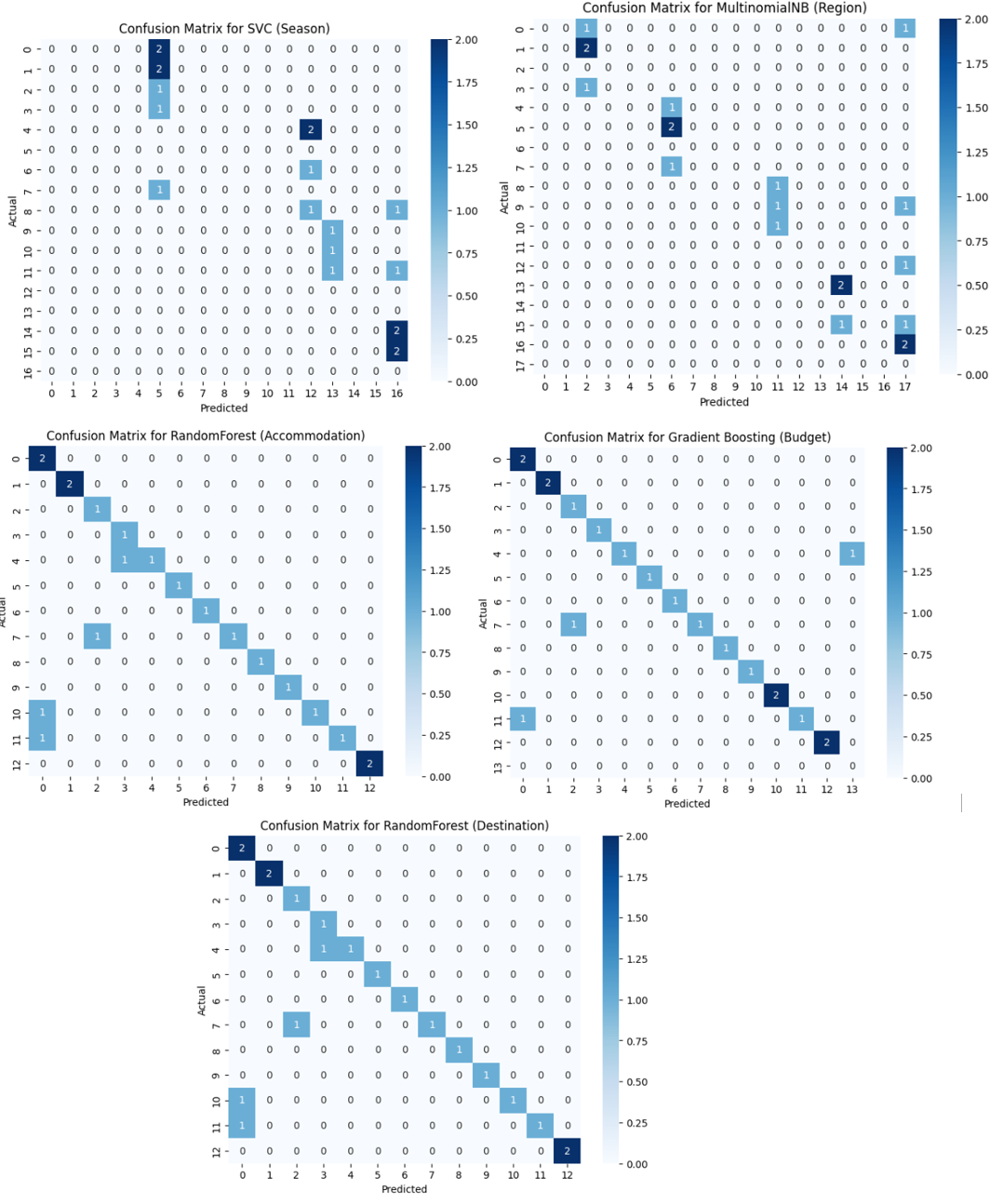


Figure 4. Confusion Matrix

# Recommender Dataset

Our reference dataset, which forms the basis of our recommendations, comprises 100 cities across India. This is separate from the model training dataset which was used in the previous step. Initially, this dataset included numerous features, some of which were removed for various reasons.

## Data Cleansing

* **Issue of Pedantry:** Columns such as "Tourist Attraction" and "State" were too specific for each record, leading to redundancy. For example, if a user intends to visit the Taj Mahal, they would simply visit the city of Agra. Similarly, someone interested in the Chandigarh Carnival would head to Chandigarh. Including such specific details defeats the purpose of the recommender system, which aims to assist users who may not have extensive knowledge about their destination.
* **Alternative Features:** The "Best Time to Visit" column was removed because the "Tourist Season" column already existed. "Best Time to Visit" was not only somewhat redundant but also less inclusive compared to "Tourist Season".
* **Inconsistency in Descriptions:** The "City Description" column was removed due to its inconsistency and lack of uniform information. While some records included details about the tourist season or the type of destination, others did not. Given the dataset's structure, with one record per city, we had to eliminate columns that introduced inconsistency.

## Preprocessing

After removing the unwanted features, we applied one-hot encoding to the reference dataset to prepare it for further analysis. Using prefixes during encoding helps maintain the integrity of the dataset and makes it more comprehensible.

# User Entry Data

To recommend a city based on user responses and preferences, we follow a three-step process.

## Step 1: Initial User Input

In the first step, we ask the user six questions, each with specific answers. These questions gather the user's preferences regarding their destination's rating, season, accommodation, type, region, and travel budget. The responses are stored in a dataframe. The specific answers are as follows:

Region: Center, North, South, West, East

Season: Fall, Winter, Spring, Summer

Accommodation: Ashram, Camp, Guesthouse, Heritage Hotel, Homestay, Hotel, Houseboat, Monastery, Resort

Destination: Adventure, Backwater, Beach, Coastal, Cultural, Desert, Heritage, Hill Station, Island, Nightlife, Pilgrimage, Romantic, Scenic, Skiing, Trekking, Urban, Wildlife

Budget: Low, Medium, High

For example, if a user provides the following parameters:

Rating: 3.1

Season: Spring

Accommodation: Ashram

Destination Type: Adventure

Region: East

Budget: Low

We create the following dataframe based on these inputs:



Figure 5. Tabular user input

## Step 2: Text Mining

The last question asks the user to describe their desired destination. For example:

"I like to go to the Center of India in the Fall. I enjoy Wildlife and prefer heritage hotels for their historic charm. My budget is High."

We apply classifiers to the user's text to extract relevant features:

Rating:

Season: Fall

Accommodation: Heritage Hotel

Destination Type: Wildlife

Region: Center

Budget: High

We then create a second dataframe based on these extracted features.

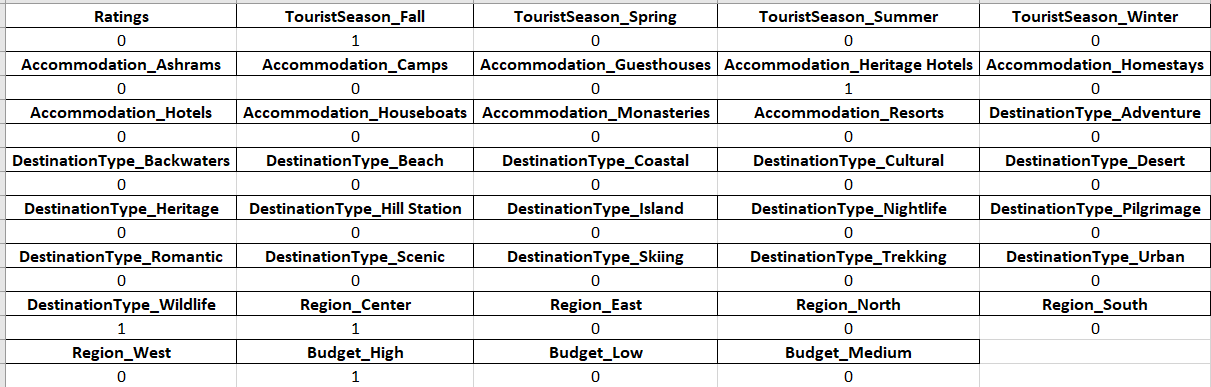


Figure 6. Text mining user input

## Step 3: Combining Dataframes

The second dataframe allows users to specify their preferences in their own words, which might capture details they missed in the initial structured input. We combine the two dataframes using a union-like operator in set theory, resulting in:

Rating: 3.1

Season: Spring, Fall

Accommodation: Ashram, Heritage Hotel

Destination Type: Adventure, Wildlife

Region: East, Center

Budget: Low, High



Figure 7. The combined dataset

# Applying Weights

Since one-hot encoding can lead to an imbalance with too many encoded categories for some columns (e.g., 17 for destination type) and too few for others (e.g., 3 for budget), we applied weights to each column. This ensures that a large number of selected destination types do not disproportionately influence the results. For example, we transform Budget\_High = 1 to Budget\_High = 0.33.

The weight for each column is calculated as:

Column Weight *=*

These weights are applied to both the combined user input dataframe and the reference dataframe of 100 cities.

# Recommending Cities

With all dataframes ready and user entries gathered, the system makes recommendations using cosine similarity. This method measures the similarity between two vectors in an inner product space, providing accurate and relevant city recommendations.

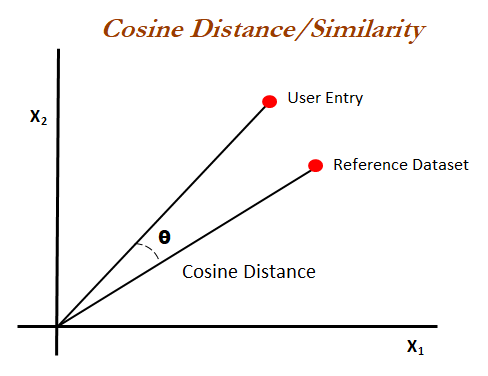


Figure 8. Cosine similarity method

# Ngrok Setup and Integration with Flask and Dialogflow

## Ngrok Setup

Ngrok is a versatile tool that creates a secure tunnel to localhost and exposes local servers behind NATs and firewalls to the public internet over secure tunnels. This capability is essential during development stages when the services need to interact with external APIs such as Dialogflow, which requires a publicly accessible URL for webhook integrations. In our project Ngrok was used to create a secure tunnel to the Flask application running locally. This setup allowed Dialogflow to send requests directly to the local development environment for processing. For installation and setup of Ngrok we went through these stages:

## Downloading Ngrok:

Ngrok can be downloaded from its official website at https://ngrok.com/download. We selected the version compatible with our operating system.

## Installation:

After downloading, we unzipped the package. For Windows, this involved extracting the executable.

## Account Setup and Authentication:

Post-installation, signing up for an account on the Ngrok website provided an authentication token. This token was used to connect the Ngrok client to the Ngrok service, which helps in managing tunnels, custom subdomains, etc. We ran the following command to add our authtoken to the default ngrok.yml:

ngrok config add-authtoken

2jNsCJK2WXczGZtVaHopxFNPSG8\_2tKU1PWcfYmh3Mwufrmgk

### Flask Application Setup:

Our Flask application was already developed and ready to run locally. It was configured to listen on localhost at port 8080. The Flask app included endpoints to handle POST requests, specifically designed to interact with Dialogflow’s webhook.

### Starting the Flask Server

We started our Flask application by running: python recommender-flask.py

### Ngrok Dashboard

Upon executing the command, Ngrok provided a public URL displayed in the command prompt. This URL forwards all traffic to our local Flask server.

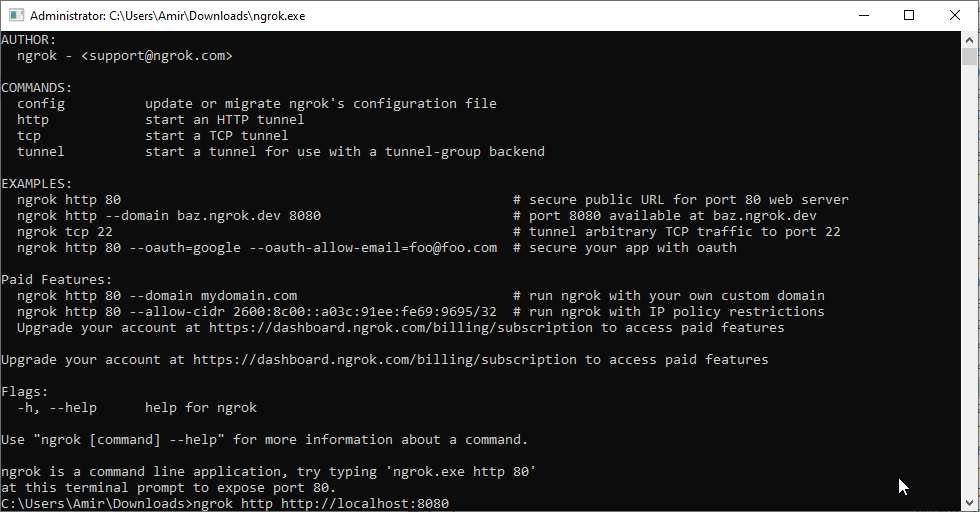


Figure 9. NGROK Dashboard

Ngrok started and displayed a screen like this:

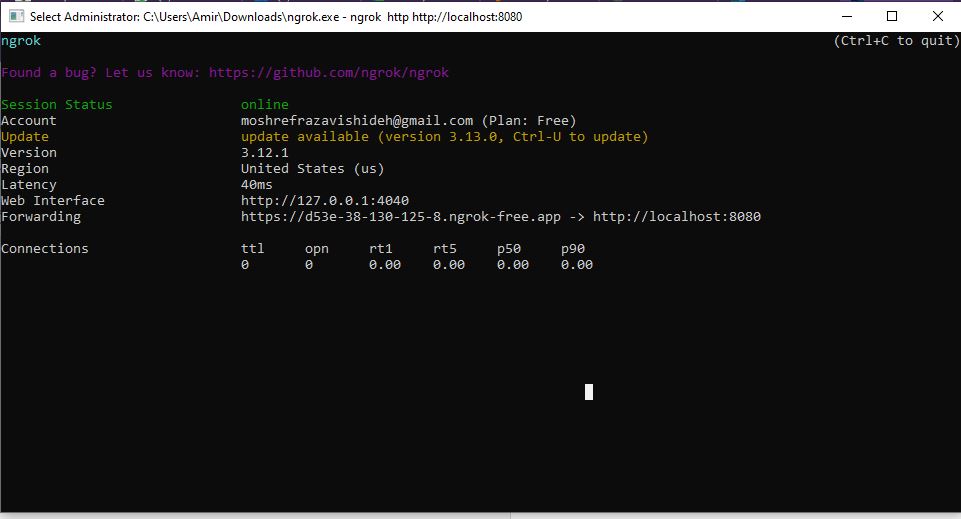


Figure 10. NGROK Connections

## Configuring Dialogflow Webhook

**Updating Webhook URL:** The HTTPS URL generated by Ngrok was copied and pasted into the Webhook URL section of the Dialogflow’s Fulfillment page, ensuring it appended the /recommend endpoint correctly as configured in our Flask app.

**Enabling Webhook:** We enabled the webhook option for the specific intents within Dialogflow that required dynamic responses generated by our Flask application. This setup allowed Dialogflow to send POST requests to our Flask endpoint via the Ngrok URL.

## Testing and Debugging

**Simulation and Testing:** Using Dialogflow’s testing simulator, we sent queries to our agent and verified that the responses were correctly handled by our Flask application via the Ngrok tunnel. Any issues in the request handling or response formatting were debugged using logs from both the Flask console and the Ngrok web interface.

## Dialogflow Setup for Tourist Recommender Chatbot

Our project utilizes Dialogflow, a tool by Google that leverages natural language understanding to process and analyze human language. The core of our chatbot's functionality within Dialogflow is defined through the creation and configuration of a specific intent that handles user requests for city recommendations based on personalized criteria.

### Agent Creation

* Initially, an agent named Tourist-Recommender was established. This agent acts as the chatbot's brain, interpreting user inputs and providing relevant responses.
* The agent is linked to our Google Cloud project, enabling it to integrate with other Google services and APIs seamlessly.

## Recommender system architecture

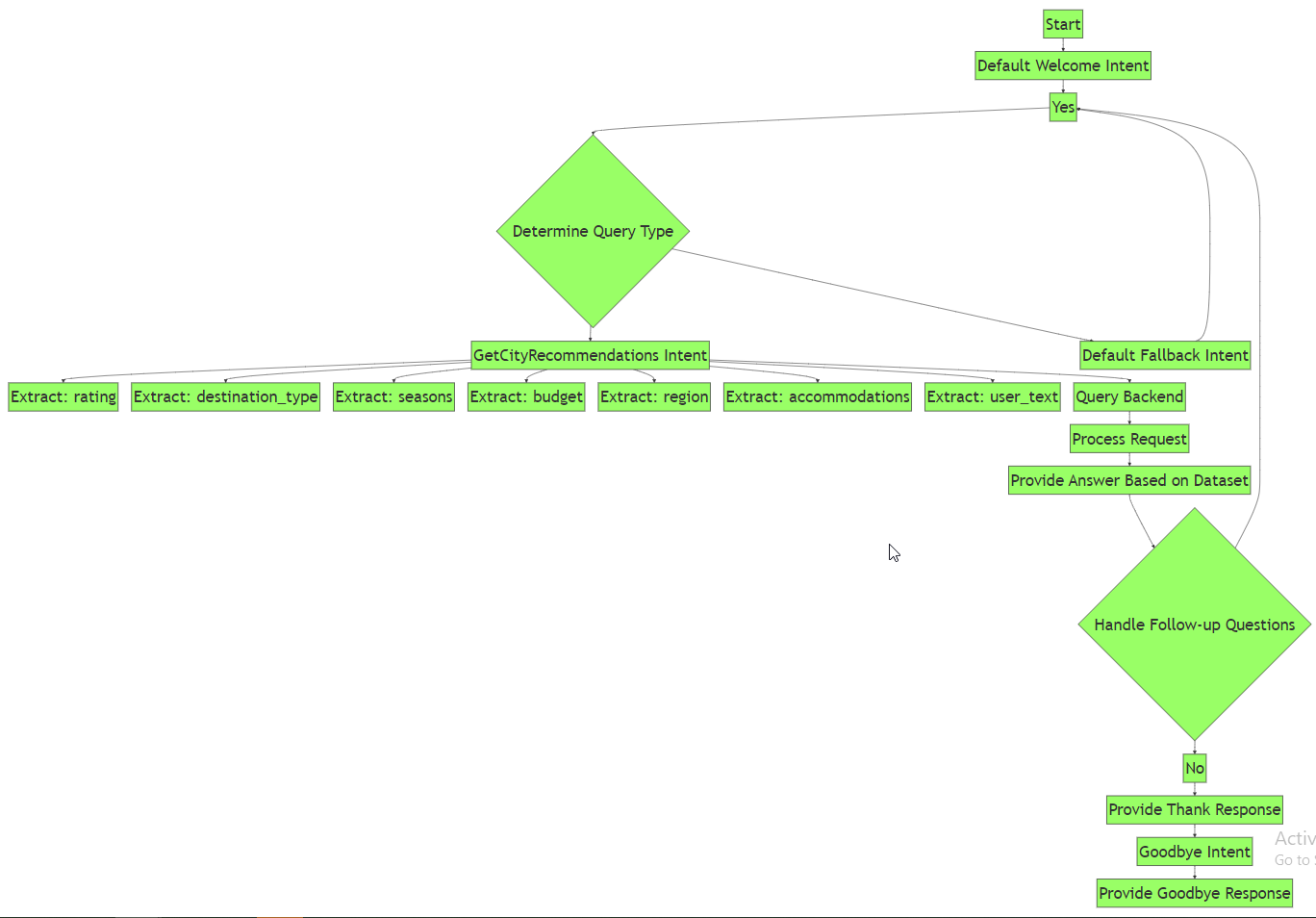


Figure 11. Tourism recommender system Architecture

## Intents and Entities Configuration

### Intents

Designed to capture different types of user queries and commands:

* **Default Welcome Intent**: Greets users and offers guidance on how to interact with the chatbot.
* **GetCityRecommendations**: Main intent that handles the core functionality of recommending cities based on user-specified criteria.
* **Thank**: Handles user expressions of gratitude.
* **Goodbye**: Manages farewell interactions.

### Entities:

They are used to extract and parameterize important information from user inputs:

* **@rating** (sys.number): Captures numerical user ratings.
* **@destination\_type**: Includes categories like Adventure, Beach, Cultural, etc., with synonyms to increase matching accuracy.
* **@seasons**: Recognizes various seasons mentioned by the user.
* **@budget**: Identifies budget levels (Low, Medium, High).
* **@region**: Detects geographical preferences. (East, West, Center, North and South)
* **@accommodations**: Lists types of accommodations like Hotels, Resorts, etc.

### Training Phrases:

* We trained the intent with phrases that are typical of how users might request city recommendations, such as "Recommend a city for a summer getaway" or " Suggest some places to visit in India "

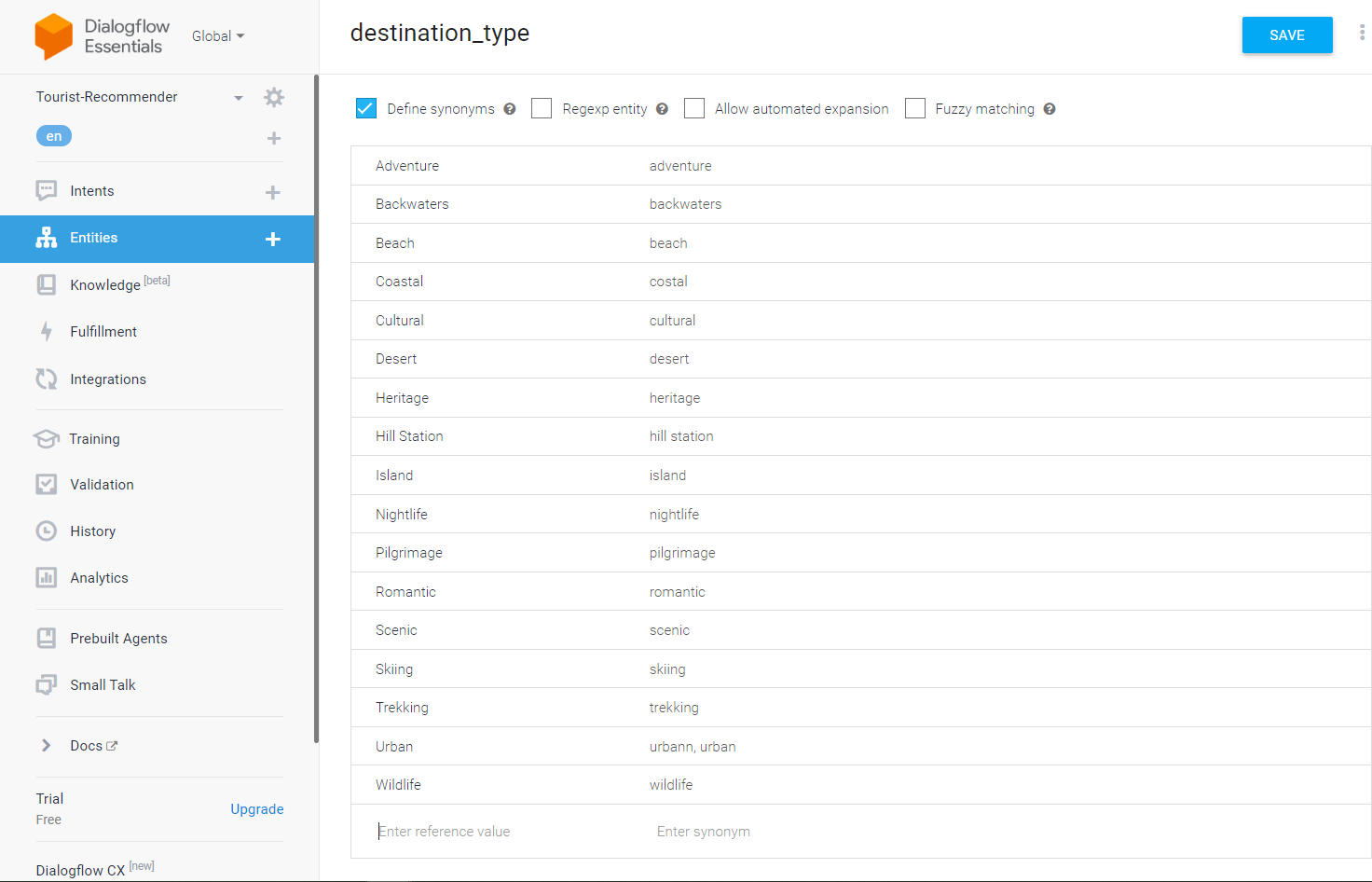


Figure 12. Entities

### Parameters and Prompts:

* The intent is configured to ensure all necessary parameters are collected from the user. Dialogflow's automatic prompts ask users for specific details if they are not provided in the initial query.

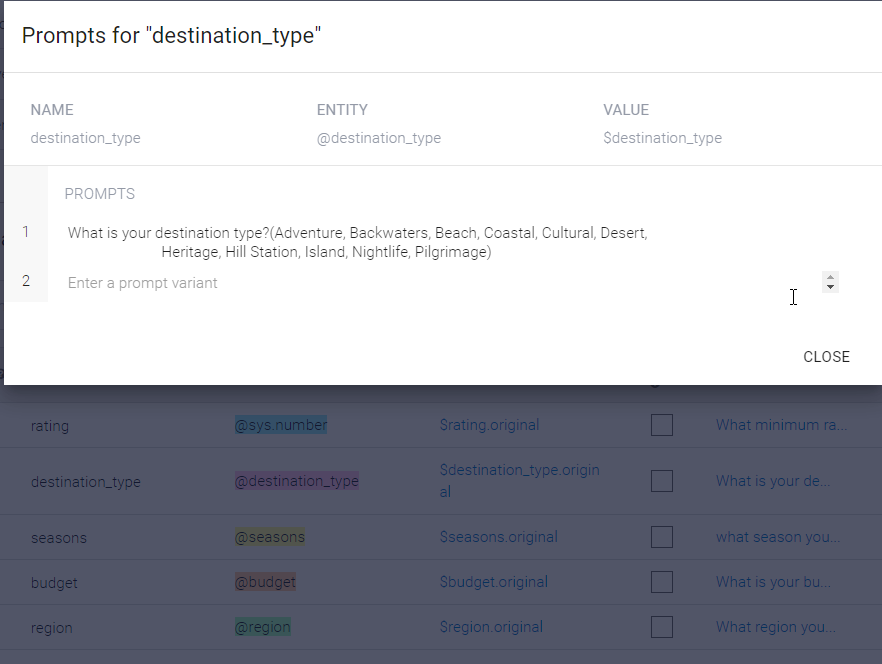


Figure 13. Prompts

## Fulfillment and Webhook Configuration

* A webhook is utilized for the GetCityRecommendations intent. When triggered, Dialogflow sends a request to our backend service (via a secure tunnel established by Ngrok during development phases) which processes the user's preferences and queries a Flask application to retrieve city recommendations.
* The URL configured in the Dialogflow fulfillment section corresponds to the Ngrok URL during development and testing, ensuring real-time communication between Dialogflow and our local server.

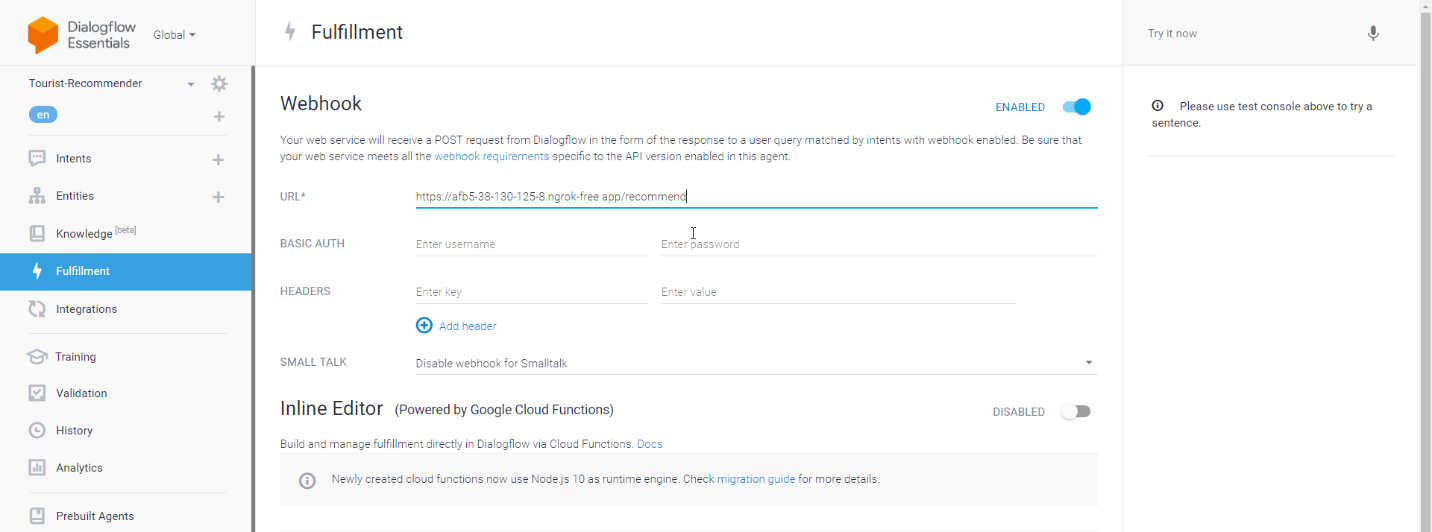


Figure 14. Fulfillment

## Integration and Testing

* Dialogflow's built-in console test feature was employed to simulate interaction and refine the intent's understanding of various user queries.
* This feature was crucial for immediate feedback and iterative adjustments to training phrases and entity recognition.

## Final Deployment

* After thorough testing and validation within the Dialogflow environment, the agent’s configuration, including intents and entities, was finalized for deployment. The final integration involves updating the fulfillment webhook URL to the production server following successful testing via Ngrok.