**Smart Road Trip Planner: Personalized Trip Planning by Capturing User Interactions**

Rishi Dinesh 19BAI1009 – [rishi.dinesh2019@vitstudent.ac.in](mailto:rishi.dinesh2019@vitstudent.ac.in) (9940190913)

Pranav Balaji 19BAI1151 – [pranav.balaji2019@vitstudent.ac.in](mailto:pranav.balaji2019@vitstudent.ac.in) (9840273517)

Abstract

Planning an itinerary before embarking on a road trip is one of the most important travel preparation activities. Recommender systems can ease this travel planning process by providing suggestions based on the user’s interests. However, most of the existing systems suffer from a cold-start problem and also do not account for changes in user’s interests with time. The goal of this project is to overcome these problems by building an AI that can be integrated into a web application that can act as a one-stop solution for its users by allowing them to plan all the aspects of a road trip with ease and comfort. This AI will be designed to capture the user’s mindset through various interactions on the website and personalize the website experience by providing recommendations that are in line with the user’s current interests. A user experience study was conducted in order to validate the efficiency of the proposed method. Results of the study showed that that accuracy of the model surpassed that of various existing methods as the number of user interactions on the website increased.

*Keywords: Trip planning, Recommender Systems, Knowledge-based agent*

Introduction

Trip recommendation and itinerary planning are challenging tasks due to the different interest preferences and trip constraints of each unique tourist. While there is an abundance of information from the Internet and travel guides, many of these resources simply recommend individual points of interest that are deemed to be popular, but otherwise do not appeal to the interest preferences of users or adhere to their trip constraints. Furthermore, the massive volume of information makes it a challenge for tourists to narrow down to a potential set of POIs to visit in an unfamiliar city.

In recent years, various trip and hotel recommendation systems have been proposed to guide the users in their travel planning process by providing suggestions that adhere to the user’s interests. However, traditional recommendation systems generally make use of the user’s previous rating history in order to understand the user’s interest. This leads to the classic cold-start problem, wherein recommendations cannot be provided for new users with no rating history. Moreover, most of these systems also do not account for the fact that a user’s interests can change with time. In this work, we have focused on exploring an alternative approach by building an AI based recommendation system that can be integrated into the backend of a trip planning website in order to capture the user’s interactions and analyse them to understand the user’s mindset.

The rest of the paper is divided as follows: The next section provides a detailed literature survey of the existing hotel and trip recommendation systems. This is followed by a breakdown of the proposed method, which is a three-piece recommender model that can provide hotel, POI and even blog recommendations on the website. Following this is performance analysis of the system and comparative analysis with existing systems. This is succeeded by a detailing of the various testing techniques performed to ensure that that system was infallible. Finally, the last section provides the concluding remarks and the references.

Literature Survey

*Hotel Recommendation Systems*

Kristian Wahyudi et al [1] proposed a simple content-based hotel recommendation system that recommends hotels based on the categories it falls under and the features of the city it is located in. It computes a similarity score between users using these attributes and in turn uses this score to get a weighted average of the user’s preference to a particular hotel that he/she has not stayed at or rated. The hotel recommendations given to the user are sorted by this weighted average.

One of the limitations of this system is that it can only recommend hotels that the user has not previously stayed at or rated. Another shortcoming is that it has a very limited scope, as the dataset being used contains hotel information for only one particular place in the US. Moreover, as a content-based recommender system, it does not account for a lot of hotel attributes such as price, star rating and amenities.

Our hotel recommender is designed to overcome these shortcomings as it accounts for almost 35 different attributes of each hotel while performing the user similarity calculation. Moreover, the hotel recommendations are not limited to any geographical location.

Huming and Weili [2] on the other hand, built a hotel recommendation system based on a clustering-based collaborative filtering and RankBoost Algorithm. This method overcame the cold start and scalability problem present in traditional collaborative algorithms. The RankBoost algorithm is a method of producing highly accurate prediction rules by combining many “weak” rules which may be only moderately accurate.

In order to generate recommendations, users with similar interest were first grouped into clusters by applying a similarity formula on the user-rating matrix. Next, the appropriate cluster was identified for the target user and the hotel’s rating was predicted based on the target user’s nearest neighbours in that cluster. This predicted rating was then used to generate hotel recommendations. The RankBoost algorithm was used in case the user was a new user.

However, this algorithm uses a binary weight to indicate the user’s preference to an attribute of the hotel: the user either like it or he doesn’t. Our algorithm overcomes this limitation by capturing different weights for different attributes of the hotel based on the type of interaction made by the user on the website.

Zhang et al [3] took a different approach in building a hotel recommendation system based on user preference analysis. It not only integrates collaborative and content-based filtering methods, but also introduces the intents of a trip to solve the cold start problem with a higher accuracy. It also uses diversity techniques to optimize the hotel recommendation list.

Their hotel recommendation framework includes three steps. In this first step, they determine the user’s preference on a specific hotel, by predicting missing ratings based on original user-item rating matrices and reviews. This is done using a hybrid preference factor model. Next, this complete user-rating matrix is improved by adding the trip intent; the idea is that users with the same trip intent usually share same hotel preferences. Finally, diversity techniques are used to optimize the hotel recommendations.

However, this algorithm does not account for changes in a user’s interest with time. Our recommender aims to overcome this limitation by introducing a time-based weight parameter that ensures that more recent interactions are given more priority than older ones.

Bushra Ramzan et al [4] presented an intelligent approach for hotel recommendation systems using sentiment analysis and machine learning models. They proposed a novel CF recommendation approach in which opinion-based sentiment analysis was used to achieve hotel feature matrix by polarity identification. Their approach combined lexical analysis, syntax analysis and semantic analysis to understand sentiment towards hotel features and the profiling of guest type (solo, family, couple etc). Their system recommends a list of suitable hotels to a user on the basis of both ratings (numerical) and reviews (textual).

This heterogenous data is crawled from the internet and stored in a database. The numeric data is then normalized while the text data is processed using NLP models. Then weighted average polarity scores are calculated by aggregating the normalized scores from numeric data and polarity scores from textual data. Finally, recommendations are computed by using a fuzzy logic approach.

One shortcoming of this approach is that a new hotel with no reviews or ratings of any sort cannot be accounted for in the recommendations. Our algorithm overcomes this disadvantage as it accounts not only for the guest ratings and the star rating, but also for the other basic attributes of every hotel such as price and amenities.

Nikolaos et al [5] focused on developing a hybrid multi-criteria hotel recommender system using online reviews and ratings. The new recommendation system uses as inputs real reviews and ratings for hotels, as well as static hotel features and user’s choices, based on standard criteria.

Three different methods of analysis are used in generating recommendations. First is the Multicriteria Satisfaction Analysis (MUSA) which is used to define the preference model of customers who have previously chosen specific products/alternatives. Second is the Sentiment Analysis which is used in order to define sentiment score for each one of the alternatives. Third is the On/Off filtering, which is used so that the user of the system opts for specific characteristics of the alternatives, which he/she considers as prerequisites for the alternatives which will be recommended to her. This is used in combination with weight assessment through prioritization of Criteria (WAP). WAP is used in order to define the preference model of a user of the recommendation system.

Final recommendation comes as a result of the comparison of the preference model of the user of the recommendation system (WAP output), with the preference models of customers who have chosen specific alternatives (MUSA outputs). In the case of convergence of these models, there is a new set of alternatives. This set of alternatives is filtered by the On/Off user’s choices as well as by the sentiment threshold. The final subset of alternatives constitutes the recommended alternatives to the user.

While this algorithm follows a holistic approach is generating recommendations, it also does not account for the change in user’s interests with time. Our system overcomes this problem by capturing the timestamp of each of the user’s interaction and factoring it while generating recommendations.

Biswarup Ray et al [6] proposed a hotel recommendation system using sentiment analysis of hotel reviews, and aspect-based review categorization which works on the queries given by a user. They have followed a systematic approach which first uses an ensemble of a binary classification called Bidirectional Encoder Representations from Transformers (BERT) model with three phases for positive–negative, neutral–negative, neutral–positive sentiments merged using a weight assigning protocol. They then fed these pre-trained word embeddings generated by the BERT models along with other different textual features such as word vectors generated by Word2vec, TF–IDF of frequent words, subjectivity score, etc. to a Random Forest classifier. After that, they also grouped the reviews into different categories using an approach that involves fuzzy logic and cosine similarity. Finally, they created a recommender system using the aforementioned frameworks.

This algorithm suffers from the same limitations of the other sentiment-model based algorithm; a hotel with no reviews cannot be accounted for in the recommendations. In addition to this, this algorithm is purely content-based, it does not capture the mindset of the user and provides the same hotel recommendations for every user. Out algorithm overcomes these challenges by personalizing the recommendations for the user by using multiple attributes of the hotel interactions on the website.

Alia Karim Abdul Hassan and Ahmed Bahaa Aldeen Abdulwahhab [7] developed a location aspect-based sentiment analyser for hotel recommender system. These systems work by sensing the location of the person and suggest the best services to him in his area.

They built a location-sentiment based recommender system that depends on user location and previous visitor reviews, which give more details about hotel quality and services They believed that reviews would be more adequate than a rating system to reflect the opinion of users about a service. This was done using a hybrid sentiment analyser that consisted of two parts, a supervised sentiment analyser that was trained on hotel reviews and an unsupervised or lexicon-based analyser that could handle new reviews that were coming in with idioms and words not used in the training dataset.

This approach faces the same issues identified in the other sentiment-based models detailed in this survey. In addition to this, as a text-based processing system, its scope is confined to only English reviews and thus cannot provide recommendations for hotels in non-English speaking countries. Our recommender, on the other hand, is not bound by language restrictions and is designed to generate relevant recommendations for any country in the world.

Fatemeh Abbasi et al [8] followed a similar approach when they combined sentiment analysis with the Collaborative Filtering (CF) based on deep learning for user groups in order to increase system accuracy. The proposed system uses Natural Language Processing (NLP) and supervised classification approach to analyse sentiments and extract implicit features. More specifically, Multinomial Naive Bayes (MNB) and Linear Support Vector Classification (Linear SVC) from Python NLTK library were used for modelling. In order to design the recommender system, the Singular Value Decomposition (SVD) was used to improve scalability. This method again faces the same issues of the other sentiment-based models.

The hotel recommendation system’s literature survey revealed that most of the recommender algorithms rely on hotel review data and follow sentiment analysis-based approach to generate recommendations. While this does eliminate some of the problems of a traditional collaborative filtering algorithm, it creates new issues as mentioned in the previous paragraphs. These models also do not account for change in user’s interests with time and only provide static time independent recommendations. Our recommender system is however designed to account for these dynamic interests and also overcome the limitations of the existing methods by capturing user interactions and using this to understand the user’s mindset.

*Trip Recommendation Systems (POIs)*

Silamai et al [9] presented a recommendation system for tourists who do not have a trip plan when they visit Chiang Mai, Thailand. The recommendation is made based on the user’s preferred tourist destination type, images, current location, appropriate distance, time period, and place’s popularity. The system also recommends restaurants and coffee shops that are nearby each recommended attraction, and it also displays the suggested route with street views so that the user can get an idea of what the journey is going to look like. A few constraints were added to ensure user satisfaction: The distance between the hotel and the farthest destination does not to exceed 100km, each destination is less than 20km from the main route and the total time of travel must is not more than 10 hours.

This algorithm has a very limited scope as it was designed to generate only a one-day trip in one particular city (Chiang Mai). Moreover, the system does not account for any past experiences of the user in the same city as recommendations are provided based on user’s current inputs. Our algorithm overcomes these limitations as it can keep track of the user’s POI history and also encompasses over 10 million tourist attractions and facilities around the world.

WenJing Luan et al [10] proposed a maximal-marginal-relevance-based personalized trip recommendation method that considers both relevance and diversity of trips in trip planning. Specifically, a novel calculation method of POI similarity is presented based on a predefined category hierarchy, and then a new evaluation strategy is proposed to compute trip diversity. An ant-colony-optimization-based trip planning algorithm is then developed to efficiently plan a trip i.e., a sequence of POIs.

The proposed framework has two modules: POI scoring (offline phase) and trip planning (online phase). During the offline phase, POI similarity is calculated according to POI categories using their novel method. In addition, POI scores of users at different time slots can be obtained by using some existing methods, such as ranking-by-preference (RBP) and normalized-by-time (NBT) methods. During the online phase, when users propose requests, constraints-satisfied trips with higher qualities should be provided. The diversity and relevance of a trip are fused into trip quality according to users’ preferences, and then trip quality is used as the objective function during the planning procedure. In this study, trip relevance reflects the degree of users’ interest in a trip, which can be calculated by a trip score. Trip diversity reflects differences among POIs in a trip, which can be provided by using POI similarity.

Despite the various advantages provided by this algorithm, its recommendations are based on the assumptions that the user has previous history of check-ins at different POIs. This system thus suffers from a cold start problem as relevant recommendations cannot be provided for a new user unless he/she has made enough check-ins at different venues.

Kwan Hui Lim et al [11] on the other hand, proposed an algorithm for recommending personalized tours using POI popularity and user interest preferences, which are automatically derived from real-life travel sequences based on geo-tagged photographs. Their tour recommendation problem is modelled using a formulation of the Orienteering problem and considers user trip constraints such as time limits and the need to start and end at specific POIs.

They introduce the concept of time-based user interest for tour recommendation, where a user’s level of interest in a POI category is based on his/her time spent at such POIs, relative to the average user. They also compare this time-based user interest to the current practice of using frequency-based user interest and show how time-based user interests results in recommended tours that more accurately reflect real-life travel sequences. They also further enhance this time-based user interest by implementing an update rule such that user interests are refined based on the recency of their past POI visits. This updating works by giving more emphasis to recent POI visits than those in the more distant past. The algorithm also gives tourists the flexibility to indicate their preferred weightage between POI popularity and his/her interests, while also using two schemes to automatically determine an appropriate weightage based on the tourist’s activity level, relative to the general tourist population.

The given algorithm personalizes the trip planning by personalizing the visit duration at each POI. That is, the goal of this paper is to recommend a sequence of given POIs that fits within a certain budget that is calculated as the sum of travel time and personalized visit time. Whereas, the goal of our recommender algorithm is to personalize the trip planning by recommending a sequence of POIs for the user given his/her user history. We will thus be using this paper as a source of reference and inspiration.

Daniel Herzog et al [12] presented a mobile based recommender system for tourist trips and sequence of POIs along enjoyable routes. The system architecture is composed of three layers: presentation tier, application logic tier and data tier. The main motivation for this modular architecture is to facilitate the development and integration of new algorithms, clients and data sources.

The user requests a new tourist trip recommendation by specifying an origin, a destination, the starting time and the maximum duration of the trip. The system then collects POIs in the vicinity from the data sources, scores them according to the user’s preferences and context factors and combines them along an enjoyable route. The system can recommend tourist trips to individuals as well as groups. Group recommendations are generated by either aggregating user models or individual recommendations.

The drawback of this application is again the cold start problem. Users have to manually rate different categories of POIs before the recommendations become more relevant. Moreover, the application does not account for the change in user’s interests over time. Our recommender algorithm will be designed to overcome these two limitations.

Sara Migliorini et al [13] considered the trip planning problem that takes into account the balancing of users among the different POIs. To this aim, they considered the estimate of the level of crowding at POIs, including both the historical data and the effects of the recommendation. They then formulated the problem as a multi-objective optimization problem, and designed a recommendation engine that explores the solution space in near real-time, through a distributed version of the Simulated Annealing approach.

[14] built a road trip recommendation system using user preferences. This system takes source, destination and user’s interests / preferences as the input to generate the most optimal path from the source to the destination considering most of the places which falls under the users‟ interest along the way. This generated optimal path is displayed as recommended route map to the user including the intermediate places. In addition to generating the most optimal path, this system also recommends the best season to visit the places selected in the route using decision tree classifier.

This system has a limited scope as it is bound by the range of values in the dataset. It also does not keep track of user history or change in user’s interest with time. Our recommender algorithm will overcome these shortcomings by using a powerful API that can give it access to over 10 million places around the world. Our algorithm will also capture user ‘s history and also account for dynamic interests while providing recommendations.

Amine Dadoun et al [15] proposed an approach that leverages contextual, collaborative and content information in order to recommend personalized destinations to travellers. They present a deep neural network model that takes as input these three different types of information and generates recommendations. Given a traveller, his demographics information (age, nationality, etc.), his historical bookings and the contextual data related to those bookings (day of week, number of passengers, stay duration, etc.), the aim of this paper was to recommend to this traveller a ranked list of destinations he would like to go to (A destination is represented by a city that has an airport).

This paper also suffered from the data sparsity problem. Although they had over 2 million booking history data, they had a very sparse traveller feedback matrix. Filtering out travellers that fit this matrix led to only 300 bookings from 26 unique travellers. Our algorithm overcomes this problem as it captures different types of user interactions in order to understand user mindset, thereby eliminating the need for a user-rating matrix, which can often be very sparse. Moreover, this paper only recommends travel destinations whereas our algorithm recommends POI categories that the user would like to visit at these destinations.

The Trip recommendation system’s literature survey revealed that most of the algorithms suffer from the cold start and data sparsity problem as they rely on user-rating matrix in order to generate recommendations. Moreover, some of these algorithms are limited in the number and types of POIs they can recommend and do not account for change in user’s interest with time. Our proposed system will be designed to overcome these limitations by capturing user interactions on the website and using that to generate time-relevant recommendations that are in line with the user’s current interests.

Proposed Method

Our proposed system is broken down into three models, each one assisting a different functionality on the website. These three models are: Hotel recommender, POI (places of interest) recommender and Blog recommender. These models are built on top of the knowledge base gathered by the AI through the various interactions on the website. The structure of this knowledge base is detailed in the following section.

The Knowledge Base – User Profile

The knowledge base of the AI is represented as a “user profile” in the database of the website. Every user on the website has an associated user profile. This profile contains all the information that the AI has gathered about the user. This user profile is made up of six attributes:

* *Hotels:* This is the list of all hotel interactions made by the user on the website. Each hotel interaction is stored as an object containing the following features
  + *Hotel Id:* The Id of the hotel
  + *City Id:* The Idof the city in which the hotel is situated
  + *Weight:* The priority level assigned to the captured hotel interaction.
  + *Timestamp:* The time at which the weight was captured
  + *Amenity Ids:* A string of numbers indicating the various amenity filters used to pick out the hotel
  + *Theme Ids:* A string of numbers indicating the various theme filters used to pick out the hotel
  + *Evaluated:* A Boolean value that tells the AI if the latest hotel capture has been accounted for or not.
* *Interest Vector:* This attribute is the weighted total of all hotel interactions made by the user and is representative of the user’s interest towards each aspect of a hotel. It is used by the AI to make hotel recommendations and is an object that is made up of the following features:
  + *Stars:* A number that represents the weighted average star rating of the hotels that the user has interacted with.
  + *Price:* A number that represents the weighted average price per room per night of the hotels that the user has interacted with.
  + *Guest review:* A number that represents the weighted average guest review of the hotels that the user has interacted with.
  + *Amenities:* A string of numbers where each number indicates the user’s inclination toward that particular amenity. A positive value indicates a liking and a negative value indicates a disliking.
  + *Themes:* A string of numbers where each number indicates the user’s inclination toward that particular theme. A positive value indicates a liking and a negative value indicates a disliking.
  + *Facility filters:* A list of all the amenity filters that the user has used while filtering out hotels in the website.
  + *Theme filters:* A list of all the theme filters that the user has used while filtering out hotels in the website.
* *Blogs:* This is the list of all the blog interactions made by the user on the website. Each blog interaction is stored as an object containing the following features:
  + *Blog Id:* The Id of the captured blog
  + *Weight:* The priority level assigned to the captured blog interaction
  + *Timestamp:* The time at which the weight was captured
  + *Evaluated:* A Boolean value that tells the AI if the latest hotel capture has been accounted for or not.
* *Blog Interest:* This is a list of three features that are specific to blogs and is used while making blog recommendations. These three features are:
  + *Cost:* The cost of the trip mentioned in the blog interaction
  + *Duration:* The duration of the trip mentioned in the blog interaction
  + *Likes:* The number of likes received by the blog
* *POI:* This is the list of all the POI interactions made by the user on the website. Each POI interaction is stored as an object containing the following features:
  + *XID:* The Id of the place of interest
  + *Weight:* The priority level assigned to the captured POI interaction
  + *Timestamp:* The time at which the weight was captured
  + *Categories:* A list of categories that the captured place of interest can be broken down into.
* *POI Categories:* This attribute is a list that stores the top three favourite categories of POI of the user. It is used by the AI to make POI recommendations.

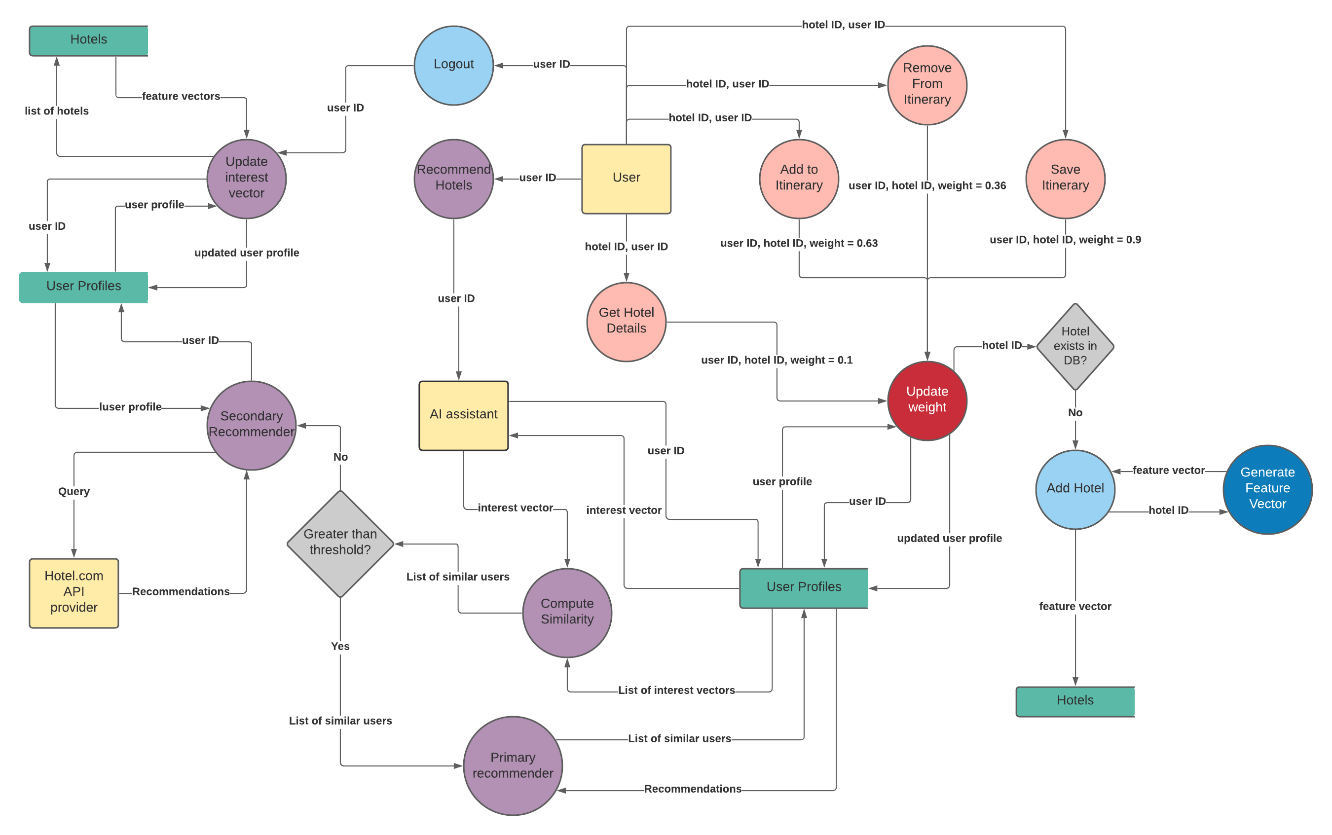
*Hotels*, *POI* and *Blogs* store the all the interactions made by the user. *Interest Vector*, *POI Categories* and *Blog Interest* represent what the AI knows about the user based on these interactions and is updated every time after the user logs out of the website. These attributes are used by the AI to make recommendations.

Hotel Recommender

The data flow diagram given below represents the working of the hotel recommendation system. Four different types of hotel interactions are captured by the AI:

* When the user clicks on a hotel to view more details, that interaction is captured with a weight of 0.1
* When the user adds a hotel to an itinerary, that interaction is captured with a weight of 0.63
* However, if the user removes a hotel from an itinerary, that interaction is captured with a weight of 0.36
* Finally, when the user decides to save an itinerary, all hotels part of the itinerary will be updated to a weight of 0.9

These captures are performed by the *Update Weight* function. At this juncture, it can be mentioned that since the website relies on API hotel data, the AI does not have a hotel database to use. Therefore, every hotel interaction on the website is followed by breaking down that hotel into its basic attributes (known as a feature vector), and storing the feature vector to build a hotel database. The AI then uses this hotel database to power its primary hotel recommender (which will be explained in the next section).



*Function: Update Weight*

This function first checks if the given hotel exists in the database or not. If it doesn’t it calls the *Add Hotel* function. It then locates the user profile of the given user. If the user has not interacted with this hotel before, it creates a new hotel entry with the timestamp. Otherwise, it locates the existing entry in the user profile and updates the weight and the timestamp along with the amenity and theme filters. The Boolean value *evaluated* is set to *false* to let the AI know that this interaction has to be accounted for while updating the user profile.

*Function: Add Hotel*

This function adds the given hotel to the database. It first calls the *Generate Feature Vector* function to get the corresponding feature vector of the hotel and stores the feature vector in the database. The reasons for generating a feature vector are two-fold: one, it is used by the primary recommender to make recommendations and two, storing the raw data of every hotel can be very memory consuming; breaking it down into its basic features and storing it is a memory saving alternative.

*Function: Generate Feature Vector*

This function breaks down the given hotel details into its basic features (34 features, to be precise). This is done by analysing the hotel API data and looking for keywords and phrases that can indicate the presence or absence of a certain feature. The list of features is given below (each feature is also numerically mapped to a unique value)

Facilities

* 24-hour front desk - 2063
* Smoking areas - 537
* Bar - 515
* Breakfast included - 2048
* Airport transfers - 513
* Parking available - 16384
* Business facilities - 519
* Meeting facilities - 1
* Restaurant - 256
* Pool - 128
* Spa - 539
* Pet friendly - 64
* Wifi included - 527
* Childcare - 521
* Gym - 2
* Bathtub in room - 517
* Connecting rooms available - 523
* Casino - 2112

Themes

* Beach - 6
* Business - 14
* Casino - 8
* Family-friendly - 25
* Luxury - 15
* Spa hotel - 27

Accessibility

* Accessibility equipment for the deaf - 2097152
* Accessible bathroom - 131072
* Accessible parking - 524288
* Accessible path of travel - 65536
* Braille or raised signage - 4194304
* In-room accessibility - 1048576
* Roll-in showers - 262144
* Wheelchair accessible rooms - 541

The presence of a feature from the above list is indicated by a value of 1 and the absence is indicated by a value of 0. In addition to facilities, themes and accessibility, the price, star rating and guest review is also stored in the feature vector.

*Function: Recommend Hotels*

At this point, it can be mentioned that these recommender models are made up of two smaller models: A Primary recommender and a Secondary recommender.

The reason for this is that the Primary recommender relies entirely on the users’ interaction on the website; the more the number of users and more the number of interactions, the better the recommendations. This can lead to a cold start problem when the website is first launched due to insufficiency of users. This is where the Secondary recommender comes in.

The Secondary recommender acts as a backup when there are insufficient users or when the similarity between users isn’t good enough. This recommender only takes into account the given user’s interest and uses just that to make recommendations.

Therefore, when the user asks for recommendations, the AI gets the interest vector of that user and the interest vectors of all other users on the website. If the number of users is insufficient, the AI resorts to the Secondary recommender. Otherwise, it calculates the cosine similarity between the given user and the other users based on the interest vectors. It then gets the top 5 most similar users and passes this information on to the Primary recommender. However, if the similarity between the users doesn’t cross a certain threshold, the AI again resorts to the Secondary recommender.

*Function: Primary Recommender*

The primary recommender uses the list of similar users to the given user to make recommendations. It first gets the hotels interacted with by the similar users (which can be found in their user profiles) and filters out the hotels that are not a part of the city for which recommendations have to generated. It then sorts the remaining hotels by the associated weights and returns the result as recommendations.

*Function: Secondary Recommender*

The secondary recommender uses the user’s user profile alone to build an API call to retrieve a personalized list of hotels. This is done by retrieving the most commonly used facility filters and theme filters used by the user and the average price, star rating and guest rating of every hotel interaction made by the user. This information will be put together to build an API call to the website’s third-party hotel data provider and the results will be returned as recommendations.

*Function: Update Interest Vector*

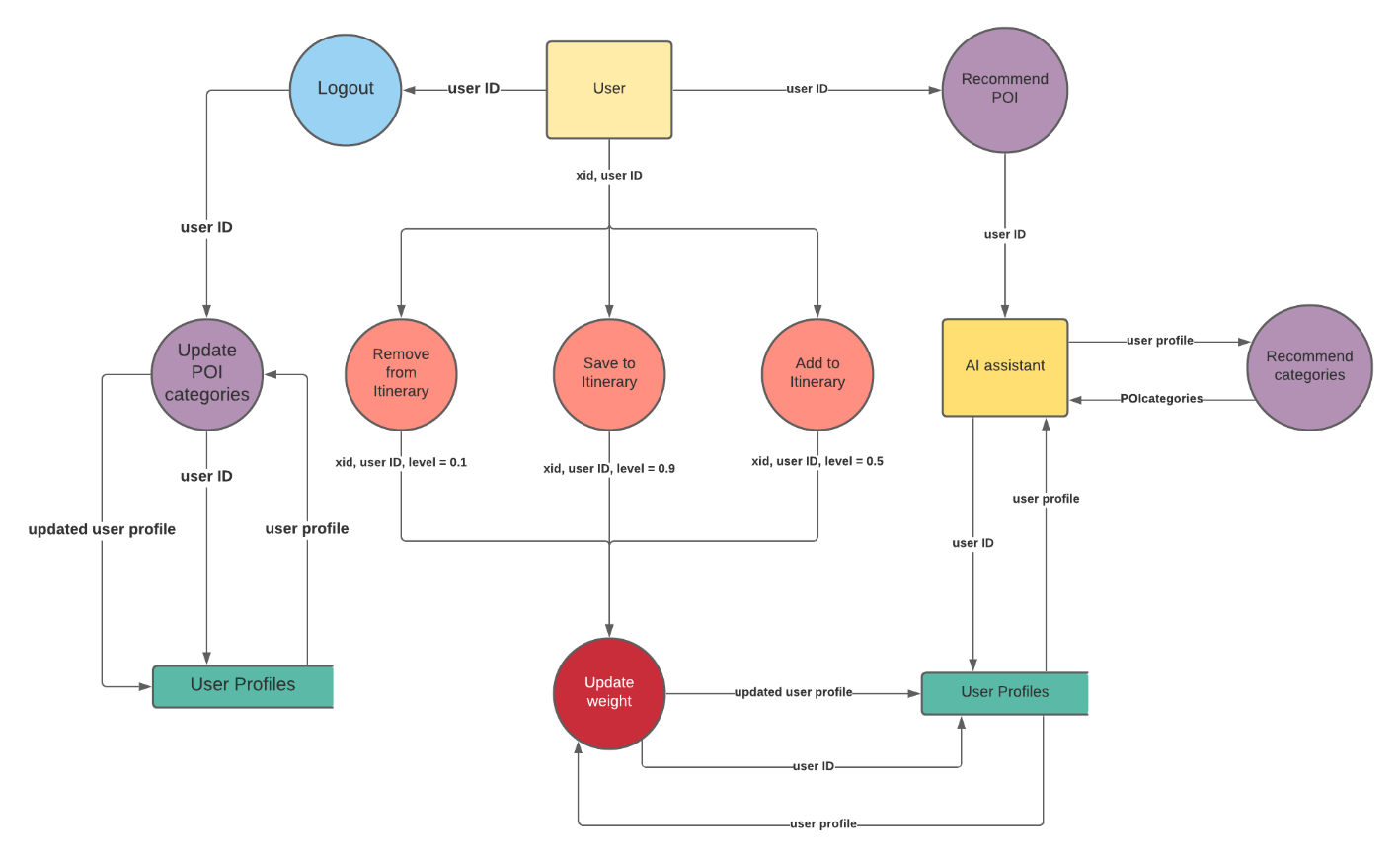
When a user logs out, the interactions made by that user during that session have to be accounted for. This is the purpose of the *Update Interest Vector* function. It first locates the user profile of the user and extracts the hotels that have not been accounted for. This can be done by locating those hotels where *evaluated* has been set to *false*. It then proceeds to update the various attributes of the interest vector, starting with the facility filters and the theme filters.

The stars, price, and guest review attributes are all features that represent a weighted average of the user’s interest with the weight being a combination of timestamp and the captured weight. These are thus updated by calling a *weighted average* function that updates the average value.

On the other hand, attributes such as themes and facilities are features that represent a weighted count of the user’s interest (with the weight being the same as before). The reason it’s a weighted count and not a weighted average is because the amenities and themes are a string of 1s and 0s and thus cannot be added but rather have to be counted. Hence these are updated by calling a *weighted count* function that updated the count value.

By considering the timestamp as part of the weight, the model captures the user’s changing interest with time and makes sure the recommendations are in line with the user’s current interests. Once all the updates have been completed, the *evaluated* status of all the hotels is set to *true*.

POI Recommender

****

The above data flow diagram represents the working of the POI recommendation system. Unlike the hotel recommendation system (or the blog recommendation system), the POI recommender is fairly simple mainly due to the amount of information it has to capture and analyse from every POI. In this case, three different types of POI interactions are captured by the AI:

* When the user saves a POI to the itinerary, that interaction is captured with a weight of 0.5
* If the user removes a POI from an itinerary, that interaction is captured with a weight of 0.1
* Finally, when the user decides to save an itinerary, all POIs part of the itinerary will be updated to a weight of 0.9

These captures are again performed by an *Update Weight* function. This function works very similar to that of the hotel recommender model with a few exclusions

*Function: Update Weight*

For POIs, the only information we need to store is the type of categories it can fall and hence we do not need to maintain a separate database for this purpose. Similar to the hotel recommender model, the *Update Weight* function locates the user’s user profile and updates the weight of the capture (or creates a new entry if the POI does not exist). The only addition here is that the various categories that the given POI belongs to is obtained by making an API call to our third-party POI data provider and that information is stored as part of the user profile itself.

*Function: Recommend POI*

This recommender model, unlike the other two, is not made up of a Primary and Secondary component. The reason is because the recommendations are only based on one feature (the categories) and thus does not necessitate the usage of the similar user’s approach. Instead, the user’s POI history alone is sufficient to generate relevant recommendations.

*Function: Recommend Categories*

This function first locates the user profile of the given user and extracts the list of POI interactions. It combines the weight of these interactions along with the timestamp to get a time-based weight. It then compiles a list of all the categories that all the POI interactions are a part of and assigns the respective time-based weight to each of these categories. It finally returns those categories with the highest weight as the recommendations.

By considering the timestamp as part of the weight, the model captures the user’s changing interest with time and makes sure the recommendations are in line with the user’s current interests.

*POI mapping system*

The different POI categories (over a 100 categories) are stored as a tree structure where every child is a subcategory of its parent (for example, “beaches” are a subcategory of the “natural” category). The mapping system follows three rules:

* The scale of the number of a node (number of digits to the right of a decimal point) is equal to the level of the node (for example, nodes at level 2 of the tree will have 2 digits to the right of the decimal point).
* Child nodes inherit values from the parent nodes.
* Nodes at one level are numbered starting from the left extreme of the level.

Using these rules, say “natural” is a child node of “interesting places” (value 3) and is the fourth node on its level (level one). Its value will therefore be 3.4. Similarly, if “beaches” was the first child node of “natural”, then its value would be 3.41.

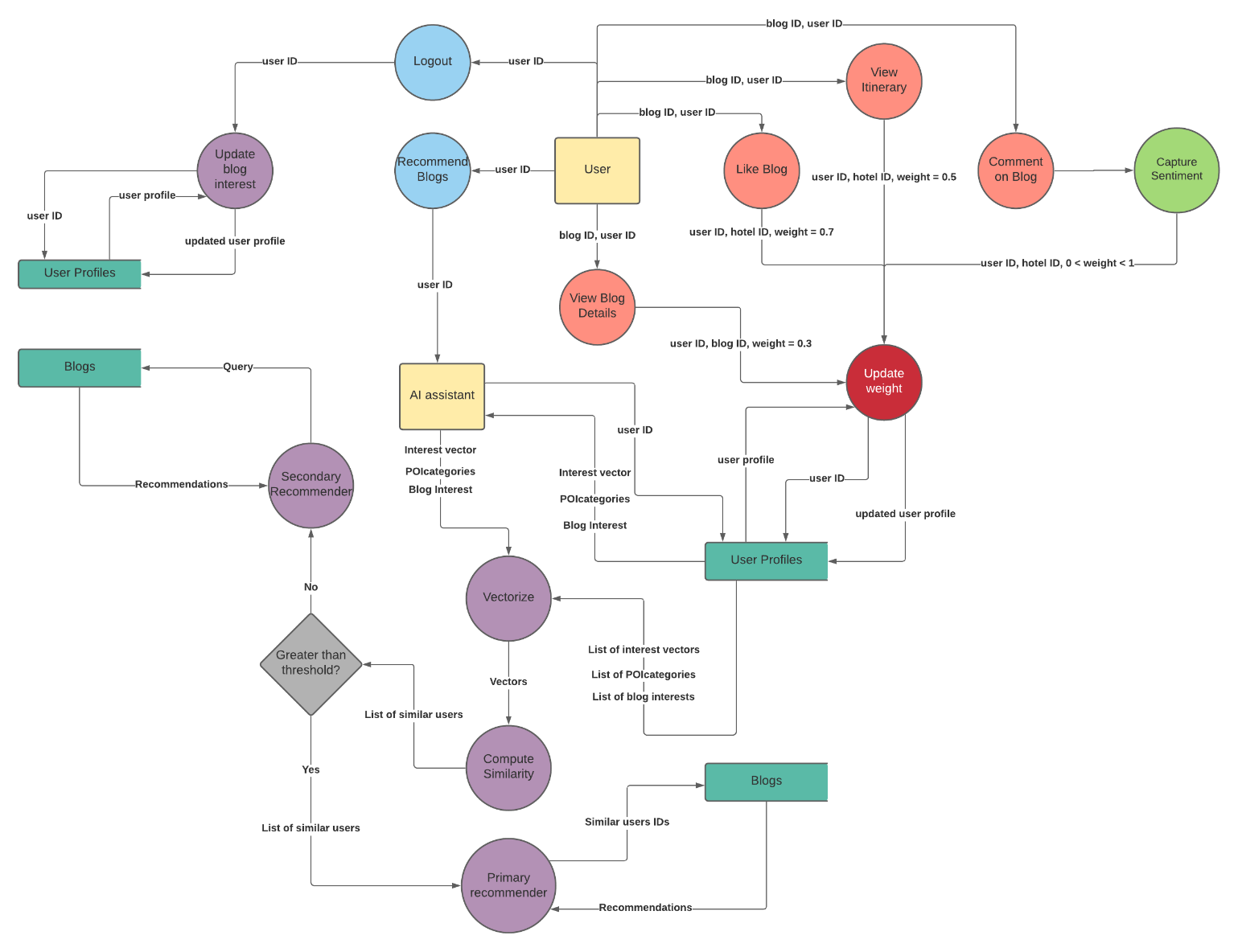
*Function: update POI categories*

Although not used to make recommendations, the *POI Categories* attribute of the user profile stores the top 3 most preferred categories of the user and this function is used to update that list every time after the user logs out. These categories are stored as numeric values (for reasons that will be explained in the blog recommender). These numeric values are obtained through the unique mapping system that ensures that similar categories have similar values.

The reason this attribute is not used to make recommendations is because the approach mentioned in the recommender can make sure that the interactions of the user in the current session are also accounted for (which is harder in the case of blogs and hotels due to the sheer number of attributes).

The *Update POI categories* function instead calls the *recommend categories* function to get the most preferred categories of the user. It then uses the above-described mapping system to store the equivalent numeric values of the top 3 preferred categories in the user profile of the user.

Blog Recommender

****

The above data flow diagram represents the working of the blog recommendation system. It can be quickly seen that the blog recommender is an extension of the hotel recommender model. Four different types of blog interactions are captured by the AI:

* When the user opens a blog to read the article, that interaction is captured with a weight of 0.3
* When the user opens the tab to view the itinerary associated with the blog article, that interaction is captured with a weight of 0.5
* If the user likes a blog, that interaction is captured with a weight of 0.7
* The AI can also capture the sentiment behind the comments made by the user on the blog. This sentiment is assigned to be the weight of the comment interaction and can vary from 0 to 1 depending on the nature of the comment.

*Sentiment Analysis*

As mentioned above, the AI can also capture the sentiment behind every comment made by the user on the blogs and use this to better understand the user mindset. This is done with the help of a machine learning sentiment analyser model that has been hosted on the cloud. This model takes in a comment as an input and returns a value between 0 and 1 depending on the type of comment (positive or negative).

*Function: Update Weight*

This function, much like the others, simply captures the above-mentioned interactions into the user’s profile. It creates a new entry if the blog interaction does not exist or update the existing entry’s weight and timestamp value. It then sets the Boolean variable *evaluated* to *false* to let the AI know that this interaction has to be accounted for while updating the user profile.

*Function: Recommend Blogs*

The blog recommender is also made up of a Primary and Secondary recommender (for reasons already mentioned in the Hotel recommender model). When the user clicks on the recommend blogs button, the AI locates the user’s user profile and the user profile of all other users. It then extracts three attributes from these user profiles: the *interest vector*, which represents the user’s hotel interests, the *POI categories*, which represents the user’s most preferred sightseeing places, and the *blog interest*, which is representative of the kind of blogs that the user has frequently interacted with. It then passes these three attributes to the *vectorize* function that groups these features together and returns a vector that represents the AI’s complete understanding of the user.

The rest of the procedure is similar to the hotel recommender model. The AI compares the cosine similarity between these vectors and if the similarity does not cross a certain threshold, or if the number of users for comparison are insufficient, the AI resorts to the Secondary recommender. Otherwise, it uses the Primary recommender.

*Function: Vectorize*

As mentioned above, the vectorize function takes in three parameters: the interest vector, the blog interest and the POI categories (represented as numbers), and groups them into one 40-dimensional vector. This is the reason why the POI categories had to be mapped to a number system: in order to facilitate the cosine similarity calculation. This 40-dimensional vector represents the AI’s complete understanding of the user’s mindset and is used to make blog recommendations.

*Function: Primary Recommender*

The primary recommender, similar to the hotel recommender model, makes use of user similarity to generate recommendations. It takes the top 5 similar user’s user profiles, and extracts all their blog interactions and blogs posted. These blogs are then sorted by weight and returned as recommendations.

*Function: Secondary Recommender*

The secondary recommender, being the backup, simply gathers all the blogs in the database and sorts them by the number of likes received. It however, also accounts for the user’s interest by factoring in the average cost and duration of each blog interaction into the sorting process. The final results are then returned as recommendations.

*Function: Update Blog Interest*

This function servers a similar purpose to the *update interest vector* function. When a user logs out, the blog interactions made by that user during that session must also be accounted for. This is done by first locating the user profile of the user and extracting the blogs that have not been accounted for. This can be done by locating those blogs where *evaluated* has been set to *false*.

This is then followed by updating the various attributes of the *blog interest vector* which includes cost, duration and likes. These are features that represent a weighted average of the user’s interest with the weight being a combination of timestamp and the captured weight. These are thus updated by calling a *weighted average* function that updates the average value.

By considering the timestamp as part of the weight, the model captures the user’s changing interest with time and makes sure the recommendations are in line with the user’s current interests. Once all the updates have been completed, the *evaluated* status of all the hotels is set to *true*.

Performance Analysis

Various aspects of the website were looked into in order to evaluate the performance of the website. This included analysing the time complexity of the different recommenders in order to understand how these functions would scale with the website. This was followed by a graphical analysis of the primary and secondary components of the recommenders in order to gauge the performance of the AI as the number of users (and user interactions) increased in the website. This was consequently followed by a website performance analysis wherein the response time and other metrics where evaluated.

Time Complexity Analysis

In order to assess the time complexity of the recommenders, we have to first evaluate the complexity of the underlying utility functions. All API calls take a constant amount of time to run whereas the MongoDB find queries take a linear time to run. Using this, the time complexities of the other utility functions can be evaluated as shown in the table below.

*Hotel Recommender Time Complexity*

|  |  |  |
| --- | --- | --- |
| **Function** | **Time complexity** | **Description** |
| Add Hotel  Generate Feature Vector | Constant – O(k) | k is the length of the feature vector. |
| Update Weight | Linear – O(n) | n = x + y + z where x is the number of hotels in the DB, y is the number of user profiles, and z is the number of hotel interactions made by that user profile. |
| Weighted average  Weighted Count  Vectorize | Constant – O(k) | k is the length of the interest/ feature vectors. |
| Primary recommender | Linear – O(n) | n is the number of hotel interactions made by the similar users |
| Secondary recommender | Linear – O(n) | n is the number of hotel interactions made by the target user. |
| Recommend hotels | Linear – O(n) | n = x + y, where x is the number of user profiles in the DB and y is derived from either the primary or secondary recommender |
| Update Interest Vector | Linear – O(n) | n = x + y + z where x is the number of hotels in the DB, y is the number of user profiles, and z is the number of hotel interactions made by that user profile. |

From the above table, it can be seen that the hotel recommender system has a linear time complexity and grows with respect to the number of user profiles in the system and the number of hotel interactions made by these user profiles.

*POI Recommender Time Complexity*

|  |  |  |
| --- | --- | --- |
| **Function** | **Time complexity** | **Description** |
| Update Weight | Linear – O(n) | n = x + y where x is the number of user profiles in the DB and y is the number of POI interactions made by that user profile. |
| Recommend Categories | Linear – O(n) | n = x + y where x is the number of user profiles in the DB and y is the number of POI interactions made by that user profile. |
| Update POI Categories | Linear – O(n) | n is derived from the recommend categories function |

From the above table, we can see that the POI recommender also has linear time complexity and grows with respect to the number of user profiles in the DB and the number of POI interactions made by the users.

*Blog Recommender Time Complexity*

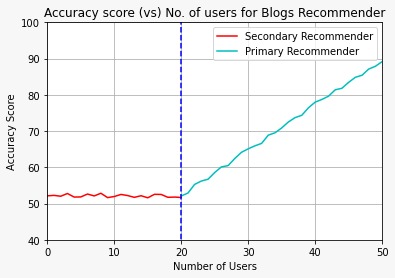
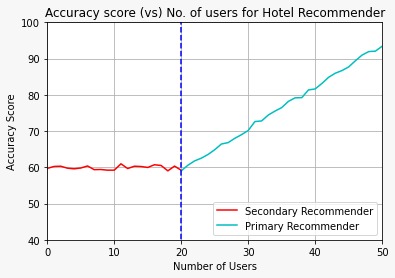
|  |  |  |
| --- | --- | --- |
| **Function** | **Time complexity** | **Description** |
| Update Weight | Linear – O(n) | n = x + y + z where x is the number of hotels in the DB, y is the number of user profiles, and z is the number of hotel interactions made by that user profile. |
| Weighted average  Vectorize | Constant – O(k) | k is the length of the blog interest vectors |
| Primary recommender | Linear – O(n) | n is the number of blog interactions made by the similar users |
| Secondary recommender | Linear – O(n) | n is the number of blogs in the website |
| Recommend Blogs | Linear – O(n) | n = x + y, where x is the number of user profiles in the DB and y is derived from either the primary or secondary recommender |
| Update Blog Interest | Linear – O(n) | n = x + y + z where x is the number of blogs in the DB, y is the number of user profiles, and z is the number of blog interactions made by the user profile. |

From the above table it can be seen that the blog recommender, not unlike the hotel recommender, also has a linear time complexity and grows with respect to the number of blogs and users in the DB and the number of blog interactions made by these users.

Hence, it can be concluded that the recommender system of the website has a linear time complexity that grows with respect to the number of users in the website and the number of interactions made by these users.

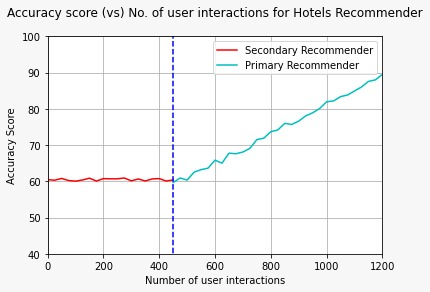
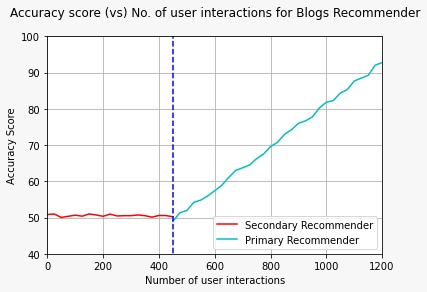
Primary vs Secondary Recommender

In order to determine the accuracy of the recommender systems, a simple feedback mechanism was setup wherein users of the website were asked to rate the satisfaction of their recommendation results on a scale of 1-5. This was translated into an accuracy score and used to evaluate the primary and secondary recommender components.



From the graphs given above, we can see that the accuracy of the primary recommender increases steadily as the number of users increase (more users mean more user interactions and hence stronger user profiles). We can also notice the spike in accuracy as the recommender shifts from secondary to primary. Since, the primary recommender uses the concept of user similarity, it is able to provide more accurate recommendations.

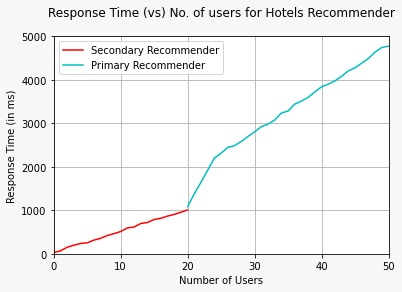
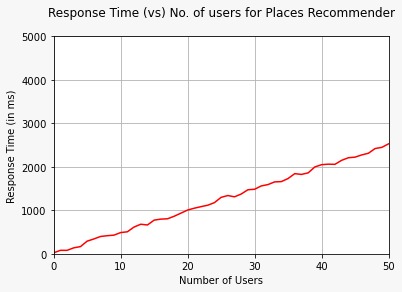
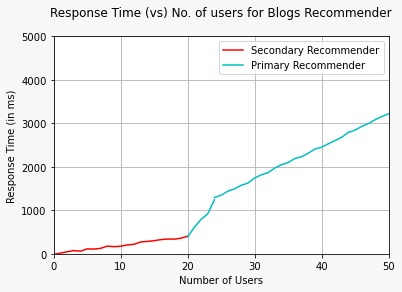
The reason why the threshold criteria is so high is because the primary recommender needs to have enough user interaction in the knowledge base in order to make accurate recommendations (Especially the hotel recommender, which builds the hotel DB itself only through user interactions). The graph given below depicts how the accuracy changes for both the primary and secondary recommenders as the number of user interactions on the website increases.



We can see that the primary recommenders rely more on the number of user interactions than the secondary recommenders. This is why the secondary recommender is used as the backup and is launched along with the website. It can avoid the cold start problem as it is not heavily dependent on the level of user interaction on the website. Another thing to note is that the slope of the blog recommender is greater than that of the hotel recommender. This can be attributed to the fact that the blog recommender accounts for more features than the hotel recommender while making recommendations.

Website Performance Analysis (Load testing)

Since the AI is integrated into the backend of the travel planner website, various tests were conducted to check the speed and response time and thereby the scalability and reliability of the application under different workloads.

From the above graphs, we can see that the hotel recommender has the slowest response time of all three. This can be attributed to the fact that the hotel recommender functions involve a greater number of API calls than its POI and blog counterparts. However, all the three recommender models scaled well with the workload (number of users) and did not fall short of its expected performance.

The website was designed to work in real-time, which means multiple users can simultaneously generate recommendations and changes in one user profile is almost immediately accounted for in the other user’s recommendations.

Comparative Analysis

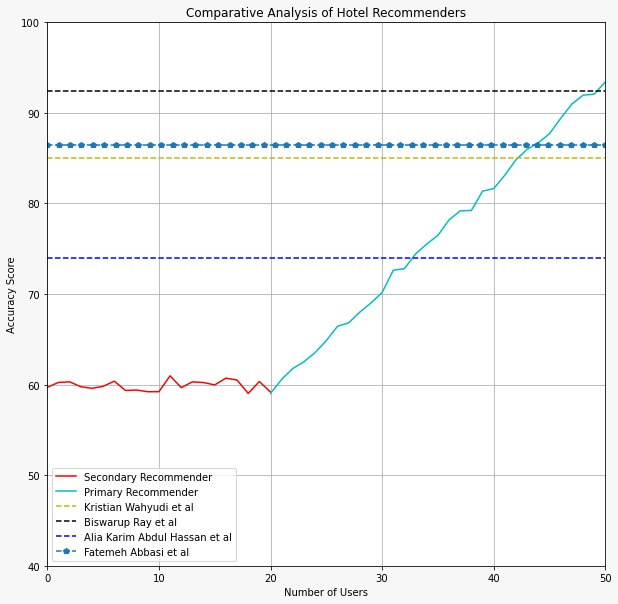
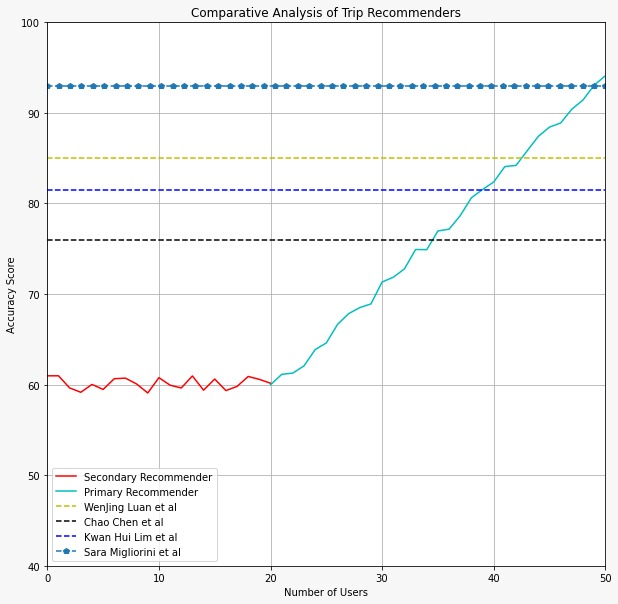
In this section, the results of the proposed method were compared with that of the various other existing methods (with similar metrics) detailed in the literature survey. Four different hotel recommenders and trip recommenders were considered for this purpose. The tables given below provides information about these existing methods and their results.

*Hotel Recommenders for Comparative Analysis*

|  |  |  |
| --- | --- | --- |
| **Author** | **Paper** | **Accuracy** |
| Kristian Wahyudi et al | Content-based hotel recommender | 85% |
| Biswarup Ray et al | An ensemble-based hotel recommender system using sentiment analysis and aspect categorization of hotel reviews | 92.36% |
| Alia Karim Abdul Hassan et al | Location aspect-based sentiment analyser for hotel recommendations | 74% |
| Fatemeh Abbasi et al | A Grouping Hotel Recommender System Based on Deep Learning and Sentiment Analysis | 86.5% |

*Trip Recommenders for Comparative Analysis*

|  |  |  |
| --- | --- | --- |
| **Author** | **Paper** | **Accuracy** |
| WenJing Luan et al | A Maximal-Marginal-Relevance-Based Personalized Trip Recommendation Method | 85% |
| Chao Chen et al | Personalized Trip Planning Leveraging Heterogeneous Crowdsourced Digital Footprints | 76% |
| Kwan Hui Lim et al | Personalized trip recommendation for tourists based on user interests, points of interest visit durations and visit recency | 81.5% |
| Sara Migliorini et al | Adaptive Trip Recommendation System: Balancing Travelers Among POIs with MapReduce | 93% |



From the graphs given above, we can see that the accuracy of the proposed system eventually surpasses the accuracy of the existing methods as the number of users (and consequently user interactions) increases. This can be attributed to the fact that our system is essentially a knowledge-based agent, and hence as the number of user interactions increases, the knowledge base of the AI expands. This helps the AI to understand its users better and provide more accurate recommendations that are in-line with their current interests.

Testing

Various types of testing were performed to check if our product matched the expected requirements and to make sure that it was defect free. Through software testing, we were able to produce an application that was infallible, reliable, secure and delivered high performance. This testing was integrated into the development process to ensure quality control at every step of the way. The different types of testing that were performed can be grouped under the category of functional testing.

Functional testing: The system was tested against the functional requirements and specifications. It verified the operations and actions of the application by testing *what* the product does. The different types of functional testing performed are:

* Unit Testing
* Integration Testing
* System Testing
* Interface Testing

Unit Testing

This is the first stage of testing where each individual unit of the application was tested with different test cases using a *white-box testing* approach (wherein the internal code structure and the flow of each unit was tested rigorously). In order to do this, each callable function that made up the different recommender models was considered as an individual unit. The success of unit testing was critical to the working of the application as these units formed the base of the entire model.

*Hotel Recommender*

The different functions (units) that were tested in the hotel recommender model are:

* The *update weight* function, that captures different types of hotel interactions
* The *recommend hotels* function, that calculates similarity scores between users and appropriately chooses either the primary or secondary recommender
* The *update interest vector* function, that accounts for a user session’s hotel interactions after logout.

*POI Recommender*

The different functions (units) that were tested in the POI recommender model are:

* The *update weight* function, that captures different types of POI interactions
* The *recommend POI* *categories* function, that returns a list of recommended POI categories
* The *update POI categories* function, that accounts for a user session’s POI interactions after logout

*Blog Recommender*

The different functions (units) that were tested in the blog recommender model are:

* The *update weight* function, that captures different types of blog interactions
* The *capture sentiment* function, that captures the sentiment behind a given comment
* The *recommend blogs* function, that calculates similarity scores between users and appropriately chooses either the primary or secondary recommender
* The *update blog interest* function, that accounts for a user session’s blog interactions after logout.

Each of the above-mentioned functions take in a set of parameters and return an output. Therefore, in order to test each of the functions, a combination of valid and invalid arguments was passed as parameters and the returned output was compared with the expected result. The summary of the tests performed for each of the different functions is given below.

*Hotel Recommender Unit Testing*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| *Update Weight (hotel Id, city Id, user Id, weight, filters)* | | | |
| A list of valid arguments | “Hotel weight captured as <weight>” | “Hotel weight captured as <*weight*>” | Pass |
| Invalid hotel Id | “Hotel does not exist” | “Hotel does not exist” | Pass |
| Invalid city Id | “Unable to retrieve city information” | “Unable to retrieve city information” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |
| Invalid weight | “Invalid interaction” | “Invalid interaction” | Pass |
| *Recommend Hotels (user Id, city, filters)* | | | |
| A list of valid arguments passing the check conditions | “Using primary recommender” | “Using primary recommender” | Pass |
| A list of valid arguments passing the check conditions with not enough hotels in the DB | “Primary recommender failed, resorting to secondary” | “Primary recommender failed, resorting to secondary” | Pass |
| A list of valid arguments with less than 5 users in the DB | “Resorting to secondary recommender, not enough users” | “Resorting to secondary recommender, not enough users” | Pass |
| A list of valid arguments with very dissimilar users in the DB | “Resorting to secondary recommender, low similarity threshold” | “Resorting to secondary recommender, low similarity threshold” | Pass |
| Invalid city Id | “Unable to retrieve city information” | “Unable to retrieve city information” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |
| *Update Interest Vector (user Id)* | | | |
| Valid user Id | “Interest vector updated” | “Interest vector updated” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |

*POI Recommender Unit Testing*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| *Update Weight (user Id, weight, xid)* | | | |
| A list of valid arguments | “POI weight captured as <weight>” | “POI weight captured as <*weight*>” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |
| Invalid xid | “POI with given xid does not exist” | “POI with given xid does not exist” | Pass |
| Invalid weight | “Invalid interaction” | “Invalid interaction” | Pass |
| *Recommend POI Categories (user Id)* | | | |
| Valid user Id | A list of recommended categories | A list of recommended categories | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |
| *Update POI Categories (user Id)* | | | |
| Valid user Id | “POI categories updated” | “POI categories updated” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |

*Blog Recommender Unit Testing*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| *Update Weight (blog Id, user Id, weight)* | | | |
| A list of valid arguments | “Blog weight captured as <weight>” | “Blog weight captured as <*weight*>” | Pass |
| Invalid blog Id | “Blog does not exist” | “Blog does not exist” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |
| Invalid weight | “Invalid interaction” | “Invalid interaction” | Pass |
| *Recommend Blogs (user Id)* | | | |
| A list of valid arguments passing the check conditions | “Using primary recommender” | “Using primary recommender” | Pass |
| A list of valid arguments with less than 5 users in the DB | “Resorting to secondary recommender, not enough users” | “Resorting to secondary recommender, not enough users” | Pass |
| A list of valid arguments with very dissimilar users in the DB | “Resorting to secondary recommender, low similarity threshold” | “Resorting to secondary recommender, low similarity threshold” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |
| *Classify Comment (comment)* | | | |
| Positive comment | Weight between 0.5 and 1 | Weight between 0.5 and 1 | Pass |
| Negative comment | Weight between 0 and 0.5 | Weight between 0 and 0.5 | Pass |
| *Update Blog Interest (user Id)* | | | |
| Valid user Id | “Blog interest updated” | “Blog interest updated” | Pass |
| Invalid user Id | “User does not exist” | “User does not exist” | Pass |

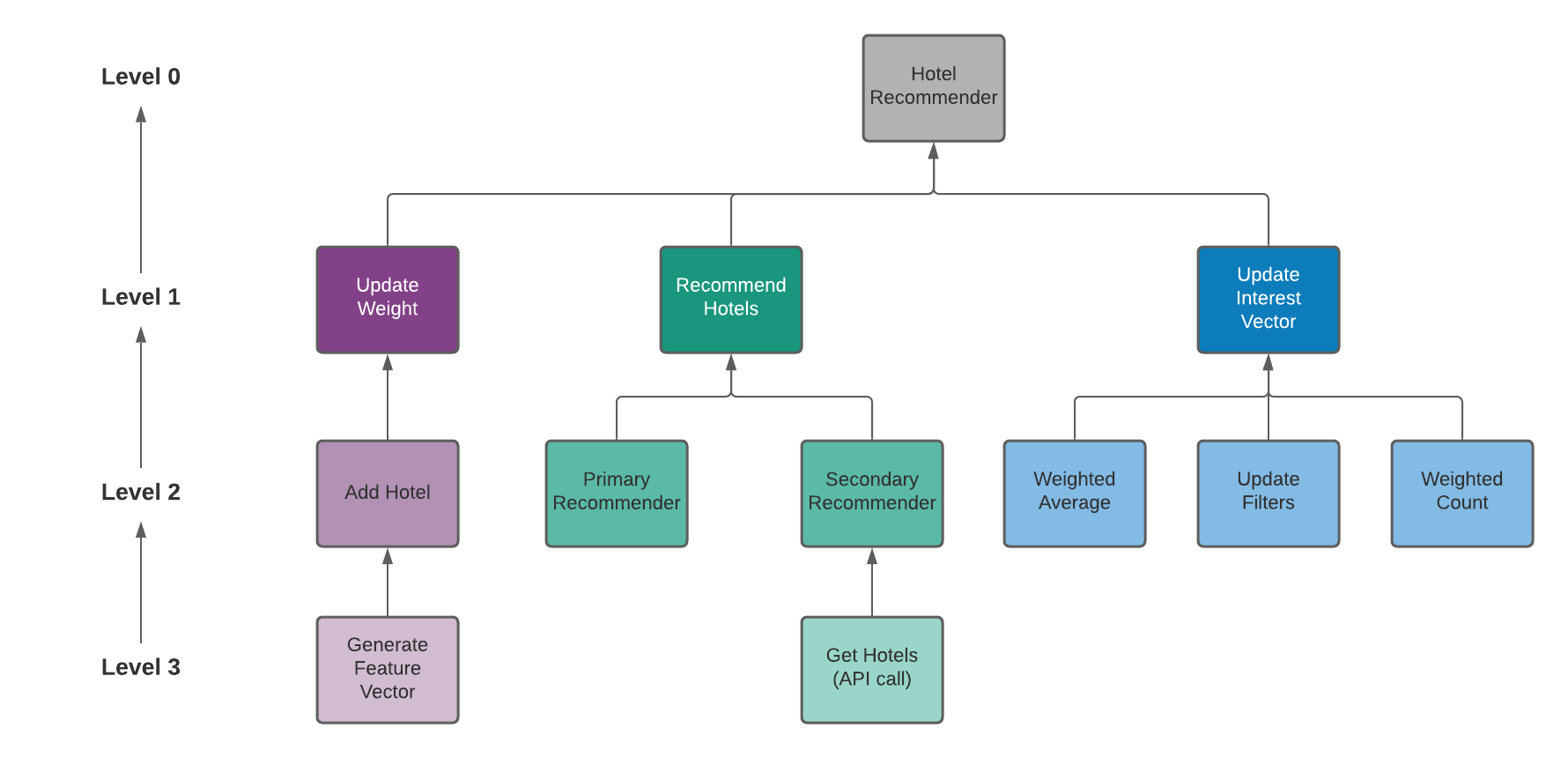
From the above three tables, it can be seen that all the units (functions) performed as expected in different scenarios. The average code coverage of all the test cases for all the units was calculated to be around 85%, which indicated an acceptable degree of unit testing.

With these results in hand, the next step was to check how well the various units of these recommenders worked together when integrated by testing them as a group.

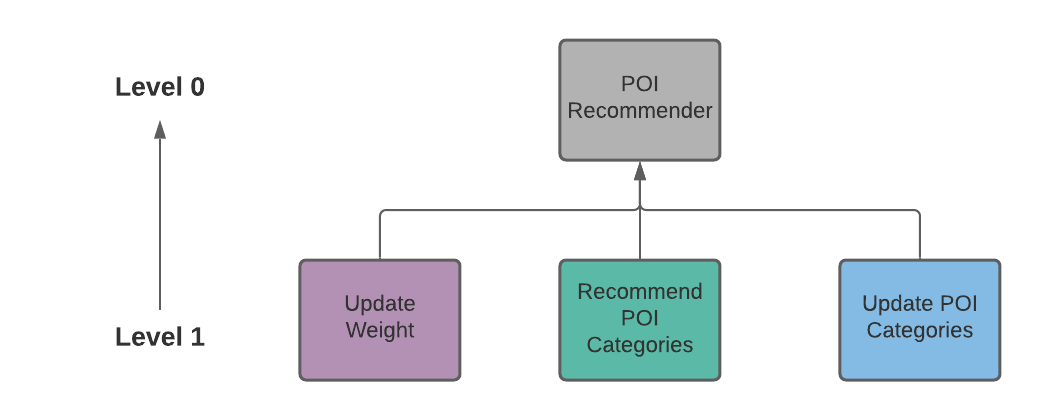
Integration Testing

This second stage of testing took place after the different units of each model were integrated to build the three recommendation systems. In this stage, the recommenders were tested as a whole to see how well the different parts of each model could work together and deliver recommendations. In order to do this, a *black-box testing* approachwas followed wherein only the functionalities of the different recommender models were tested without looking into the internal code structure or implementation details.

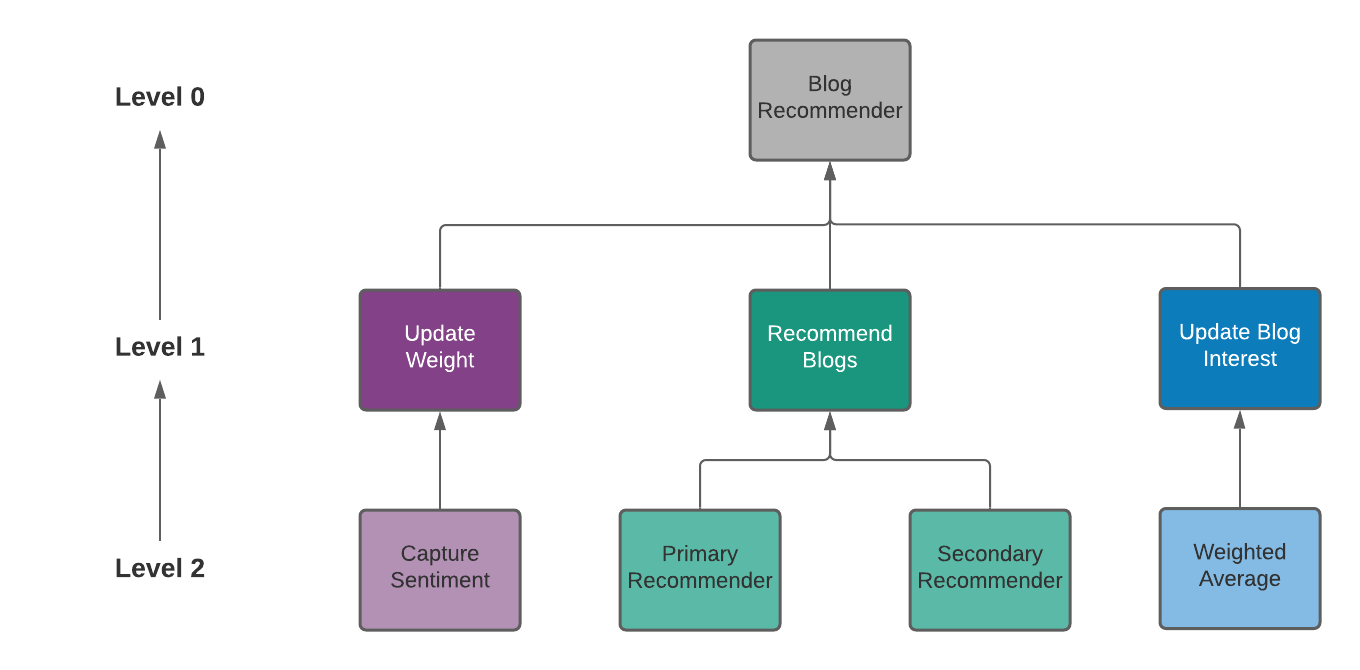
The various modules were integrated in a *bottom-up* manner. The lower-level modules were tested first and these tested modules were then used to facilitate testing of higher-level modules. This process was continued until all the modules at the top level were tested.

 *Hotel Recommender Integration Testing*

*POI Recommender Integration Testing*



*Blog Recommender Integration Testing*



*Integration Testing Procedure*

The above diagrams show the different modules that were integrated and tested in each level in a bottom-up manner while building the recommender models. For each model, the following steps were followed during integration:

* Clusters were formed by merging low level modules. These clusters, also known as builds, are responsible for performing a subsidiary function of the recommender
* Control programs (drivers) were written to coordinate the input and output of different test cases.
* Testing was done on the entire build using different test cases
* Finally, the drivers were removed and the clusters were integrated by moving upward from bottom to top in program structure with help of control flow.

The integration testing proved to be successful as all the different modules were able to pass all the given test cases by working together and also finally generating proper recommendations.

With these results in hand, the next step was to test these recommenders as whole by introducing them to various user personas. This was done in the system testing stage.

System Testing

In this stage, different user personas were first created by interacting with the website. The aim of this was to allow the AI to capture all these interactions and build its knowledge base and understand each of its user’s mindset. This was followed by asking for recommendations for each of the users. This was again a *black-box testing* approach wherein the recommender models were tested as a whole without looking into their different underlying components.

The different user personas created for this purpose are detailed in the table given below.

*User Personas for System Testing*

|  |  |  |  |
| --- | --- | --- | --- |
| User (hometown) | Hotel History | POI History | Blog History |
| John (Chennai) | John is a middle-class man who has used the website to plan trips to various metropolitan cities. | John likes to add restaurants and other food joints to his itinerary. He also sometimes likes to visit clubs and bars. | John has shared his 4-day trip to Bangalore on the website which costed 30$. He has also liked and commented on a few beach-themed trips. |
| Matt (Bangalore) | Matt is a rich businessman who uses the website to plan business trips. | During his trips, Matt likes to visit museums and theaters for entertainment. | He has not neither shared nor interacted on the blog page of the website. |
| Susan  (Mumbai) | Susan is a middle-class woman who loves to plan beach trips and also visit hill stations. She always travels with her dog. | Susan likes to add all the beaches and hill station viewpoints she is going to visit on her trip in her itineraries. | She has shared two 3-day trips, one posted a long time ago (50$) and another posted more recently (20$). |
| Sophia  (Delhi) | Sophia likes to visit hill stations and mountains. She is a wheelchair user but this does not stop her from travelling the country. | Sophia likes to visit religious places such as churches and temples wherever she goes. She also likes to explore different natural POIs. | Sophia is an avid writer and has shared all her trips on the blog section of the website. |

The testing showed that the AI was able to accurately capture its user’s personality from the different interactions on the website. The results of the different recommendations are shown in the table below. We can see that the results are in-line with the user’s interests.

*System Testing (Recommendations for user personas)*

|  |  |  |  |
| --- | --- | --- | --- |
| User (destination) | Hotel Recommendations | POI Recommendations | Blog Recommendations |
| John (Bangalore) | Mid-range 3-star and 4-star hotels. | [Foods, Bars, Restaurants, Cafes, Alcohol, Pubs] | 3-4-day beach and city trips in the 25$-35$ price range. |
| Matt (Chennai) | Luxury 5-star and 4-star hotels with business facilities. | [Cultural, Museums, Art Galleries, Cinemas] | List of blogs sorted by likes. |
| Susan  (Goa) | Mid-ranged pet-friendly beachside hotels. | [Beaches, Natural, Lagoons, Waterfall] | Beach themed trips in the $20-$30 price range. |
| Sophia  (Kodaikanal) | Mid-ranged hotels with wheelchair accessible rooms and accessible path of travel | [Natural, Waterfall, Mountain Peaks, Lakes] | Blogs related to beach-themed and mountain-themed trips. |

Interface Testing

Prior to the three levels of testing mentioned above, an interface testing was performed for the various utility functions that would assist the recommenders by making API calls and querying the database for the required data. Connections existed between the application and MongoDB and also between the application and APIs such as Hotels.com (for hotel data), OpenTripMap (for POI data) and MapBox (for route data). A *black-box testing* approach was therefore adopted to make sure that these interfaces were setup properly.

Similar to unit testing, each of the functions that made an API call or made a query to the database was tested by passing in different combinations of valid and invalid arguments. The summary of the tests performed for each of the different functions is given below.

*Application-API Interface Testing (Hotels.com)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| *Get Hotels (city Id, filters)* | | | |
| A list of valid arguments | A list of hotels | A list of hotels | Pass |
| Invalid city Id | “Invalid destination Id” | “Invalid destination Id” | Pass |
| Incorrect filter Ids | “Invalid filter object” | “Invalid filter object” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |
| *Get City Info (city name)* | | | |
| A list of valid arguments | City Id | City Id | Pass |
| Invalid city name | “Invalid destination” | “Invalid destination” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |
| *Get More Info (hotel Id, filters)* | | | |
| A list of valid arguments | Hotel details | Hotel details | Pass |
| Invalid hotel Id | “Hotel not found” | “Hotel not found” | Pass |
| Incorrect filter Ids | “Invalid filter object” | “Invalid filter object” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |
| *Get Photos (hotel Id)* | | | |
| A list of valid arguments | Hotel photos | Hotel photos | Pass |
| Invalid hotel Id | “Hotel not found” | “Hotel not found” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |
| *Get Reviews (hotel Id)* | | | |
| A list of valid arguments | Hotel reviews | Hotel reviews | Pass |
| Invalid hotel Id | “Hotel not found” | “Hotel not found” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |

*Application-API Interface Testing (OpenTripMap)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| *OTM API (method, query)* | | | |
| method = “xid” | Information about given POI | Information about given POI | Pass |
| method = “geoname” | POIs in a given city | POIs in a given city | Pass |
| method=” radius” | POIs within a given radius | POIs within a given radius | Pass |
| Invalid city name | “Cannot retrieve info” | “Cannot retrieve info” | Pass |
| Invalid radius value | “Unable to retrieve info” | “Unable to retrieve info” | Pass |
| Invalid xid | “Unable to retrieve POI details” | “Unable to retrieve POI details” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |

*Application-API Interface Testing (MapBox)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| *Get Optimized Route (points)* | | | |
| A list of valid points | “Optimized route found” | “Optimized route found” | Pass |
| Missing points | “No points provided” | “No points provided” | Pass |
| Invalid/expired API key | “Access denied” | “Access denied” | Pass |

*Application-DB Interface Testing (MongoDB)*

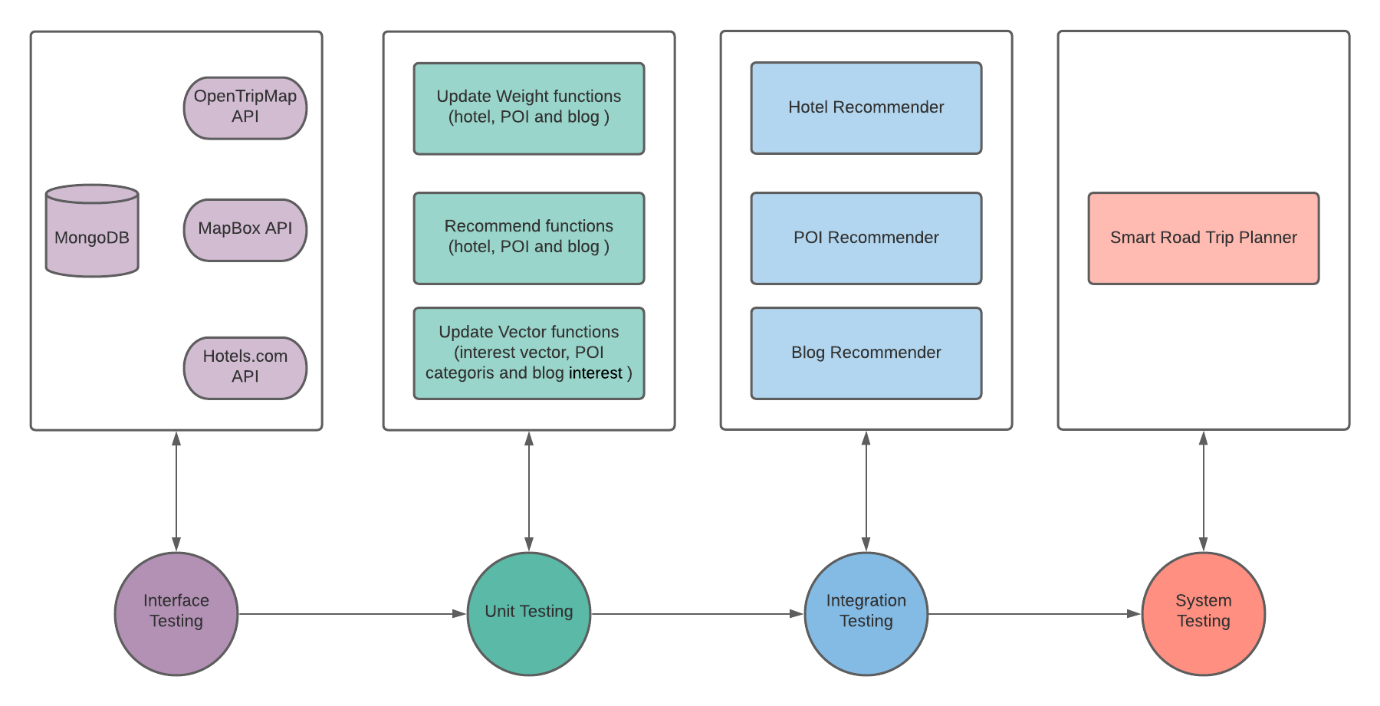
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case Description** | **Expected Result** | **Actual Result** | | **Pass/Fail** |
| *mongoose.connect(mongoose connection string)* | | | | |
| Valid string | “Database connected” | | “Database connected” | Pass |
| Invalid string | “Database connection error” | | “Database connection error” | Pass |
| Valid DB query | Query results | | Query results | Pass |
| Invalid DB query | “Invalid query” | | “Invalid query” | Pass |

From the above four tables, it can be seen that the application passed the interface testing by performing as expected in all test scenarios. These results, along with the results from the other three testing techniques mentioned previously conclude the functional testing of the application.

Testing Summary

Various types of functional testing techniques were performed on the recommender models. A brief workflow of all the testing methods has been shown in the diagram below. These testing techniques were integrated into the development process to ensure quality control at every step of the way. The results of interface testing showed that the application was successfully able to communicate with the database and other third-party APIs. Next, the results of unit testing showed that all the fundamental components of the recommender models performed as expected in different scenarios. Following this, the results of integration testing showed that these components could successfully work together in order to make up the recommender models and generate recommendations. Finally, the results of system testing showed that the AI was successfully able to capture the user’s mindset through the various interactions on the website and provide recommendations that were in line with the user’s current interests.

*Summary of Functional Testing*



Conclusion

In this project, three different recommendation models were designed in order to guide the user in the trip planning process and also to provide inspiration to travel. The system built using these models could provide hotel recommendations, POI recommendations and also travel blog recommendations on the blog section of the website. This system was built with an AI that could capture user interactions on a trip planning website and use it in order to understand the user’s mindset. These captures were also made to be time-sensitive so that the system could account for changing user preferences with time and generate recommendations that were in line with the user’s current interests. In order to overcome the cold-start problem, these recommenders were divided into primary and secondary components, with the secondary recommender being in charge of recommendations until the AI could gather enough information to build strong user profiles and switch to the primary component. Various testing techniques were integrated into the development process to ensure quality control at every step of the way. In the end, a feedback system was setup in order to understand user’s satisfaction with the recommendation results and thereby evaluate the accuracy of the model. Analysis of the feedbacks showed that the system’s accuracy increased as the number of user interactions on the website increased, eventually surpassing that of existing recommender systems.

References

[1] Wahyudi, K., Latupapua, J., Chandra, R., & Girsang, A. S. (2020, March). Hotel Content-Based Recommendation System. In Journal of Physics: Conference Series (Vol. 1485, No. 1, p. 012017). IOP Publishing.

[2] Huming, G., & Weili, L. (2010, April). A hotel recommendation system based on collaborative filtering and rankboost algorithm. In 2010 Second international conference on multimedia and information technology (Vol. 1, pp. 317-320). IEEE.

[3] Zhang, K., Wang, K., Wang, X., Jin, C., & Zhou, A. (2015, April). Hotel recommendation based on user preference analysis. In 2015 31st IEEE International Conference on Data Engineering Workshops (pp. 134-138). IEEE.

[4] Ramzan, B., Bajwa, I. S., Jamil, N., & Mirza, F. (2019). An Intelligent Data Analysis for Hotel Recommendation Systems using Machine Learning. arXiv preprint arXiv:1910.06669.

[5] Zisos, I., Matsatsinis, N., & Tsotsolas, N. (2018, June). A Hybrid Multi-Criteria Hotel Recommender System Using Online Ratings and Reviews. In 7th International Symposium and 29th National Conference on Operational Research The contribution of Operational Research, new technologies and innovation in agriculture and tourism (p. 1).

[6] Ray, B., Garain, A., & Sarkar, R. (2021). An ensemble-based hotel recommender system using sentiment analysis and aspect categorization of hotel reviews. Applied Soft Computing, 98, 106935.

[7] Hassan, A. K. A., & Abdulwahhab, A. B. A. (2019). Location Aspect Based Sentiment Analyzer for Hotel Recommender System. Iraqi Journal of Science, 143-156.

[8] Abbasi, F., Khadivar, A., & Yazdinejad, M. (2019). A Grouping Hotel Recommender System Based on Deep Learning and Sentiment Analysis. Journal of Information Technology Management, 11(2), 59-78.

[9] Silamai, N., Khamchuen, N., & Phithakkitnukoon, S. (2017, September). TripRec: trip plan recommendation system that enhances hotel services. In Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers (pp. 412-420).

[10] Luan, W., Liu, G., Jiang, C., & Zhou, M. (2018). MPTR: A maximal-marginal-relevance-based personalized trip recommendation method. IEEE Transactions on Intelligent Transportation Systems, 19(11), 3461-3474.

[11] Lim, K. H., Chan, J., Leckie, C., & Karunasekera, S. (2018). Personalized trip recommendation for tourists based on user interests, points of interest visit durations and visit recency. Knowledge and Information Systems, 54(2), 375-406.

[12] Herzog, D., Laß, C., & Wörndl, W. (2018, September). Tourrec: a tourist trip recommender system for individuals and groups. In Proceedings of the 12th ACM Conference on Recommender Systems (pp. 496-497).

[13] Migliorini, S., Carra, D., & Belussi, A. (2018, July). Adaptive trip recommendation system: Balancing travelers among pois with mapreduce. In 2018 IEEE International Congress on Big Data (BigData Congress) (pp. 255-259). IEEE.

[14] Kavitha, P. Road Trip Recommendation System Using User Preferences.

[15] Dadoun, A., Troncy, R., Ratier, O., & Petitti, R. (2019, May). Location embeddings for next trip recommendation. In Companion Proceedings of The 2019 World Wide Web Conference (pp. 896-903).