Name: Shubha Changappa Palachanda
Student ID: 2129848

Uber-Lyft Pricing Application (EasyCabs)

Introduction

There are multiple cab provisioning apps in the market. As such we see that Uber and Lyft are currently on top of it. When booking a cab, users must input their source and destination locations in 2 different apps before determining which surge price option they are most comfortable with. EasyCabs, here, tries to reduce this user action by providing a single platform for the user to visualize the surge prices for different cab options in Uber and Lyft.

EasyCabs is a machine learning-powered web-based tool that allows its users to derive and compare the surge prices for a particular journey from point A to point B using different types of Lyft VS Uber cabs to help them choose the best economical option for travel. This application uses a pricing model that figures out the demand-supply (surge multiplier) in a particular zone by incorporating information about features like weather, rush hour, location, and distance and decides the cab fares accordingly. Furthermore, a feedback loop architecture is also maintained to leverage the outputs of this ML system and the corresponding end-user actions to retrain and improve the model over time.

Functional Specification

The platform's dynamic price calculation tasks take its motivation from Uber and Lyft's surge pricing microservices. This task mainly has two components. The weather details and the rush hour interpretation derived from the user's location inputs and the time of input respectively are used to first calculate the surge multiplier. This surge factor is then used along with the cab types and distance metrics in the pricing model to obtain the final price for each of the Uber and Lyft cab types chosen.

User profile: Boston City Demographic looking for the most cost-effective alternatives between Uber and Lyft cabs

Use case: Choosing a cab from Uber and Lyft as per convenience

User: Logs into the application
User: Once authenticated, enters the source and destination locations and the cab types of Lyft and Uber.
Program: Runs the model for surge prediction
Program: Runs the model for dynamic price calculation using the output from the surge predictor
Output: The application displays the surge prices corresponding to the cab types chosen and the distance and duration for the journey from source to destination inputted as well.

Component specification

Component 1: login and logout functionalities (app module)

Login: Calls the Google Oauth API for authentication; User Input: User's Google email ID and password. If login is successful, the user is taken to the next page else the home page reloads.
Logout: Log the user out from the application once the logout button is pressed or the session is timed out.

Component 2: GetCabPrice function (app module)

User Input: Source, destination, Uber, and Lyft cab types; The component then calls the following subcomponents to get the required values:

Geospatial Data: Input: Source and Destination; Output: Latitude and Longitude, ETA and Distance.
Weathermap API: Input: Source Latitude and Longitude; Output: Required weather parameters.
Uber API: Input: source and destination; Output: dynamic Price and surge of the required cab *
Lyft API: Input: source and destination; Output: dynamic Price and surge of the required cab *
(Components 1 and 2 are structured under the app module which handles the main functionality of the whole project. This script acts as an interface between the UI, the API calls, and the machine learning models)

Component 3: surge_inference module (model_scripts package)

Uses weather parameters obtained in Component 2 and the rush hour indicator derived based on the time of user input and runs the classification model to calculate surge multiplier. (Returns surge multiplier)

Component 4: dynamic_price_inference module (model_scripts package)

Incorporates surge multiplier obtained in Component 3, distance, and cab type parameters from Component 2 in regression models to determine the Uber and Lyft cab fares. (Returns surge price for the selected Uber and Lyft cab types separately)

Component 5: Result - UI component

Displays an output comprising the dynamic prices obtained from Component 4 and ETA and distance values obtained from Component 2 on a blank page (final html page) for user visualization.

Component 6: feedback_app package (*to be used with a cron job for CICD)

Used to retrain Components 3 and 4 by utilizing the parameters obtained in Component 2(the prices from Uber and Lyft APIs act as ground truth)

Component 7: Database

*To be used to store general user information, results of Component 3 and 4, and the inputs for the feedback application (All this information is currently available in CSVs)

Name: Shubha Changappa Palachanda
Student ID: 2129848

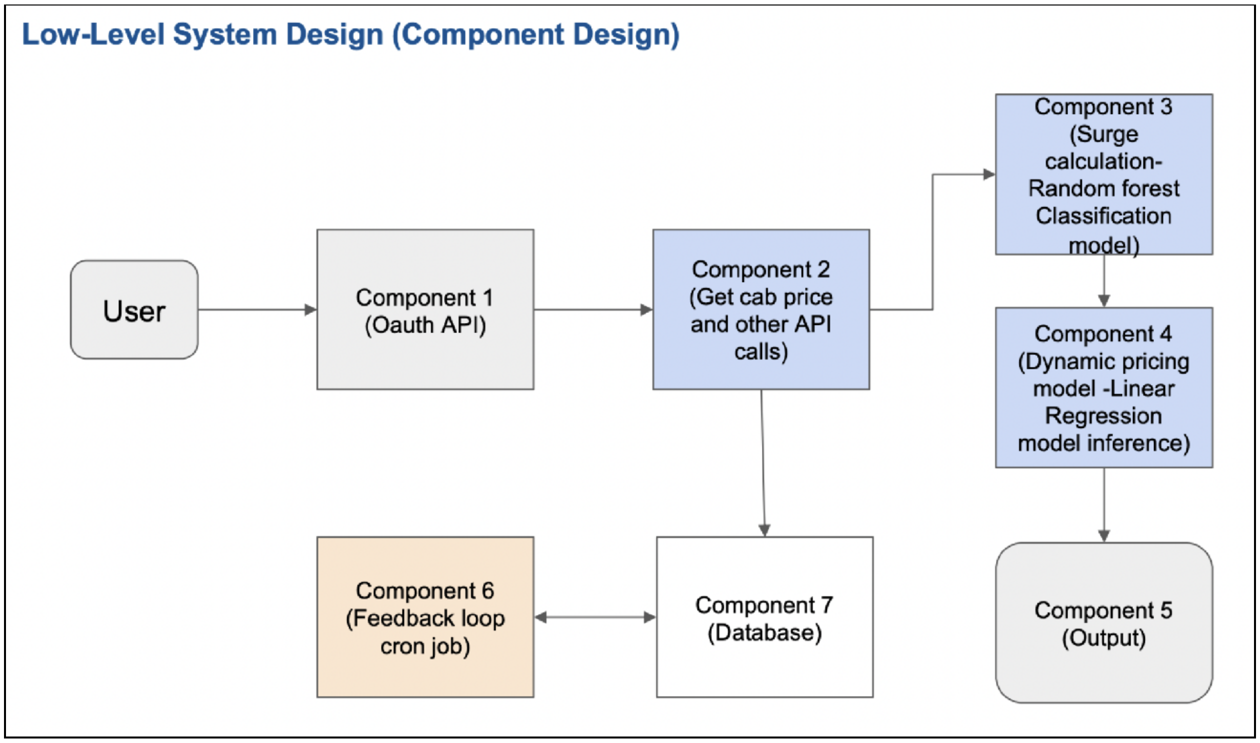
Low-Level System Design (Component Design)
User
Component 1 (Oauth API)
Component 2 (Get cab price and other API calls)
Component 3 (Surge calculation- Random forest Classification model)
Component 4 (Dynamic pricing model -Linear Regression model inference)
Component 5 (Output)
Component 6 (Feedback loop cron job)
Component 7 (Database)

Fig 1: Low-level design architecture representing the component interactions

Design decision

Separation of Concerns

This principle of separate responsibility to each function or class has been maintained across all components of the application. For example, the software's inference modules in model_scripts package have been separated based on the work they perform i.e., surge_inference module is used for classification, to determine a surge multiplier for a particular trip and dynamic_pricing_inference is used for prediction, to determine the cab fares. The same distinction of duties can be seen in the utility programs/libraries in the Utils directory. This type of separation helped ensure that the overall business model is easy to test and can evolve without being tightly coupled to low-level implementation details.

Single Responsibility

This design principle similar to the separation of concerns is used to ensure that every class/method/module has a single responsibility. For example, all modules in feedback_app (which itself is a microservice that's obtained when a single responsibility principle is applied to the application architecture and taken to its logical endpoint), have a single responsibility i.e., SurgePriceModel class (in surge_model_training module) and PricePredictModel class (in dynamic_pricing_model_training module) have a specific responsibility of retraining the surge classifier model and the surge price prediction model respectively. This ensures that any change in process of retraining the surge classifier model doesn't affect the code for the retraining surge price prediction model or vice versa as each class focuses on a single concern. Overall, following this principle has improved code readability and helped to produce more loosely coupled and modular systems, since many kinds of new behavior can be implemented as new classes, rather than by adding additional responsibility to existing classes.

Open-Close Principle:

This principle is incorporated when designing some of our software entities(functions) to ensure that developers have the ability to build on top of our existing code without having to change it to add any new feature or functionality, thus avoiding any refactoring process that could result in potentially messy, hard to read/understand code or increase code debugging operations. For example, we have separate methods in our retraining modules to save classifier/regression model output data into different CSVs. In the future, we can add methods to save the data in a different file format (say XML) or save the data to a database without having to alter the existing method in place for data storage in CSVs.

Comparison of the design with an existing software/library

https://github.com/sushantparkhi/Uber_Price_Prediction

In the above comparison example, the developer has tried to develop a Uber Price Prediction model using linear regression. The regression model for price prediction considers only distance and time of day for surge price prediction and linear regression model is used instead of multiple linear regression model to predict price. Its important to note that distance and time alone are not enough predictor variables to predict the surge price. When calculating a surge price, a surge multiplier which accesses the demand against the supply of rides is essential. Weather information along with the time features is necessary to calculate the surge multiplier since weather changes are one of the primary causes for surge/traffic congestion. However, surge multipliers are not considered in this model. The code snippet for the regression process also doesn't show any evidence of testing conducted on it. It's essential to test the model after training it, to ensure that the model chosen has high accuracy (determined by the R squared value of the model)

All the operational code is written in a single file (R Code.R file). The design principles like separation of concern are not taken into consideration. There are no distinguishable methods for data cleaning tasks or for data manipulation tasks. There is no specific function for calculation of distance, or the rush hour factors. The entire program is written in an imperative manner, hence code reusability, and extension becomes very difficult. Its also not easy to read the code as there very few comments provided against each operational snippet of code.

The user input is hard coded into main R code file which again doesn't promote code extensibility. User input in this application is meant to be flexible and not fixed/unchangeable. Moreover, latitude and longitude information are expected as user input which is not a common practice in real-life. Users generally are required to input source and destination locations/addresses and the latitude and longitude corresponding to these locations are then derived during data manipulation tasks.

All the above-mentioned inaccuracies and limitations in the model are rectified in EasyCabs application (https://github.com/rohitl17/cab-dynamic-pricing)

Name: Shubha Changappa Palachanda
Student ID: 2129848

Extensibility of the software

Different features have relatively different levels of implementation ease when it comes to their extensibility. Some extensions that can be made in the near future along with their respective caveats (if any) are as follows:

Currently, we have developed functionalities with the purpose of comparing cab prices between Uber and Lyft cab types. An additional independent feature of price comparison between different cab types within Uber or Lyft services can also be added with ease if the user has any preference over Uber/Lyft cabs

Apart from comparing prices between Uber and Lyft cabs, the EasyCabs platform can be further expanded to added other upcoming Cab services like Curb, mytaxi, etc., This is currently difficult to implement as the training of classifier/regression models are for these can services are currently on hold as we don't yet have free access to their APIs

EasyCabs customer-base is currently limited to the Boston city population for travels within the city as our models are currently trained on cab journeys across different locations in Boston City. We would like for this application to be accessible across different metropolitan cities in the United States once our models are retrained with different city-specific data fetched from the different cab services and Google APIs. (This implementation is currently in progress as the tokens were not available from the Uber or Lyft's developer platforms)

Retrain the classifier/regression models (from feedback_app) can be incorporated relatively easily into the application by utilizing a cron job (cron.yaml in the root directory of the app) to configure scheduled deployments of the retrained models.

Rush-hour indication input to the surge classifier model is currently determined by the time that the user inputs their options into the app. The model can be further optimized if we were to include weekends or holidays when determining the rush-hour parameter. This, however, requires an addition of a method to classify and list the holidays/weekends for a year and also modify the existing method in place for data manipulation before running the inference method, which can result in violation of the open-close principle maintained in our code thus prompting more refactoring tasks.