March 17th, 2022

Uber/Lyft Dynamic Pricing Application

CSE/DATA 515

ROHIT LOKWANI

1. Introduction

Uber and Lyft account for the major market capitalization for offering cab services on mobile applications. But their prices are not constant like public transportation. They are greatly affected by the demand and supply of rides at a given time. So what exactly drives this demand? Some of the factors include weather changes, rush hours, and location. Easy Cabs takes into account these factors and tries to replicate a prototypical version of the pricing and surging microservices for these apps. It also allows retraining models using a feedback loop as a separate microservice to continuously improve the pricing model's performance.

Easy Cabs is an ML-assisted web-based application that helps you in getting the dynamic pricing of Uber and Lyft cabs. The user enters the source and destination. Easy Cabs converts that to latitude, longitude, gets the weather information, and predicts the estimated price for your rides using machine learning. The user can then decide on taking a cost-optimized cab.

2. Functional Specification

While booking a cab, one of the aims is to find a cost-optimal one. Users have to put in the location in Uber and Lyft apps to get the optimal cost. Our ML-based solution incorporating surge overcomes this by being a one-stop solution for comparing prices for different types of cabs. The surge calculation tries to replicate Uber's microservice for pricing and surging calculation by taking the rush hours, weather changes, and location as the input features. The dynamic pricing ML model uses the surge, distance, cab type source, and destination location features to calculate the final price. Our goal is to suggest the user an appropriate cab type from Uber and Lyft as per their budget constraints and convenience minimizing the user's effort.

User profile: People who want to compare and book cabs at optimal cost and convenience.

Use cases

1. Choosing a cab from Uber and Lyft as per cost
    i. USER: Logs into the application using Google Email ID and password
    ii. USER: Enters source and destination.
    iii. USER: Provides **similar** cab types for uber and Lyft.
    iv. PROGRAM: Calls geospatial and weather APIs to get the required information
    v. PROGRAM: Runs the model for surge prediction.
    vi. PROGRAM: Runs the model for dynamic price calculation using the surge prediction model output.
    vii. OUTPUT: Outputs the price for uber, Lyft respective cab types and ETA and distance between the source and destination.

2. Choosing a cab from Uber and Lyft as per convenience
    i. USER: Logs into the application using Google Email ID and password
    ii. USER: Enters source and destination.
    iii. USER: Provides cab types for uber and Lyft as per **luxury/convenience** (similar or different)

iv. PROGRAM: Calls geospatial and weather APIs to get the required information
v. PROGRAM: Runs the model for surge prediction.
vi. PROGRAM: Runs the model for dynamic price calculation using the surge prediction model output.
vii. OUTPUT: Outputs the price for uber, Lyft respective cab types and ETA and distance between the source and destination.

3. Component Specification

3.1 Component 1: (app/googleAuthorize)

Calls the Google Oauth API for authentication and uses flask_login functionality to store and manage user-session information. It also helps the user log out of the session that needs to be closed.

Input: User's Google email ID and password.

If login is successful, take the user to the next page else reload the home page

3.2 Component 2: (getCabPrice)

Handles the business logic for the whole application. This API calls all the required APIs to get weather and geospatial information and calls components 3 and 4 to get the outputs from the ML model.

Input: Source, destination, and type of cabs.

The component then calls the following subcomponents to get the required values:
1. Geospatial_information:
   Input: Source and Destination; Output: Latitude and Longitude, ETA and Distance.

2. Weathermap_information:
   Input: Source Latitude and Longitude; Output: Required weather parameters.

3. Uber API (Development in progress):
   Input: source and destination; Output: dynamic Price and surge multiplier of the cab

4. Lyft API (Development in progress):
   Input: source and destination, Output: dynamic Price and surge multiplier of the cab

3.3 Component 3: (surge_inference)

Calculates the rush hour and runs the classification model to calculate surge.

Input: Weather and geospatial parameters obtained in getCabPrice Component 2; Output: surge_multiplier

3.4 Component 4: (dynamic_price_inference)

Runs the data manipulation and the regression model to calculate the dynamic price for uber and Lyft individually.

Input: Surge_multiplier, distance, source and destination geospatial information; Output: Price for Uber and Lyft selected cabs

3.5 Component 5: (result_generator / UI component)

Creates a JSON and shows the output on a new page with a logout option or back button to try other combinations.

Input JSON for the output from component 4 to show the final user the report; Output: Price, ETA, and Distance estimates for the user

3.6 Component 6: (feedback app)

Append the new datapoints to the already available training data frame with parameters obtained in Component 2, using price from Lyft and Uber API as their ground truth. A cron job can be configured by the developer to run and save the results every time. The default configuration is running every Friday at 11 pm.

Input: None; Output: Retrained models for surge classification and dynamic pricing.

3.7 Component 7 (database)

This component comprises of CSVs now. We store the basic user information for storing user context for future logins. This database will also store updated model results and entries for feedback application.

Components interaction

1. User visits the home page. Signs in using a google email ID. The backend for this authentication is handled by component 1. In case of a successful login, the user is directed to Component 2.
2. For component 2, the user inputs the required source, destination and selects uber and Lyft cab types from the dropdown. This then directs the context to the involved subcomponents to get the required information using the APIs.
3. Once the information is available, component 3 is called and calculates the surge using the ML model.
4. The surge multiplier is sent to component 4 which calculates the dynamic price for the respective cab prices.
5. Once the price is calculated, the final report is generated as a JSON and sent to the reporting frontend which shows the output.
6. Component 6 is an independent component that interacts through component 7 (database) with the other components. Components 1 and 2 store the user-related and model-specific information in the database. This information is used to retrain the models at specified intervals using a cron job.

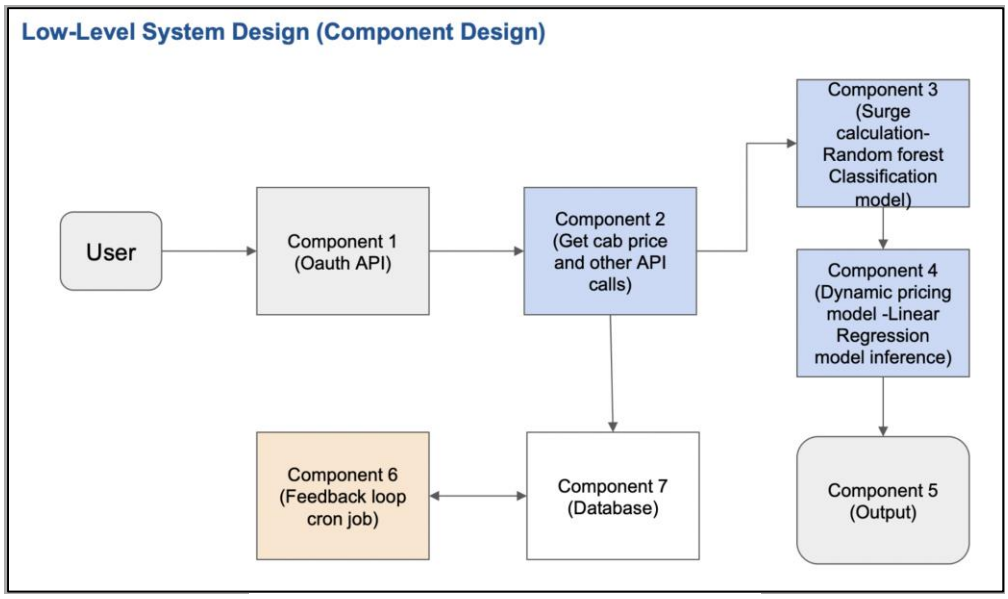

Figure 1: Component Specification Diagram

4. Design decisions

The application works on a request-response model. The client sends the request to the server for information and the server processes and responds to it. This section highlights the design decisions made at high-level (architecture) and low-level (code).

4.1 High-Level Decisions

1) As we see the components in the high-level design in figure 2, the Presentation layer, Application layer, Business logic layer, Data access layer are kept separate. This aids separation of concerns and supports the extensibility of the software on all fronts.
2) We keep a feedback loop as a separate microservice for retraining models at pre-specified intervals. This not only helps in monitoring and improving ML model performance but microservices make the software scalable and reliable and easy to debug in case of any failures.

Overall, this architecture facilitates easier and cleaner app maintenance, feature development, testing, and deployment compared to a monolithic architecture. The microservice architecture fits best for complex use cases and for apps that expect traffic to increase exponentially in the future, like a social network application.

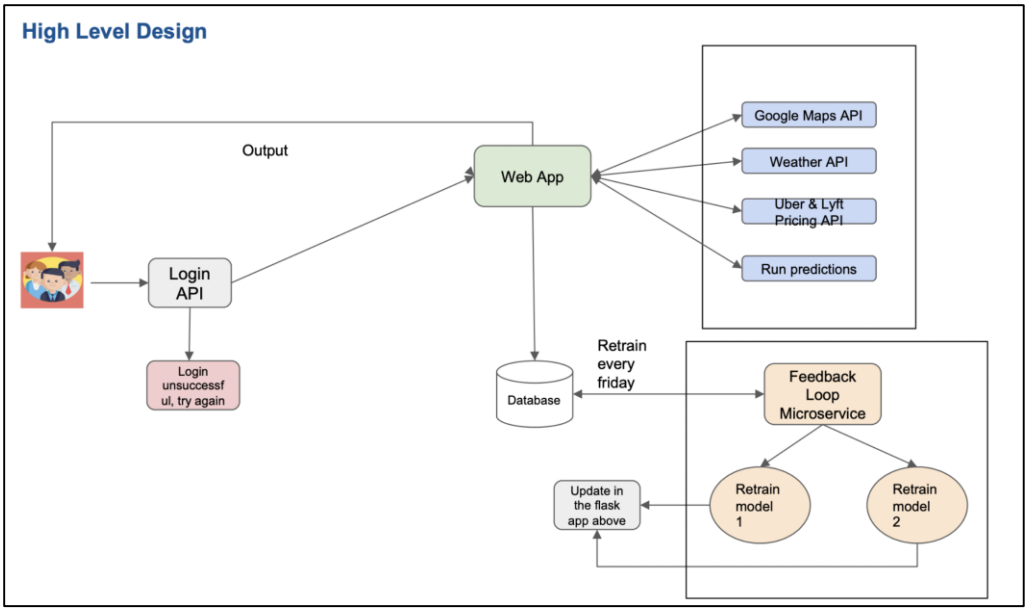

Figure 2: High-level Design Diagram

4.2 Lower-level Design decisions

1) Separation of concerns

The main app file has separate modules which handle the user login functionality using Google OAuth. The module for handling user-related information is in user.py which has a User class. A similar design is followed for model inference of Surge classification and Dynamic pricing regression models. The model-specific cleaning and transformations are done in their respective inference scripts to keep the concerns separate. The main app file acts as the aggregator for the information and the actions are in separated modules. This separation ensures the low-level inter-component dependency is reduced and helps in the software's evolution.

2) Abstraction

As mentioned in point 1, the concerns are separated depending on functionality. We ensured that the interface is made simpler with deep implementations. Also, after every call to another module, the control is transferred back to the main app file. Developers in the future can work on one aspect of the software and only important details are mentioned in the code which helps in following the abstraction principle.

3) Single Responsibility Principle

Each of the modules or components handles just a single responsibility. For example, a class of SurgeModelInference will just handle data manipulation and inference for that model. The training module for this model is a separate class in the feedback loop, making sure the code is not redundant. Overall, following this principle has improved code readability and helped to produce a more loosely coupled and modular system.

4) Use of UserMixin to store context information (Encapsulation, Inheritance)

We combined two libraries (Oauth from Google) and flask-login to make a user class. We inherit the Usermixin class (a stand-alone base class to extend functionality in flask) to store context and user-related information in the database. This class ensures the privacy of user information. It also helps in storing the user information which is encapsulated as protected variables to ensure the access to these variables is limited in future methods. The static methods ensure the class state is not modified. The exceptions are handled if the user creation fails in the database. The code is as follows

```
from flask_login import UserMixin

class User(UserMixin):
'''
The user class of type Usermixin from flask_login handles the user information.
’’’

    def __init__(self, id_, name, email):
        self._id = id_
        self._name = name
        self._email = email


    @staticmethod
    def get(user_id):
        #Retrieve user information if account already exists
        return User(specific_user_details['user_id'], specific_user_details['name'],specific_user_details['email'])

    @staticmethod
    def create(self):
        try:
            # Try Creating a user in the database if authentication succeeds
        except BaseException as error:
            #Return to the home page if user creation in database fails
```

Overall, this design decision incorporates inheritance, encapsulation, use of mixin classes, and exception handling to function with privacy and handle failure.

5. Comparison of the design with an existing software

The repository used for comparison is https://github.com/bopas2/surge-surfer.

In this section, we will compare the above repository with our repository and how we overcome the limitations of this repository in our work.

1) Objective and end-goal perspective

The surge surfer application uses past data and current traffic information to help the user for booking a ride by tagging if it is a prime-time to have a surge. We extend this objective in our work by estimating the surging value and the dynamic price of the cab.

2) Differences from ML modeling perspective

i. Their models are trained on Lyft data, hence their use case is limited. On the other hand, we showcase our results for Uber and Lyft, thus adding to the functionality and deliverables.
ii. The weather information is not incorporated in the models which further makes the performance questionable for detecting a surge.

3) Code and software design perspective

i. Code explainability: The repository has no comments for the code. Hence making it less explainable.
ii. No separation of concerns: The code is not modular and all actions are carried out by the index API. All the code is put in a single app file.
iii. No abstraction/encapsulation: The calls to geocoding APIs and models could be separate from business logic and implementation can be hidden from the main code, which is not done.
Iv Hardcoded values for variables and data features. Does not support code extensibility.
v. Commented lines of code in the main app file make the code less readable and difficult to understand.
vi. The length of print statements, blank lines don't follow PEP8 standards (Potential E501 and W293 flake8 errors)

Hence, overall, we think our library comes up with a more reformed and deeper objective/end-goal, handles the Machine learning inference and training through feedback well, and exhibits design principles and coding standards missing in this library. We also, implement Continuous Integration using Travis and have test cases that make the code reliable.

6. Extensibility of the software

The limitations and future work of our software include:

i. We are still trying to get access to server tokens from Uber and Lyft to make our feedback loop fully functional
ii. Our software exhibits scalability in design but from a deployment point of view, we need to deploy on a web server like Nginx and a development server like Gunicorn in the backend to handle massive amounts of traffic, with an open option for autoscaling servers.
iii. The database now is a CSV, which could be extended to MongoDB or SQL for reliability
iv. We can add logging mechanism to debug a large codebase in the future.
v. The ML models can further incorporate new features which could imitate the actual pricing and surging Uber microservices. Limitations of data can be improved to make the models ubiquitously accurate.

We think from the design point of view having the feedback loop and every service modular will help in adding features in the future. The code has been written in a way that allows extension for specific classes, API calls and is easy enough to add API without breaking the original code. This makes the code and design both easily extensible.

7. Software Requirements for Easy Cabs

Authlib, flask-login, flask, google maps, geopy, requests, numpy, openweathermap, pandas, pytest, setuptools, scikit-learn