Boston University
Electrical & Computer Engineering
EC464 Senior Design Project

Second Prototype Testing Report

WhereTo

by
Team 5

Team Members:
John Burke - jwburke@bu.edu
Muhammad Ahmad Ghani - mghani@bu.edu
Haochen Sun - tomsunhc@bu.edu
Erick Tomona - erickshi@bu.edu
Marta Velilla - martava@bu.edu

Required Equipment:

Hardware:

- Personal Computer
    - To host both the backend of the application, as well as the database
- Personal Smartphone
    - To experience the frontend user interface of the application

Software:

- Python Park API
    - This is hosted on our machine, makes use of ML models as well as algorithmic design and API calls in order to analyze parking information for the user
- React Native UI
    - This is hosted on our machine, React code to display our user interface on mobile. This interacts directly with the backend via an API call.
- Expo Go
    - A mobile application which is designed for testing react native applications on iOS or Android devices
- SQLite Database
    - This is a cache, to reduce excessive calls to the Google Services APIs

Pre-Testing Setup Procedure:

In order to set up the test correctly, every component of the software must be correctly configured as well as actively running. Because of this, each component has its own setup that must be performed prior to attempting the test:

ParkAPI Component:

1. Ensure the backend repository is in the expected location on the personal computer being used for testing
2. Ensure that all dependencies for the backend are installed via pip
3. Ensure that the correct configuration variables are set in the config.py file. This is vital to allow for both use of the OSM Extract API as well as the Google Maps API.
4. Run the application by navigating to the root directory of the API in terminal and executing: ‘python3 app.py’
    - This will run the application on every open host possible, so we can access it through our React Native Application

React Native UI Component:

1. Ensure that the smartphone being used for testing has the Expo Go application installed, this is necessary to run the application prototype
2. Once the ParkAPI is hosted (in the above steps for ParkAPI Component) it will output its URL to the terminal. One must ensure this URL is the URL that the React application attempts to access via axios.
3. Run the application by executing 'npx expo start'
    a. This will output a QR code to the terminal, scan this and open Expo Go to download the application and begin testing

Database Component:

1. We want an empty database for the start of the test, so any existing db file should be deleted and a new empty database should be created using sqlite commands.
    - This database should have the schema present in the .sql file in the backend repository. This schema allows us to store two tables: coordinates we have looked at, and detections we have made.

Testing Procedure:

The testing procedure for the application was as follows:

1. Load the application by scanning the QR code outputted from the 'npx expo start' command
2. For each address and radius combination for which we want to examine:
    a. On the InputDisplay screen for the UI, enter a valid address string and radius value
    b. Press the "Find Parking" button on the UI
    c. Wait for the backend to finish completing the request
    d. Examine the UI/map output present after the request is done processing
3. Additionally, we should perform each combination of inputs twice, to ensure that proper caching is being performed; that is, we are not performing duplicate calls to the Google API services if we have analyzed them previously/recently

During the test, we followed the testing plan exactly. We went through a few example inputs, two times each, in order to prove that we were able to locate relevant parking information in cities, as well as cache the important results. During the testing procedure, we also tested an example address and radius pair of a CVS in Newton, MA, at Professor Osama's request. This example was able to detect the parking meters at this location fairly accurately.

Measurements Taken:

The measurable criteria for the Python API was as follows:

I. The Python API should be capable of analyzing the streets in a given area, this is validated by a local file in the map/ directory which displays markers on every analyzed coordinate
II. The Python API should be capable of using the pretrained model to make and deliver predictions with a confidence level of possible parking meters in the area as a JSON response. This is also verified partially in the map/ directory, where markers displaying detections are placed
III. The Python API should return an error in the case of an invalid parameter, and should be capable of receiving multiple requests in a row and processing them accurately to the requirements of the application

The measurable criteria for the React Native UI was as follows:

I. The React Native UI should allow for the user to input a radius and address that they desire
II. The React Native UI should be capable of graphically displaying the JSON response in the case of a successful API procedure
III. The React Native UI should be capable of handling and informing the user of an error in the case of an unsuccessful API procedure

The measurable criteria for the SQLite Database was as follows:

I. The SQLite Database should be used instead of a duplicate call to the Google API services and machine learning model if we have examined a coordinate before, this is going to be displayed/shown through API logging

Results:

Overall, this second test was a very successful one for our design. Our design performed as we had hoped and expected that it would during the test.

The Python Park API was capable of analyzing streets within the given radius, and was able to forward the detections it saw to the frontend as well as store the detections inside of the database. When we fed incorrect parameters to the API, it was able to return an error to our frontend. We could determine that the caching was working with the SQLite database as the second time we ran any given input, it completed the request much faster than the original time. This would be due to less calls to the Google API and AI components, as we were using our database for the information.

The React Native UI was also successful in all of the criteria we demanded of it during the test. It allowed for user input, and was able to correctly handle both an error and success response from

the backend API. The UI was also able to visually graph the locations of detected parking meters on a map for the user.

Conclusion

Our main conclusion from the test was that, save for adding some features and cleaning up the UI, our parking meter detection algorithm works as intended from an end to end perspective. This was great news, as it will allow us to begin to focus on the final additions that we want to have as components of our design before the final testing. After speaking with our client on the day after the test, he informed us that the system was in a good place, and that we should focus on adding features for road sign detection and text recognition on those signs. Due to the already modular nature of our design, it will not be too exhaustive of a task to add this road sign and text recognition component.The other item which we have to add is a modal display for in depth information about individual road signs and parking meter detections on the front end. This will be a challenging task for the front end, and will probably require modification of the existing backend API, or an additional endpoint to deal with forwarding image data to the frontend.This test overall showed us that our design was functional, and it is now time to add more features to our existing design.