

Explanation of Backend and Model Integration

Initially, we wanted to use a Kaggle dataset to train our model, and use user input as the test dataset to make the actual recommendation predictions. Due to the lack of datasets that fit our specific need, we decided to create two of our own datasets: `cities.csv` and `queries.csv`.

The `queries.csv` file contains the training data containing various different responses to the questions asked in the profile setup stage. The `cities.csv` file contains information on various different cities. For example, New York will contain information that it is in the United States, the budget would be “premium” and it is a “busy” city. The model will use this dataset as a way to find the most similar match of cities based on what the user inputted as their preferences in their profile. For example, if a person had also listed their preferences as a busy, premium city, within the continent, then New York could be recommended to them since it has similarities to their preferences.

After we have prepared the data, we created a Python file in Google Colab to create the actual model and train it based on these two datasets using TensorFlow recommenders. Once it had finished training, then it would be saved as a `.h5` file and converted to a TensorFlow Lite (`.tflite`) file. We need to save the model to import it into our mobile app, so this is why we save the model in these two extensions. Later, we found that the `.h5` model file is too big to run on our mobile app, so we converted it to a `.tflite` file since that is a more compressed version of our model that will work with mobile apps better.

Once we finished creating the model and saving it, we needed to create a backend for our app. Usually, `.h5` or `.tflite` files can just be saved directly into the assets folder in a mobile app, and then it will run and be called by the front-end of the app. This did not work for us, however, since we are using Expo Go to host our app, and this approach only works with a bare React Native app. Therefore, to integrate the `.tflite` model into our app, we had to create a backend. The backend consists of `app.py` and it will load the `.tflite` model and create a `/recommend` route. This function essentially will take in a vectored feature input (the user data) and ensure that the input is safe for the model to take in. Then, it will run the model on this test data, and receive an output of top 3 recommendations from the model. From there, it will return a JSON string of these recommendations.

We also needed to host our backend so that it can actually run in the mobile app. We chose Render, since it had a free tier. First, we connected our GitHub repo (specifically the `elysian_backend` folder) to Render, and deployed the backend. Once it was running live, the mobile app should now be able to post the user input data as a vectorized input and then use the `app.py` code to run the model and wait to receive the response of the recommendations in a JSON output so that it can display on the mobile app page. This is done in `home.tsx`, where it posts the input data to the recommendations page and then waits for a response and displays it.