Trailblaze - Formal Report

Team Ornithomimus

CPEN 291 Computer Engineering Design Studio I

The University of British Columbia

Andrew Forde, Taylor Mcouat, Leo Sun, Jayden Wong

Introduction

Trailblaze is a Tinder-style mobile application created to connect hikers with the best views offered by British Columbian trails. It combines two machine learning models to recommend hikes tailored to the user's preferences. The first model, an embedding-based recommender system, is used to generate a list of hikes similar to the user's previously liked hikes; the second model, an image classifier, is used to identify visual features in liked images, and recommend hikes which contain similar features. These two models are connected behind the scenes to a mobile app, so the user can find new hikes from wherever they are. The combination of powerful machine learning and the convenience of a mobile application make Trailblaze the ideal tool to help hikers on the move find their next destination.

Once opening Trailblaze, users start by filling in a short profile, including their preferred hike difficulty, trail length, and elevation gain. Next, users are able to swipe through a deck of cards, with each card showing a collection of images from a hike. Each hike and its photos have been sourced from the popular hiking website, AllTrails.com. Cards which are swiped right are marked as likes, and are added to the user's matches screen; cards which are swiped left are marked as dislikes, and are ignored in the app. On the matches screen, the user can see more details and photos, and they can follow a link to the AllTrails website for more reviews. The user's likes and dislikes are taken from the hike screen and sent to the backend using a RESTful API. Based on these ratings, a recommender machine learning model will find hikes similar to that of the user's liked hikes. These hikes are then sent to an image classifier to predict if the user will like them, based on photos of past liked and disliked hikes. All hikes that are predicted as liked are sent back to the mobile application, where the new hikes can be viewed and rated by the user. The more hikes the user reviews, the more refined and accurate the recommended hikes become.

Project Overview

From the user's perspective, the mobile application is where the action happens. Upon opening the application, users are greeted by the sign-in page where they can sign in with their email and password. If this is the user's first time on Trailblaze, they can create an account with their name, email, and a password. This information is used to save their preferences when they are not using the app. Once signed in, the user is on the profile screen; here, they can input their preferred hike distance, elevation gain, and difficulty. Switching to the hike screen using the navigation bar, the user is shown a stack of cards, each displaying a unique hike. Users can scroll up and down to see photos from the hike, and can swipe left or right to rate the hike; swiping right likes the hike and swiping left dislikes the hike. Once the user nears the end of the card stack, an API call is made to get more hikes from the backend based on the user's previously liked hikes. Moving on to the matches screen, all of the user's liked hikes can be found again in a list format. Tapping on any of the hikes will bring up more details. On the unique hike page, the user can scroll vertically through the available images; the hike name,

difficulty, length, elevation gain, location, and hike type are all shown below the images. Two buttons at the bottom of the screen allow the user to take more action. The first button takes the user to the AllTrails webpage in their mobile browser, where they can see more photos and reviews; the second button removes the hike from the matches page if they are not interested.

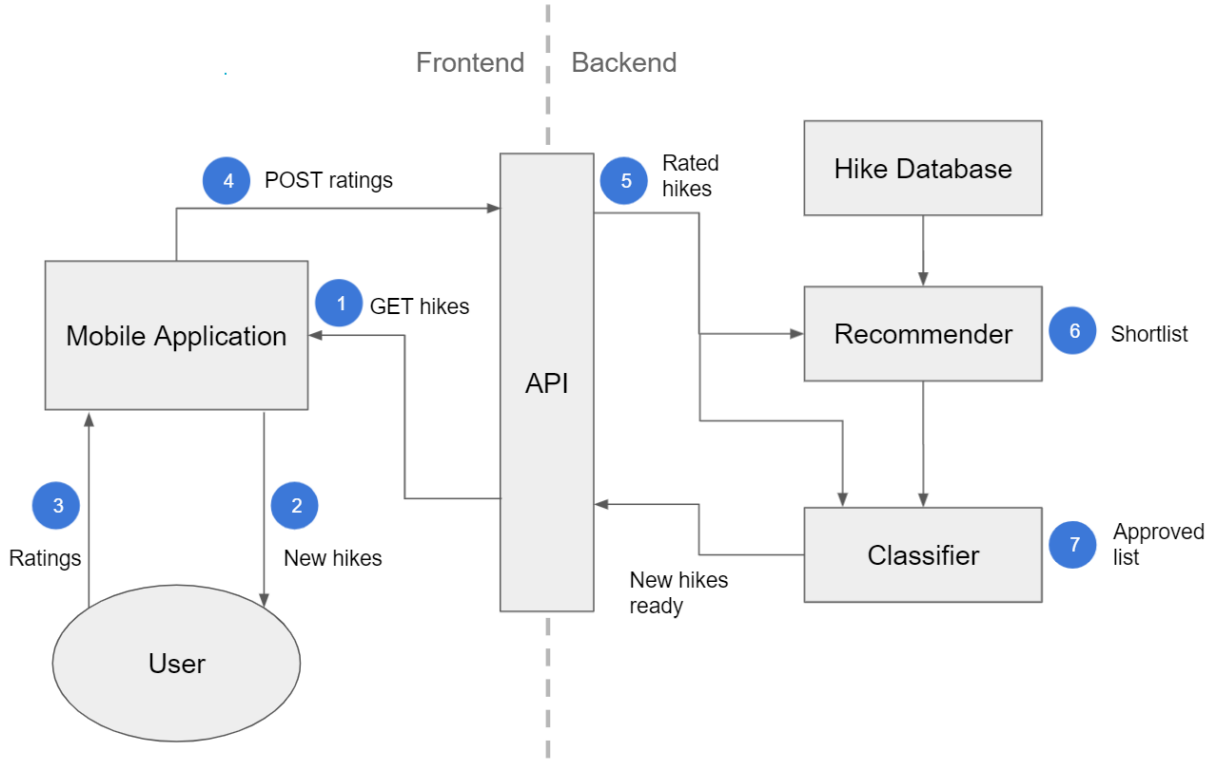
Moving behind the scenes, there are 3 API calls that connect the mobile application to the backend: posting user preferences, posting reviews, and getting hikes. In the app, when the user switches from the profile screen to any other screen, an API call is made to post their preferences to the backend. Upon making this call, the backend creates a new user if the username is not found. The new user's profile information is stored and used to curate the next batch of hikes; any hikes that do not meet the user's preferences will not be considered for recommendation.

On the hike screen, every time the user swipes on a hike, an API call is made to post that review to the backend. The backend receives the name of the hike and the rating, and uses this information to update the models. First, the image classifier dataset is updated; the hike name is used to identify any images in the unrated folder and move them into the liked or disliked folder, depending on the rating provided. Next, the recommender model receives the review, finds the matching hike in the database, and updates its model. When 10 reviews have been received by the backend, the image classifier trains for one epoch using its updated dataset. After it is done training, the backend prepares the next list of hikes to send to the user.

To produce a list of hikes, the recommender model starts by producing a shortlist of the 20 best hikes for the user. It does this by first filtering all the hikes in the hike database which meet the requirements for hike length, elevation gain, and difficulty specified in the user's profile. It then compares these hikes with the user's preferences, which are determined by the user's previously liked hikes. The similarity between the hike and the user is calculated using cosine similarity, and the most similar hikes are selected for the shortlist. This shortlist is then sent to the image classifier, where hikes are either rejected or approved based on the mean predicted rating of the hike's images. This final list of approved hikes is saved for the next API call from the application.

When the user nears the bottom of the stack of hike cards, a third API call is made to get more hikes from the backend. When the backend receives this call, it sends the list of approved hikes from the image classifier immediately. Shortly after the list is sent, the image classifier will retrain with its updated dataset. Although the hikes sent to the application are not based on the most recent ratings from the user, this is done to improve the user experience. The models train at the same time as the user swipes through their current hikes so that there is a list of recommended hikes ready to be sent to the application as soon as the API call is made. This slightly decreases the accuracy of the hike recommendations, but significantly reduces the wait time between reaching the end of the stack and receiving more hikes.

Project Diagram



1) App sends request to API for new hikes, API responds with list of hikes
2) App displays new hikes to user
3) User rates hikes by swiping
4) App posts rated hikes and user preferences to API
5) Model datasets are updated and models are trained
6) Recommender produces shortlist of best hikes and sends to classifier
7) Classifier approves or rejects hikes from shortlist and saves approved list

Repeat as necessary

Detailed Component Descriptions

Flutter Mobile Application

Upon opening the app, the user is presented with a login page where they may either log in or create a new account. Each new user inputs a name, an email, and a password to create a new account with the app. This information is stored in a Firebase Cloud document for each new user, which is used to allow for later authentication and to persist user preferences and matched hikes. Upon creating an account, each user is assigned a randomly generated UID. After creating or logging into their account, the user is directed to their profile page where they can set their preferences. The user may use the bottom navigation bar to view the three main pages: the profile page, the hikes page, and the matches page.

Hikes are received from the backend as JSON encoded strings, and are then decoded by the app into Hike objects. Each hike has a name, location, difficulty rating, length, elevation gain, 3 images, and a corresponding url which leads back to its AllTrails page. HTTP posts are sent back to the API with the user's UID and a list of encoded hikes with user ratings.

On the hikes page, the user can rate new hikes based on the three images shown in a Tinder-card format. The Flutter application outputs HTTP requests for new hikes which are filtered by the user's preferences, and the user then uses the tinder cards to provide feedback via more HTTP requests. For each hike card, the user may either swipe right to like or left to dislike; alternately, they can use the corresponding buttons below the card stack to rate hikes. The app manages the swipes using callbacks to add "liked" hikes to the matches page. All hikes that have been swiped get added to a list of "rated" hikes to send back to the API for training the machine learning models. When the list of unrated hikes runs out, the app will automatically send an HTTP request for more hikes; if this fails, the user may press the yellow "refresh" button which sends another request to the API for more hikes. Each card features the current hike's name and 3 images which may be viewed by swiping vertically on the image.

On the profile page, the user can change their hike preferences. The hike length and elevation gain can be changed using the range sliders with values between 0km and 100km and 0m and 1000m respectively. The difficulty can be changed by using the toggle buttons for easy, moderate, and hard hikes. As the user updates the sliders and buttons, the preferences are automatically synchronized to the Firebase Cloud document corresponding to the user's UID. We originally planned for the user to have the ability to customize their profile picture and background, but we decided against implementing these features due to time constraints. The user may also log out using the dedicated button below their name, or they may also delete their account if they so choose. Doing either navigates them back to the login page.

On the matches page, the user can browse their matched hikes in a list format. By tapping on a hike in the list, they can view more details about the hike (length, elevation gain, difficulty, etc.). Similarly to the hikes screen, the user may view the images corresponding to the hike by swiping vertically on the image. Each hike may be rejected if the user decides they do not like it

after seeing its details, or they may also view the hike on AllTrails.com using the dedicated button, which opens the web page on the device's default browser.

Flask Python RESTful API

In order to connect our Flutter application to our complex machine learning models, we needed an application that was robust. To structure our application, we used the Flask micro web framework to create a RESTful API that was capable of sending and receiving data from the Flutter application. RESTful (representational state transfer) API's are typically hosted on remote servers and function by sending and receiving data via HTTP requests. Our RESTful API is capable of creating users, receiving reviews for hikes (likes/dislikes) and recommending hikes based on past reviews. All of this data is sent over HTTP requests, so there doesn't need to be any formal connection between the Flutter app and the server.

The three resources managed by our Flask RESTful API are user data, hike data, and review data. Each of these resources have a unique endpoint that is reached through a unique URL matched with the user's username. For example, to reach the Hike resource you would use the URL address: "http://127.0.0.1:5000/hike/{username}" and inform the server whether you are making a POST or GET request. Each endpoint also has JSON formatted arguments that are required for a valid request. For example, to send a POST request to review a hike you need to send the hike's name and a boolean value representing if the hike was liked or disliked. If you send a valid request, the system will respond with a response code and (in some situations) a JSON formatted response. The response code informs the Flutter application if the request was successful or not. For example, if the Flutter application sent information for a user that did not exist, our RESTful API would respond with a 404 status code, informing the Flutter application that a user with that username was not found in the system.

The backend of the application is also connected to our RESTful API. When information is requested, the server polls the backend of our application that stores the user, hike and review information and connects them to the machine learning models.

Recommender System

The recommender system is the first of the two machine learning models in this project. It was designed to be very lightweight, as the image classifier was expected to take a while to run, and we didn't want users to wait longer than necessary when requesting a hike. Since the image classifier was required to be retrained every time a new batch of reviews came in, we wanted the list of potential hikes to be smaller than the original dataset (3000 hikes with 3 images/hike). It's purpose is to send a short list of hikes to the image classifier that could be used to predict liked images. It works by examining past "liked" hikes by the user and finding hikes that are most similar to these hikes.

Each user has a vector of 19 attributes that relate to the users' ideal hike. Each hike contains the same 19 attributes that relate to the information for that hike. The attributes included a mix of embedded categorical data (such as relevant keywords that are present in hikes, the difficulty

of the hikes, and the route type) and continuous data (such as hike length and gain). Each time a hike is reviewed on the app, the user's vector is updated - each value is increased if it is present in the hike, and decreased if it is not present in that hike. Each value is given the same weight between 0 and 1, so each attribute is weighed equally. When the shortlist is requested, the cosine similarity between each hike's attribute vector and the user's attribute vector is calculated, and the hikes with the highest similarity to that of the user are sent to the image classifier. The calculation for the equation is:

$$Similarity(User,\ Hike)\ =\ (User_{\ vector}\cdot\ Hike_{\ vector})/(\left|User_{\ vector}\right|*\left|Hike_{\ vector}\right|)$$

The input to the model is the list of available hikes, and the output of the model are the top *50* most similar hikes. After the hikes are recommended to the image classifier, these 50 hikes are removed from the list of available hikes for future iterations of the model.

Image Classifier

The image classifier is the second of two machine learning models in this project. Its purpose is to use the broad features found in a hike's images and predict if the user would like the views that the hike has to offer. In the backend workflow, the image classifier receives a list of hikes from the recommender system and outputs a list of approved hikes, ready to be sent to the mobile application at the next request. Thinking of the classifier system as a module, it has two sub-components: the ResNet18 model and the classifier dataset.

The image classifier's dataset consists of two categories - likes and dislikes - and is updated regularly by the user's most recent ratings. Each time the application posts a new review to the backend, the classifier dataset is updated. The name of the hike is used to identify the corresponding images in the unrated folder; these images are then moved into the liked or disliked folder, depending on the rating provided. When 10 reviewed hikes have been received, and the image folders have been updated accordingly, the datasets will recompile. At this point, the model can be trained using the updated dataset for one epoch.

At the core of the image classifier is a pretrained ResNet18 model with the output modified to work as a binary classifier. This machine learning model is used to predict if the user will like or dislike an image. The general principle behind the use of this model is that the user will prefer images with some features more than others, and the model will be able to predict which features will lead to a preferable rating. For example, if the user generally likes images with mountains and lakes, and dislikes grey skies and beaches, the model will begin favouring images containing mountains while rejecting images with beaches.

The reason for using a ResNet18 model is (counterintuitively) because it is simple. Since the classifier dataset is directly dependent on the number of hikes the user has rated, it is relatively small; a more complex model would easily memorize the images used in training and not produce desirable results on new images. To compensate for the small dataset size, we paired our simple model with lots of data augmentation to increase the effective size of our training

dataset. Another benefit of using a simple model is that it is faster, meaning shorter wait times for the user before they can keep looking at new hikes.

When the classifier model receives a shortlist of hikes from the recommender system, it will produce a list of approved hikes. The classifier takes each hike from the shortlist and predicts how the user will rate it. It does this by looking at all 3 of the hike's images, computing a score for each image individually, and predicting the hike's rating based on the mean score of all images. If a hike is predicted to be a dislike, it is removed from the list; if a hike is predicted to be a like, it is approved and moves on. The resulting list of approved hikes is then saved until it is sent to the application, where the new hikes can be viewed and rated by the user.

Image Database and Connections between Backend Components

Flutter is not well suited for the calculations required by our machine learning models. Especially since it will be housed on a mobile device, it is important that the frontend application is connected to a more powerful backend. Because our team is familiar with Python and Pytorch, we decided to design and implement both our machine learning models in Python.

Our Flask API connects the front-end Flutter application to the back-end calculations of the image classifier and the recommender system. Every time the server is started, a list of hikes is initialized using a csv file containing hike information scraped from AllTrails. A list of users is also initialized and updated every time a user is created in the system. Both the User and Hike objects store information required for the machine learning models.

The User object stores information about the user's preferences, number of hikes reviewed, a list of liked hikes, a list of disliked hikes, and their attribute preference vector. A list of available hikes is also initialized and stored in the User object upon creation. The list of available hikes is filtered based on user preferences inputted on user registration. List comprehension is used to convert the ~2,400 hikes to a smaller, more refined list using the user preferences (for example, hard and moderate hikes will be ignored if the user prefers easy hikes only). Each Hike object stores information on a hike, AllTrails links, and its attribute vector. The list of all hikes is a global variable shared among all files and remains static after creation.

When a list of hikes is requested from the flutter application, the Flask application finds the user making the request and loads the associated User object. It then uses the stored information in the User object to poll the recommender system for a shortlist of hikes, based on the user's attribute vector. These shortlisted hikes, along with liked and disliked hikes by the user, are sent to the image classifier to predict which hikes the user will like. The model is retrained after each batch of reviews to refine the model and increase accuracy.

The dataset for the image classifier is a folder system located in the same directory as the Flask application files. When the image classifier is run, liked photos are sent to the "liked" photos folder, disliked photos are sent to the "disliked" photos folder and unrated photos remain in the "unrated" folder. These folders are reset every time the image classifier is run.

Team Contributions

Andrew Forde

Created image scraper for pulling hike photo links from AllTrails.com, gathered links in csv file, and downloaded images for classifier dataset. Implemented image classifier, including functions for training and testing the model, updating the dataset, running the model on individual samples, and displaying output predictions. Implemented and refined Flutter mobile application, including preference selectors on profile screen, multi-image view of hikes on Tinder cards, and matches screen with selectable hike pages. Also added API calls to connect with the Python backend, test methods to simulate API calls locally, and the button on individual hike pages to link to their AllTrails webpage in a browser.

Taylor Mcouat

Created a Selenium web-scraper for pulling data from the top ~2400 hikes in British Columbia, and manually ran the scraper to gather the data into a csv file. Also assisted in the running and downloading of images from AllTrails for the moderate difficulty hikes. Designed and implemented the RESTful Flask API program from the bottom up. Built the relevant API endpoints required by the app and ran unified testing to ensure the server was functional for use by the app. Implemented the recommender model using cosine similarity, designed the user and hike models, and took the image classifier from Google Colab and integrated it into our Flask application.

Leo Sun

Did research on the Flask-Restful API and compared it with Django. Designed an initial object model of the user and hikes. Proposed an initial datapath between frontend and backend. Contributed to the recommender model construction. Did some preparation work on testing environment setup. Revised the final report.

Jayden Wong

Downloaded hike images for “hard” hikes. Developed the flutter application. Added tinder cards and swiping functionality. Implemented bottom navigation bar with routes. Implemented user account creation and email and password authentication using Firebase. Implemented profile page, which persists user preferences using corresponding Firebase cloud document. Edited final project video.