Wanderlust: Discovering New Hiking Trails

Nicolaas Kaashoek

1 Code

The code for my project can be found at https://github.mit.edu/nicolaas/6.S080-Wanderlust. Let me know if you have trouble accessing it; the TA's should be added as collaborators. Thanks for a great course this semester!

2 Introduction

With the enormous number of trails present in the United States, it can be overwhelming as a hiker to find new ones to try. While systems like AllTrails and Gaia exist and publish all these trails, they do little to aid in the discovery process. Different hikers prefer different kinds of trails depending on their preferences, and these sites do nothing to take those preferences into account. Wanderlust aims to address that problem.

Wanderlust's main goal is to aid in the discovery of new trails. To do so, it computes metrics and features for the trails, identifying things like switchbacks and sections of variable steepness. Based on these features, Wanderlust is able to recognize similar trails, which in turn drives a simple discovery system that allows users to quickly look for new trails based on ones they've enjoyed. Along the same lines, Wanderlust recommends trails to users based on what similar users have enjoyed. This is done using collaborative filtering.

Additionally, Wanderlust aims to address the wide difficulty range that many current hiking websites have. For example, on both AllTrails and GaiaGPS, there are only three difficulty ratings: easy, medium and hard. This is borderline absurd, as it results in the "Hard" difficulty rating being full of any trails that anyone *might* find difficult. For example, the Mount Washington Trail is marked as hard, with an elevation gain of 4,712 feet and a distance of 16.6 miles. The Lake Angeles Trail, is also marked as hard, but has an elevation gain of 2,450 feet and a distance of 3.5 miles. There is no reason that these two trails should be in the same difficulty category, and doing so bloats the upper end of the spectrum. To address this, Wanderlust aims to adapt its difficulty estimates based on how difficult a given user has found the other trails they've hiked.

At the moment, I've implemented a prototype version of Wanderlust that achieves all three of these goals. While I haven't been able to test the system extensively due to a lack of users, simple tests and manual checking demonstrate the effectiveness of the system.

3 Prior Work

As mentioned in the previous section, most prior work in the area has been done by websites like AllTrails and GaiaGPS, both of which are used by beginner and advanced hikers. While these sites do a fantastic job of displaying information about a hike to users, this is only possible if users actually know about the hike. Neither site is particularly personalized, and neither presents anything outside of the basics like a trail map and some elevation information. While both sites allow users to create accounts, neither seems to do much with the information about what hikes a user does, outside of recommending popular hikes in the same area. If a hiker wants to discover something less well traveled, neither site does a good job of exposing infrequently hiked trails.

What both sites do have is the ability to enter data about how long a hike took a particular user, and for users to upload their own GPS data to the site. Both of these are features that Wanderlust lacks at the moment.

4 Design

Wanderlust is made of a number of separate components, each described in more detail in the following sections. At a high level, Wanderlust begins by actually collecting the data. Once the data was collected, I built a parser that took as input GPX files (the standard output format used to describe trails) and processed the information contained in each file to build up the metrics I mentioned in the introduction. From there, the GPX points along with the metrics are stored in a Postgres database to be accessed by the web application that users interface with.

The web application enables users to search for trails, view similar trails, and to look at trails that Wanderlust recommends they try out. All recommendation generation occurs separately from the web application, which reads in the same data and produces a new set of recommendations for a given user based on whether or not they've hiked anything on their list.

4.1 Data Collection

The first step in building Wanderlust was actually obtaining the data that described the trails. To do so, I scraped GaiaGPS' backend API for all of the GPX files that I could find. All code for the scraper can be found in my git repository under the "scraper" folder. The entire component is written in Golang, and runs many workers in parallel. Gaia's API assigns each trail a value, and stores the trail under that value. Wanderlust exploits that fact to guess the indices and try and download as many trails as possible. Each worker guesses a different index, downloading the corresponding GPX file if it exists, and then pauses to avoid any throttling restrictions imposed by Gaia's API.

Using this method, Wanderlust was able to scrape around 32,000 hikes (the entire United States has a total of 31,176 trails registered). Unfortunately, around half of these trails were duplicates, and so had to be avoided. This left Wanderlust with 16,723 trails to start. In the future, I hope to more completely scrape the website and get access to all 31 thousand

trails that Gaia makes available. A quick manual check confirmed that the trails heavily overlapped with those present on AllTrails, so I avoided scraping that website.

4.2 Parsing and Metrics

Once Wanderlust had access to all of the raw GPX files, I wrote a parser to actually break those files down into numbers I could use to compute metrics and identify key trail features. Once again, I used Golang to implement the parser, which also works in parallel. All of the code can be found in the "parser" directory in my repository. Each worker handles one trail at a time, loading the GPX file into an in-memory data structure on which Wanderlust can run computations.

The first thing the parser does after loading the data is to compute basic metrics on the set of points. The exact metrics can be found in "parser/types.go". They include things like average number of meters climbed per meter hiked, elevation gain, distance, and altitude amongst other values. These basic metrics help form the feature vector that is used later in the nearest neighbors algorithm and the recommendation system.

After computing the basic metrics, Wanderlust goes on to try and identify some key features of the trails. It looks for flat sections, steep sections of varying grades ($\geq$ 15%, $\geq$ 50%, and $\geq$ 100% grade), and switchbacks. It also computes the total distance spent on these sections. Finding flat and steep sections is relatively straightforward, as it simply involves maintaining a window while iterating through the list of points and adding points to the window if they fit in with the current section being evaluated. The identification of switchbacks is slightly more involved.

Effectively, the algorithm draws lines between successive points along the trail, dividing the entire trail into variable length segments that are all considered "straight". I defined straight to have a $\geq 120^\circ$ angle between points for the pair to be added to a straight segment. This was necessary, as some GPX files only include the corners of the switchbacks, while others have points along switchback that aren't at the corners. Then, I measured the angles between all the successive straight segments, looking for sets of straight segments that had significantly acute angles between them. If more than 3 of these segments are found in a row, then they make up a switchback.

In the future, I plan to change this to use machine learning so that the algorithm is hopefully more flexible, as the numbers I chose to identify the switchbacks are relatively arbitrary. While manual inspection shows that they work well and certainly identify any obvious sections of switchbacks, there are segments that lie somewhat on the border that my algorithm fails to identify. I believe that this is fine for now, as hikers tend to care most about switchbacks when they are more defined, which Wanderlust does catch.

The basic metrics, grade sections, and switchbacks are combined to form the feature vector that drives Wanderlust's recommendation and identification system.

4.3 Frontend and Backend

Users interact with Wanderlust via a simple web application that allows them to query for trails, view recommended trails, and look at basic information about the trails such as a map and the metrics computed by Wanderlust. Both the frontend and the backend are written

in javascript, with the frontend using react.js. The backend uses the express.js library, and simply serves as a RESTful API that the frontend can query in order to gain access to the data.

All of the data is stored in a PostgreSQL database that is hosted on Microsoft Azure. While neither the frontend nor the backend are currently being hosted anywhere. The data is divided into 5 separate tables. First, a trails table that contains the trailnames, unique ids, and the computed metrics. This table also contains some aggregated metrics regarding ratings, such as the number of users who hiked the trail and the average rating given by users. Second, a points table that just contains the list of GPX points making up each trail identified with the same unique id used in all other tables. Third, a users table that stores basic user information and session ids to enable cookie-authenticated login. Fourth, a ratings table containing each rating issued by a user for a trail. I chose to store the ratings separately from everything else to isolate all updates to the data to just that table, rather than having to constantly update either each user or each trail. Finally, a neighbors table containing the results of the nearest neighbors algorithm from the next section.

The frontend uses the trails table to allow users to search for trails and to display the basic metrics to users. It uses the points table in order to draw the trail on a map for the user. The maps are implemented using OpenStreetMap, an open source service that exposes map data. The frontend also uses the neighbors table to identify similar trails to the one that the user is currently looking at, to make trail discovery even easier. In order to access the tables, it issues a simple REST request to the server, which does the actual communicating with the database.

All the code for the frontend can be found in "wanderlust-fe" and all the code for the server can be found in "server", which also contains the descriptions of the different tables. These descriptions can be found in "server/tables/TABLENAME.sql".

4.4 Trail Similarity

In order to compute the similarity between trails, Wanderlust simply uses Nearest Neighbors. Wanderlust loads in all the trails from the stored database, and feeds them into SciKit Learn's NearestNeighbors implementation. I chose to use the brute force algorithm as the computation only needs to happen once (the trails are constant, there aren't new ones being added). I also chose to use a weighted Minkowski distance as the distance metric as opposed to the default Minkowski implementation.

Before feeding the trail metrics into the nearest neighbors algorithm, I scaled the data so that the different metrics all lay within the same range. This means that trail distance (which is in thousands of meters), doesn't completely outweigh the average elevation gain per meter, for example. I also assigned weights to the different metrics to more heavily emphasize certain characteristics like distance and elevation gain as opposed to altitude.

Once the nearest neighbors instance was fit to the trail data, I queried it for each trail's 20 closest neighbors, and stored those results in the neighbors table of my database. Because no new trails are being added, I didn't need to worry about running the algorithm more than once. The neighbors table is then used by the web application to display the most similar trails to users in case they want to find something like the trail they are looking at.

Evaluating this piece of the project was difficult, as in order to truly test it, I would

have needed to actually have people hiking the trails to determine whether the nearest neighbors actually returned similar trails. As this was impossible, I instead went through some examples to manually confirm that the suggested trails were actually similar. Initial results were promising, and only got better as I adjusted the weights of the algorithm, leading to the point where I'm reasonably confident that two close by trails are actually quite similar. Again, this is all through manual inspection of the map. To actually test it, I would need to have people hike pairs of estimated similar trails and confirm their similarity.

The code for this can be found in "processing/neighbors.py".

4.5 Some Methodology

For both of the following sections, I populated the ratings database with some dummy information in order to evaluating different methods. To do so, I generated 1000 fake users. I then created 10 trail groupings, each full of 40-50 trails, selected using the nearest neighbors algorithm from the previous section (each group contained similar trails). Then, I divided the fake users into 10 groups of 100 users, and had each group rate 3 of the trail groups, assigning consistent ratings and difficulty scores across within a single group. At least 2 of the trail groups rated would be rated highly. Each user in the group randomly selected a subset of the trails to rate, in order to simulate the traditional sparseness of a ratings matrix. I then generated 10 more users, assigned each of these test users to 1 of the 10 trail groups, and had them rate a subset of those trails. I used this data to test out the algorithms, as it seems to simulate what a real system would have, as similar users would assign similar trails similar ratings and difficulty scores.

4.6 Collaborative Filtering for Recommendations

Besides just allowing users to view similar trails, Wanderlust also recommends trails to users based on their ratings of hikes they've previously done. To accomplish this, I implemented a standard collaborative filtering system using the Scikit-Surprise library. I fed the library the dataframe generated by loading in all the ratings in the system, and then trained the model on that data.

I tested two kinds of models for the collaborative filtering, one using k-nearest neighbors with means, and one using the SVD++ matrix factorization algorithm. Feeding them the entire dataset minus the test users, I evaluated the RMSE of both using scikit-surprise's out of the box cross-validation techniques. The SVD++ produced a slightly lower RMSE for ratings of 0.59, so I went with that algorithm.

From there, I created a flask server to serve the recommendation system, and plugged it into the front end. I then took each of the 10 test users and checked what trails were recommended to them. The algorithm recommends trails by looking at all unhiked trails and seeing which result in the highest predicted rating, and recommending the top 10 of those. Given the max trails in the system is 17K, this was fast enough for a starting point. I looked at each of the recommendation groups for the test users, and confirmed that the recommended trails were from the groups of users that had rated the assigned trail group highly as well. Having verified this, I feel satisifed that the recommendation algorithm does a

decent job of recommending trails that similar users liked but that aren't necessarily similar to trails the user has hiked in the past.

The code for this can be found in "recommendations/filter.py"

4.7 Adaptive Difficulty

In order to guess how difficult a given user might find a trail based on how difficult they found other trails they've hiked, I used Scikit-Learn's K Neighbors Regressor. To create the data that is fed into the regressor, I select all the trails the user has hiked (all the metrics associated with those trails) along with the difficulty they assigned to the trail, and train the regressor on that. Then I ask the regressor to predict the difficulty for the metrics of the trail the user wants the difficulty of, and return that prediction. This runs on the same flask server as the recommendations.

To evaluate the effectiveness of this, I first computed the RMSE of the regressor using scikit learn's built in cross validation methods. This returned an RMSE for a user who had rated the difficulty of 10, 20, 50, and 100 trails. Obviously, it was most effective when the user had rated 100 trails. I also performed a sanity check using the same methodology from the above section. This time, I had each user group rate the trail groups assigned to them with a consistent difficulty score, and then asked for an estimate of the difficulty score of one of the trails the users hadn't hiked. This confirmed that similar trails do indeed get assigned similar difficulties.

In the future, I hope to improve this section by feeding the regressor the trails rated by users that are similar enough to the requesting user. I am currently unsure what the threshold for "similar enough" should be for users, which is why I've avoided it. However, it would greatly increase the training dataset's size for the regressor, which should improve accuracy of the results in the future.

The code for this can be found in "recommendations/difficulty.py"

5 Implementation

As noted in the design section, the frontend and backend are implemented in javascript, the nearest neighbors, difficulty estimation and collaborative filtering portions are all implemented in python, and the parser and scraper are built in Golang. The entire project is around 6000 lines of code.

6 Future Work

There are three things I hope to work on in the future. First, geographic boxing of trails. The similar trails feature currently suggests any trail across the country, which isn't always useful. I plan to bucket the trails based on their geographic location and only suggest similar trails from the same bucket.

Second, I want to make a well-developed search feature. While users can search for trails currently, they need to know the exact name of the trail as registered in the database. I hope

to improve the search feature in the future to allow for more forgiving parameters, which would greatly improve the usability of the system.

Finally, I hope to fine tune and improve the machine learning algorithms that drive the difficulty and recommendation pieces of the system. Both could probably benefit from having their parameters more finely tuned, and as mentioned in the previous section, the difficulty estimation could be made better by taking into account the data from more than one user.

7 Conclusion

While there are obvious areas in which Wanderlust could be improved, I feel satisfied with what I was able to build for the final project. Many of my hypothesis regarding difficulty were confirmed, as I was able to more accurately segregate trails and identify which ones would be more difficult for different users in a way that other websites simply don't do. I feel as if I've confirmed that this idea is worth spending more time on, and hope to do so in the future, developing and improving upon the features I built this semester. Perhaps I'll even be able to release it as a fully featured website in the future.