XploRun: Running routes recommendations and navigation

Big Master Project

at

University of Bayreuth

Chair of
Applied Computer Science 8

Vivek Mahto, Feruz Davletov, Andrej Garaj
Supervisor: Prof. Dr. Daniel Buschek
25 October 2022

Contents

Introduction **1**
0.1 Compared to similar applications 1

1 Interaction concept **2**
1.1 Login and Sign-up 3
1.2 Main activity 4
1.3 Page 'Run' 5
1.3.1 Loading of the routes 6
1.3.2 Route Navigation 6
1.3.3 Voice navigation 7
1.3.4 Analysis of the run 8
1.4 Page 'Statistics' 8
1.5 Page 'User profile' 9
1.5.1 Badges and gamification 10

2 Technical Concepts **12**
2.1 Frontend 12
2.1.1 Displaying of routes recommendations 12
2.1.2 Layouts 13
2.1.3 Libraries 13
2.1.4 Routing and Navigation 14
2.2 Backend 15
2.2.1 Route Generation 15
2.2.2 Feature Extraction 17
2.2.3 Matching routes to user preferences 18

3 Discussion **20**
3.1 Testing 20
3.2 Challenges 20
3.2.1 Choosing frameworks and libraries 20

3.2.2 Walking vs Running 21
3.2.3 Difficult to handle situations 21
3.2.4 OSM Map Lies 22
3.3 Outlook 26

Introduction

XploRun, as the name suggests, means the combination of Explore and Run. This Android application lets the user discover running or walking routes in his neighborhood. The application recommends user routes based on his running or walking preferences. The user can navigate these routes with his phone in his pocket, with the assistance of voice instructions.
The idea is to make running interesting for the user, such that he is not bound to only one route and will be inspired to try different variations and compare how he performs in different settings. People often become unenthusiastic thinking of running, but with XploRun we add an element of curiosity to this task, because one never knows what route he will discover, and what will he discover on that route.
XploRun is intelligent, it continuously works on user feedback and updates user preferences such that it can provide him with the best recommendations.

0.1 Compared to similar applications

We analyzed multiple running apps like **Travis, Nike Run, Map Runner**, etc. Many of these apps don't provide any route suggestions, the user just runs and they track it and give him the analysis. Few apps provide running routes, but if so, they are pre-created routes from other users. These traditional running apps do a good job of tracking the run and analyzing it, but we wanted to change how one looks at running/walking and make it more interesting for the user.

What sets XploRun apart from other apps is the following:

- We are generating routes for the user on the fly, so he never knows what new route will he get to explore.
- These routes are tailored to the user's preferences.
- Assisted voice instructions to guide the user throughout the run.

1 Interaction concept

In this chapter, we discuss the interaction flow of our application, and introduce different pages and activities, while explaining their purpose. We also analyze design choices and motivation behind them.
The main idea of the design concept was to keep the interaction as simple as possible, to enable the user to achieve his task quickly and without distraction. Therefore, after opening the application, existing user (whose data are cached) only needs to tap once to get the routes recommendations, select the route most suitable for him by swiping over the recommendations, and tap once more to start the run (Figure 1.3). During the run, he is guided by voice instructions and doesn't need to visually interact with the application anymore.

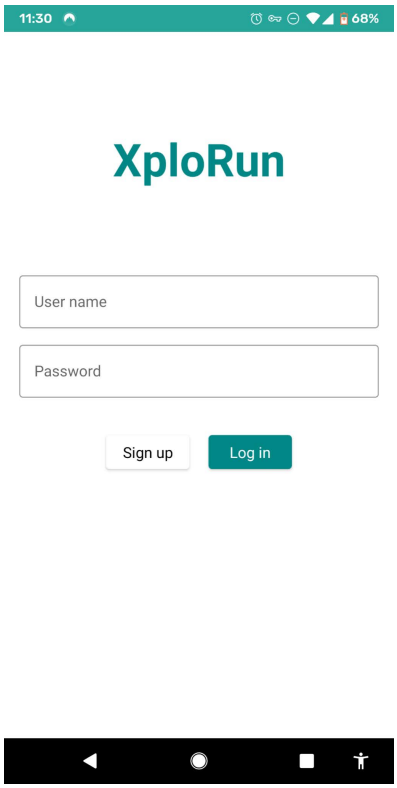

Figure 1.1: Login page

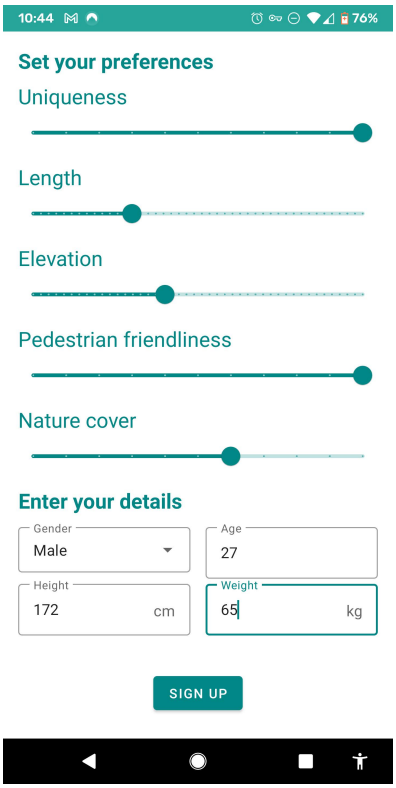

Figure 1.2: Sign-up page

1.1 Login and Sign-up

When the application starts, the login activity is shown (Figure 1.1). This activity serves both for login in case of existing user and sign-up of a new user. User logs in using his username and password. If he is signing up, he can click the sign up button after which an and additional field appears, which prompts him to repeat the given password. If the sign up was successful, the user is shown an activity asking him for his preferences about the routes that will be recommended to him (Figure 1.2). The preferences user is setting during this process reflect the features of the routes we are considering when generating the recommendations. These are:

- Uniqueness: Possible repetition of of location (nodes) in the path. Higher setting means less likelihood of route with repeated node.
- Length: Length of the route, possible values are between 2km - 8km.
- Elevation: Possible values are between 1m - 1000m.

- Pedestrian friendliness: Higher setting means less likelihood of generated routes being intersected with roads for cars.
- Nature Cover: Higher setting means more likelihood of nature cover in the routes (with respect to user's location and possibilities of nature cover in his surroundings).

User is informed about meaning of these preferences using toast messages. This allows us to solve the cold-boot problem - situation where we don't have any existing information about user preferences that can be optimized. Besides these preferences, the user is prompted for his information: sex, age, height and weight. These are used for calculation of the calories burned during the run.
In order to ease interaction, we are caching the login data, therefore the user doesn't need to manually login every time he opens the application.

1.2 Main activity

After successful login, the main activity is opened. The activity serves as the core activity of the application and contains a bottom navigation bar enabling navigating between these three pages:

- Run: The main page of the application containing the routes suggestions.
- Statistics: Page containing the statistics of previous runs of the user.
- User profile: Page containing user's details, weekly performance and badges.

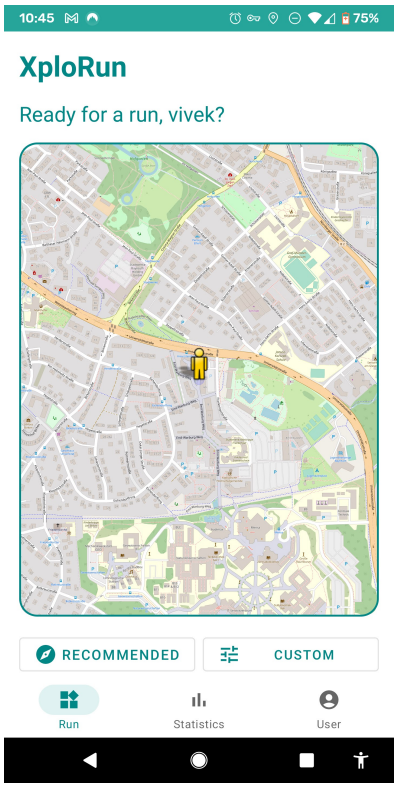

Figure 1.3: Main page

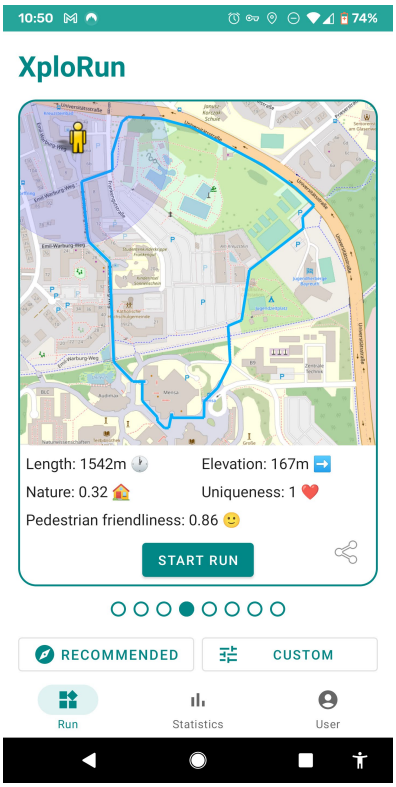

Figure 1.4: Recommended routes

1.3 Page 'Run'

This page contains the core functionality of the application, i.e. displaying the suggested routes for the user to run. If no routes are yet loaded (e.g. after login), the page contains a map with user's current location and two buttons (Figure 1.3) to generate routes. User can choose between two kinds of routes to be generated. First option are the routes recommended to him based on his preferences. These can be accessed by clicking the **RECOMMENDED** button, after which the routes are loaded. Second option are custom routes, available by clicking the **CUSTOM** button. After choosing the custom routes option, new activity similar to Figure 1.2 is opened, where user manually chooses features of the routes to be generated (his preferences are disregarded). When the user is finished with selecting the features, the routes are loaded. Loading process is described in more detail in 1.3.1.
After the routes are loaded, the page is populated with a set of cards, each representing a route suggestion (Figure 1.4). Each suggested route is drawn and highlighted in a separate map, under which features of the route are

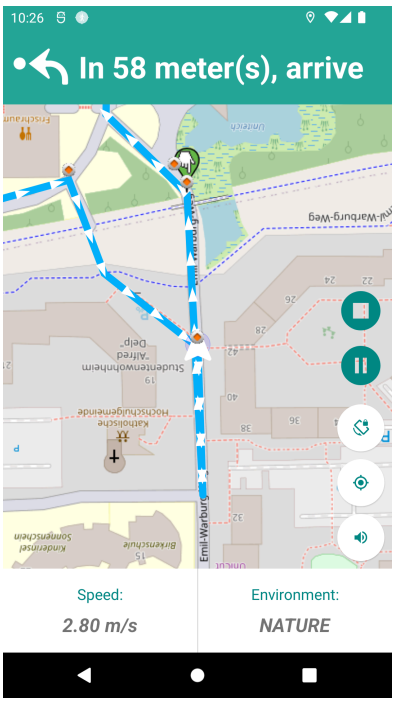

Figure 1.5: Navigation page

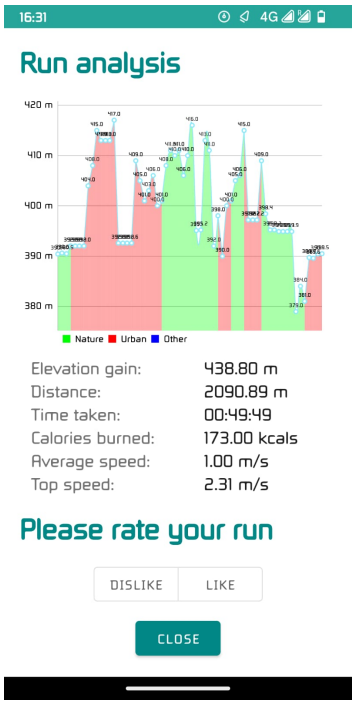

Figure 1.6: Run Analysis page

displayed. Each suggestion also has a separate button to start the run on the selected route.
Each suggestion also contains a share button, enabling the user to share the route with his friends, or save the route for future use.

1.3.1 Loading of the routes

Creating routes is a time consuming task which is explained further in the technical concept section. While the routes are being generated, periodically changing running tips are shown to keep the user occupied. This point may be the most critical for the users, since the loading time varies according to number of factors (such as distance of the desired routes), and may take up to several minutes. During this process, the application awaits response from the API, where the backend generates eight routes that fit the desired features (either user-specific or selected manually) the best.

1.3.2 Route Navigation

Route navigation page consists of map-related controls and route-related controls. Map-related controls are enable or disable map orientation, centering

user location, and voice navigation. Route controls are stopping (quitting) and pausing (not implemented yet) active route. On top of the page there is a text consisting of next maneuver(see Figure1.5).

When the user starts the run, the map will show the user's location and route with checkpoints. Additionally, the route will show direction indicators to help the user orient. In our tests, we had problems with orienting and finding the starting location. We have addressed this issue by giving the user a heads-up via an audio message that says "Please orient yourself and go to the starting location!" and have started the run when the user arrives at the starting location. When a user reaches the starting location, the app automatically starts the run and notifies the user with the audio message 'Have a great run!'. (see Figure 1.5)

After the user starts the run, there are two ways to stop it. Either by completing the route or by quitting. Completing the route means reaching the end checkpoint. Reaching the end checkpoint will automatically stop the run and show run analysis (see Figure 1.6). Quitting is explicitly stopping the run in which case the user will get a partial point and SURRENDER badge. If the user has already completed more than 80 percent of the route, quitting will not be considered.

Route navigation is inspired by Google Maps navigation. Due to technical limitations, we could not utilize Google's service for navigation. Firstly, Google does not allow navigation experience to be embedded inside other apps. Secondly, google limits number of checkpoints to maximum of 25. Instead we used open source routing tool OSRM[1] and OSMBonusPack[2] for map view. This had additional benefits like customizing the navigation according to our needs. For example, we used a third-party text-to-speech service for the navigation voice and we can also apply different map view styles if necessary.

1.3.3 Voice navigation

We used Azure TTS service for voice navigation. This service has additional features like emotional speech generation. Having experimented with this a bit, this part can be extended further. When the user starts the run, for example, the voice is excited. Further development ideas could include context-based voice emotions, like anger when the user is moving slowly or not moving for a while. Since the purpose of the app is to encourage the

[1]http://project-osrm.org/
[2]https://github.com/MKergall/osmbonuspack

user to run or walk, the effectiveness or usefulness of such a feature can be discussed.

1.3.4 Analysis of the run

When the user finishes his run, an activity informing him about the completed run is automatically opened. This activity is also opened in case the user has decided to manually quit the run (providing he started the run by moving to the start location). This activity contains a chart depicting the elevation changes during the run, along with different environments that (possibly) have been changing during the run (Figure 1.6). The number of points in the graph stays fixed, regardless of the length of the route, and the different environments are color-coded, serving as a fast overview for the runner. Along with the chart, the main statistics about the run are displayed:

- Elevation gain: Total elevation gained, meaning the total number of meters ascended during the run.
- Distance: Distance covered in the run, in meters.
- Time taken: Total duration of the run.
- Calories burned: Number of burned calories, calculated using the personal data of the user.
- Average speed: Average speed during the run in meters per second.
- Top speed: Top speed achieved during the run in meters per second.

Besides displaying the statistics, the user is asked for his feedback. He can either choose to dislike, like the run or close the activity without providing feedback. If the user provides feedback, a toast message informs him that his preferences have been updated, and the changes will be reflected the next time he chooses to load new recommended routes. In case the user receives new badge in the run (1.5.1), he is also informed using a toast message.

1.4 Page 'Statistics'

The application allows user to view his statistics for the past runs in page Statistics. At the top, overall statistics of the user are displayed. These include his best pace, overall distance he covered while using the app, total amount of burned calories, distance of the longest completed run, distances covered in nature and urban environment respectively, and his experience

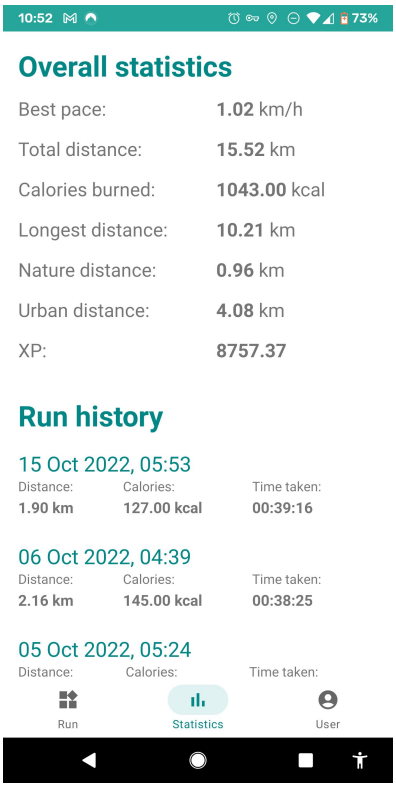

Figure 1.7: Run history

points. Part of this page is also the run history of the user, which consists of list of entries, where each entry represents a run titled by date and time of when it was finished, and few key statistics: distance, burned calories and time taken for each run (Figure 1.7).

1.5 Page 'User profile'

Page User profile (Figure 1.8) shows the user his personal details chosen during the sign-up process: username, gender, age, height and weight. This section also provides the weekly statistics of the user's runs: covered distance, burned calories and gained experience points during the week. The page displays two charts visualizing the user's performance throughout the week. First one contains covered distance, and the second calories burned for each respective day of the week, allowing the user to compare his daily performances. Both charts contain seven columns for each day in the week, with the current day of week always displayed in the last column. Next section contains the user's gained badges, and a logout button, which restores

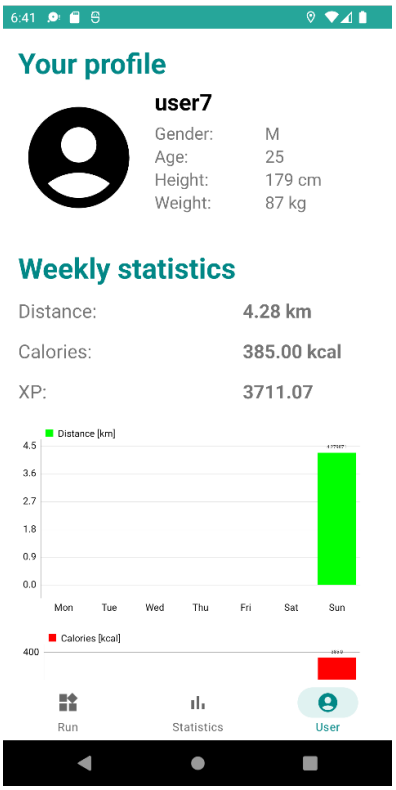

Figure 1.8: User profile

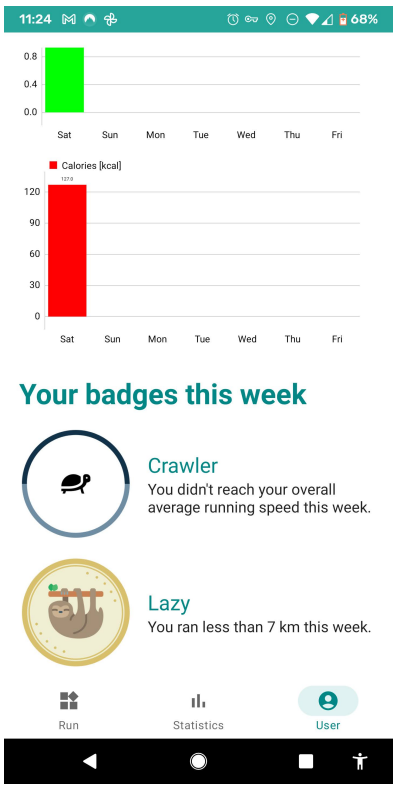

Figure 1.9: Weekly stats and badges

the cached login data and restarts the application.

1.5.1 Badges and gamification

To motivate users, user will be awarded with XPs for each of their runs. We did this considering the future prospect of this project. Like showing leader-board in a area, or a friends leader-board. We use the below formula to calculate the XP.

```
XP = run_distance(in meters) + 2*elevation(in meters) - time_taken
(in seconds)
```

Badges will also be awarded for the users weekly performance. We will award both good badges and bad badges. The idea of giving badges weekly to make sure that the badges don't stay with the user forever. Each badge comes with an expiry date, which is 7 days, this makes sure that

Figure 1.10: Badges

- User has to run every week to keep the good badges.
- In case user got a bad badge, it won't stay forever unless user breaks the bad habit.

In total user can obtain 7 different badges, 4 good and 3 bad. Below is the list of badges and how those badges are given.

- CRAWLER: Never touched the average running speed of 12 km/h in a week.
- FORREST GUMP: 10 hours spent in running.
- HIGH INTENSITY: Highest average pace of 16.1 km/h.
- MOUNTAINEER: Elevation gain of 1000m.
- LAZY: Only 7 km distance covered in a week.
- NATURE: Total distance of 10 km covered
- SURRENDER: Quit the run midway.

All the above numbers were taken from personal experiences of some of the frequent and non-frequent runners.

2 Technical Concepts

2.1 Frontend

Front-end was created using Java programming language, in IDE Android Studio. Throughout the project, parts of the project were significantly modified. The application is controlled by the main activity, which at the start of the application starts the login and sign-up activity, and waits for its results. The layout of the main activity consists of only the bottom navigation bar and container for a fragment, and the activity is responsible for switching between the fragments. The tree fragments (or 'pages': Run, Statistics, User profile) are switched using their tag. If the target fragment does not yet exist, it is created, otherwise, it is found by its tag and displayed. Therefore, on the switch between the fragments, the fragments are not recreated if they already exist, which enables smooth transitions between the pages with preserved information (e.g. recommended routes will stay loaded and ready to be started regardless of switching between different pages). Besides activities and fragments, we are heavily utilizing the singleton pattern of software design, to store data globally in final static instances for classes, for data used in multiple places in the application, such as the user details, recommended routes retrieved from the API, and singleton for Volley requests.

2.1.1 Displaying of routes recommendations

The recommended routes are displayed in the main activity, either by loading the recommended routes or by loading custom routes with manually selected features. After the start of recommendations retrieval, activity with a fragment of loading is shown, which contains running tips inside *TextSwitcher*, changing periodically, and a Material Progress Indicator. Once the routes are retrieved from the API, they are stored in a singleton class and displayed using *ViewPager*, which serves as a container for fragments, where each recommendation is a standalone fragment inflated in its own layout. *ViewPager*

uses a custom adapter, which extends *FragmentStateAdapter* and is responsible for creating the fragments for individual recommendations while passing the type of the routes being shown (recommended/custom). Routes fragments are created from one common class, which initializes the respective recommendations based on the position of the ViewPager, which translates to an index of the recommendation stored in singleton containing routes. The process of visualizing the route is described in section 2.1.4.

2.1.2 Layouts

Layouts for activities and fragments are created using *ConstraintLayout* type (transformed from the original *LinearLayout* used at the beginning of the project), for speed and adaptiveness on devices of different screen sizes. Depending on the activity, the layout was either used as a standalone or nested inside *SwipeRefreshLayout*. Besides these layouts, a number of additional layouts were needed: layout for features of the route nested inside layout for route suggestion (individual route suggestion 'card') and layouts for lists (badges, run history).

2.1.3 Libraries

For styling of the app, we used Material Design (later switched to the latest version 3) UI components. Since the core of our application is the main page with swipeable cards containing route suggestions, we used library *Material View Pager Dots Indicator*[1] to provide an indicator of the current selected page in the *ViewPager*. To create charts in the application, we used the library *MPAndroidChart*[2], which was selected for its rich documentation and customizability (which was mainly needed for the run analysis chart). For serializing and de-serializing JSON objects, we used the library *Gson*[3]. API calls to the backend were implemented using library *Volley*[4], chosen also because of our previous experience with it.

To manage dependencies between Activities and other objects we used Hilt (or Dagger). It is a simple Dependency Injection library provided by Google. This simplifies our code structure and will help in the future to maintain.

[1] https://github.com/tommybuonomo/dotsindicator
[2] https://github.com/PhilJay/MPAndroidChart
[3] https://github.com/google/gson
[4] https://google.github.io/volley/

2.1.4 Routing and Navigation

Here we will give a more detailed implementation of routing and navigation. After retrieving the list of recommended routes, this data is stored in JSON format on the user's phone. But to visualize this we need to create *Road* object required by *OSMBonusPack*[5] map view. For this we used *OSMR*[6] tool.

OSRM has additional useful features for optimizing the route and generating turn-by-turn instructions for navigation. Optimization is required to remove redundant instructions otherwise we end up with lots of *Continue* instructions. This happens because maps usually have plenty of junction points. Another important reason to optimize the route is to minimize the query time to find the nearest checkpoint to the user's location. An optimized route usually has 5 to 10 times fewer checkpoints than an unoptimized one, depending on the route generated.

After the optimized route is retrieved, it is converted into a series of *Step* objects for navigation. These steps are stored in a simple array. Initially, we used a queue to store them, but it turned out to be more complicated than expected to handle certain edge cases, e.g. when the user deviates from the route. Using an array on the other hand is static, and we can treat each checkpoint as a location-based message. So each checkpoint/step contains location, turn instruction, and distance to the next checkpoint. This way we can do a simple search in a loop to query if the user is near a checkpoint.

The query for finding the nearest checkpoint has four 'if' conditions:

1. Is the user's location in the radius of a checkpoint. Specifically, is the distance between the checkpoint and the user less than 16m + speed (meters per second). We added the user's speed into account for more accurate timing of voice instruction.
2. User should not skip more than 5 steps. We should allow the user to slightly deviate from the route.
3. Is checkpoint has higher order. We should not allow going backward.
4. Is the total distance between checkpoints less than 100m (implemented but not tested yet).

Condition four solves the problem where there are few checkpoints but the distance is long, this happens when we have a straight route. We return the

[5]https://github.com/MKergall/osmbonuspack
[6]http://project-osrm.org/

checkpoint that meets the above conditions to the Background service for voice instructions and the navigation page for textual instruction. The only difference between voice and textual instructions is voice outputs the current checkpoint instruction and textual has the instruction for the next step.

2.2 Backend

Our back-end is written in python with flask endpoints. Below is the algorithm to get recommended routes for the user:

1. Generate 50 routes based on the user location.
2. Extract the features from the route.
3. Get user's preset preferences
4. Match the preferences with the extracted route features.
5. Find 8 best matching routes and send the response.

2.2.1 Route Generation

The generated routes are a loop based on the user's preferred length. We can only use the length part to generate routes, taking more preferences increases the route generation time significantly.
We are creating the routes by using the **Open Street Map(OSM)** data. The python library **osmnx**, provides all functions required to use the OSM data.

Algorithm

1. Choose a starting point:
   The starting point is based on the user's location. We create a bounding box (bbox 50m x 50m) around the user. And any node/point is chosen at random within the bbox.
2. Find the next candidate points:

   The routes are created by finding three points as can be seen in Figure 2.1. The first point is the starting point. The second point is the candidate point.
   How do we find the candidate points?
   Given the user length preference L, The candidate points are the points that are at a distance L/3 (error of 300m is allowed) from the starting

Figure 2.1: Sample route 1

Figure 2.2: Sample route 2

point. This distance is calculated by using **partial shortest path (PSP) algorithm**. There can be many candidate points(all chosen at random).
Since we have to find the best routes for the user, we have to find multiple candidate points. PSP implements Dijkstra algorithm, which is run multiple times to get the candidate points. Thus increasing the computation time.

3. Find the third point:
   The third point has to be approximately equidistant from the candidate points and the starting point. We again use here PSP thus all in all we generate almost 50 routes. We have to keep a threshold on the number of candidate points and the total number of routes generated as it eats up the computation time.

4. Post-processing of the generated routes Removal of the unfavorable routes:

   (a) If two routes have 0.8 similarity then one of them is removed.

   (b) A route with low uniqueness is removed, if a node is repeated many times, then that is unfavorable.

   (c) Removal of unnecessary chicken neck in the routes.

Finally, all the filtered routes are sent to the feature extraction.

2.2.2 Feature Extraction

As discussed in the app flow section the user has five preferences, that he can set during signup. For each route, we have to calculate these features (for routes we call them features instead of preferences).

The features like length, uniqueness, and pedestrian-friendliness can be directly calculated from the OSM data using the osmnx library. OSM doesn't have the elevation data, although osmnx offers methods to fetch elevation, it only works with the Google API key. The difficult part was to access the nature information from the route, while researching we stumbled upon the **https://osmlanduse.org/**, which has the color-coded landuse of the map. But the problem was how to access this information programmatically for a small route encompassing bbox.
The osmlanduse is a project of the University of Heidelberg, on discussion with them we found that they provide a WMS service(open but not published) which will give a color-coded image in response to the route bbox co-ordinates. bbox coordinates are the four extreme points of the route by which we can form a rectangle that encompasses the route completely.

```
bbox = [WEST LONG, SOUTH LAT, EAST LONG, NORTH LAT]
```

Below is the code snippet, used to send the request to the service and a figure showing the response we get from that (Figure 2.3)

The response is an image, we extract the color from each pixel and map that to the color legend (Figure 2.4) to find the percentage of pixels belonging to nature.
All the features/preferences are scaled between 0-1.

Figure 2.3: WMS response

```
# Name for each color
map_legend = {'#e6004d':'urban_fabric',
              '#ffffa8':'arable_land',
              '#4dff00':'forests',
              '#cc4df2':'ind_com',
              '#ffa6ff':'artif_non_agri_vegetated',
              '#a600cc':'mine_dump_constr',
              '#e6e64d':'pastures',
              '#e68000':'permanent_crops',
              '#00ccf2':'water_bodies',
              '#e6e6e6':'open_spaces_no_veg',
              '#ccf24d':'scrub_herbs',
              '#a6a6ff':'wetlands',
              '#e6e6ff':'coastal_wetlands',
              '#ffffff':'unmapped area'
}
```

Figure 2.4: Color Legend

```
# convert the bbox co-ords to
# epsg3857/projected co-ords(These co-rds are used in the OSM)

projected_coords = hf.geoCords2ProjCords(bbox)

URL = "https://maps.heigit.org/osmlanduse/service"

PARAMS = {'SERVICE':'WMS',
            'VERSION':'1.3.0',
            'REQUEST':'GetMap',
            'FORMAT':'image/png',
            'TRANSPARENT':'true',
            'LAYERS':'osmlanduse:osm_lulc_combined_osm4eo',
            'BUFFER':25,
            'WIDTH':256,
            'HEIGHT':256,
            'CRS':'EPSG:3857',
            'STYLES':'',
            'BBOX':projected_coords}

response = requests.get(url = URL, params=PARAMS)
```

2.2.3 Matching routes to user preferences

Here we find the 8 best possible routes out of all the generated routes. For each route, a similarity to user preferences is calculated.

$\text{diff} = \text{abs}(\sum_{n=1}^{n=5} route_f - \sum_{n=1}^{n=5} user_p\ )$

Since all the features/preferences are scaled between 0-1, we calculate the

absolute value of the difference between summed route features to summed user preferences. The smaller the difference, the higher the similarity.

3 Discussion

3.1 Testing

To make testing and evaluation easy, we dockerized our backend and deployed it to the kubernetes, because running a computation intensive backend on a local machine is time consuming, and increases with number of users. Docker containers not only helped us easily deploy the backend, but route computation time was reduced significantly.

To collect crash logs on different Android phones we used Firebase Crashlytics. It is cloud service provided by Google. Additional features include build automation, distribution for testing and many more. While testing we also found out that our app will not work on devices without Play Store. For example in Huawei device our app could not get correct location and Android api does not behave like in standard api.

3.2 Challenges

3.2.1 Choosing frameworks and libraries

Our first attempt at implementing navigation functionality was using Google Maps, however we encountered a number of problems:

- Google does not allow Google Maps navigation to be embedded into other apps. We attempted to contact Google for a possibility of exception for a student project, but we were unsuccessful.
- Number of checkpoints the route may consist of is limited to 25, which represents a constraint for our use-case.

Because of this issues we re-implemented map views and navigation part with open source tools and libraries like OSM, OSRM and OSMBonusPack.

Key component of our application, the voice navigation, was implemented using Microsoft Azure TTS because open source solutions proved not to be good enough for our expectations.

In the backend, all the information except elevation are obtained from OSM data. The elevation is fetched via Google Elevation API keys, which is payed per use. We found that existing open-source solutions are not viable for our needs.

3.2.2 Walking vs Running

The initial idea of creating this app was for the running, but while evaluating and testing the app, we found that for running and walking routes different aspects need to be considered.
While walking, user has more time to interact visually with the app to verify inconsistencies between the map and voice instructions. On the other hand, while running the accuracy has to be high, since timing of each voice instruction has to be adjusted depending on the runner's pace.
Some of our generated routes lead through narrow spaces or off road around a building, which wouldn't be an issue for walking and exploring, but creates complications for running. Another issue to considerate is that the route for running should be mostly straight with less number of turns, and with high pedestrian friendliness.

The Figure 3.1 and Figure 3.2 shows two unsatisfactory cases of routes for running, the first one leads through a swimming pool and the second one contains a sharp turn not suitable for running.

3.2.3 Difficult to handle situations

In Figure 3.3 and Figure 3.4, we have two left paths right next to each other, at this point voice instruction 'turn left' might be confusing for the user. A walking user might be able to open the phone and check, but while running it would be difficult.

In Figure 3.5 and Figure 3.6, the voice instruction received is 'turn slight left', at that location in realty, this path appears to be straight. Since this instruction is received 20 m before reaching the actual checkpoint where there is another hard left as can be seen in the Figure 3.6, this leads to confusion.

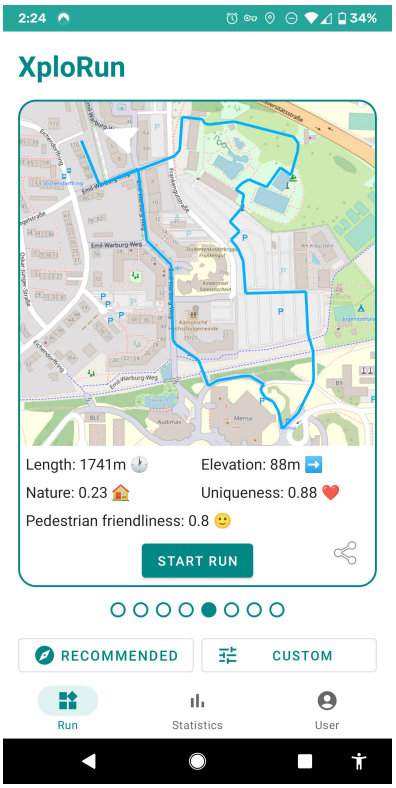

Figure 3.1: Path leads through a swimming pool

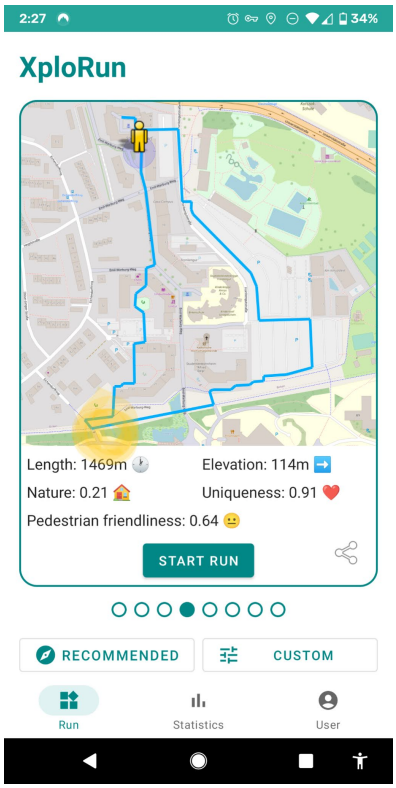

Figure 3.2: Turn not suitable for running

3.2.4 OSM Map Lies

Routes on the map might appear differently than they really are. On several occasions we found that there was a path on ground that wasn't shown on the map, which makes it difficult to navigate the user.

In Figure 3.7 and Figure 3.8, on ground we can see there is a clear left, and we should have received simple instruction like 'turn left', but on map there is not a direct left there, this makes the voice instruction navigation difficult.

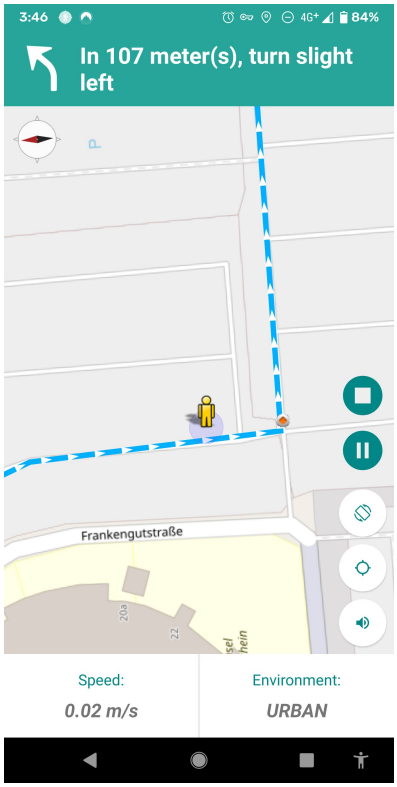

Figure 3.3: Two left

Figure 3.4: Two left on ground

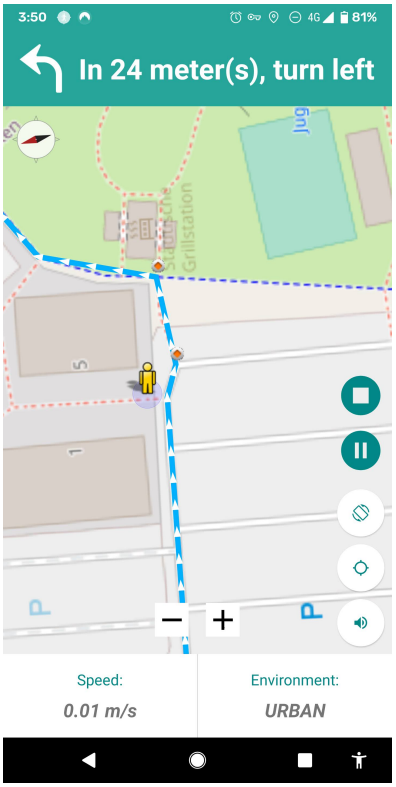

Figure 3.5: Slight left

Figure 3.6: Slight left in reality

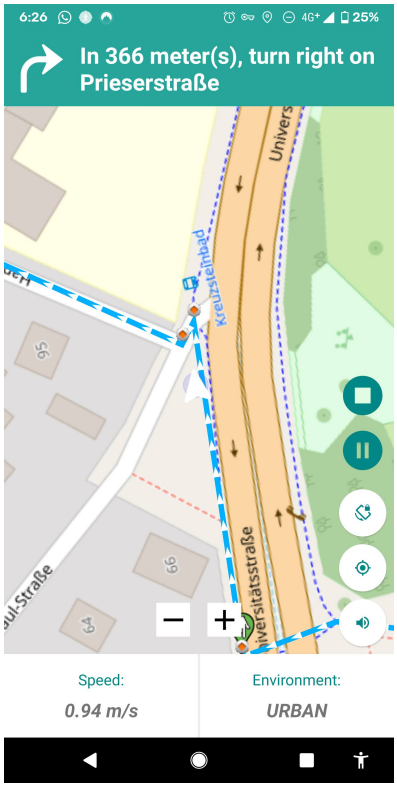

Figure 3.7: On map

Figure 3.8: In reality

3.3 Outlook

1. **Challenge the user with new routes:**
   One of the feedback we received while testing from one of the user was that people who walk prefer to change the walking routes frequently but runners prefer to stick one route for some time, so that they can see the improvement with the old performance.
   To do this we can create weekly challenge for the user, where we will create route with increased difficulty based on the user past performances. Adding this functionality can help user to challenge himself by running the same route for a week.

2. **Advanced Running analysis:**
   Implement advanced analysis which shows the improvement with previous runs, how user has performed in different environment, lengths, elevation and nature.

3. **Use user information in voice instructions:**
   Making the voice instructions more dynamic, such that users gets more personalized voice instructions, e.g. 'Running 50 more meters will give you the nature badge'.

4. **Different voice mods:**
   At the moment, we have just a robotic voice instructions, to make it more interesting, we can provide different voice mods for user to choose, like 'Angry Sargent', 'Anime' etc. More interesting addition would be adding emotions to the instructions based on user's speed and environment.

5. **Add more gamification elements:**
   We are awarding user XP on their run. At this point of the project, it doesn't have any real use for the user, besides possibly motivation to gain the highest number. We implemented this considering the future possibilities. We can for example create weekly leaderboard based on user city, which may lead to healthy competition between runners.