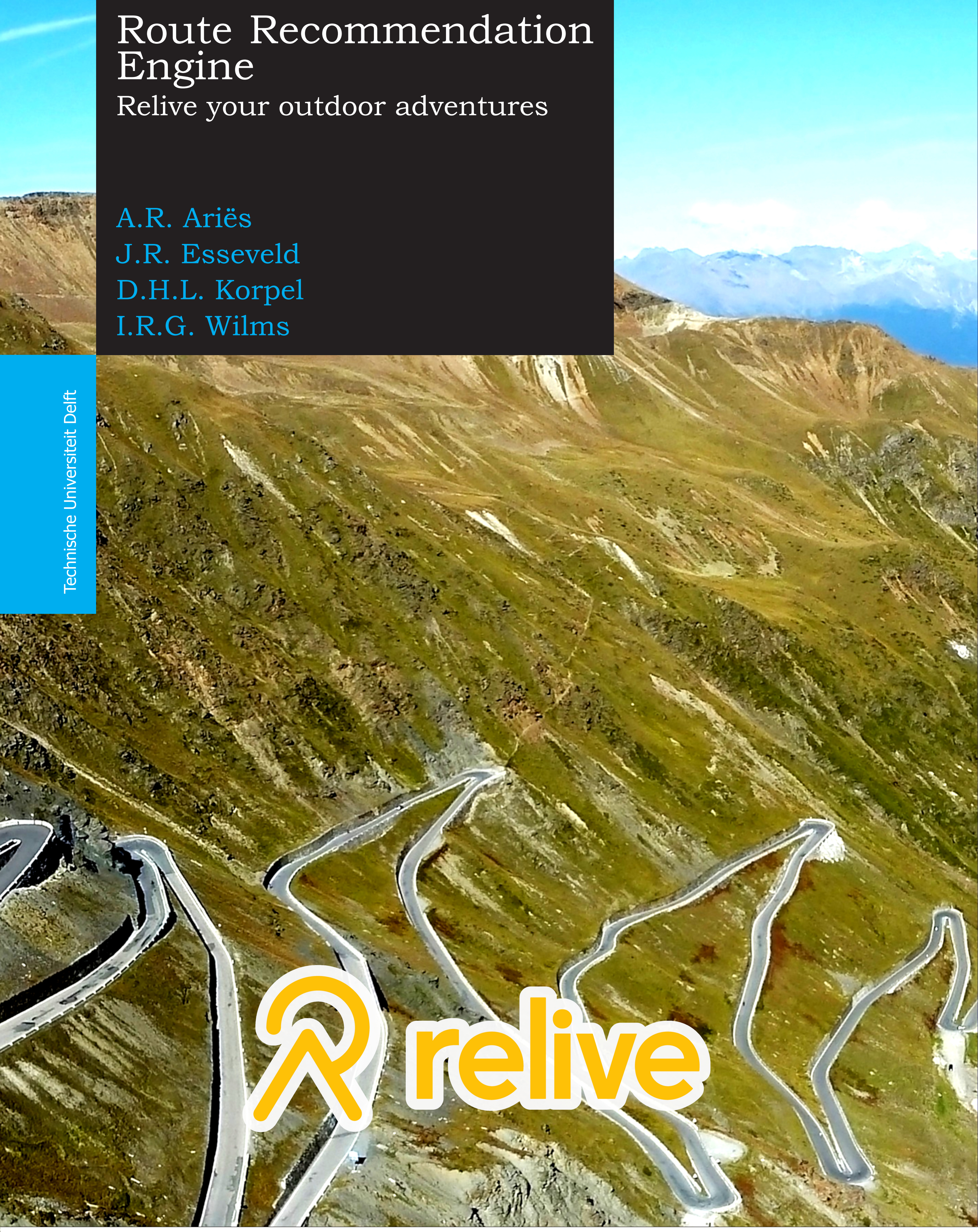
# Route Recommendation Engine
## Relive your outdoor adventures

A.R. Ariës
J.R. Esseveld
D.H.L. Korpel
I.R.G. Wilms

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

# Preface

This report describes the Route Recommendation Engine project done by Alessandro Ariës, Jeroen Esseveld, Dennis Korpel and Ivo Wilms, as the Bachelor End Project for the Computer Science Bachelor at Delft University of Technology. Over the course of 10 weeks we created a route recommendation engine and website for our client Relive. The goal of this project from the perspective of the TU Delft is for us to show what we have learned during our bachelor. The goal from the perspective of Relive was to create a Route Recommendation Engine for its users.

We would like to extend our gratitude to Lex Daniels and Yousef El Dardiry, who helped us prioritize features, offered both high level and low level advice on the code, and evaluated the quality of our route recommendations. We would also like to thank the entirety of the Relive team for testing our product and giving feedback, for helping us with the code and the design, and for making our time at Relive an awesome experience.

Finally, we would like to thank Christoph Lofi, our TU Delft coach, for helping us consider the problems and our solutions in new ways, and for giving invaluable feedback on our research report and this final report.

# Summary

For our bachelor end project, we made a route recommendation engine for the company Relive. The Relive app allows users to turn their tracked GPS routes into 3D videos, showing their run, hike or ride activity on a map from a bird's eye perspective.

The goal is to help users that are new to an area discover interesting routes nearby. Existing route planner tools exist, but they either ask users to choose the way points themselves or they let the user choose from a list of curated routes. Our client asked for a new approach: use the routes in their database from their two million users to automatically find the best tracks. Additionally, activity photos and titles that users enter can be used to give some extra information about the route.

We started with researching ways to cluster routes so we could find the most popular ones, as well as a way to filter interesting photos and titles. It turned out that using a heatmap to find routes going over popular segments was too biased towards routes going over main roads instead of interesting roads. We then used a clustering algorithm to find multiple instances of the same route and looked for routes that were used many times. This, with the addition of some basic filters, resulted in good recommendations. The quality of recommendations were determined by experts at Relive who know the best routes near certain places. Many routes that came out of the recommendation matched the ones they wanted to see, showing how the engine is successful.

Because we focused on the route recommendations, we didn't end up incorporating the titles and photos in our recommendation. Titles and photos required significant processing before being useful, since most titles where just things like ”afternoon ride” and photos were often selfies.

The end product is a route recommendation server with a website client. The most important components work, but the waiting time before receiving the recommendations can easily exceed a minute because the database is not designed with this application in mind and route clustering is an expensive operation. Further optimizations in these areas or caching routes recommendation can make the application fast enough for deployment.

# Contents

# 1 Introduction

Relive is a start-up located in Rotterdam. They built an app allowing users to re-experience their activity by generating a video of their sports activity. Over the years Relive has collected a lot of user activity data. They propose that routes could be recommended using this data. They put us to the task of researching the possibilities of this subject.

First we will discuss our research which covers the research phase of the first two weeks of our project. Here we define the research questions and goals. After that, the Product Design will be explained. This chapter discusses the requirements and the architecture of our system. Chapter four describes the Product Implementation which explains the technical details of the project. In chapter five Product Quality Evaluation will be explained, here we describe code quality, the validation of recommendations and we analyze the requirements. Followed by chapter six which elaborates on the process of the project, here we discuss our meetings, goals and different phases during the project. Chapter seven concludes this report and finally we discuss issues like ethical implications and reflect on our process. We will now first give some more information about our client and elaborate on the problem description.

## 1.1. Context

Relive is a company built around the idea of ”reliving” outdoor activities, with a focus on biking. It achieves this with an app that can keep track of a user’s location while they are biking, hiking etc. It can then make a video of their route on a 3D map, with the pictures they took and with some relevant details like maximum speed and total elevation gain. However, users currently have to decide on the route themselves.

## 1.2. Specific description

Since the initial problem description (see Appendix B) is not very specific we created a new description from the meetings with our client. The goal of this project is to make a route recommendation engine for the users. The recommendation should base its recommendation on the popularity of routes and on the preferences of the user. The first version of this route recommendation engine should recommend already existing routes. An existing route means that the route is already in the database of Relive, i.e. it is a route that has already been used by someone. However, we may implement a later version which may also use parts of already existing routes to compose new routes. The intended audience is all users of the Relive app, with special attention for users who are unfamiliar with the area, such as tourists. Finally, the recommendation engine should give context to a route. Context is information about a route other than the actual route itself, such as the title, quality of a route, photos of scenic views, road business, points of interest, etc. The goal of giving a context such as photos of scenic views and points of interest is to provide a presumed reason for the popularity of the recommended route, i.e. what distinguishes this route from other routes in the area. Because Relive is mostly a platform about experiences, we assume that users are looking for tourist routes as opposed to training routes. Differentiating between the goal of a route is something that is not part of the first version, but it may be added later.

# 2 Research

In this chapter we will explain what research has been done to develop the final product. First we will discuss existing solutions to the problem and their main features and shortcomings. After that, we will define a research question with three subquestions. Each subquestion will be answered in different sections. Finally, we conclude this chapter by answering the research question.

## 2.1. Existing solutions

Before designing our own route recommendation engine, we first considered existing solutions for making routes, because we want an innovative new solution instead of reinventing the wheel. We mention related services and the insights we gained from them.

**Strava** Strava is an app for runners and cyclists that tracks the users' route. It also has a route planner where the user has to enter which places he wants to visit. There is an option to consider popularity of routes, then it is more likely to choose paths that are often taken by other Strava users. This means it will consider staying on main roads more rather than taking shortcuts through smaller streets.

**Komoot** Komoot has a route planner where the user inputs a start point, end point and waypoints by giving an address or clicking on a map. Interesting locations can be shown on the map as points. The application then shows a route visiting all waypoints on the map. The user has to manually specify the order of waypoints though. It is similar to Google Maps in that sense, though it does give information about height difference and difficulty of the route.

**Routeshuffle.com** On this website, the user enters a place and what distance they want the route to be, and the application gives a randomly generated route near the given place and of the chosen length. It is not much more than that, it does not give any route description, extra info about heights or road quality etc. The premium version does allow you to export the route to other services (for example Komoot) that do show extra info. The routes, being randomly generated, are often uninteresting. Sometimes you're suggested to follow a long road for a while and then turn back so that it reaches the target length.

**Route.nl** This is a service with pre-made routes. The amount of meta-data per route is striking here: each route has an extensive description with photographs. Aside from showing the route, it also shows points of interests alongside it like attractions, hotels and cafes.

**MapMyRide** MapMyRide is an app to track cycling activities, though there are also other variants like MapMyRun for running. On the App, you can look for nearby routes that users have cycled previously. It doesn't seem to filter them, sometimes a route of around 100 meters is shown, or a route along train tracks. It also doesn't seem to cluster them, because almost the same route is listed four times in a row sometimes.

### 2.1.1. Conclusion
The problem with existing solutions is that they are either pre-made by users and developers, or they require manual input regarding which places you want to visit. Our solution aims to use data gathered from users of the Relive app to automatically find interesting routes of a certain length near a certain place without someone explicitly mapping it out.

## 2.2. Research questions
After we discussed the final product definition with the client we defined a main research question and sub questions we would like to answer with research. In this research we will try to find answers to the following questions:
**Main question:**

- How can we use the data from the client to find the most popular distinct routes within an area based on user preferences and provide context for those routes?

**Sub questions:**

- How do we find distinct routes?
- How do we define popularity of a route?
- How can we give context to a route?

### 2.2.1. Motivation of research questions
Since the main assignment of our client is to recommend popular tourist routes and the available data is suitable to extract the popularity of a route (see 2.3), we chose to include the notion of popularity in our main research question. Additionally the client wants to recommend routes which are distinct, since a user does not want to get recommended routes which are very similar to each other. In 2.4 we will elaborate on the distinctness of routes and answer the first sub question of our research.

The choice of researching the context (definition of context has been discussed in 1.2) of a route is based on the the clients needs for our final product. Must Have 2 in 3.1.1 states that our final recommendation engine should give the user context to a route.

## 2.3. Available Data
In order to answer the main research question of our research we must first discuss which data Relive has available for us to use.

Our client has a database of activities of their users. The start and end time of an activity are determined by the user. The user can select different activity types: ride, hike, run, skiing, snowboarding or 'other'. Ride is the most common and also the default activity type. Since the client is mostly interested in this activity type we will mostly focus on 'ride'.

Every time a user wants to make a 'Relive' (video of their route plotted on a map) it is stored as an 'activity' in the database. An activity contains the route as a list of GPS-coordinates. Additionally, a user can give a title and add photos taken along the route. Adding photos that were made earlier is possible too. An activity needs to be at least 1 kilometer long, otherwise a user can not make a "Relive" of its activity. In our final product and in the research phase of this project, the primary source of information comes from the this activity database. Especially the route, the title and photos of an activity will be of great value for us.

To give an idea of how large the dataset is, we gathered some statistics. There are at least 25,000 activities with routes passing through Amsterdam in the database provided to us by Relive. Note that this database is a subset of the database Relive has in production, so if necessary, more routes are available. In figure 2.1 the distribution of distances of routes is plotted.

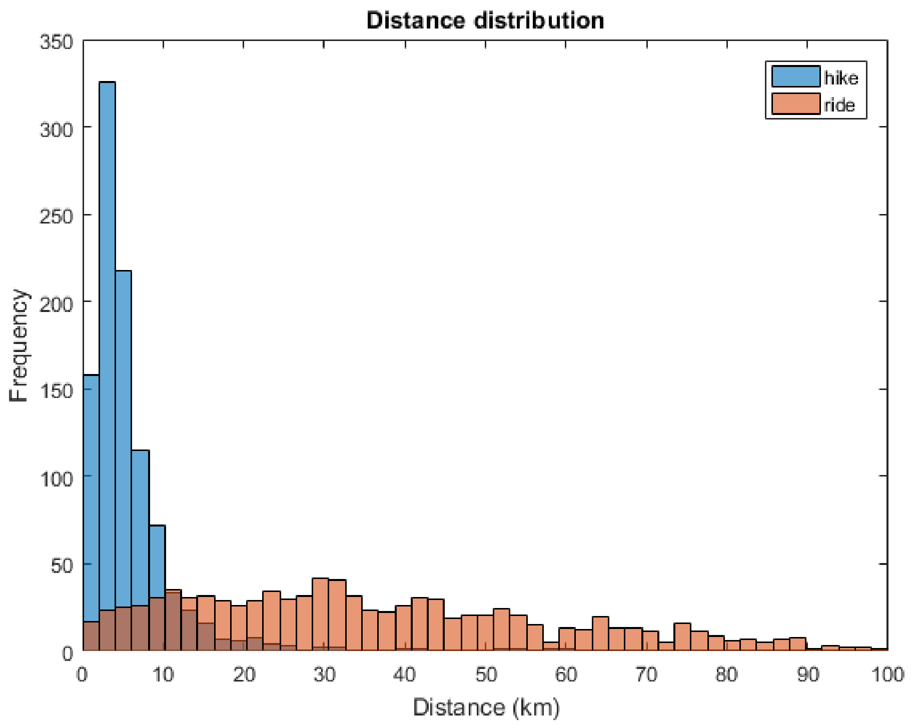


Figure 2.1: Distribution of distances of activities in the Relive database

It can be seen that hiking routes tend to be shorter than 10 km, and riding routes tend to be shorter than 100 km. A route in the database has on average 100 GPS points, though there is also a database with low resolution versions of routes with 25 points on average.

We will now introduce two terms which we are going to use extensively in the following sections. We define a route as a sequence of road segments. A GPS track is a sequence of GPS coordinates. It is not trivial to extract a route from a GPS track and the practice to do so is outside the scope of the research.

## 2.4. Distinctness of routes

Because multiple GPS tracks can map to the same route, and because we don't want to recommend routes that are similar, we came to the sub question: How do we find distinct routes? Note that distinct is used in the meaning of dissimilar, which means more than just unique: routes that are not strictly equal but still similar are *not* distinct in this context.

In this section we first give a more formal definition of distinctness. After that, we give a hypothesis on how to find distinct routes. Then, we give a method for testing the hypothesis. We consider different implementations for our hypothesis, and finally we test our hypothesis and discuss the result.

#### Notion of distinctness

A GPS track has different characteristics such as geographical location and distance. Two GPS tracks are distinct if they don't share enough common characteristics. We can do the categorization based on different properties of which the geographical locations are the most important of the GPS track.

### 2.4.1. Hypothesis

The client wants us to find distinct popular routes for different areas. We can achieve this with the help of grouping routes together if they are similar. Since the client claims that clustering of GPS tracks is an effective method, we came up with the hypothesis: *Clustering is an effective way to find distinct routes*. In subsection "Notion of distinctness" we will give a clear definition of distinctness. The idea is that similar routes are by definition not distinct, so if we group similar routes into clusters, we can take a route from each cluster and those routes should all be distinct.

### 2.4.2. Methods

We test our hypothesis by creating a prototype and seeing whether the experts agree with the results. This prototype takes GPS tracks as an input and should output clusters of GPS tracks which represent popular routes. The output should visualize the clusters in such a way that experts can judge that the algorithm produces satisfactory clusters. The experts will be our client, because they have experience with the data and they are cyclists themselves, so they know what routes are similar and which routes are distinct.

#### Distance

To determine how similar two GPS tracks are, we want to look at the shape and physical distance to each other. Besse et al. describe this as shape based distance [1]. They consider different kinds of shape based distances: Hausdorff, Fréchet and One Way Distance (OWD). None of these are perfect so Besse et al. composed their own distance function based on the existing shape based distances: Symmetrized Segment-Path Distance (SSPD). This distance is better suitable for GPS track comparison, because it takes the good parts from both OWD and Hausdorff. It is less affected by noise, takes into account whole GPS tracks and is rather fast compared to other distance functions.

The SSPD function seems to express distances between two GPS tracks very well so we use this distance function on the relive data set as well. We can now define whether two GPS tracks are distinct

#### Clustering analysis

To be able to cluster GPS tracks, we are going to use the SSPD distance function. Different clustering techniques have been considered. We will provide a brief summary of the clustering techniques and argue how applicable they are in our situation.

**K-means** K-means assumes clusters to have a spherical shape, but the GPS tracks in Relive's database turned out to have a skewed nature yielding inaccurate results.

**Expectation-maximization** Expectation-maximization clustering is more suitable in this case, because it does not assume spherical clusters. This is important because there are a lot of correlations in the GPS track data.

**Head/tail breaks** Head/tail breaks is a clustering technique suitable for a heavy-tailed distributed or skewed data set. This method emphasizes the GPS tracks of popular routes a little bit more, which is good, but unfortunately Head/tail breaks can only work with a one-dimensional input data set so it is not applicable for GPS track clustering [2].

**Hierarchical Clustering** We considered Hierarchical Clustering (HC). This method is suitable for skewed data sets, but the big challenge of this method is to determine a suitable threshold for the final clusters. Besse et al. used this method on their data set of trajectories and this yields good clustering for them.

**HDBSCAN** Finally Hierarchical Density-Based Spatial Clustering of Applications with Noise, or HDBSCAN in short, is a clustering algorithm that only returns clusters of elements with many nearby neighbors and rejects outliers. This clustering algorithm is useful for filtering out the unpopular routes.

For our research, we used HC because the clustering method works very well with the data we have. Different link types[1] for HC can be used to determine the distance between clusters: Single, Complete, Group average, Weighted, Centroid, Ward and Median[2]. Several stopping criterion for HC are available:

[1]A linkage is the way the distance of two clusters is computed, more info http://saedsayad.com/clustering_hierarchical.htm

[2]In depth information about the different links can found in this manual: https://github.com/dmuellner/fastcluster/raw/master/docs/fastcluster.pdf

inconsistent, distance, maxclust and monocrit[3]. We used HC with stopping critereon inconsistent during research and in our initial implementation, but later on we switched to HDBSCAN which gave better results due to the fact that it returns more coherent clusters since it can discard outliers.

### 2.4.3. Results

To check whether the hypothesis holds: "Clustering is an effective way to find distinct routes" we see if the prototype yields clusters which contain GPS tracks which are similar to each other. The prototype produces plots of different clusters which experts can assess by looking at them, see Figure 2.2. In this figure we can see different clusters denoted by their color. We can see that some clusters are quite good, for example the big cluster in the middle around Hoofddorp. The experts looked and saw that GPS tracks with the same approximate shapes group in the same clusters. We see that GPS tracks intersecting with other GPS tracks but taking other streets group in different clusters. Furthermore, GPS tracks which are a subset of longer GPS tracks do not group together if the subset is significantly smaller. Experts conclude that these result satisfies the detection of distinct GPS tracks, meaning that the hypothesis is correct. Therefore we conclude that clustering is an effective way of finding distinct routes.

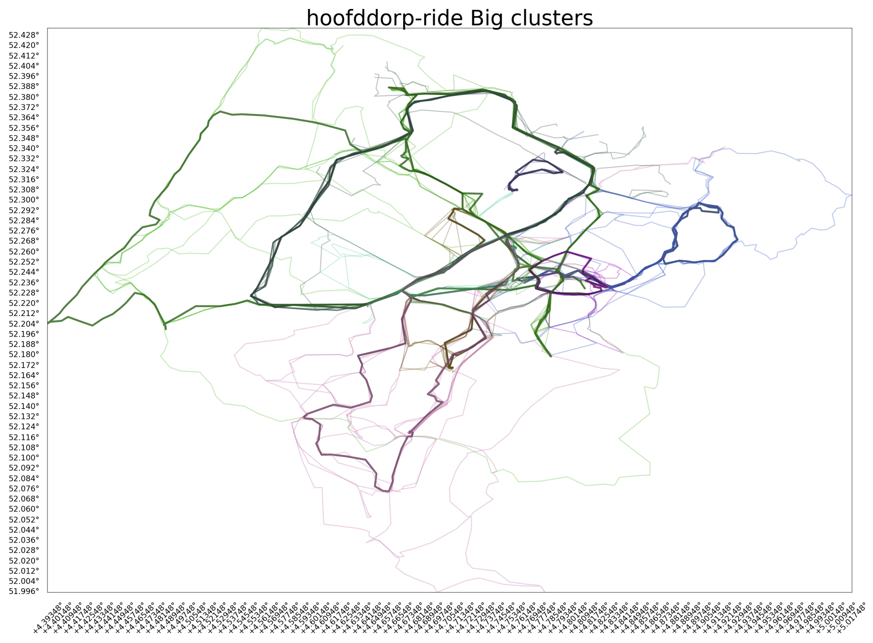


Figure 2.2: Route clusters around Hoofddorp

## 2.5. Definition of popularity

The client wants us to find the different popular routes in an area. To do this, we need to find a good definition of popularity. In this section we will give a hypothesis on the definition of the popularity of a route. After that we will give a method to test our definition, and finally we will discuss the results of the tests we performed.

### 2.5.1. Hypothesis

Before we can give the hypothesis, we need to define a few terms. A *segment* is a part of a road between two intersections. We give segments a *popularity score* by summing the number of routes that pass through that segment. Finally, we can then define the *popularity of a route* by taking the weighted average of the popularity of the segments of that route, where the weight of a segment is its length.

[3]More about the stopping criterion can be found in the documentation of scipy: https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.fcluster.html#scipy.cluster.hierarchy.fcluster

Because routes are stored in the database as GPS tracks and it is not easy to map them to road segments, we introduce an approximation so that such a mapping will not be needed. This approximation works analogous to our definition, but it is optimized to be easier to compute with the data available to us. First, we divide the world into a grid with cells of 200m $\times$ 200m. Then, we give each grid cell a popularity score by summing the number of routes that go through each cell. Finally, we can calculate the popularity of a route by averaging the popularity of the cells that the route passes through.

Because we want to use the approximation for finding popular routes, and we only need a good definition (and not a perfect definition), we have the following hypothesis: *The approximation is a good definition of popularity*.

### 2.5.2. Methods

We test whether the approximation is a good definition by giving it a trial: the client gives us five areas and the routes that are popular in those areas. Our approximation should also mark these routes as popular. If the approximation correctly identifies the popular routes as given by the client, we know that there is a high chance that the approximation is a good definition of popularity. If it did not correctly identify the popular routes, it is definitely not a good definition, and we will need to make a new definition or approximation, or we will need to extend our current approximation so that it does pass the test.

### 2.5.3. Results

The popularity of all routes combined can be expressed in a heat map. The heat map highlights popular routes. We tested this in different areas and experts saw that interesting roads are more highlighted on the heat map than other roads. This means that there is a good correlation between popular routes and GPS tracks which intersect a lot with popular roads.

## 2.6. Context of a route

After receiving a route recommendation, the user might be interested in more information than just an outline on a map: What kind of route is it? What can I expect to see? The client suggested that the titles users give to their activities as well as the photos users attach to their activities can be used for giving context to recommended routes. In this section we look into ways information about the context of the route can be extracted from these sources. We first cover extracting significant terms from activity titles and then we look into what kind of photos users add to activities.

### 2.6.1. Activity titles

The first data source to be considered is the activity titles that users enter. Extracting useful information from titles can be challenging since it involves natural language processing. To get more feeling about the data, a dataset containing around 4.000 entries of activities within 2km of Delft was exported from the MySQL database and imported into an Elasticsearch client. Elasticsearch provides a database system that has text searching and analysis functionality.

It was observed that many titles are very generic (”morning run”, ”afternoon ride”) and looking at common terms is not very effective. We want to find a way to extract terms that are actually specific to a route.

#### Hypothesis

Our hypothesis is that *significant terms analysis can be used to find context for a route.* As we saw, just looking at popularity doesn't yield useful results. But in a significant terms analysis, you look at a relative increase in popularity between a foreground set and background set. For example, the term ”Delft” may not be common in general or even in Delft, but it's more common if you only consider activities near Delft than if you consider the entirety of South-Holland. This way common titles won't appear, but instead only interesting terms for a certain place.

#### Methods

A data set containing around 55.000 entries of activities within 20km of Rotterdam will be considered, covering most of South-Holland. A significant terms aggregation is performed with as foreground set a smaller circle on the map around certain places: Rotterdam, The Hague, Leiden, Delft, Pijnacker, Zoetermeer, Bergschenhoek and Westland. For more information about how significant terms analysis works, refer to the elasticsearch documentation.[4] The 10 buckets with the highest score as determined by the algorithm from Elasticsearch were examined. This test aims to see whether a circle around a city is a good enough foreground set for significant terms or whether a more specific foreground set should be considered. We expect to see some terms related to the places, but many false positives. When it still results in merely common terms like ’morning ride’ we can reject our hypothesis.

As a second test, a cluster of Routes near the Rotte (see section 2.4 for clustering methods) will be used as a foreground set to see if the ’Rotte’ will come up as a significant term. The background set is still a circle of 20km around Rotterdam. Similar experiments will be done on other locations, but in the interest of time this report only covers this example. By doing this we test if a specific foreground set of routes improves the result. We expect the term ’Rotte’ to be a significant term.

### Results

First we will discuss the results and observations from the significant terms in South-Holland. Words like ’evening’, ’run’, ’ride’ and ’afternoon’ are significant in Rotterdam and The Hague. This may be because of the higher amount of international residents in cities, making the English variants of the common Dutch titles such as ’middagrit’ more frequent. More relevant significant terms include places, clubs and events. Examples of these are:

- Noord AA (watercourse in Zoetermeer)
- jzc (Joop Zoetemelk Classic, cycling event)
- WTOS (a student cycling club in Delft)
- tcp (toerclub Pijnacker)
- wwv (Westland Wil Vooruit)

Even in this larger set, significant terms can easily be dominated by one person in a certain area using a similar title every time. One particularly bad case is when someone has titles like ”Training X of 63” for every X from 1 to 63, thus using the term ’63’ 63 times. Another example is ’06’ becoming significant because of one person who often starts around 6:00 AM and includes his start time in the title.

It becomes clear that there are a lot of false positives with a foreground set spanning an entire city and the background set spanning a province. The significant terms from the cluster along the ’Rotte’ showed that ’Rotte’ was the number 1 significant term, followed by terms related to the ’PETZL Night Trail’ which is a running event. This suggests that with more targeted foreground sets, we can successfully extract useful information from titles.

To recognize terms from single users and one-time events, we can filter by user and date variance. More research for this will be required. To answer the hypothesis of the question whether titles can give context, we conclude that a carefully tweaked algorithm can find relevant terms that add context. The things to watch for are:

- Users don’t have incentive to write attractive, descriptive titles. When titles aren’t generic they’re often very specific about something unrelated or personal. They are often left to the default or something very personal. A lot of processing is needed to extract anything useful.

- The background set and foreground set have to be carefully chosen. If the background set is the entire world and the foreground set is a city in the Netherlands, significant terms are likely dominated by terms significant to the Netherlands as a country, not terms specific to the city.

- A term can be made significant by one person entering the same specific title every time. We don’t want significant terms coming from one person doing many activities with similar titles where they live, we want them to be significant because they are related to the place. We could try to filter titles that are significant because of only one user. Also, a single event or organization can cause significant terms to appear. These are actually relevant to the route, but if we are strictly looking for features alongside the route we could try to filter these as well.

[4] https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-significantterms-aggregation.html

We conclude that relevant information can be extracted from the titles as they are, but it is not valuable enough to pursue further. The terms that can be extracted can also be retrieved from open map info. If users are first encouraged to enter more interesting titles or provide other textual metadata, then a significant terms analysis could yield truly interesting results.

### 2.6.2. Photos

#### Hypothesis

Some users want a route with nice landscapes or cool landmarks. If a route has many vantage points, we expect a relatively high amount of photos taken along the route. But just because a scenic route implies many photos doesn't mean a route with many photos is a scenic route. If most photos are random selfies or photographs of meals for example we would end up with a lot of false positives when searching for scenic routes by the amount of pictures. We expect however that scenic photos out-number the useless (not related to the route) ones.

Our hypothesis is: *The presence of photos alongside a route implies the route has nice scenery*. From the following analysis, we hope to get an impression of what kind of photos users upload so we can check our hypothesis.

#### Method

To test our hypothesis, we manually analyzed a sample of all photos present in Relive's database. We first browsed some photos to get an idea what we could expect, and then made a list of categories to which a photo could or could not belong. Note that these attributes are not mutually exclusive: a photo can theoretically have all of them or none of them. The categories are:

- Scenic photos. Photo's that show landscapes, mountains, beaches or other nature. It is sometimes hard to judge whether a view is 'good', but having a wide view and nature in the shot made photos qualify.
- Man-made structures. These include bridges, buildings or other architecture. These are also signs of interesting routes.
- Shows user. Many photos include the user doing the ride as well as people accompanying them. Whenever a face was identifiable, it was assumed to be of someone doing the ride. These photos are more personal and not related to the scenery.
- Photos taken on a road. When the road was visible on the photo and a normal car could reasonably drive on it, it was considered "on road".
- Photos taken off-road. This can be inside a forest, on a mountain trail, in a meadow etc. This, in combination with the on-road property, aims to see if routes with photos tend to be 'adventurous' or more laid back.
- In or near a building. Our routes are for outdoor enthusiasts, so the presence of photos showing people at a restaurant or in their home doesn't give context to the route itself.
- Shows bike. It turns out that many photos feature the user's bike clearly in shot without any people. These people are probably riding by themselves but still want something personal in their shot.
- Unrelated. A rest category for photos that are not related to the route: screen shots or photos of a screen, brochures, a map, pets, insects, photos too overexposed or dark to make out anything etc.

We took a random sample of 200 photos that have location data associated to them and manually determined whether each property applied to them. The random sampling was simply done by taking the first photo of a random activity entry in the database and filtering those who do not have photos with location data. Activities without photos were therefore excluded but routes with a high amount of photos per activity did not have a higher chance to be represented than routes with a low amount. While a purely uniformly distributed photo sample may have been better, the extra complications from getting such a sample were not deemed worth it for this experiment.

#### Results

| Property | Percentage |
|---|---|
| Scenic view | 38% |
| Man-made structures | 8% |
| Shows user | 45% |
| On road | 30% |
| Off road | 20% |
| In or near building | 10% |
| Shows bike | 12% |
| Unrelated | 12% |

Table 2.1: Statistics of photos

It is clear that many photos are personal: almost half of the photos show the riders. When we show pictures along a recommended route in our tool without a filter, they are likely to show off a person instead of the route.

There also were a lot of scenic photos. There were not so many photos of man-made structures though. This can be because those tend to be in cities, and cities are full of traffic which makes riding there unpleasant.

The ”on road” and ”off road” properties turned out to be less successful, because many photos (especially the scenic ones) don't show the road. Therefore, many photos couldn't be classified. Still, it did became clear that many photos are taken off-road, more than expected.

The ”shows bike” category turned out to be pretty useless in hindsight. Twelve percent is unrelated. It was expected that there would be unrelated photos, but luckily these don't dominate at all.

We therefore conclude that photo's are not dominated by useless 'meal photos' and they can be used as a measure of how scenic a route is. To show photos along a route, we should first find a way to filter them: we don't want too many personal photos or totally unrelated photos (for example photos of a screen or photos of an injury) to appear.

A follow-up hypothesis is that photo clusters (many photos taken around the same location) imply scenic photos while personal photos are uniformly spread along routes. This way, we can use a clustering algorithm to separate photos taken from nice 'photo spots' from selfies taken on the road. Testing this requires a lot of implementation work however, so we left this out the scope of this preliminary research.

## 2.7. Conclusion

Our client Relive requested that we make a route recommendation engine. It should be able to recommend already existing routes to users based on the popularity of the route and the user's preferences. The main focus is on cyclists who are unfamiliar with an area and who would like to ride a tourist route. The recommendation engine should also give context to routes, which means it should give extra information about the routes. This extra information should give users an idea of what makes this specific route unique from the other recommended routes.

To achieve this, we came up with the following research question: *”How can we use the data from the client to find the most popular distinct routes within an area based on user preferences and provide context for those routes?”* We divided this question into multiple subquestions:

- How do we find distinct routes?
- How do we define popularity of a route?
- How can we give context to a route?

**How do we find distinct routes?** We need to find distinct routes, because we don't want to recommend routes that are too similar. The use of clustering algorithms is the answer to the first subquestion. By first clustering all the similar routes together, we can then take a representative route from each cluster and these routes will then be distinct. We use the Symmetrized Segment-Path Distance (SSPD) function to determine the distance between two GPS tracks, and we use hierarchical clustering to cluster them.

**How do we define popularity of a route?** We made a definition using segment popularity, but that definition was hard to implement because of a mismatch with our data. Therefore we used an approximation of the definition. The approximation divides the world in a grid of cells, and each cell has a popularity based on the number of routes that pass through it. Then, each route gets a popularity by averaging the popularity of the grid cells it passes through. This method quite accurately describes the popularity of a route.

**How can we give context to a route?** Giving context to routes was divided into two hypotheses, one for extracting context from titles and one for extracting context from photos.

To extract information from titles, we need to consider the slight complication that many titles are completely useless, because users will often leave titles on the default. This leads us to the hypothesis *significant terms analysis can be used to find context for a route*. Significant terms works with a foreground and background set, and it finds terms that are relatively more popular in the foreground than in the background set.

We did multiple experiments, one with a cluster of routes near the Rotte as foreground set, and 8 experiments with each a different city or town as foreground set. The background set for all of them was most routes in South Holland. While there were false positives, we felt that there were also enough true positives, so we conclude that the hypothesis is true: we can use significant terms to extract context from titles. However, more research is required to actually make an implementation that can reliably give relevant context for routes.

We would like to show users photos of the recommended routes, so that they know what the routes looked like. We assume that scenic routes have lot of photos, but that doesn't mean that routes with many photos are scenic routes. This leads us to the hypothesis *the presence of photos alongside a route implies the route has nice scenery*.

To check this, we manually looked at 200 photos randomly selected photos and categorized them based on what they contained, such as scenic view, shows user (selfies) and unrelated. 38% of the photos shows nice views, so we might be able to use photos, either directly or by first filtering them. Such filtering would require extra research into how scenic photos are distributed.

**Main research question** To answer the research question we connect the dots. First, we select a group of routes based on the preferences of the users, which means that they go near the start and end location of the user, and that they are of the same activity type (e.g. biking). These routes are then clustered into clusters of similar routes. For each of these clusters, a representative GPS track is chosen based on its popularity and its SSPD to other GPS track in the cluster. Finally, from these GPS track we select the 10 best recommendations based on their popularity and their distance from each other.

# 3 Product Design

In this chapter we discuss the different functional requirements prioritized using the MoSCoW method. We also explain the architecture design by showing the different components of the product and how they interact with each other. Finally, we give the APIs for the different components.

## 3.1. Requirements

To describe the requirements and possible features of the final product, we will divide the requirements into functional and non-functional requirements. Functional requirements are concerned with the types of input and output the product should accept and produce, while non-functional requirements are everything else.

### 3.1.1. Functional requirements

To describe the functional requirements, we will use the MoSCoW method. The MoSCoW method divides features and requirements into four categories: Must have, Should have, Could have and Won't have. The must haves are hard requirements: the product is not complete without these features. The must haves define the minimum viable product. Should haves are features that are not essential, but that would greatly improve the product. Could haves are features that would be nice to have but that are certainly not essential. Finally, won't haves are features that definitely will not be included in this iteration of the product.

#### Must Haves

1. The product must be able to recommend suitable already existing (see 1.2) popular tourist routes based on the input.
2. Input:
   The user must be able to select a start and end location. The recommended routes should start and end within a given radius of the start and end locations respectively.
3. Filter: distance.
   Allow the user to give a preferred distance. Recommended routes should be approximately that distance.
4. Visualization of the route on a map.
   This does not have to be integrated with the app, a simple web interface is enough.
5. The user must not have to wait more than 10 seconds for loading the routes.

#### Should Haves

6. The product should give a context (see 2.6) to recommended routes
7. Show pictures of the routes
   Users can share pictures they take on their routes with Relive, so we could use those pictures to give an idea of the route.

8. Differentiate between tours and paths in the user interface.
   A path is a route from a start point to an end point, while a tour ends on the same place it started.

9. Show relevant information about the route, such as time, distance, and climb. Allow the user to pick one of the recommendations.

### Could Haves

10. Saving routes
    Allow the user to save the route so that they can find it later. Could also be used to allow users to share their routes.

11. Filter: time
    The user can filter routes based on the time they take

12. Filter: climb
    The user can filter routes based on the climb

13. Filter: region
    Allow the user to choose a region that the route should pass through / stay inside.

14. Filter: tourist or training route
    The must have is only recommending tourist routes, but in the future we could allow the user to make the distinction between tourist and training routes.

15. Way points
    Allow the user to give a set of way points that the route should include. Advanced options are

    - Way point types
      Not a specific place, but a type of place, such as a restaurant or a public toilet.
    - Middle of the route
      Make sure that the way points are not near the end or the beginning of the route.
    - Expected way point time
      Allow the user to specify a preferred expected arrival time for the way point

16. Search for location
    Allow user to type in a location and give them suggestions for a location where we can recommend routes.

17. Compose new routes
    The must have is only recommending already existing routes, but this feature could be extended to also compose new routes based on (parts of) the existing routes. Advanced features are

    - Interactive route composition
      Allow the user to interactively change recommended routes.

18. Filter: intensity
    Allow the user to filter routes based on the intensity of the route. This could for example be measured by the average speed, heart rate, or the climb.

19. Personalized recommendations
    Base the recommendations on previous routes of the user. Examples of things to consider would be recommending routes through locations that the user hasn't been to previously, or using their average speed to better estimate the time a route will take them.

20. Ratings
    Allow users to give a rating to recommended routes. These ratings could then be used to improve the recommendations.

21. Multiple recommendations
    Show multiple recommendations for the user to choose from. Also clearly state the differences between these recommendations (why did the engine choose to show both instead of choosing one of these routes?)

22. Current location as start point
    Allow the use of the current location as a starting point (i.e. make a shortcut to enter the current location as start point).

23. Saved locations
    Allow the user to save locations and quickly use them as start, end or way point.

24. Heat map
    Show a heat map of the routes other people took

25. Elevation
    Show the elevation in the visualization.

26. Feasible routes
    Make sure that the route is actually feasible for the user. This means that it shouldn't use private roads or go the wrong way on one way streets.

27. Export routes
    Allow the user to open the route in other apps, such as Google maps. These other apps can then provide features the Relive app won't have, such as providing driving instructions.

#### Won't Haves

28. App integration
    We will not integrate our recommendation engine and visualization into the Relive app. This has been excluded from the project on request of Relive.

29. Offline capabilities
    The recommendation engine will not be available offline, and routes cannot be viewed offline.

30. Navigation system
    The product will not give instructions on how to drive. However, it might allow exporting routes to an app that can give such instructions (such as Google maps).

### 3.1.2. Non functional requirements

Non functional requirements are not concerned with possible input or output, but with other properties of the product. They are often not easily testable or not objectively testable at all.

- The User Interface must be available in a web browser
- The User Interface should be easily usable on a mobile web browser
- The interface should be easily understandable
- The product should be maintainable and extensible

## 3.2. Architecture design

After defining the requirements of the product, we designed an architecture of the system. This section starts of with a general overview of the five modules used. It then describes the API between the clients and the server. Finally, it describes the database module in more details.

### 3.2.1. Existing architecture

Even though Relive already has an app live, the only existing architecture that we could use was the databases. The databases contain two tables that are of interest to us: the Route Table in the Routes database, and the Activity Table in the database called Skyhawk. Routes and Activities map one-to-one: each route has an activity, and each activity has a route. They are matched via an `activity_id` column in the Route Table. Activities and Routes are not merged into a single table for reasons that are beyond the scope of this report; This is just something we had to work around.

### 3.2.2. General overview

Before we describe our architecture in more detail, we start with a general overview of the system architecture and the files.

#### System architecture

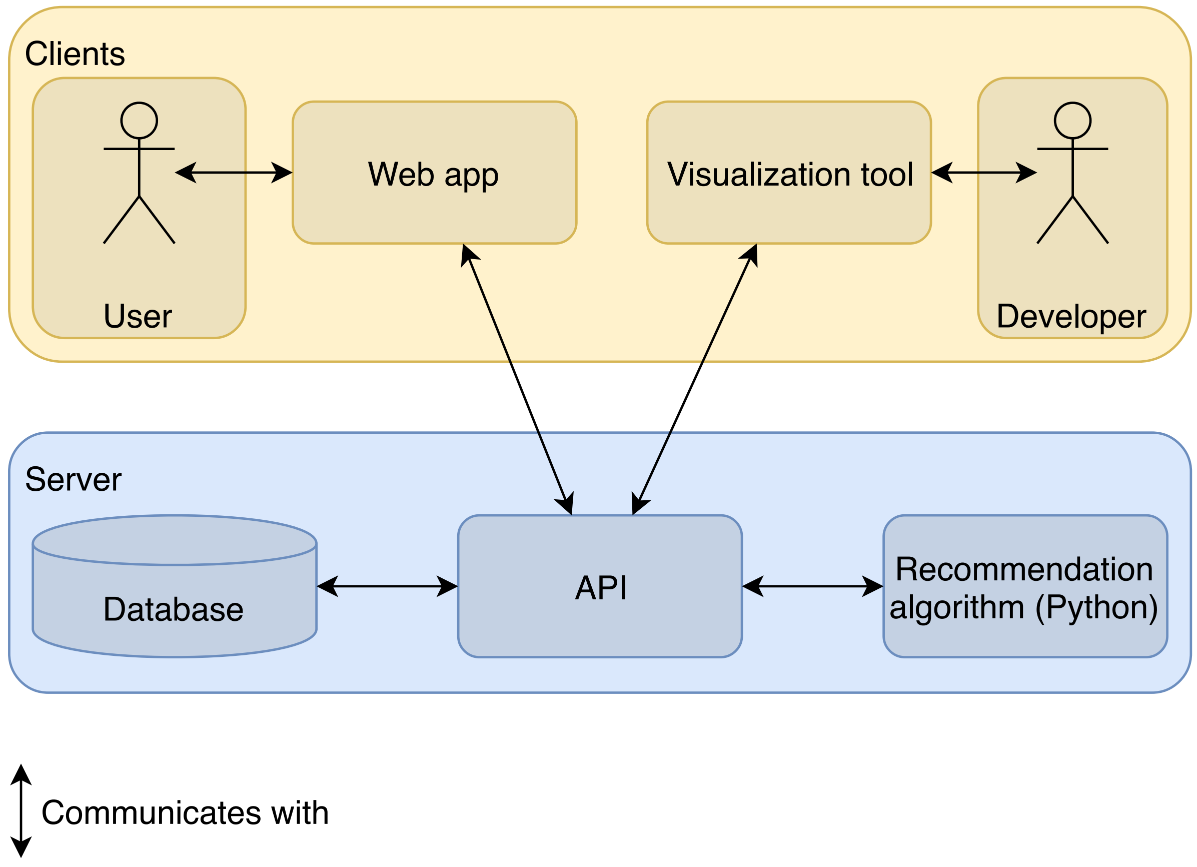


Figure 3.1: General overview of the system architecture

To reduce the complexity of our code, we used a modular approach. Each module has a single responsibility, and they get connected by the API. There are 5 modules, divided over clients and servers:
Clients:

- The Web App is the software the user uses to get recommendations (see section 4.4.2).
- The Visualization Tool is a tool to visualize the results of requests, which helps in development (see section 4.4.1).

Server:

- The API is responsible for handling requests from the clients by calling methods on the server modules. It has multiple inputs and outputs, in the form of HTTP-requests and responses to certain URLs:
  - `/routeFromLocationRequest`: get a set of routes that go through a circle.
  - `/clusterFromLocationRequest`: get the set of clusters created when recommending routes.
  - `/recommend`: get a few routes recommended from a location, a radius and the preferences of the user. Currently the only supported preferences are minimum and maximum length of the route.

- The Database module is responsible for getting data from the databases and merging it together into a single data object. Its input is a circle in the form of a location and a radius, and its output is a set of Routes that go through that `Circle`.
- The recommendation algorithm is responsible for recommending a few routes. Its input is a set of routes, and its output is the routes from that set that are recommended. It can optionally also output the clusters it made for debugging purposes; this feature is used by the visualization tool.

#### Module interaction

Figure 3.2 gives a more detailed overview of the way files within the modules interact with each other. More detailed descriptions of the interaction within and between modules are given in the following sections.

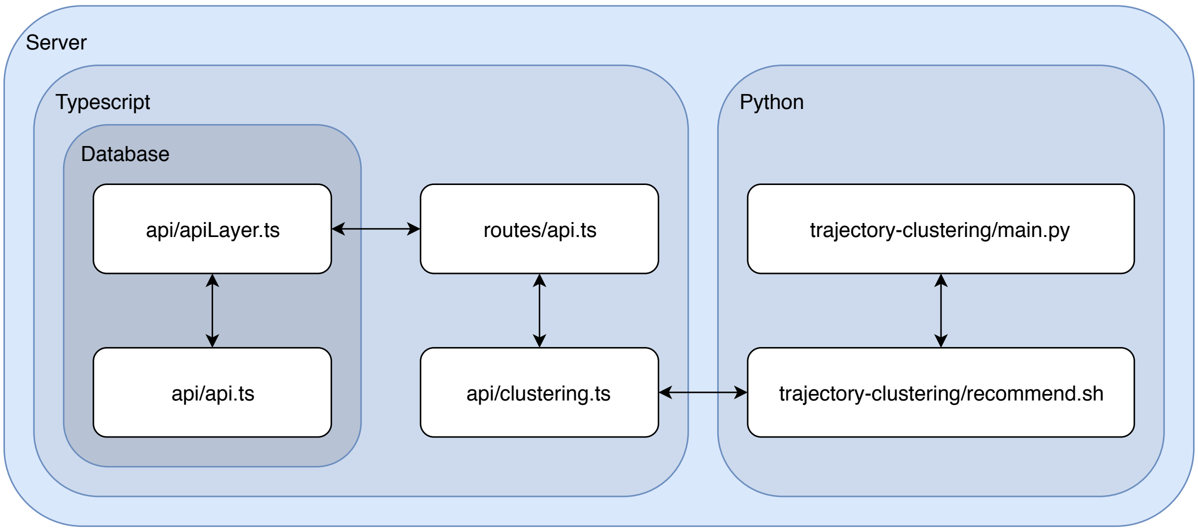


Figure 3.2: Detailed overview of the file interactions within the server

### 3.2.3. Server API

The server API is accessible at `[host]/api/1/[request]`, with `[host]` the host address (like `localhost` or `www.relive.cc`), and `[request]` one of the URLs mentioned in the general overview. The requests to these URLs need to be HTTP-POST requests, with a JavaScript Object Notation (JSON) body of the following form:

```
{
    request: {
        location: {
            center: {lat: <number>, lon: <number>},
            radius: <number>
        },
        activityType: <"hike"|"ride"|"run"|"snowboard"|"ski"|"other">,
        preferences: {
            distance: {min:<number>, max:<number>}
        }
    }
}
```

The `activityType` can optionally have the value `"all"` for the `/routeFromLocationRequest` and the `/clusterFromLocationRequest`.

### 3.2.4. Database

When a call to the server API is made, the first thing that needs to be done is retrieving data from the database. Since the data is divided over two databases, getting data can't just be done with an Object

Relational Mapping (ORM) framework; Some additional processing is needed to get `Route` objects from the data in the databases. Thus, loading data from the database has to be done in two stages. Each stage is done in a separate file. First, the required data is retrieved from the databases, which happens in `api.ts`. The data is then combined into the required objects in `apiLayer.ts`. No one should ever need just the raw data, thus no one should ever need to call functions in `api.ts` directly. Therefore the interface of the module is in `apiLayer.ts`, which then gets the raw data from `api.ts`.

### 3.2.5. Recommendation Module

After the data has been retrieved from the database, it can be processed by the recommendation module to make recommendations. The recommendation algorithm itself is written in Python, which has some consequences for the way this module is integrated with the TypeScript code.

#### Input and output

First, we can't just call the Python functions from within TypeScript. Instead, the Python code is started as a child process from within the TypeScript code. The input and output for the recommendation algorithm is done via `stdin` and `stdout`: all input is converted to a JSON string which is piped into the Python process. We chose to use JSON `stdin` as input mechanism over several other options, namely command line arguments, writing and reading from a file, and sockets. Command line arguments would be easiest, but they have limitations on the amount of input they can take and we would easily pass those limitations, so they didn't meet our requirements. We expected writing to and reading from a file to be inefficient and prone to garbage when the process exited with errors, so that was disregarded as well. Finally, we though of sockets after we had already implemented the `stdin` approach. While sockets may have been faster, we deemed it not worth the effort of revamping the input/output system as it would most likely still require doing input/output as text.

The Python process runs the recommendation algorithm and prints for some routes the id, the score, popularity and whether it is recommended. Not all routes that were in the input are printed in the output; some of them are discarded for being too noisy or otherwise not suitable.

#### Type definitions

Second, because Python and TypeScript are different languages, it is not possible to type check with only one type definition file. This means that the integration of Python and TypeScript requires careful communication between the developers of the TypeScript code and the Python code. The TypeScript code has the input types documented as type definitions within the module, and the expected output is written as documentation within the module. The Python code does not contain the expected input and output format within the code itself; Instead, it is documented in the `README` of the repository.

#### File system

Finally, we need a way to locate the Python code, so that we can call it from the TypeScript code. To call the Python code, it needs to have a defined place within the file system. But we also don't want to pin down the main entry point of the Python code within the TypeScript repository. To achieve this, we add a level of abstraction: there is a file named `recommend.sh` in the root of the Python repository, which is responsible for calling the actual main file of the repository. This way, the main file can be moved and renamed freely by updating `recommend.sh` instead of the TypeScript code, which reduces coupling.

# 4 Product Implementation

In this chapter we will discuss some details regarding the implementation of our product. These details provide the information required for a developer to understand and possibly take over the project. We will elaborate on technology considerations, the back-end, the design of the web application and the internals of the recommendation algorithm.

## 4.1. Database

The first thing the server does upon receiving a recommendation request is retrieving routes from the database that will be fed into the recommendation algorithm. This responsibility is handled by the database module. See section 3.2.2 for its place in the system architecture and 3.2.4 for its API.

The interface of the module is defined in `apiLayer.ts`. The module takes a `Circle` and it returns the routes that go through that circle.

The first stage is retrieving the required data. This is done by passing the request on to the appropriate methods in `api.ts`. The sequence diagram for these calls is shown in figure 4.1. First, a request for all ids within the `Circle` is made to the Routes database. From these, the 5000 highest are selected. We take the 5000 highest as we assume that these will be the 5000 most recently added routes. This is not done by a simple `SORT BY id LIMIT 5000` because this takes longer than performing the sorting manually. A request for the `RouteInstances` of these ids is then made to the Routes database. Each `RouteInstance` only contains an id, the path and an id for the activity. Finally, it gets all the `ActivityInstances` from the Skyhawk database using another function in `api.ts`.

In the second stage, the sets of `RouteInstances` and `ActivityInstances` are merged into a set of `Routes` with a method defined in `apiLayer.ts`. This set of `Routes` is then returned as output of the database module.

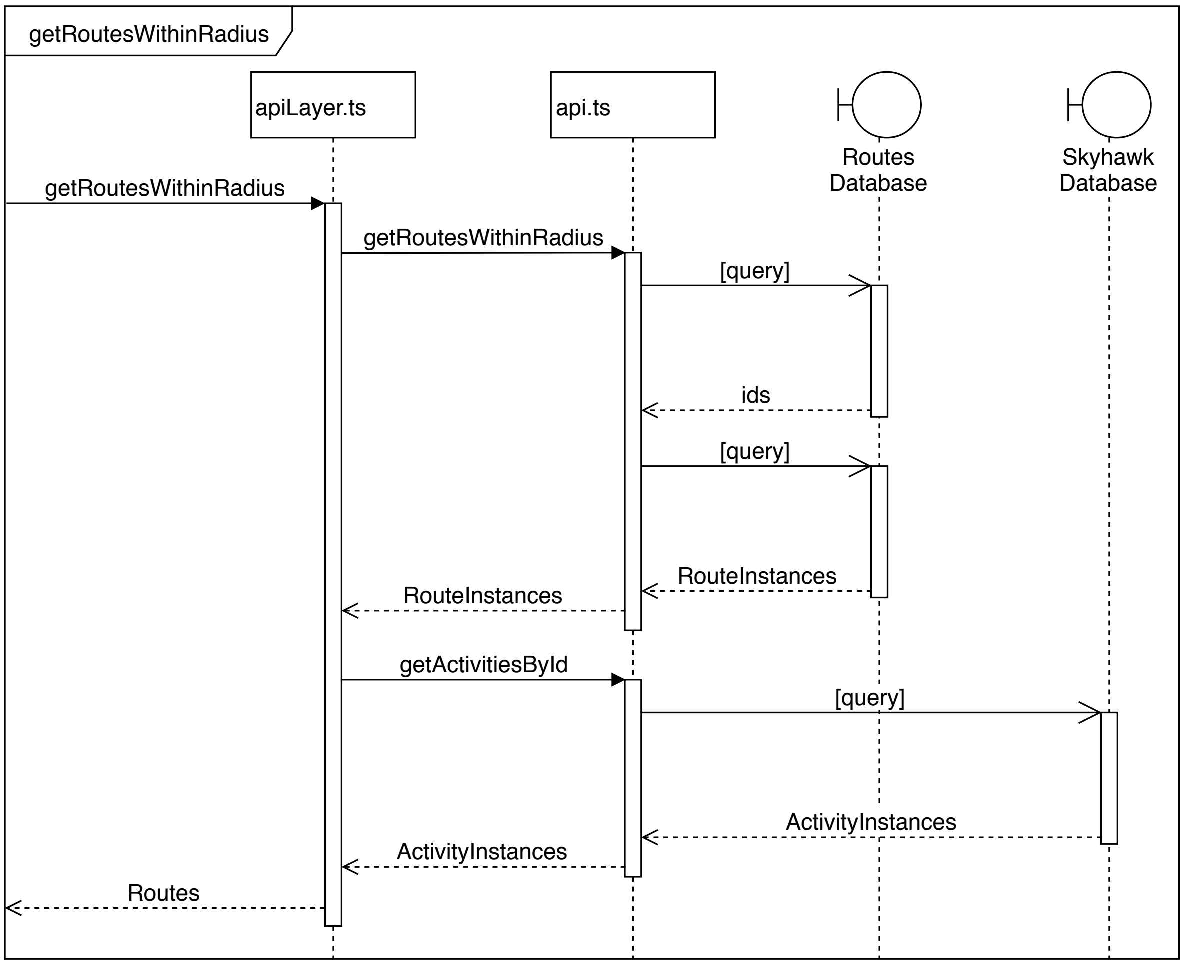


Figure 4.1: Database sequence diagram

## 4.2. Technology considerations

In this section we describe what technologies we have chosen to use to build our product. We will tell about which programming languages we used, which version control and continuous integration.

### 4.2.1. Existing infrastructure

Relive has a big part of their code written in Typescript, which is a strict syntactical superset of JavaScript, making it more scalable since type-errors can be caught at compile-time rather than run-time. This allows them to use the same language for the back-end and the front-end. Relive stores their data in a MySQL[1] database. They have written an in-house library skyhawk-dal for communicating between the database and TypeScript, which we will use too. Furthermore we discussed the structure of our project with our coordinators at Relive and they wanted us to create two different repositories for the client and the server, because they think it is more structured to have two separate repositories.

### 4.2.2. Language and frameworks

The server and client are both written in Typescript. While we were free to choose our own language (with proper argumentation), Relive preferred Typescript as most of their code is already written in Typescript.

We use Node[2] as a server back-end and the React framework[3] is used for the client. At Relive they have a lot of experience with these frameworks, which makes it easy for us to get started and get

[1]https://www.mysql.com/
[2]https://nodejs.org/en/
[3]https://reactjs.org/

help during the project. React allows insertion of HTML elements into Typescript code which makes programming the client side of our application easier.

Yarn is used for package management. We use Yarn as package manager, since Yarn has better caching capabilities than Node Package Manager (npm). Yarn also uses a lock file which ensures that all contributors use the same versions of the packages.

For the implementation of the recommendation algorithm we used Python. Python libraries such as: numpy, scipy, fastcluster, matplotlib are used for data analysis. The program does not outperform compiled languages but the development is quite fast. Development is done according to the PEP8 style guide of Python. Python allows development of a contained environment and quick visualization capabilities, useful for tweaking the algorithm.

### 4.2.3. Testing

We use the Mocha [4] and Chai [5] frameworks for automated testing. Mocha is used, because it is a testing framework for Node.js and Relive already has a good experience with it. Chai adds fluent asserts to Mocha, which makes tests more readable and portable. To write tests for the python code, we used the built in unittest package.

### 4.2.4. Code tools

Relive suggested us to mainly use two code tools: TSLint and Prettier. TSlint is a linter tool for Typescript that checks for correct formatting of the code, ensuring a consistent and explicit code style.

Prettier automatically formats the code so it uses a consistent indentation and bracing style and ensures lines of code are not too long.

### 4.2.5. Github and Travis

For version control we used Git and Github. We had five repositories separating our concerns: the Server, Visualization Tool, Web Application, Route Cluster Engine and a Util repository were kept separately. The Visualization Tool and Web Application (see 4.4) are added as dependencies for the server. This is consistent with how other Relive services were managed and it makes separate development of these components easier.

Our contributions were committed to separate branches which would later be merged into the master after code review and continuous integration approved it. It was decided that at least one review was necessary, because we were working on separate parts and having to involve each one of us with all code bases would slow development down. When a pull request was very large or important, it was reviewed by multiple people however. Additionally, programmers at Relive would sometimes also take a look and share their feedback.

The server, Visualization Tool and Web App all used Travis for continuous integration. For each pull request it would check on a central build server whether the code compiles, is properly formatted according to TSlint, and passes the test suite. This way the reviewer knows that the code has proper syntax and style so the review can focus on the semantics instead.

### 4.2.6. Other

For significant terms analysis, we use Elasticsearch[6] and Kibana. Significant terms analysis and why we use it will be explained in section 2.6. Elasticsearch is a search engine that is specialized for text analysis and search engines. Because MySQL's text query functions are limited, we transfer part of the records from the MySQL database to an elasticsearch server and analyze them there. Kibana is a visualization tool for Elasticsearch. It enables visual exploration and real-time analysis of the data in Elasticsearch, which makes it easy for us to research the data needed for our terms analysis.

[4]https://mochajs.org/
[5]http://www.chaijs.com/
[6]https://www.elastic.co/

## 4.3. Recommendation implementation

In this section we will elaborate on the implementation of the recommendation algorithm. The complete algorithm consists of different major components. First off we'll give an overview of the algorithm with its components and after that we will go over each part in more detail.

### 4.3.1. Algorithm overview

The route recommendation program combines different algorithms to come up with a recommendation. The recommendation is a list of GPS tracks which we can present to the user as recommendation for a certain area. The program takes as input a JSON document containing GPS tracks from a certain area along with metadata. It also contains recommendation preferences, which are currently only the minimum and maximum distance of the route. In our case, the server provides this set by passing the preferences it received from the client and querying for routes near the location that the user specified.

First all noisy routes are filtered from the data set. Routes are considered as noise if they do not add any useful information to the route recommendation algorithm. After that a heat map is generated from the data set. The heat map is used to filter out the unpopular routes. After that we make area clusters, these are clusters which divide the data set in different areas. These areas are used to be able to recommend one route per area cluster, so we get a variety of recommended GPS tracks. Inside every area cluster we make route clusters. Route clusters are clusters with GPS tracks which are very coherent, meaning that GPS tracks in one cluster really look like one another. From these route clusters we select the best cluster based on the score of a cluster. The score is determined by combining different factors such as the cluster cardinality and average Mean Squared Error (MSE) of every GPS track from the cluster center. From the best route cluster we take the best GPS track, this is based on the route with the lowest distance to other routes. With this approach we get several GPS tracks as recommendation.

Now we will discuss the components in greater detail.

### 4.3.2. Heat map

The heat map is a grid of heat cells of locations. The grid is stored in a hash table because the matrix equivalent would be very sparse. The keys of the hash table are the locations and the values are the heats of that corresponding location. Every GPS track is rasterized by generating a sequence of lines represented in the form of points by converting GPS locations to list of points using the Bresenham algorithm. For every unique point from a rasterized GPS track one unit of heat is incremented in the hash table.

To determine the heat of a route, the average heat is taken of every location from a GPS track. GPS track heat is positively correlated with popularity of routes.

### 4.3.3. Filters

Before clustering routes, we first filter out a portion of the routes. This has two reasons: it improves the run time performance since calculating trajectory distances scales quadratically with respect to amount of routes, and some routes can be determined to be poor just by the track alone. Our three filters are:

- The circular test. The routes that our experts wanted to see where all circuits. To filter our routes that go from A to B, we check whether the distance between the start- and endpoint are not more than 10% of the total length of the route.
- The repetitiveness tests. This is used to filter out routes of for example someone ice skating many laps or riding on a race track. It works by plotting the route on a grid and counting both every square it passes through and every square it passes through more than once. If the ratio of revisited squares and total squares is higher than one half, the route is deemed to repetitive.
- GPS tracks with geographical leaps in them. Sometimes the GPS signal gets lost for a while. This can result in routes with big leaps that look like a perfectly straight line that doesn't follow roads. These are not presentable to a user, so if a track contains two consecutive points more than 5 km apart it gets filtered out.

### 4.3.4. Hierarchical clustering

After filtering, quick hierarchical clustering is performed based on centroids of routes. The centroid is calculated as the average latitude and longitude of each point. Since consecutive longitude lines have a lower distance between them near the poles than at the equator, we divide them by the cosine of the average latitude. Then we use fast hierarchical clustering from the fastcluster package. We ask for about 6-8 clusters depending on the amount of routes left after filtering. The goal here is to divide the routes in batches in different wind directions. If all routes are clustered at once, it takes more time and can result in two recommended routes to be very close. With this approach, we recommend not more than 1 route per area cluster, ensuring the recommended routes have some variation.

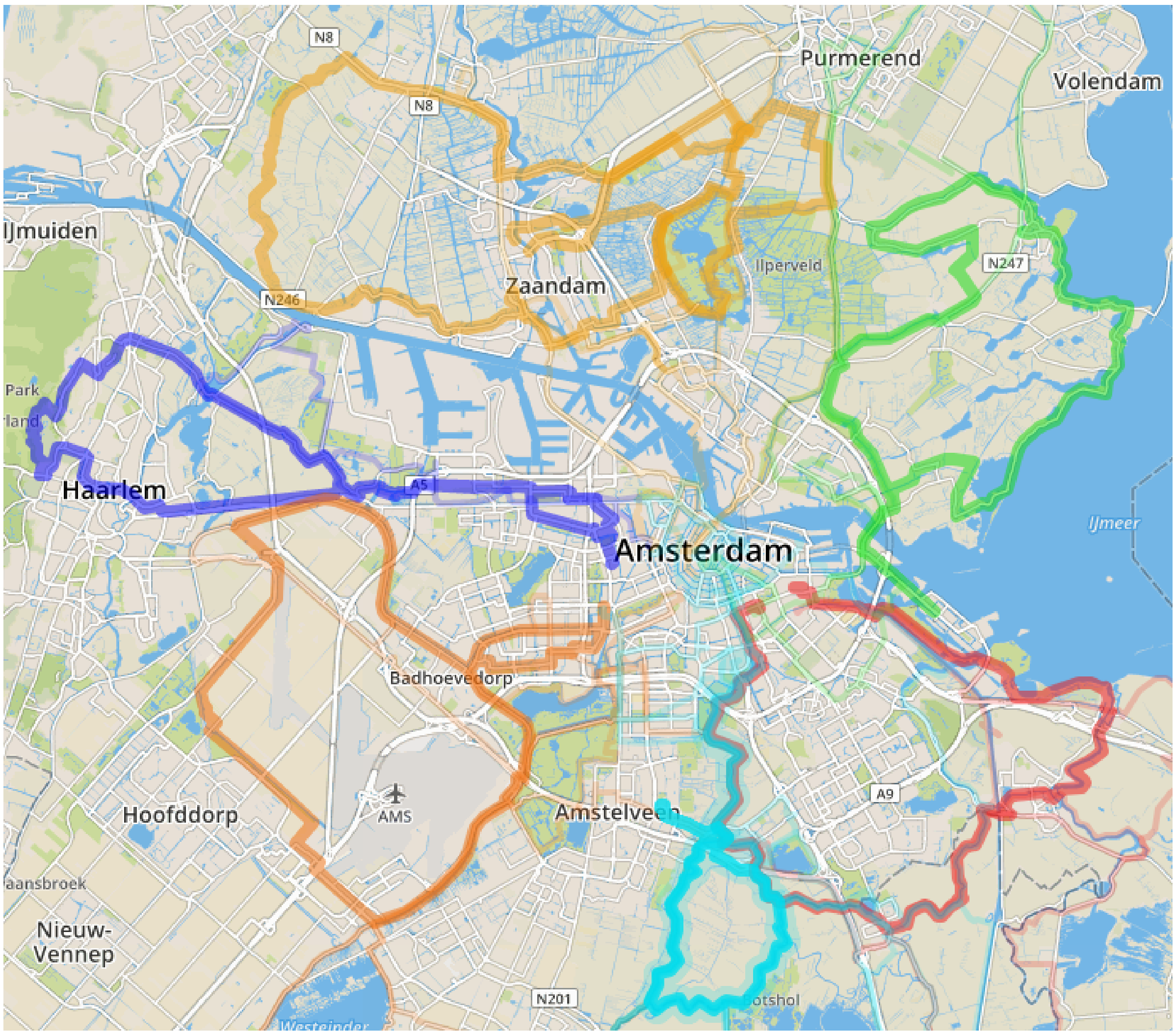


Figure 4.2: Different area clusters represented by different colors, the most important route clusters per area cluster are highlighted.

### 4.3.5. HDBSCAN

Once area clusters have been made, route clusters are created for each one of them. The goal here is to find routes that are nearly identical, so a simple centroid comparison will not suffice. Here we used the trajectory distance algorithm as described in the research chapter on a low-resolution version of the GPS track. This low resolution version is created by omitting points until 24 segments of roughly equal length are left. Once all pairwise distances are calculated, they are fed into an HDBSCAN clustering algorithm. We used HDBSCAN instead of K-means or hierarchical clustering because, as our TU Delft coach told us, the latter are actually partitioning algorithms: they force each route to be in a cluster while HDBSCAN allows outliers to be ignored. A large portion of routes are not part of a common route, and we don't want those to be merged into clusters to which they don't belong. In our final implementation, around 25% of filtered routes don't end up in any cluster.

### 4.3.6. Score calculation

After clusters have been formed, a recommendation score is calculated for each one of them. The score of a route cluster is determined by their cardinality, average distance and average Root mean squared error (RMSE). A high cardinality gives a higher cluster score since it means the route is more popular. The RMSE can be seen as the 'average distance from one route to an arbitrary other route in the cluster'. A higher RMSE means the routes in the cluster are actually quite different, and that lowers the score of a cluster because it means the cluster is more noisy. Finally a portion of the route distance is added to the score to compensate for the fact that longer routes naturally tend to have a higher RMSE. Additionally, we found empirically that longer routes tend to be more interesting and that without this, short uninteresting routes in a busy area tend to be recommended.

Per cluster the optimal route is selected, the optimal route is the route with the highest score, the score is determined by the following formula:

$$score = \frac{|routes\_in\_cluster| + route\_distance \cdot 0.5}{1 + route\_RMSE}$$

The multiplier of distance (0.5) was found empirically. The addition of 1 to the RMSE prevents an otherwise lackluster cluster with a very low RMSE to explode in score, since dividing by a very small number results in a very large number.

The final recommendations are the best route clusters of the best four area clusters. For each cluster, the route best representing the cluster is chosen by looking at the route with lowest RMSE. The whole algorithm as described was built by iteratively trying things, looking for problems and trying to solve them. We looked at clusters we liked, clusters we didn't like, and looked for ways in which these metrics differed. We started out by recommending the cluster with highest heat for example, but that resulted in routes going over busy but uninteresting roads to come up as recommendation, so then we tried to look at cluster sizes.

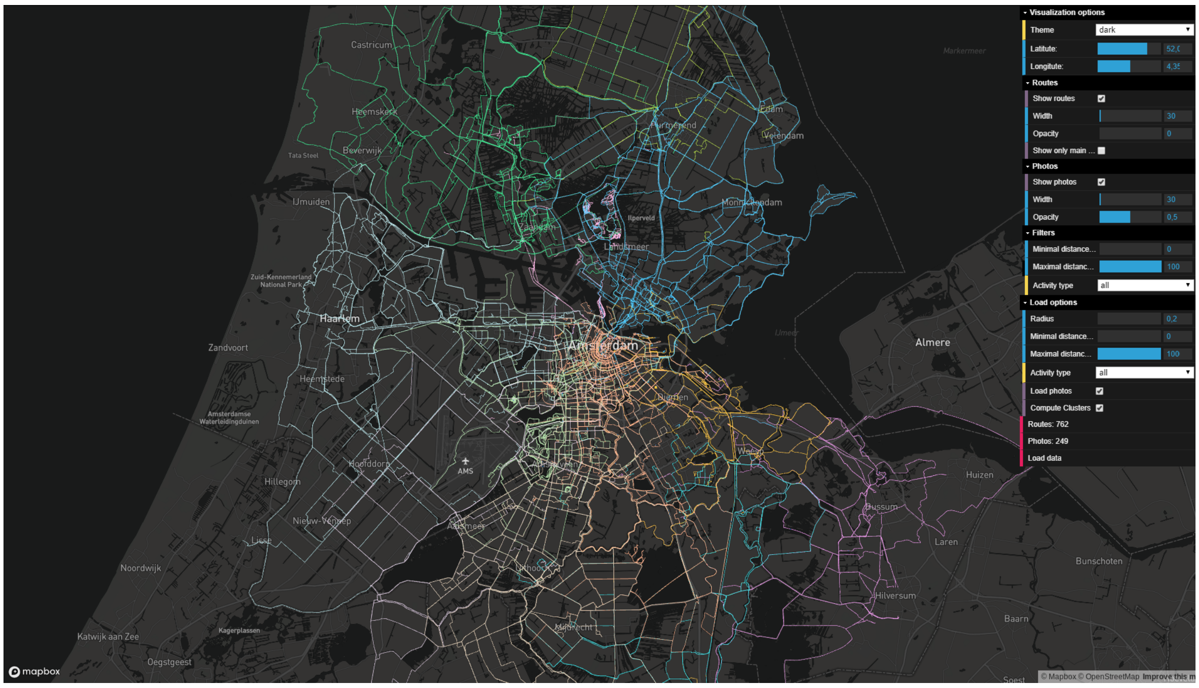
There is room for improvement in our score calculation. The best formula also depends on the application, but to get the routes our experts wanted, the way we described above turned out to work.

## 4.4. Web Application and Visualization Tool

As briefly mentioned in 3.2.2 we created two web applications, the actual Web Application (final product) and the Visualization Tool. In this section we will discuss the difference between these two applications and discuss the design and features of the Web App and the Visualization Tool.

### 4.4.1. Visualization Tool

The Visualization Tool is a tool we created to be able to visualize the results of our clustering algorithm. This tool is only designed for the developers and is therefore not accessible for the Relive users. With this tool we were able to load routes within a selected area and see (by color) which route belongs to which cluster. By selecting a route, only the routes in the same cluster are visualized and additional information (such as title, distance and elevation) are shown. The tool also has the ability to only show the best route of each cluster. Furthermore, the tool has options to, for example, filter on activity type and a distance range, so only routes with a certain activity type and distance will be loaded. The opacity and width of the routes can also be changed. We also added the ability to load photos, which are then visualized by dots on the map. When such a ”photo dot” has been clicked, the photo on that particular location will be shown. The opacity and width of the ”photo dots” can also be changed, just like with the routes. All of this proved to be very useful when testing out the results of our clustering algorithm. With the tool we could easily change parameters and see if the results met our expectations.



Visualization options
Theme: dark
Latitute: 52,0
Longitute: 4,35
Routes
Show routes ☑
Width 30
Opacity 0
Show only main ... ☐
Photos
Show photos ☑
Width 30
Opacity 0,5
Filters
Minimal distance... 0
Maximal distanc... 100
Activity type: all
Load options
Radius 0,2
Minimal distance... 0
Maximal distanc... 100
Activity type: all
Load photos ☑
Compute Clusters ☑
Routes: 762
Photos: 249
Load data

Figure 4.3: Screenshot of the Visualization Tool

### 4.4.2. Web Application

#### Initial design

After the creation of the Visualization Tool, we started working on the actual final product, the Web App. The Web App is the web application which eventually is going to be used by the users of Relive. The User Interface of the Web App has been designed by our client. They decided how our final product should look like and drew some sketches so they could show us their ideas.

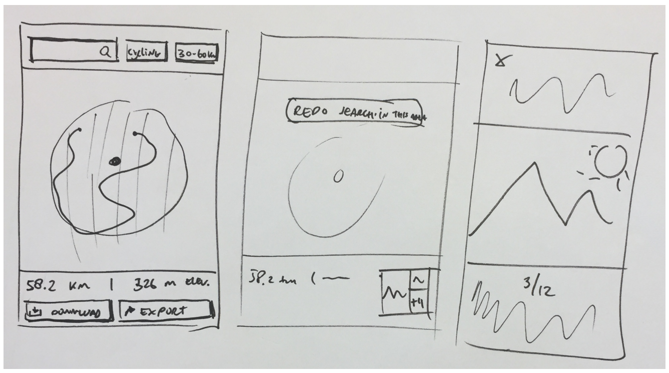


Figure 4.4: Design of the Web Application

Since Relive is a mobile application, a lot of Relive users also visit the Relive website on their mobile device, therefore Relive wanted us to first focus on a User Interface that works on a mobile browser and then make small changes so it also works on normal (PC) web browsers. The design shows the following features:

- Location search bar
- Activity type filter
- Distance filter
- Map with routes visualized
- Visualization of area in which the routes are loaded
- Route information
- Download button
- Export button

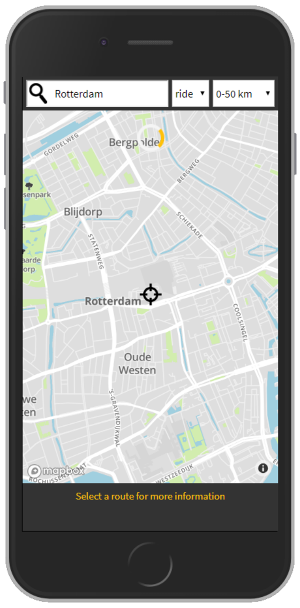
What these features exactly do and which features we incorporated in our final product will be discussed next.

#### Final design

The final design of our product is almost the same as the initial design of the client. We chose to extract the visualization of the area and the export button from the features of the final design. The visualization of the area has been removed, since it created ambiguity and was not clear for the user what it actually meant. With the export button the user would be able to export the selected route for use with third party applications. Since this is almost the same as downloading a file of the route, we decided it was not necessary to include the export button in our final product. All the other features designed by our client were included in the Web App.

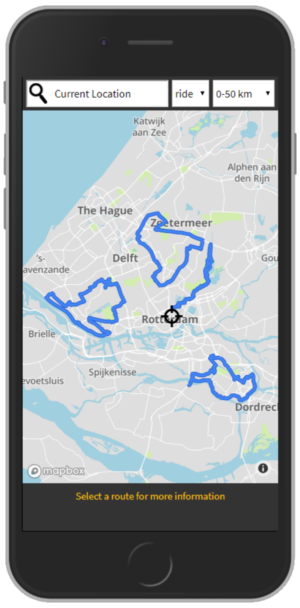
With the Web App you are able to select a location by clicking on the map or search for a location using the search bar in the top left corner. When a location has been selected, the Web App will start loading the best routes within a certain area of the selected location. On start up the Web App will ask permission to use the current location of the user. When permission has been given, the app will use the current location as search location the next time the user starts the app. The user will also be able to select its current location from the suggested locations of the search bar when permission has been given. Furthermore the user has the option to choose a particular activity type for which they want to get routes recommended and select a range of distance they want the routes to be within. The possible ranges of distances the user is able to select are dependent on the activity type they have currently selected. When the user location, activity type or distance range changes, the app will automatically start loading routes which satisfy the new preferences of the user, while the old recommended routes are still visible and interactive.

When the recommended routes are loaded they will be shown in blue on the map. The window of the app will zoom out and show all the routes within the screen. The user is then able to click on a route to see information about that route. When a route has been selected, the route will turn yellow and the window will zoom in again, so the selected route will be in shown in more detail. Distance and elevation of the selected route will be shown at the bottom of the screen together with a download button, to download the selected route as a GPX[7] file. The user is also able to rotate through the routes with the arrow buttons. Finally when a location which does not have any routes in the area has been selected, a message will pop up which communicates this to the user. Below we included some screenshots of the Web App showing different stages.



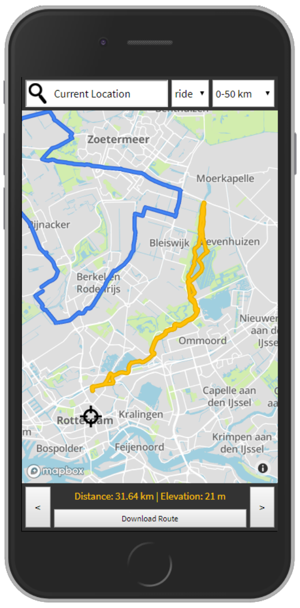
Select a route for more information

Loading routes



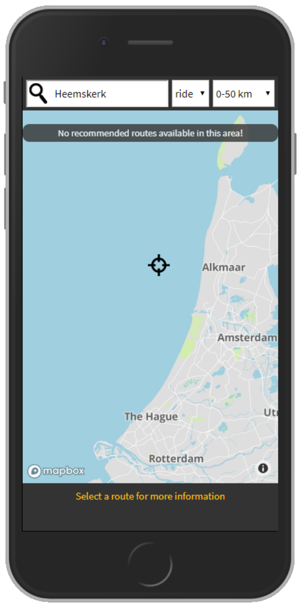
Select a route for more information

Routes are loaded



Distance: 31.64 km | Elevation: 21 m

User selected route



No recommended routes available in this area!

Select a route for more information

No routes available

Figure 4.5: Four screenshots of different stages of the Web App

[7]https://wiki.openstreetmap.org/wiki/GPX

# 5 Product Quality Evaluation

In this chapter we list the various ways in which we evaluated the quality of our product, both from a software quality perspective and a user perspective. We discuss three points: The quality of the code base, the quality of the recommendations, and whether we have reached the goals we set out at the beginning of the project.

## 5.1. Testing

Testing our route recommendation engine was a challenge because of a variety of reasons. We will discuss how we approached evaluating the quality of the components and the challenges that we had.

### 5.1.1. Route quality

The first question that our TU Delft coach brought up during our first meeting was "how do you define a good route". This question is at the core of determining success for this project, but it is also a tough one to answer.

The quality of a route is highly subjective and it depends on the use case of the user. For example, some users might want a short intensive route while others want a longer, more relaxed route. Our plan was to use the experts at Relive to examine the quality of routes. They ride regularly and know the good routes in many areas. If our tool would recommend the routes that they know to be good, then the client is satisfied.

However, our TU Delft coach suggested that we should come up with a set of heuristics that determine the quality of a route. We spent some time analyzing route properties like average speed or distance, but those are not able to discriminate routes very well. More complicated heuristics (like road quality) were harder to obtain since they aren't in the Relive database and require the use of external sources. Our client suggested to stick to our original plan of assuming popularity is the most important heuristic and tweaking our algorithm, filtering out any false positives as they appear, until it would give route suggestion they agree with. In section 5.4 the results of this testing are discussed.

In retrospect, with the knowledge we gained from being involved in this project, we suspect that it is possible to come up with heuristics that estimate the quality of a route by looking at the shape alone. Nice routes tend to be roughly circular for example, while uninteresting routes tend to have many sharp turns or stay for a long time in the same small area. While the feedback of experts is very useful, we recommend to additionally create a test set of curated routes and finding patterns in the shape that distinguish them from randomly sampled routes. This way, the feedback loop can be tighter: it can be immediately seen whether a tweak in the algorithm results in better or worse identification of good routes.

### 5.1.2. Clustering algorithm
Our clustering algorithm is at the heart of our route recommendations. To help finding the best parameters for clustering, a 'report' is created after every clustering job. This report contains specific statistics like the size of all clusters, and the average distance of routes within one cluster. If for example this average distance was too big in many clusters, then distinct routes were incorrectly placed in the same cluster. The cluster algorithm would then be tweaked to be more conservative in allowing routes into clusters. Before the clustering process, some routes were filtered (as discussed in section 4.3). These filters were very easy to test by giving some example routes of which we know whether they should be filtered or not, so we wrote unit tests for these filters.

### 5.1.3. Testing the server and database
The server retrieves routes from the database containing all routes. Some test code was written that performs test queries and prints the result in the command line. It could then be manually verified that the query results were correct. Writing automatic tests for these would take much time and slow the test suite down, so we decided after our manual verification that it was good enough.

### 5.1.4. Testing user interfaces
Our two components with a graphical user interface were the Visualization Tool and the Web App. The Visualization Tool was only for internal use in order to see the output of the clustering algorithm on a map instead of in a report; the Visualization Tool was therefore a means of testing itself. For this reason, it wasn't important to make this tool intuitive or robust. The Web App on the other hand did need to be user friendly since any user of the Relive app should be able to use it with minimal effort. We let employees at Relive (who are also users of the App) try out our app, and then listened to their feedback. One thing they noticed for example is that the start- and end-point of the route, which were initially indicated by a green and red circle, were not useful to show. Therefore, we decided to remove them.

## 5.2. SIG
The Software Improvement Group (SIG) is a company that specializes in measuring code quality and giving recommendations to make code more robust and maintainable. They use various metrics to look at aspects such as unit complexity, test coverage, and code duplication. Halfway through the project, we submitted our code base to them for feedback. One week later they responded with recommendations to make the code better.

### 5.2.1. Unit complexity
The group found that our unit complexity was too high. By splitting large blocks up into smaller units they become easier to understand, test and maintain. Our longest method was `getSignificantTerms` which is supposed to give related tags to a route by searching for significant terms in titles that users give to their routes. This function is divided into multiple steps, like generating a foreground set and background set, retrieving significant terms, filtering false positives et cetera. These steps could be separated into separate functions. This ended up not being necessary however, since we removed the functionality of the code. In the end we shifted focus on improving the route clustering so the route context code wasn't fully integrated, so this issue resolved itself.

We kept the feedback in mind for other code however. One example we found ourselves was our route clustering code: at first, we had a method that did parsing of input routes, filtering bad routes, and converting them to a different data structure all in one go. Now our different route filters have their own function making them easier to understand and easier to test.

### 5.2.2. Unit test
The presence of test code was deemed promising, but the amount of tests is low compared to the amount of production code. Since the evaluation, we improved the amount unit tests. There are around 20 functions with clear input and output which we wrote tests for. Much of the server code was about interfacing with a MySQL database and unit tests are not suitable for testing database interactions. This code was checked with manual end-to-end tests. Writing automated tests for these would strain the

database and continuous integration server. Also, the client and visualization tool where all about input and output from the user interface so instead of unit tests, we focused on making the user interface intuitive and easy to use on mobile devices. For pure functions we did write unit tests. These mostly consisted of routines that compute properties of a route (like distance or centroid) that are easy to test. Because of time constraints we could not get a high coverage however. Our focus was getting good route recommendations and these are too subjective to test automatically. We used experts that inspected the quality of routes to test this.

## 5.3. Requirements Analysis

In this section we will compare our final product with the initial requirements we made and discuss why some requirements were not met or not feasible. See 3.1 for reference to requirements.

### 5.3.1. Must Haves

Our final product meets 3/5 of the Must Haves. Our product does not meet requirement 2 and 5. Requirement 2 (*The user must be able to select a start and end location*) has not been met since our client decided they don't want the user to be able to select a start and end location, because the product is focused on users who do not know the area they want to ride in. Instead of being able to select a start and end location, the user is able to select just one location. Routes will be recommended within a particular radius from this selected location.

Requirement 5 (*The user must not have to wait more than 10 seconds for loading the routes*) has not been met, because things like queries already take more than 10 seconds. Despite the fact that we tried hard to optimize the clustering algorithm and the SQL queries, we did not manage to get routes recommended within 10 seconds. In general the recommendation takes about 45 seconds. The fact that we do not pre-compute recommendations, is one of the main reasons we did not manage to recommend routes within 10 seconds, since, without pre-computing, our current design of the recommendation engine does not allow computing of recommendations to be fast enough. Pre-computing of data is something which is outside the scope of this project, since we did not have enough time to do that.

### 5.3.2. Should Haves

Our final product meets 1/4 of the Should Haves. Requirement 6 and 7 both deal with giving context to a route. Although showing pictures of routes and showing a descriptive text about a route is promising, those features both deal with privacy issues. Since we focused more on the quality of the recommended routes, we did not have enough time to filter out privacy sensitive information from photos and titles. Therefore we did not manage to give more context to a route other than the distance and the elevation of that route.

Furthermore we do not differentiate between tours and paths in the user interface (Requirement 8). We chose to not make this difference, because our client wanted us to only recommend tours, instead of tours and paths, so we filter out any route that is a path.

### 5.3.3. Could Haves

Our final product only meets Requirement 16, 22 and 27 of the Could Haves. Although the Could Have requirements are good features to include in our final product, we just did not have enough time to implement these features.

### 5.3.4. Won't Haves

Our final product does not have any features stated in the Won't Haves, so as expected meets 3/3 Won't Haves.

### 5.3.5. Non functional requirements

The final product does meet all the four Non Functional Requirements.

## 5.4. Route validation

As discussed in section 5.1, we used experts to get a list of good routes that our algorithm should be able to find. In table 5.1 we show how many of the expected routes were recommended by our tool. The 'expected routes' column shows the amount of routes that were given by experts in the area specified in the 'place' column. The 'Recommended expected routes' column shows how many of the expected routes were among the routes recommended by our tool. The score column shows the percentage of 'correctly' recommended routes.

| **Place** | **Expected routes** | **Recommended expected routes** | **Score** |
|---|---|---|---|
| Amsterdam | 3 | 2 | 67% |
| San Francisco | 3 | 1 | 33% |
| Copenhagen | 2 | 2 | 100% |
| Almere | 2 | 2 | 100% |
| Rotterdam | 1 | 1 | 100% |
| Total | 11 | 8 | 73% |

Table 5.1: An overview of how many expected routes were output by our algorithm.

It can be seen that while it sometimes misses a route, often the expected routes show up in the recommendations. It should be noted that sometimes, expected routes are 'almost recommended', meaning that a route is given that overlaps a lot with the expected route. For example, figure 5.1 shows that a back-and-forth trip was recommended, while a circle around Amsterdam north was expected. Also, our recommendation tool always gives four recommendations (unless there isn't enough data available in the area), while experts gave fewer. In figure 5.2 it can be seen how one route is among the expected routes, but the other three are not.

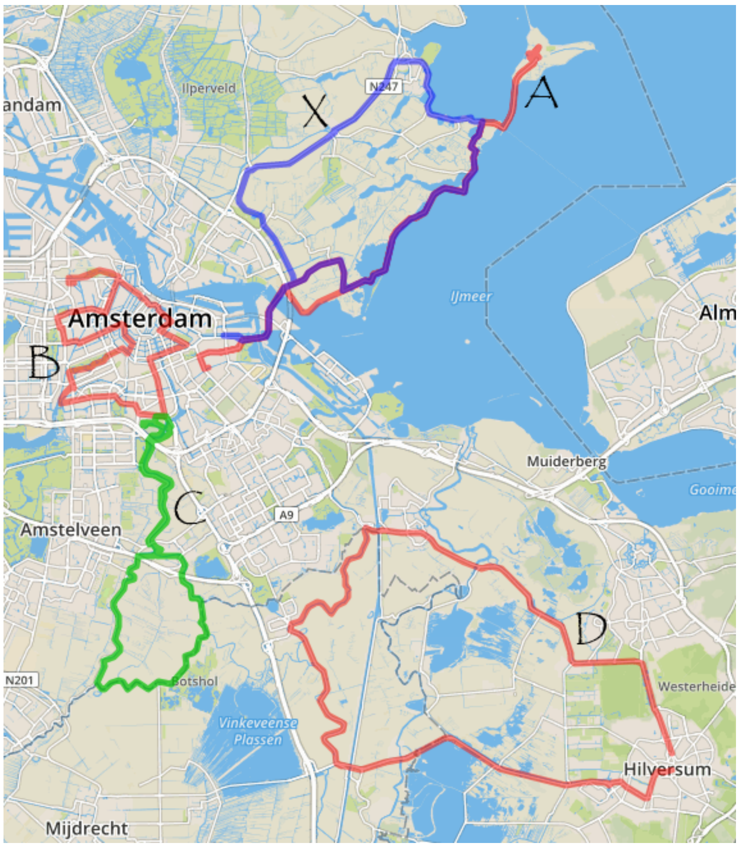


Figure 5.1: Routes A-D are our recommendations of routes near Amsterdam under 50 km. Route C is one of the expected routes. Route X is an expected route that is not recommended by our tool, but recommended route A shares a significant portion with it.

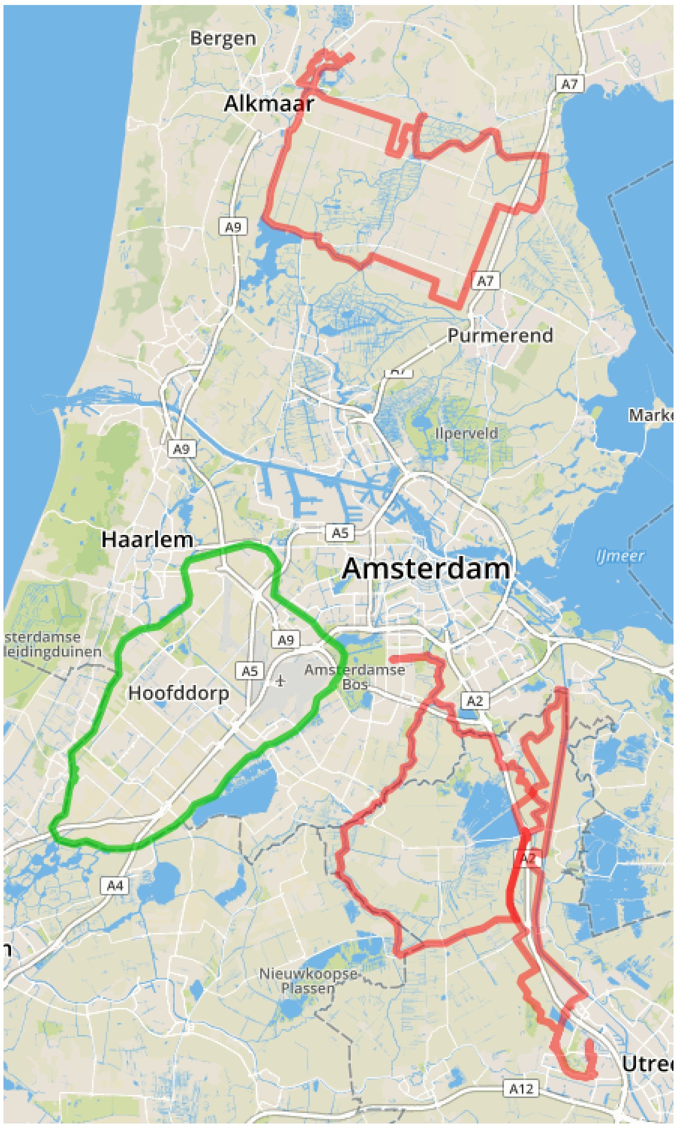


Figure 5.2: Recommendations near Amsterdam with a length between 50 and 100 km. The route around Hoofddorp (marked in green) is also an expert route.

Some of the feedback we got from experts was that routes, while nice, started very far away from the specified location. It is undesirable to travel very far before you get to the part of the route that is actually good. For routes under 50 km, we intend to only consider routes that pass through a circle of radius 15 km. We considered lowering this radius, but we found that this lowered the clustering quality significantly. A solution might be to pass the user's chosen location to the clustering algorithm and give clusters that are closer a higher score. Because of time constraints we didn't implement this however.

# 6 Process

In this chapter we will briefly describe the different phases which we went through during the project. Additionally, we discuss how we communicated with our client and set our goals every week.

## 6.1. Phases

In the first two weeks of the project our client recommended us to start building a Visualization Tool which we could use to visualize our routes and use for debugging purposes (see 4.4.1). In this phase Alessandro started setting up the Visualization Tool and the Web App, while Ivo started with setting up the server. In the meantime Jeroen started researching different clustering algorithms and Dennis researched if we could retrieve valuable information from route title. After this phase, the whole team did research to answer different research questions (see 2.2). After the research phase, Alessandro finalized the Visualization Tool and started working on the Web Application (the final product) (see 4.4.2), while Jeroen and Dennis continued working on improving the clustering algorithm and Ivo worked on the server API. Finally there was the phase of finalizing the final product and writing the Final Report. For optimizing and tweaking the clustering algorithm, the Visualization Tool turned out to be quite handy. We choose to first start building a tool and get familiar with the data, because then we could do more efficient research, since we know what kind of possibilities there are with the data.

## 6.2. Meetings and Goals

To come to an end product that satisfied the client's needs, we had meetings with Relive at the beginning of each week to set the goals of that particular week. This approach worked decently in the first couple of weeks, but after a while it became challenging to define relevant and specific goals. To be able to still define those goals per week, we decided to set more high level goals for the end of every week and divide those high level goals into sub goals, which needed to be achieved at the end of every day. This approach led to a much more structured way of working and kept each team member more focused on their particular task. It also created possibility to ask more specific questions when something was unclear, since we had a more specific goal we were trying to achieve.

# 7 Conclusion

We implemented a software tool that allows outdoor enthusiasts to find interesting routes for their rides. The end products are a recommendation engine that takes a large dataset of routes and outputs the routes that are deemed the best, a server that retrieves routes from a route database and passes them through the recommendation engine, and a client interface in the form of a website that allows users to click on a world map and then view and download recommended routes.

The project started by exploring the data that our client, Relive, has available. We looked both at the routes themselves as well as the titles and photos users attributed to their activities and looked what can be used for a good recommendation. The assumption was that we could find good routes by looking at routes that were popular. It was found that while indeed interesting routes (as determined by experts) tend to be popular, routes that go over popular roads are not always interesting: when merely making a heat map and looking for 'hot' routes, routes that simply go over many main roads become recommended. A clustering algorithm was used to find routes that are truly taken by many users. We used a trajectory distance algorithm to find routes that were close to each other and put similar routes together in a cluster. This resulted in adequate recommendations.

Supplying context for a route turned out to be challenging. The two most common types of photos are photos of the user and landscape photos. While the latter are interesting, they are hard to distinguish from the former. Titles contained mostly the default "Afternoon ride" and variations, or very personal titles. A smart significant terms algorithm can be used to find relevant terms for a route from user titles, but they are still mostly just the name of the place of the route which can also be retrieved from map data. With extra research, an algorithm can be found that adds context such as titles or photos to routes, but our implementation does not include this functionality yet because we focused on getting the recommendations right.

In conclusion, the basic functionality works well but there are still quality and performance issues that need to be solved before it is production-ready.

# 8 Discussion

## 8.1. Process reflection

In the first weeks of our project we were very content with the process we made. Everyone had its own tasks and we created the visualization tool and a first prototype of the clustering algorithm very fast. After that we had the research phase in which we mostly worked on the Research document and did not do any coding. This phase took longer than expected and did not go flawless. We first created a research report without any input from our coach, because we had trouble finding a coach at the beginning. When we handed in the first version of our research report we basically had to change the whole approach and make an almost completely new report. After this phase we started coding again, but soon we noticed that this did not go as well as the first weeks. This was the reason for changing our approach to set a goal at the beginning of each day (see section 6.2). When this change of approach was applied we made a lot of progress again and we managed to get into the same workflow as the first two weeks again. The final phase of writing the final report also went very well and did not bring up any issues or problems.

## 8.2. Ethical Implications

While a route recommendation engine does not have great ethical implications, there are certain problems regarding privacy that might arise. For example, a route gives information about when and where a certain user could be found, and the activity information contains the title, photos and even health information if they have the right hardware for that.

Because we have access to such personal information, it is critically important that this information does not become accessible to users of the route recommendation engine, i.e. it should not be possible to derive any information about the creator of a route. This can be implemented by only sending the required information, such as the path, distance and total elevation gain to the web clients. Additionally, the start and end of routes may need to be cut to avoid revealing where people live.

In the interest of time, this has not been implemented yet. We and the client deemed it more important to get the product working than to ensure full security, since the product is currently in development. As such the final product is not production-ready. We urgently request Relive to fix these privacy issues before putting the product live.

## 8.3. Recommendations

Although we were able to deliver a fully working product, there is still a lot of room for improvement. The current version is not able to recommend routes within 10 seconds. This is something that must be fixed before our product is usable by Relive users. This could be realized by using precomputed data. Another possibility is the use of a combined database instead of separate databases. For a more detailed explanation of the these existing databases see subsection 3.2.1. The Routes and Activity database would be merged into one database, so only one query is needed and it is easier to filter out routes. Another recommendation would be switch to a database which is good at handling spatial queries, so searching for routes, given a location, would be faster. Furthermore we would like to tweak the clustering algorithm parameters to optimize the results of the algorithm.

Security and privacy issues are also problems that need to be considered before releasing the route recommendation engine. Finally we would recommend Relive to enable users to add more context to their routes, so this context can be shown as additional information of the recommended routes. Relive could also encourage users to think of more descriptive titles for their routes, so these can also be used in our route recommendation engine.

# A SIG Evaluation

Below is the unedited feedback we received from our code submissions to the Software Improvement Group.[1] We received our second feedback after the deadline for this report. It was not included in the report we handed in originally, but it has been included for the sake of completeness for the upload to the TU Delft repository.[2]

## A.1. First feedback

De code van het systeem scoort 3.6 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Unit Complexity vanwege de lagere deelscores als mogelijke verbeterpunten.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes.

In jullie project is getSignificantTerms() met afstand de grootste methode. Jullie geven in het commentaar bovenin al aan dat de methode het hele algoritme implementeerd. Vervolgens worden de verschillende stappen met commentaar aangegeven. Door de stappen naar aparte methodes te splitsen zorg je er voor dat het algoritme op lange termijn onderhoudbaar blijft.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.

Ook hier zou het helpen om de structuur van het algoritme meer in de code te verwerken.

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid tests blijft nog wel wat achter bij de hoeveelheid productiecode, hopelijk lukt het nog om dat tijdens het vervolg van het project te laten stijgen.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

[1] https://www.sig.eu
[2] https://repository.tudelft.nl

## A.2. Second feedback

In de tweede upload zien we dat het project iets groter is geworden. De score voor onderhoudbaarheid is in vergelijking met de eerste upload gestegen.

Bij Unit Size en Unit Complexity, de verbeterpunten uit de eerste upload, zien we een kleine verbetering. Daarnaast hebben jullie zelfstandig een aantal andere verbeteringen in de code doorgevoerd. De combinatie van deze acties heeft uiteindelijk geleid tot de stijging van de totaalscore, die nu op 4 sterren uitkomt.

Naast de toename in de hoeveelheid productiecode is het goed om te zien dat jullie ook nieuwe testcode hebben toegevoegd. De hoeveelheid tests ziet er dan ook nog steeds goed uit.

Uit deze observaties kunnen we concluderen dat de aanbevelingen uit de feedback op de eerste upload grotendeels zijn meegenomen tijdens het ontwikkeltraject.

# B
# Original Problem description

The initial pitch for the project provided by the client is the following: ”Based on our data set of nearly 100 million GPS-tracked activities, can we build the perfect route recommendations for any outdoor enthusiast? Can we answer questions like; do you have suggestions for a 50k paved cycling route starting and ending at EWI? Or; I'm looking for a beautiful 10k within a one-hour drive of Amsterdam? This project would start with researching our data set of tracked activities. In a language of your choice, see whether you can crunch the data and build a tool to give generated recommended routes to our users.” [3].

# C
# Info sheet

On the next page the info sheet has been given.

**Title** Route Recommendation Engine

**Name of client organization** Relive CC

**Date of final presentation** July 2nd 2018

## Description

How do you plan a route for hiking, running or riding in an area that's unknown to you? You probably want nice roads with little traffic and great sights. Existing solutions require you to set the way points yourself or select from a curated list of routes. We tried a new approach: looking at routes that users have already taken to find the best routes in the area. Our client Relive makes 3D videos of your outdoor adventures. With over 2 million users, their database contains many existing routes.

Our challenge was to find the best route that complies with the user's location and distance preference. In our research phase, we found that clustering approaches can be used to find which routes were taken by many users and therefore more likely to be interesting. We also looked into the possibility of using activity titles and photos to give context to a route, but that didn't pan out.

During the development, we focused on getting visual results from early on so that experts at Relive could easily give feedback on recommended routes. We struggled with making the recommendations fast enough since the route database was not well suited for this task and route clustering scales poorly.

The final product is a website that communicates with a route recommendation server using our route clustering engine. Optimizing the database and clustering algorithm or caching recommendations can help speeding up the process so the waiting time becomes low enough to be acceptable for deployment.

## Members

**Jeroen Esseveld**
Interests: Algorithmics, Data analysis, Optimization
Role: Recommendation Algorithm and route clustering

**Ivo Wilms**
Interests: No.
Role: Server API, database interface

**Dennis Korpel**
Interests: Assembly, Gaming, Composing music
Role: Route context research, route clustering

**Alessandro Ariës**
Interests: Web Development, Algorithms, Databases
Role: Web Application, Visualization Tool

## Contact

| Name | Affiliation | email |
|---|---|---|
| Christoph Lofi | Web Information Systems | |
| Yousef El-Dardiry | Relive cc | |
| Lex Daniels | Relive cc | |
| Dennis Korpel | student | dkorpel@live.nl |
| Alessandro Ariës | student | alessandro.aries@gmail.com |
| Ivo Wilms | student | ivoatinternet@gmail.com |
| Jeroen Esseveld | student | jeroenesseveld@gmail.com |

# Glossary

**climb** The total positive change in elevation. 13

**GPS track** A sequence of GPS coordinates. 21, 22

**JSON** JavaScript Object Notation. 16, 17, 21

**npm** Node Package Manager. 20

**ORM** Object Relational Mapping. 16

**RMSE** Root mean squared error. 23

# Bibliography

[1] Philippe C. Besse, Brendan Guillouet, Jean-Michel Loubes, and François Royer. Review and perspective for distance based trajectory clustering. *CoRR*, abs/1508.04904, 2015.

[2] Bin Jiang. Head/tail breaks: A new classification scheme for data with a heavy-tailed distribution. *The Professional Geographer*, 65(3):482–494, 2013.

[3] Relive. Route recommendation engine. *BepSys*, 2018.