

Title of the Thesis

Subtitle if needed

Jack Jibb

School of Computing and Mathematical Sciences, University at Greenwich

2024-2025

Abstract

In professional cycling, athletes are often sent to training camps in locations such as the French Alps, Andorra, Sierra Nevada, Colorado Springs, or other high altitude destinations. While a big part of this is altitude acclimatisation, many cyclists report that the biggest effect is actually the suitability of the roads for training. Long, car-devoid mountains, where athletes can just put their head down and focus on their effort means that their training quality is improved, as opposed to having to ride through small villages, stopping at intersections and being constantly vigilant of overtaking cars or traffic furniture. This thesis explores the possibility of being able to track and quantize "Training Suitability" of roads. What is proposed is an integrated framework that can semantically partition continuous trajectory data from a cyclist's GPS device, and meaningfully define each segment in the context of training suitability. Starting from raw GPS traces, we apply geometric filtering and map-matching to correct noise and align positions with a reference network. An adaptive segmentation algorithm then identifies breakpoints using curvature statistics, speed variance, and road metadata (such as road name, or speed limit), yielding segments that have a homogenous quality. Each segment is characterized by spatial, temporal, and contextual features and subsequently classified through a Wavelet Transform function that quantizes variability and changepoints in data streams. The scalability and robustness of the framework will be evaluated through a small-scale live web application.

Contents

Abstract	1
1 Introduction	4
1.1 Background	4
1.2 Research Objectives	4
2 Literature Review	6
2.1 Training Suitability and Segmentation Algorithm Design and Analysis	6
2.1.1 Research Objectives	6
2.1.2 Algorithmic Approaches to Segmentation and Categorization	6
2.1.3 Applications in Cyclist Route Planning	6
2.1.4 Methodological Considerations and Assumptions	6
2.2 Training Suitability Scoring	6
2.2.1 Research Objectives	6
2.2.2 Approaches to Suitability Scoring	6
2.2.3 Applications in Training Suitability Scoring	8
2.2.4 Methodological Considerations and Assumptions	8
2.3 LSEPI Analysis	9
2.3.1 Legal	9
2.3.2 Social	9
2.3.3 Ethical	9
2.3.4 Political	9
3 Requirements	10
3.1 Software Requirements Specification (SRS)	10
3.1.1 Purpose and Scope	10
3.1.2 Intended Audience and Reading Suggestions	10
3.1.3 Overall Description	10
3.1.4 Specific Requirements	10
3.2 Standards and Compliance	12
4 Methodology	13
5 Design	14
5.1 Scope and Purpose	14
5.2 Architectural Design	14

5.2.1	Architectural Views	14
5.2.2	Design Rationale	16
5.2.3	Design Patterns and Styles	16
5.3	Module-Level Design	16
5.3.1	Module Decomposition	16
5.3.2	Module Specifications	17
5.4	Interface Design	17
5.4.1	User Interface	17
5.4.2	Software Interfaces	17
5.5	Data Design	17
5.5.1	Data Models	17
5.5.2	Database Schema	17
5.5.3	Data Dictionary	17
5.6	Standards and Compliance in Design	17
6	Implementation	18
6.1	Component Breakdown	18
6.1.1	Input Pipeline	18
6.1.2	Segmentation Engine	21
6.1.3	TSS Engine	21
6.2	Testing and Results	21
7	Results and Conclusions	22
8	Further Discussions and Research Gaps	23
A	Appendix	24
A.1	Further Reading	24
A.2	Source Code	24
A.3	File Structure	24
A.4	Additional Documentation	24

List of Figures

2.1	The 7 Zone Training Model, courtesy of Indoor Cycling Association	7
2.2	My Power Curve for August, 2024. The power curve can estimate what would be considered "hard" for an athlete at a given duration. Image courtesy of intervals.icu	8
5.1	High level system architecture for the Route Segmentation and Analysis System	15

6.1	A visual representation of the Road network used for testing the pipeline. Green roads are the downloaded .pbf map edges, and the blue and yellow line is the series of GPX points plotted over the top to verify that the bounds had successfully covered the route	19
-----	--	----

List of Tables

Chapter 1

Introduction

Modern cycling training for professional athletes has been perfected down to a science. Since the days of Team Sky, "marginal gains" and hyper optimisation in all aspects of training, such as nutrition, periodization, strength, endurance and race calendar have been optimised to such a degree that the difference between the top echelon of professional cycling and the middle tier is less than a few percent. Pro riders have the advantage of having everything done for them when it comes to planning training, from their schedule to their locations. However, amateur riders do not have this same advantage, and as I have experienced many times, often we go out for rides, looking for new, longer routes to do training on, only to find that we end up down a country lane with blind corners, or a town with really annoying road furniture. The first step in planning a route has always been to go onto a mapping website, and draw a route, that you could later save to your GPS device to follow, or to write down turn by turn on the stem of your handlebar. While I won't mess with the sacred art of taping a piece of paper on your handlebars in the most aerodynamic way you can, I can maybe just revolutionise the plotting of routes for cyclists looking to get the most out of their training. Enter the concept of route "segments". In Strava, a Segment is a piece of road that has been given a place in their database, for athletes to compete for times on. Getting a "KOM" (King of the Mountain) on a Strava segment is a coveted title by many amateurs. In our context, a segment will be keeping track of more than just performance metrics. It will keep track of the inherent data of the road from an open source mapping software called OpenStreetMap, and all the data will be combined to give an overall "score" of the segment in the context of training suitability.

1.1 Background

1.2 Research Objectives

1. To develop a system that can take user cycling data in the form of GPX files, and provide feedback in the context of training suitability
2. To design and implement a Route Segmentation algorithm that splits a GPX file into meaningful sections by enriching a route-matched path from the data with road information from OpenStreetMap.

3. To be able to store the segments in a database, and future GPX files can be converted into arrays of these segments, by matching their segments with the database, and returning segments if there is a margin of error less than a certain threshold.
4. To implement batch processing of GPX files so users can submit large quantities of GPX data to aid in segment generation.
5. To create a two-way pipeline system that takes either GPX data, or a custom-plotted route, and feeds back a list of curated segments from the database.

Chapter 2

Literature Review

2.1 Training Suitability and Segmentation Algorithm Design and Analysis

2.1.1 Research Objectives

2.1.2 Algorithmic Approaches to Segmentation and Categorization

2.1.3 Applications in Cyclist Route Planning

2.1.4 Methodological Considerations and Assumptions

2.2 Training Suitability Scoring

2.2.1 Research Objectives

2.2.2 Approaches to Suitability Scoring

Sharifzadeh et al. "Change Detection in Time Series Data Using Wavelet Footprints" An interesting approach to Suitability scoring relates to Fast Fourier Transforms, and a convolution approach comes from Medhi Sharifzadeh et al, from the University of Southern California, introducing a concept known as "wavelet footprints" as a compact, multi-resolution approach to representation of spatial-temporal trajectories. Wavelet footprints are a more granular version of the Wavelet Transform, which in turn is a version of the Fourier Transform. The approach involves using Wavelets to transform a signal into the "Wavelet Domain". The smaller the wavelet, the more reactive it will be to change in the original signal, so by adjusting the size of the wavelet, the signal can be filtered to be more or less reactive to change. Wavelet footprints have an advantage over the general Wavelet transform, where they only retain the most significant components. This is done by having wavelets occur in orthogonal sets. Due to Heisenberg's Uncertainty Principle, it is impossible to perfectly describe both the frequency content of a signal and the location in time of the signal. Time-domain and Frequency-Domain analysis in this regard are at opposite ends of the spectrum, but the Wavelet Domain sits in the middle, allowing for a sliding scale value, where a larger scale gives more frequency and less time resolution, while smaller scales give less frequency, and more time resolution.

The advantage of using wavelet footprints over the Fourier Transform means that it is possible to isolate points in the signal where significant changes occur in specific metrics, or combination of metrics, allowing flexibility in choosing what conditions must be met to enact a segmentation.

Indoor Cycling Association: "How Much Time in the Red Zone?" This article details a breakdown of the 7-training-zone model, which provide a good template for cutoff times for tuning Wavelet Footprints for analysing the GPX signal. The general consensus is having 5 zones is a good compromise between continuous training definitions (specific power numbers) and binary (hard or easy). While the article describes 7 zones, Zones 1-3 all fall outside of the hour range, which for the purpose of segment analysis would be fairly useless. The 5-Zone model approach has good suitability to wavelet analysis, since small scale values for a wavelet would detect short burst efforts, while larger scales will detect longer, sustained effort. Having 5 zones allows for a reduced scale array, contributing to higher performance. Here is a graphic, courtesy of the Indoor Cycling Association, that breaks down each zone. Zones 3-7 will be used for Wavelet Analysis.


 Power and Heart Rate Training Zones and RPE Chart							
ZONE	Name/Purpose	%FTP	%LTHR	RPE	RPE Description	How Does It Feel?	Duration
7	MAXIMAL	>150	n/a	10	Maximal, Explosive power	Gasping for air, can't say one word	5-20 seconds
6	Anaerobic Capacity	121-150%	n/a	9	Very, very hard	Breathless, ragged breathing, talking impossible. Severe sensation of leg effort.	30 seconds to 3 minutes
5	VO2 Max	106-120%	>106%	8	Very hard	Cannot talk, laboured breathing. Strong sensation of leg effort	3-8 minutes
4	Lactate Threshold	91-105%	95-105%	6-7	Hard	Deep forced breathing, but still sustainable. Moderate to greater sensation of leg effort.	10-60 minutes
					Moderately hard	Deep breathing; talking is very challenging	
3	Tempo	76-90%	84-94%	4-5	Somewhat hard	Heavier but rhythmic breathing, greater sensation of leg effort.	1-3 hours
					Moderate	Talking becomes uncomfortable.	
2	Aerobic Endurance	56-75%	69-83%	2-3	Easy	Light rhythmic breathing. Can maintain for hours.	Many hours
					Very easy	Can talk in complete sentences	
1	Active Recovery	<55%	<68%	<2	Very, very easy	Restful breathing; can sing.	All Day

Figure 2.1: The 7 Zone Training Model, courtesy of Indoor Cycling Association

Sean Hurley: "Normalized Power: What It Is and How to Use It" <https://www.trainerroad.com/blog/normalized-power-what-it-is-and-how-to-use-it/> TrainerRoad writer Sean Hurley provides a useful and concise definition of Normalized Power (NP), a metric invented by Dr. Andrew Coggan in his book *Training and Racing With a Power Meter*. NP "reflects the disproportionate metabolic cost of riding at high intensity, by weighting hard efforts and deemphasizing periods of easy spinning", according to Dr Coggan. Essentially, NP approximates what power a rider could have put out for the same effort, if their effort was steady-state. While it is not a completely accurate metric for effort, it is a really good indication of power variability, and as such is useful in determining the type of effort of a segment. The algorithm for determining NP is as follows:

1. Calculate a rolling 30 second average power for the duration

2. Raise each rolling average value to the fourth power.
3. Determine the average of all the rolling values
4. Take the fourth root.

For an input power signal, $P(t)$ over interval $(0, N)$, the formula for Normalized Power (NP) is:

$$\text{NP} = \left(\frac{1}{N} \sum_{i=1}^N (\overline{P}_r(i))^4 \right)^{1/4}$$

where P_r is the 30 second rolling average power starting at point i in the power data array.

2.2.3 Applications in Training Suitability Scoring

2.2.4 Methodological Considerations and Assumptions

- **Power:** All cyclists have different power thresholds; in other words, two cyclists may be going the same speed up a climb, but one may be going all out, and another just going easy. They may not even have the same power output. The one going easy could have a higher power output than the person going all out, depending on their weight. As such absolute power is a bad reference point for training suitability. Rather, we need to focus on power variation, as well as power curve. A power curve is the integration of all power numbers over a duration of exercise, plotted average power in watts on the y-axis, and max sustained duration for that average power on the x-axis. The graph tends to look like an exponential decay function. An example of my own personal cumulative power curve in 2024 is shown here:

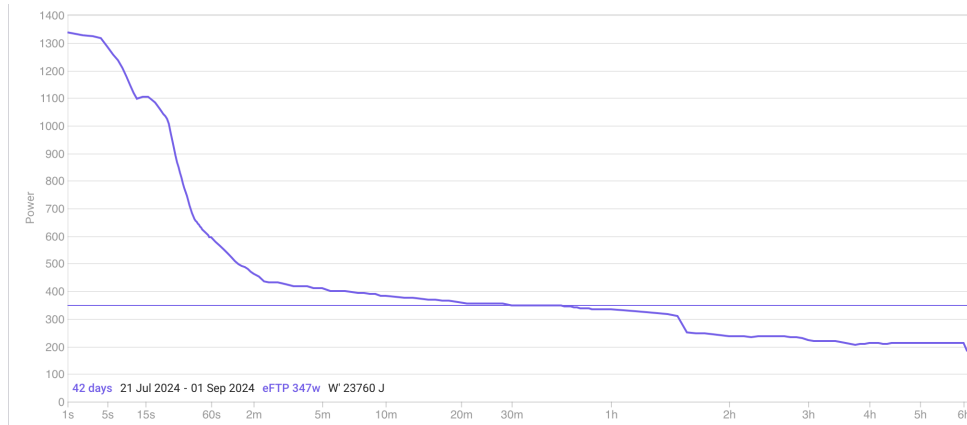


Figure 2.2: My Power Curve for August, 2024. The power curve can estimate what would be considered "hard" for an athlete at a given duration. Image courtesy of **intervals.icu**

- **Route Terrain:** Hilly or technical terrain (such as lots of sharp corners or non-pavement roads) can significantly affect how good a route is for training, but in different ways. Hilly terrain could be really good for consistent efforts, if the gradient is sustained, but if there are a lot of short, steep climbs, it would be harder to maintain any consistency in effort. Likewise, gravel or dirt roads could be good for endurance if they are consistent, but throw

in some sharp corners, and suddenly you have to brake a lot more, accelerate, and even focus on balancing more, which can induce more fatigue. Terrain is probably the most important metric that is intrinsic to the route itself.

- **Safety:** Safety is obviously very important in general when cycling, but it also plays a big part in performance. Having to focus on keeping safe on the road often means being ready or having to brake or slow down to avoid getting into dangerous situations. If a road can be considered "safe" (think long straight bike paths, or straight roads with very little traffic), then the athlete is free to focus more on their effort. To consider safety is a complicated problem, since there are many aspects. One method to consider could be to come up with a points system, and apply danger points as a suitability metric. Every "unsafe" property of a road can add danger points to the segment, and a subjective scoring system could be put in place initially, and in the future, a machine learning approach could be used.

2.3 LSEPI Analysis

2.3.1 Legal

2.3.2 Social

2.3.3 Ethical

2.3.4 Political

Chapter 3

Requirements

3.1 Software Requirements Specification (SRS)

3.1.1 Purpose and Scope

3.1.2 Intended Audience and Reading Suggestions

3.1.3 Overall Description

Product Perspective

Product Functions

User Characteristics

The typical user will be of a reasonable proficiency with other online mapping softwares, such as route creation in Strava or MapMyRide. This software is mostly proof of concept, so usability considerations will be considered less important over functionality.

Operating Environment

Design and Implementation Constraints

Assumptions and Dependencies

3.1.4 Specific Requirements

External Interface Requirements

User Interfaces Description of UI requirements.

The user interface only requires two features:

1. The ability to upload GPX files to the server
2. The ability to map a route via any interactive map service (Mapbox, Leaflet.js or Folium.py).

Software Interfaces APIs and protocols.

The system will link the front and back ends with a RESTful API. Also it is important to choose a front end framework that supports uploading multiple relatively large files (between 5-10MB

each). Since a lot of the application is written in Python, Flask is a good option for this. Other API connections will be implemented to communicate with the Segmentation and TSS engines. This allows them to be hosted on separate servers in the future, to give them more processing power.

Functional Requirements

The functional requirements of the system are as follows:

- FR1: The system must accept multiple GPX files as input via a file loading system
- FR2: The system shall allow users to draw a route using an interactive map UI
- FR3: The system shall align raw GPS Tracks to the road network for consistency.
- FR4: Upon receiving a matched route, the system shall identify route segments that are related by road features.
- FR5: The engine shall perform a two-pass segmentation. One to match the route with existing segments, and one to find new segments.
- FR6: Upon uploading a GPX file, the system shall check if a matching segment exists in a database (within spatial and directional tolerances). if found, the segment's hit count is incremented and Training Suitability Score values are added to the database's list.
- FR7: If no existing segment is found for a segment generated by the Segmentation engine, and the generated segment is sufficiently significant, the system shall create a new segment entry in the database.
- FR8: When the user draws a route in the interactive map, and queries the system without uploading a GPX, the system shall send the route to the Segmentation engine, and identify known segments along the route, along with associated TSS data. It shall **not** create new segments during the query.
- FR9: The system shall evaluate each segment's Training Suitability Score whenever a new GPX file is uploaded to the input pipeline by sending the segments to the TSS Engine.

Non-functional Requirements

- NF1: **Accuracy** -
 - Every GPX file must be converted into a GeoJson file that has less than 4% difference in distance
 - For a segment to be considered "matching", it must fall within a similarity parameter of 90
- NF2: **Performance** - The system must be able to process an average of 1,000 GPX points per second.
- NF3: **Extensibility** - The system must be built with scalability in mind, and each major component should be able to be isolated on a separate system. All components should communicate via API calls to keep this a reality.

NF4: **Privacy** - A default culling of the first and last 500m of each GPX file will provide some privacy for users' home addresses.

- No training specific data will be stored on the database, only the Training Suitability scores, to increase the data security of users.

NF5 **Compliance** - All data regulations by the UK government must be abided by.

Logical Database Requirements

Software System Attributes

Reliability

Availability As a proof-of-concept system, the availability of the system is only when required for testing and presenting.

Security Security isn't as important for the proof of concept, but a lot of standard considerations can be made that relate to LAMP stack web applications.

Maintainability Every component of the system will be independent, and input and output formats are documented in the documentation in A.

Portability The whole project will be made available via GitHub, and possibly a dockerized version of the system will be developed in the future for portability.

3.2 Standards and Compliance

Chapter 4

Methodology

Chapter 5

Design

5.1 Scope and Purpose

The following section is a comprehensive design plan for the Route Segmentation and Analysis System. The architecture will be described in compliance with ISO/IEC/IEEE 42010:2011 [?], which standardizes how systems and software architectures are documented. The System allows users to upload GPX files, or draw routes via an HTML interface. The uploaded routes are processed to identify meaningful **segments** (portions of the route that have similar road conditions). The segments are then analysed to gather training insights. Every segment will be sent to a database, where it is cross referenced with all values to see it falls within a determined margin of error (<5%). If it is, then the matching database entry will have its count increased by 1, and the training score averaged into the running average. Users can also query the system by drawing a prospective route, and will get in return a list of any known segments along the route, along with their score. The main architectural challenge in this system is to detect and match route segments directionally (direction matters), while also considering reliability.

5.2 Architectural Design

The overview of the system is presented in ???. Present a high-level system architecture diagram using UML or SysML.

5.2.1 Architectural Views

Concept View

The architectural concept of the system can be broken down into 3 layers:

- **Presentation Layer** - This is the front end; A web application where users can either upload their GPX files, or draw a route on an interactive map. This layer will communicate with the back end through HTTP API requests. This layer is mostly outside the scope of the project, since I am not a front end engineer.
- **Processing Layer** - The beef of the project, this includes the back-end of the system, and consists of input/output pipelines, and several processing engines. It will also contain

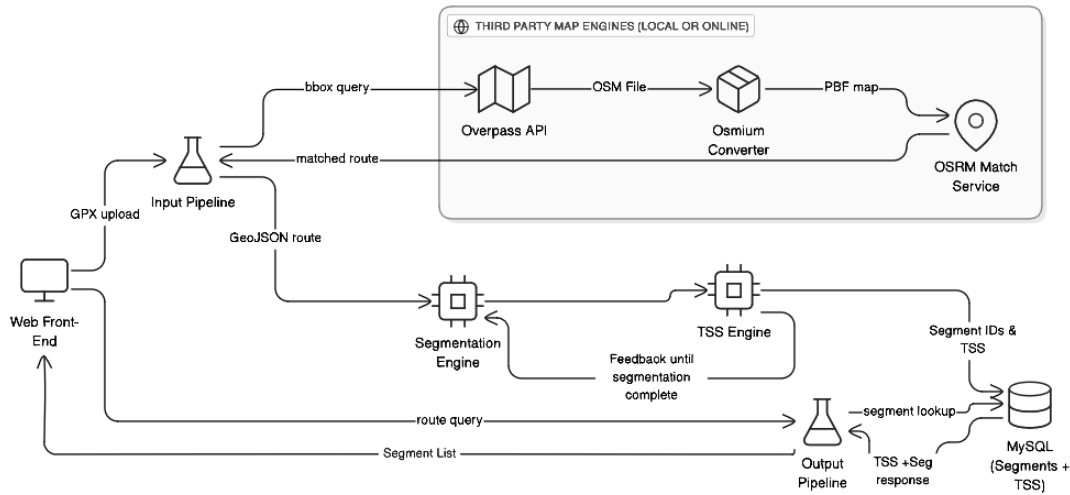


Figure 5.1: High level system architecture for the Route Segmentation and Analysis System

several controllers, written in Python, to coordinate the pipelines and to send information to and from the different engines.

- **Data Layer** - This layer is represented by a SQL database that stores permanent information about route segments. External APIs also fall into this category, such as the Overpass API to fetch map data from online. These components provide necessary data but are mostly abstracted away, in the processing layer.

Functional View

The functional view includes key modules and their responsibilities.

- **Front end Web App:** Serves HTTP pages and gets information from the user.
- **Front End Controller:** Takes user information and routes it to correct API endpoint.
- **Input Pipeline Orchestrator:** Controller that handles input information routing, sending the GPX files to the pipeline, and pulling the resulting GeoJSON objects out.
- **Engine Orchestrator:** Takes routes, passes them to Segmentation algorithm, and handles the "two pass" system between Segmentation and TSS Engines.
- **DTO Orchestrator:** Formats the resulting segments, into the correct Data Object and passes it to SQL Database
- **Output Pipeline Orchestrator:** Receives the Route data from the Input Pipeline Orchestrator, validates it and sends it to Segmentation Engine. Gets segment list, and routes it back to front end in GeoJson format.
- **Segmentation Engine:** Written in C++, the engine is a specialized service that performs route segmentation and matching logic.
- **TSS Engine:** Takes a list of segments and the input GPX data, and assigns scores based on the training data that happens during each segment

- **SQL Database:** Stores all persistent segments, along with a hit count, and a list of TSS objects that have been calculated on matching segments.
- **Osmium Engine:** A docker service that runs a .osm to .pbk conversion and merging process. Used in the input pipeline
- **OSRM Matching Engine:** Another docker service that takes small GeoJson chunks, and matches them to a map network.

Physical View

Hardware deployment and network design.

Information View

Data Flow Diagram, and Database Schema diagrams

Behavioral View

Use case and sequence diagrams for the system

5.2.2 Design Rationale

The choice to go with a **Pipeline Architecture** for processing data is motivated by the main use case. Users who are looking to find a route to ride are not interested in uploading GPX, but conversely, users who are done with their ride aren't looking to find a route. So cyclists can be considered to be in one of two states when wishing to interact with the system, either wanting a route, or wanting to upload their data. The two pipelines reflect this with one being a GPX upload-only pipeline, and the other one is a visualisation pipeline that shows users what their mapped route's suitability looks like. A simple LAMP-like stack approach is a simple, but effective pattern that can implement this project. As of right now, this is a small scale experiment to prove the concept. If this application becomes popular, there will be several avenues to explore for scalability, such as deploying the system to expandable cloud services such as AWS or Azure. This is beyond the scope of the project however.

5.2.3 Design Patterns and Styles

Identify and describe any applied patterns (e.g., MVC, layered) and coding conventions.

5.3 Module-Level Design

??

5.3.1 Module Decomposition

List all system modules, their interfaces, and dependencies. Include a module dependency diagram.

5.3.2 Module Specifications

For each module:

Name: Purpose and description.

Interfaces: Inputs, outputs, and protocols.

Behavior: Algorithmic overview or pseudocode.

Dependencies: Internal and external module links.

5.4 Interface Design

5.4.1 User Interface

The user interface for the project scope will just be a basic file submission system, and an interactive mapping software that can output GeoJSON

5.4.2 Software Interfaces

The front end will send GeoJSON objects to the backend via POST. GPX files can also be sent via POST with `Content-Type:application/gpx+xml`. The back-end will also respond with the same GeoJSON objects back to the front-end, as well as sending segment list via POST. REST APIs are also used to interface with third party tools such as the OSRM route matching engine, and the Overpass API.

5.5 Data Design

5.5.1 Data Models

ERD showing data entities and relationships.

5.5.2 Database Schema

Detailed table definitions, keys, indexes, and normalization rules.

5.5.3 Data Dictionary

Definitions and formats for all data elements used in the system.

5.6 Standards and Compliance in Design

List all applicable ISO/IEC, IEEE, and domain-specific standards adhered to (e.g., ISO/IEC 27001 for security).

Chapter 6

Implementation

6.1 Component Breakdown

This section details the method and process of implementing each component of the system.

6.1.1 Input Pipeline

This was the first component written. It started very simple by prototyping isolating GPX files and obtaining their coordinate pairs in a list of duples. Initially, it was written as a single file parser, but was later modified to include batch processing.

1. Prototyping: Included learning about GPX file structure, and isolating data values in lists.
2. I created a Python module that pulled data from the Overpass API, and tested it by putting in some bounding box (bbox) coordinates.
3. I then set up the Python virtual environment, which would gather the required modules from `requirements.txt`.
4. The next logical step was to take the GPX file, and parse it for the coordinates, then extract the min and max latitude and longitudes. I then created a function that generated a bbox from the GPX file, and used that to get OSM data.
5. I researched OpenSourceRoutingMachine (OSRM), which was a framework that had several useful functions, one of which was match GPS points to an OSM network. Unfortunately, the files downloaded from Overpass were only of the .osm file type, which is essentially just XML. I needed to convert it to Protocolbuffer Binary Format (.pbf) for OSRM to be able to use it.
6. I did more research, and found that Osmium provided the tools necessary to convert to .pbf, as well as other tools such as map merging (which would prove useful later for batch processing). However, since I wanted to keep all the necessary environments enclosed within the pipeline directory, I couldn't use osmium directly in Python, since there was no proper module for it, but I decided instead to set up my own Osmium engine, by creating a docker image from Ubuntu, and using a Dockerfile to install and set up Osmium on the

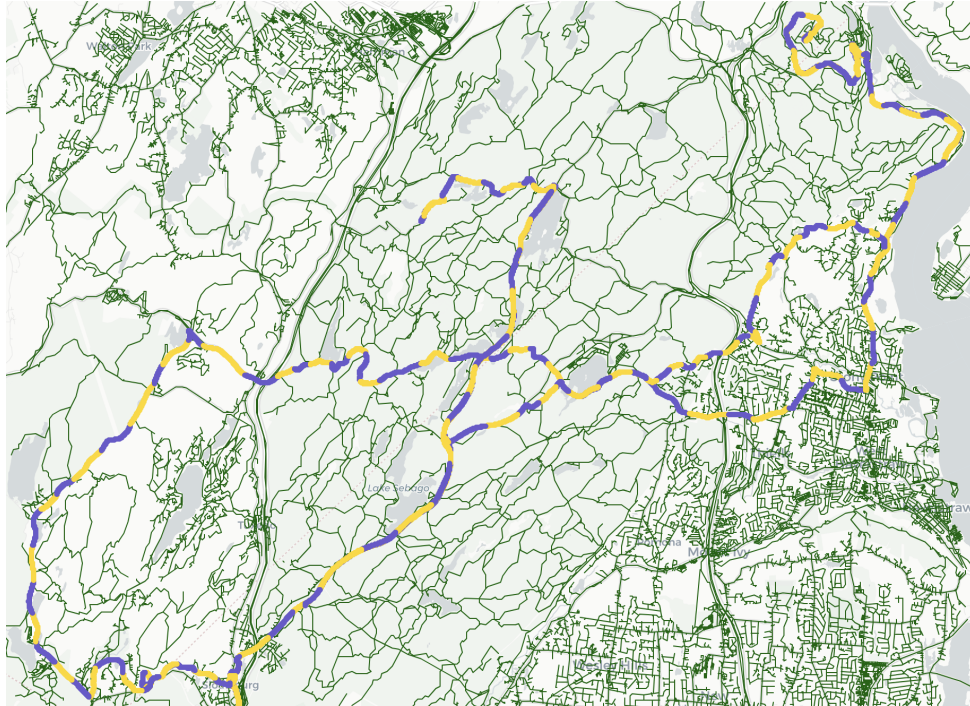


Figure 6.1: A visual representation of the Road network used for testing the pipeline. Green roads are the downloaded .pbf map edges, and the blue and yellow line is the series of GPX points plotted over the top to verify that the bounds had successfully covered the route

container. I then wrote a Docker Compose file that would set up the necessary volumes in the pipeline directory, and set the entrypoint to "osmium". I also wrote a few scripts that would merge and then convert every .osm file in the input directory.

7. The `run_pipeline.sh` bash script was the main file that orchestrated all of the input directions, such as creating and removing directories, and running Docker. I created the script around this point, after the osmium docker compose, so I could start automating the process as we went along.
8. After Osmium, I had successfully created a script that took GPX files and generated a map from them in .pbf format. At this point it was prudent of me to start actually doing formal testing, so I created a few test functions that would run in the background and collect variable information for me to see. I also created a test script that would convert the .pbf file into a viewable map, with the .gpx coordinates overlaid. ?? shows this map:
9. Now I started creating another Docker instance for OSRM. There exists a docker container image "osrm-backend", which I ran in detached mode, and assigned input and output volumes. In the backend, there are a few functions that were necessary to call before I started the match service. I first had to download a profile.lua file from OSRM's website. This contained all the settings and optimisations that I could change to adjust the matching algorithm. This worked by giving certain road types different speed limits. I adjusted them to roughly what I felt I could do on those types of roads on a bike. For example, on a road, I put 35 (units are Km/hr), on gravel I put 25, and I put even less on other types of roads that I felt were unattractive to cycle on. I set the weight parameter to "duration", which

essentially made it so the speed value became a weighting unit. The higher the number, the more likely the match engine would choose to go on that route. When testing, I ran into an issue where no matter what, I couldn't get the route to go the right way, but it turned out I actually didn't have the right road defined. The road turned out to be a "trunk" highway, when that type didn't exist by default in the profile.lua. This showed me that the file isn't perfect, and will require tweaking over time. However, after a few days of tweaking, I came up with a profile that seemed pretty good and could match 95% of my test routes with 99% or more accuracy. I feel this is highly individual, but in the future, it could be fun to try to automate the profile.lua values to hyper-optimize the matching algorithm.

10. After running the profile.lua through the OSRM engine, I then created another Python module called `batch_route_calc.py`. This file, as named, was the first attempt at batch processing files. However, this wasn't at the time because I needed to parse multiple GPX files. It actually came about because at this point I realized that matching an entire GPX file in one go is a bad idea. Some of my test files had over 10,000 GPX points, and needless to say, sending all 10,000 through a REST API is difficult at the best of times, but because OSRM is slightly outdated, the only way to process the map data is through GET requests to the `/match` endpoint, with the data in the URL itself. Sending 10,000 coordinates, each with between 15 and 20 characters would be impossible, so I had to come up with a solution.
11. My solution was two-fold. Firstly, I would break the GPX file into chunks. I went back to my first python script, and added a function that would take the gpx files, format into .json files of a smaller length, and batch process them with OSRM. I was successful in creating the "chunks" as I called them. But the issue I ran into was the fact that I had to still make over 100 GET requests, since the largest reasonable amount of data I could send in 1 go was 100 GPS coordinates.
12. I solved this problem by parallelizing the script. I used `ThreadPoolExecutor` to run multi-threaded batch processing on the GPX file. This sped things up significantly, and allowed me to process the GPX file in seconds rather than closer to a minute.
13. I started also parallelizing other processes, such as the API calls to Overpass, and the input/output file operations. These are the tasks that benefit the most from multithreading.
14. Next I had to merge the many route chunks that were generated from OSRM. I was able to optimize this to $O(\log n)$ with tree recursion, as well as multithreading within each tree level. To do this, I essentially just performed merge on every pair, and recursively called the merge function until the base case where there was only 1 value, in which case we were finished. This worked well, although I had a few instances where the recursion broke because the merge function wasn't working correctly, and I got infinite recursion, which is always fun.
15. After merging, I visualised the connected route with Folium, and compared the route to the GPX file. I overlaid both on the same map to check for issues. This is also where

I started manipulating the `profile.lua` file in OSRM directory to try and get matching as accurate as I could. After a few days I got it to the previously stated accuracy of roughly 99% for most of my files. The 5% of files that I could not get that accurate ended up being outliers anyway. Most of them either had large amounts of GPS noise, or were not on mapped roads, or were going through junctions or along bike paths that ran with the road I was on. All of these are not huge considerations however, as the main goal of the system isn't to accurately map routes, but to quantify training suitability. If there is a bike path nearby, then most of the time, that will end up being a little bit more suitable than the road, and if not, they're close enough together that anyone out on the road can choose which one they prefer.

16. The very last step in the pipeline is to enrich the route map with OSM metadata, such as road names, road width, road type, speed limit, etc. This was done with a python script called `enrich.py`. The module took the waypoints in the GeoJSON file, and mapped them onto OSM ways using OSMnx's nearest node function. Each node was assigned a "way", and a file was created that stores that way, along with the array of waypoints, in the order they occurred. This basically provides a mapping of GPS coordinates to OSM ways, and also provides the lowest level of granularity for our segmentation algorithm to work with.

Data Stream

The data going in was the GPX file, and the `settings.yml` that controlled parameters such as chunk size (of the gps chunks), and dynamic radius window, which determined how wide of a window to check for variance to adjust the match search radius.

6.1.2 Segmentation Engine

6.1.3 TSS Engine

6.2 Testing and Results

Chapter 7

Results and Conclusions

Chapter 8

Further Discussions and Research Gaps

Appendix A

Appendix

A.1 Further Reading

A.2 Source Code

A.3 File Structure

A.4 Additional Documentation