Exercise 2: Wearable Activity Monitoring

Have you ever wondered how smartphones and smartwatches can monitor your physical activity? In this exercise, it's time to step up your game and become the R&D expert yourself! Using 676 collected recordings from smartphones and smartwatches of students commuting between Zurich main station and ETH main building, your task is to infer the activities performed, the number of steps taken, the route followed, and the location of the device on the body!

The exercise is designed to allow you to gain practical skills and learn to work with real-world data. Throughout the project, you will develop processing techniques for motion signals, evaluate them, and analyze their results.

1 Task description

For this exercise, you will be working on a crowd-sourced activity dataset gathered along five distinct paths connecting Zurich main station and ETH main building.

During the recording process, participants wore a LilyGo smartwatch at one of three designated body locations: left wrist, fastened to a belt, or right ankle. The smartwatch streamed the signals captured by its accelerometer, gyroscope and magnetometer to the participant's smartphone. The smartphone then stored this data alongside additional telemetry data from its sensor suite, encompassing GPS, motion sensors, and temperature readings. Participants covered each route by walking, running, cycling, and/or interspersed periods of standing still, ensuring each activity lasted for at least one minute.

Your task is to develop an algorithm that can infer the performed activities, the covered path, the number of steps taken, as well as the location of the smartwatch on the body based on the **available and accepted** input signals (see Section 5) captured throughout the duration of a single path.

More formally, your algorithm should predict the following:

- Step count $s \in \mathbb{N}_0$
- Smartwatch location $l \in \{0, 1, 2\}$, (0: left wrist, 1: belt/waistband, 2: right ankle)
- Activities performed for more than 60 s uninterrupted throughout the recording of a data trace ("standing": True/False, "walking": True/False, "running": True/False, "cycling": True/False).
- Path index $p \in \{0, 1, 2, 3, 4\}$

In addition to taking part in the Kaggle competition and submitting your algorithms in the form of Python code, provide brief documentation about how you tackled this problem. Please refer to the subsequent sections for detailed instructions and specific requirements.

1.1 Step Counting

Your algorithm should infer the number of steps ($s \in \mathbb{N}_0$) taken over the completion of a path. Your algorithm should be as accurate and robust as possible.

Please note:

- Only a fraction of the available recordings in the dataset contain a valid entry for the ground-truth step count. The step count had to be manually tracked by the participant using a hand counter. Thus, to ensure data quality, the respective recordings were all conducted by members of the TA team. Your step count algorithm will also **only** be tested on traces recorded by the TA team. Some recordings may contain an additional entry for *phone_steps*, which, if available, has been automatically inferred by the smartphone. It may vary in accuracy, and thus should be used with caution. *phone_steps* **must not be used as input** for your step count algorithm but could serve as additional "low-confidence" labels for development purposes.
- We define a step as the action of lifting and placing one's foot, which contributes to the movement of the entire body. Activities such as walking stairs or running are counted as steps, while activities like cycling, sitting, or standing still do not contribute to the step count. Your algorithms will be assessed based on natural body movements only (i.e., we have not recorded data involving dragging the feet along the floor to move without generating step events or similar actions).
- Participants were instructed to wear the LilyGo smartwatch at one of the three designated body locations throughout the whole recording period for one path. No guidance was given regarding the placement of the smartphone, which could vary during the recording (e.g., pocket, hand, etc.). Consequently, exploring the acceleration signals of the smartwatch may be a sensible starting point for data analysis.
- Step counting is a problem for which several algorithms have been proposed. You are free to build on existing algorithms or Python modules, but it may be necessary to optimize them for this specific context.

- The following paper may serve as one possible starting point: Walk Detection and Step Counting on Unconstrained Smartphones by Brajdic et al.. Note that the algorithms they propose are designed for use in a smartphone rather than a smartwatch, but they still offer insight into possible approaches. Google Scholar is useful to find additional related publications on the topic. Starting with Brajdic et al.'s paper, you can find related work in the articles they cited as well as by using 'cited by' in Google Scholar.

1.2 Smartwatch Location

For each recording, you algorithm should classify the on-body location of the smartwatch. Participants were instructed to select one of the following three placements (see the 'LilyGoLocations.pdf' file for illustrations):

- **Left wrist (0):** The smartwatch was worn similar to a regular watch around the left wrist. The display faced towards the participant when looking at the back of their hand. Both possible rotations were allowed: touch button aligned with thumb *or* pinky.
- **Belt/waistband (1):** The smartwatch was strapped on the participant's belt or a belt loop, so that it was centered in front of their body. The orientation of the smartwatch was not specified.
- **Right ankle (2):** The smartwatch was positioned around the lower right leg, slightly above the right ankle (approximately where the leg is slimmest). If the leg was too large to accommodate the watch strap, participants had the option to detach the watch body from its wristband and secure it in place using a sock, tape, or alternative strap. The watch was worn on the inner side of the ankle, with the display facing towards the left foot.

Compared to step counting, on-body location classification is a less commonly solved problem and you may find it more difficult to find related work. When looking for related work on this topic, you can start with Where am i: Recognizing on-body positions of wearable sensors by Kunze et al. and proceed with Google Scholar as described above. Note that it will be challenging to find a pre-existing algorithm tailored precisely to the requirements of this subtask, but related work may still serve as inspiration (e.g., features to extract).

1.3 Activity Recognition

Activity monitoring can offer valuable insights into health and fitness. Your algorithm should recognize the activities performed during the completion of a path, which are limited to standing still (0), walking (1), running (2), and cycling (3).

Please note:

- Your algorithm should output a boolean value (True or False) for each activity—standing, walking, running, and cycling—indicating whether the respective activity was performed uninterrupted for more than 60 seconds at any point during the recording.
- Every sample in the training set comes with a label indicating which activities were carried out during the recording. However, these labels are considered "weak" because they only tell

us that certain activities occurred, not how long they lasted or exactly when they happened. Participants were asked to note every activity they performed (0 for standing still, 1 for walking, 2 for running, and 3 for cycling). They needed to list each activity at least once but were not required to put them in any specific order. Some participants might have also included the order in which they did these activities, but we cannot guarantee the accuracy of this additional detail. Part of the challenge of this exercise is to handle this uncertainty, and it is ultimately up to you to come up with an appropriate strategy or method to enable the development of a suitable algorithm.

- To avoid edge cases in activity recognition, participants were asked to adhere to the following rules when recording and labeling data:
  - Activities must be performed for more than 60 seconds uninterrupted to be valid. Participants were especially instructed to avoid any durations that lasted less than 60 seconds but more than 30 seconds.
  - Participants should not walk or run in place or circles.
  - Walking/running stairs is considered as a walking or running activity, respectively.
  - To differentiate between walking/running and standing still, participants should minimize steps during a phase of standing still. A period of less than 8 seconds of standing still between two steps may still be considered as uninterrupted walking or running, and a period of standing still is still considered if the number of steps within a minute is less than 7.
  - The difference between walking and running is that running is characterized by an aerial phase in which all feet are above the ground. This is in contrast to walking, where one foot is always in contact with the ground. Participants were instructed to minimize walking or standing still during a phase of running. Up to 8 seconds of standing still or walking between running intervals may still be counted as uninterrupted running.
- Activity recognition is a core challenge in the literature and many approaches have been published. You may want to take a look at Activity recognition from accelerometer data by Ravi et al. to start your research. You may also take a look at existing Python packages that may help you to solve this problem (see Subsection 4.1)

1.4 Path Classification

For each recording, participants completed one of five possible routes between Zurich main station and the ETH main building. These paths vary in terrain, direction, and elevation, with three paths leading uphill and two paths leading downhill. Please refer to Section 3 for the label and description of each path.

Your algorithm should correctly assign the corresponding path index ($p \in \{0, 1, 2, 3, 4\}$) to each datastream.

1.5 Documentation

In addition to developing functional code, please prepare a short documentation of your algorithm. Your write-up should not exceed 1000 words and should span no more than three pages

(A4, one-sided, 2 cm margins, 10 pt minimum font size) including graphics and plots. Please describe and discuss your approach, implementation, and rationale behind your design decisions.

2 Evaluation

To be evaluated, you will have to take part in the Kaggle competition and upload your algorithm and report in a single .zip folder to Polybox as described below. You'll submit your code in a Jupyter notebook file that should follow the submission template notebook available via Kaggle. Please read the below carefully.

- Take part in the Mobile Health 2024 - Wearable Activity Monitoring Kaggle challenge. To test your code before the deadline, you can upload a .csv file with your predictions for the test data here on Kaggle. The score you receive on the public leaderboard will be based on a randomly selected 20% of the test recordings. You will only see your results on the remaining 80% of the test data after the submission deadline. Your performance on this 80% will contribute towards your grade, while performance on the public 20% will not. Out of all your submissions, select the one you want to be graded via the corresponding checkbox in your submission history on Kaggle. Name this submission file 'groupXX_submission.csv' and upload it together with the files described below.
- If your algorithm is executed on Kaggle on a CPU it must yield a prediction within 10 seconds for each individual .pkl file. This time limit applies to the inference phase of your algorithm, which may include loading potentially pre-computed features (see below). If your code fails to compute the required output within the allotted time, the output for the corresponding recording will be considered incorrect.
- Hand in your algorithm as a Jupyter notebook file (.ipynb) that follows the submission template on Kaggle. Include your group number, Kaggle team name, and information about each team member (ETH email, legi number, Kaggle username) at the top of the source code. Update the filename to 'groupXX_activity.ipynb', where XX is your group number in the sign-up sheet. Use a leading zero if it is a single digit.
- (Optional:) Submit the models you have trained and saved so they can be easily loaded into your notebook 'groupXX_activity.ipynb'. Name these models 'groupXX_model_i', where i is an identifier in case you upload multiple models. Use the recommended method for saving models according to the library you are using (e.g., joblib for sklearn). If you upload saved model(s), also provide a notebook named 'groupXX_training.ipynb' that conducts the training and saving process of your model(s).
- (Optional:) You can upload a file containing previously computed features for the test recordings, which you simply load during inference. Name this file 'groupXX_features.xyz' (xyz corresponding to the file format you choose). You can load and use these features as part of your 'groupXX_activity.ipynb' notebook. In case you upload a 'groupXX_features.xyz' file, you must still upload the code used to compute the features! Do this within the 'groupXX_training.ipynb' file.
- Submit the documentation as a PDF file, naming it 'groupXX_documentation.pdf'

Upload all files in a .zip folder to this Polybox. You may upload multiple times until the deadline. The last uploaded version of the folder will be graded.

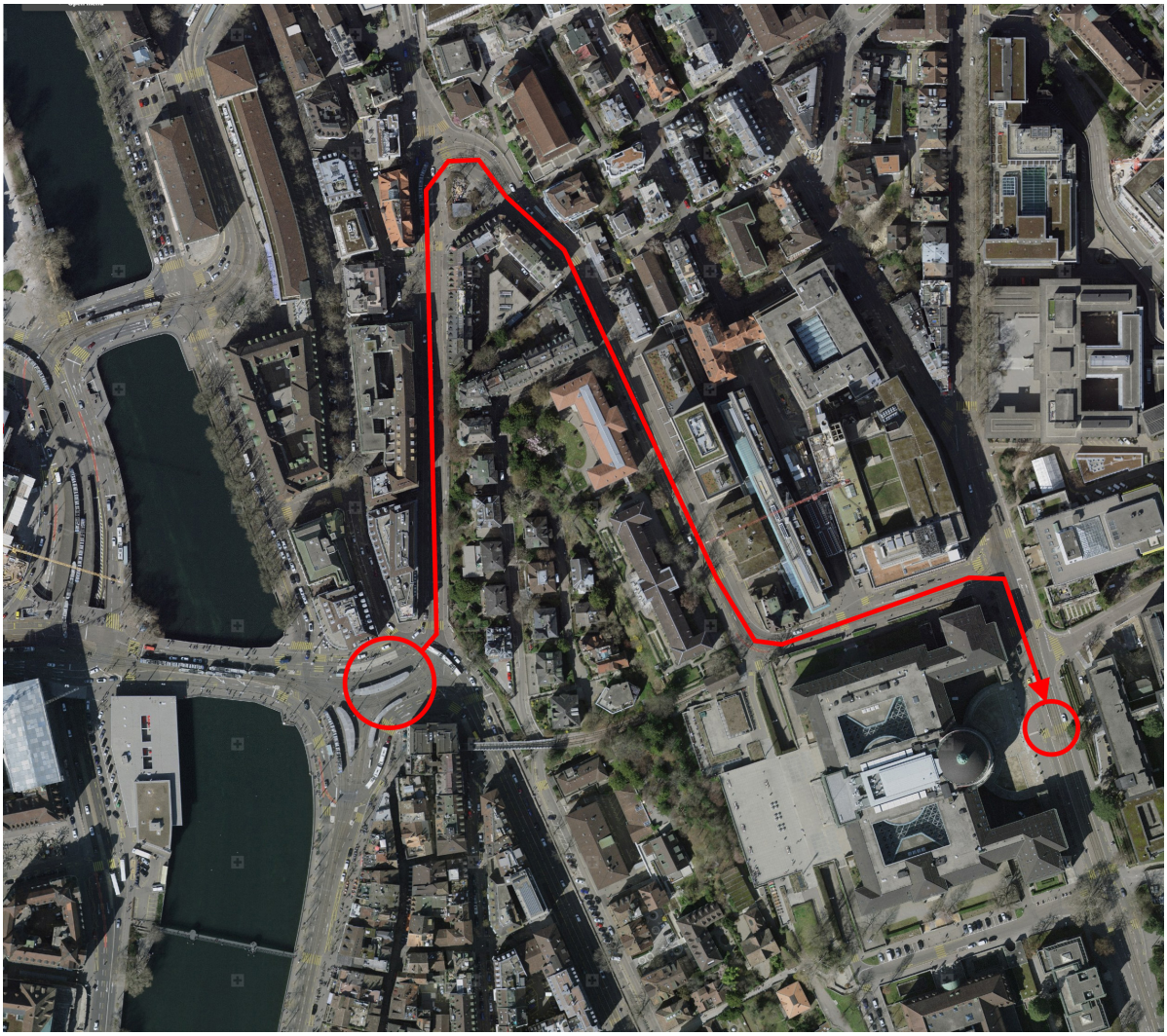
Point assignment

This exercise is worth 70 points of a total of 100 points achievable (without bonus points) for both exercises. Including 8 possible bonus points, the maximum achievable score for a team is 78 points for this exercise. No bonus points are necessary to achieve the highest grade of 6.0 for the exercise. All members of a team will receive the same grade for the exercise.

- achievable points: 70 pts
- 60 pts awarded based on algorithm performances on private leaderboard (see Section 2)
- 10 pts awarded for short documentation (1000 words upper limit)
- For each of the four categories (smartwatch location, path index, performed activity, step count) separately: 2 and 1 bonus points will be awarded to the two teams with the top algorithm performances on the private leaderboard.

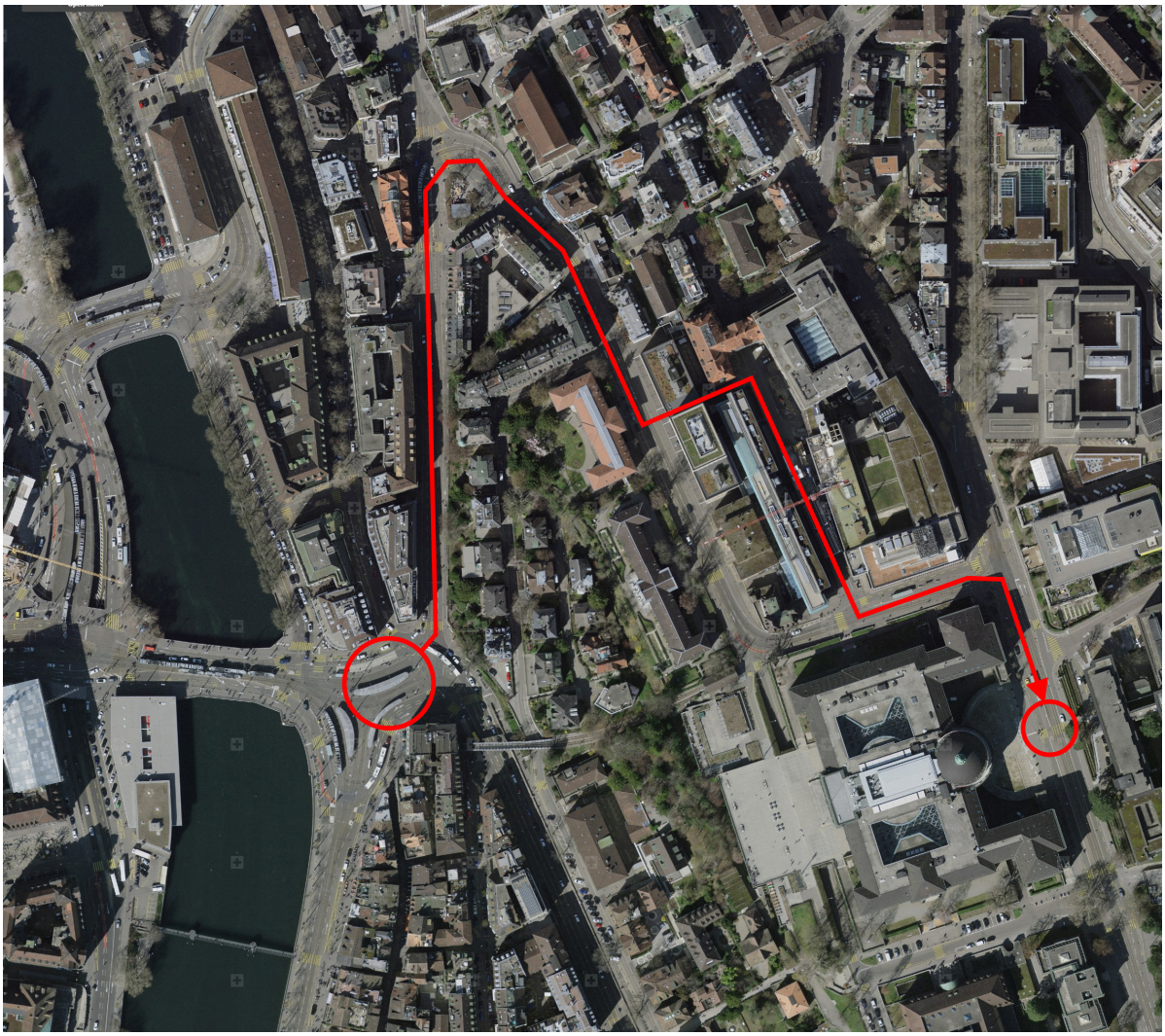
3 Zurich Paths Documentation

3.1 Path 0: Central to Main Building



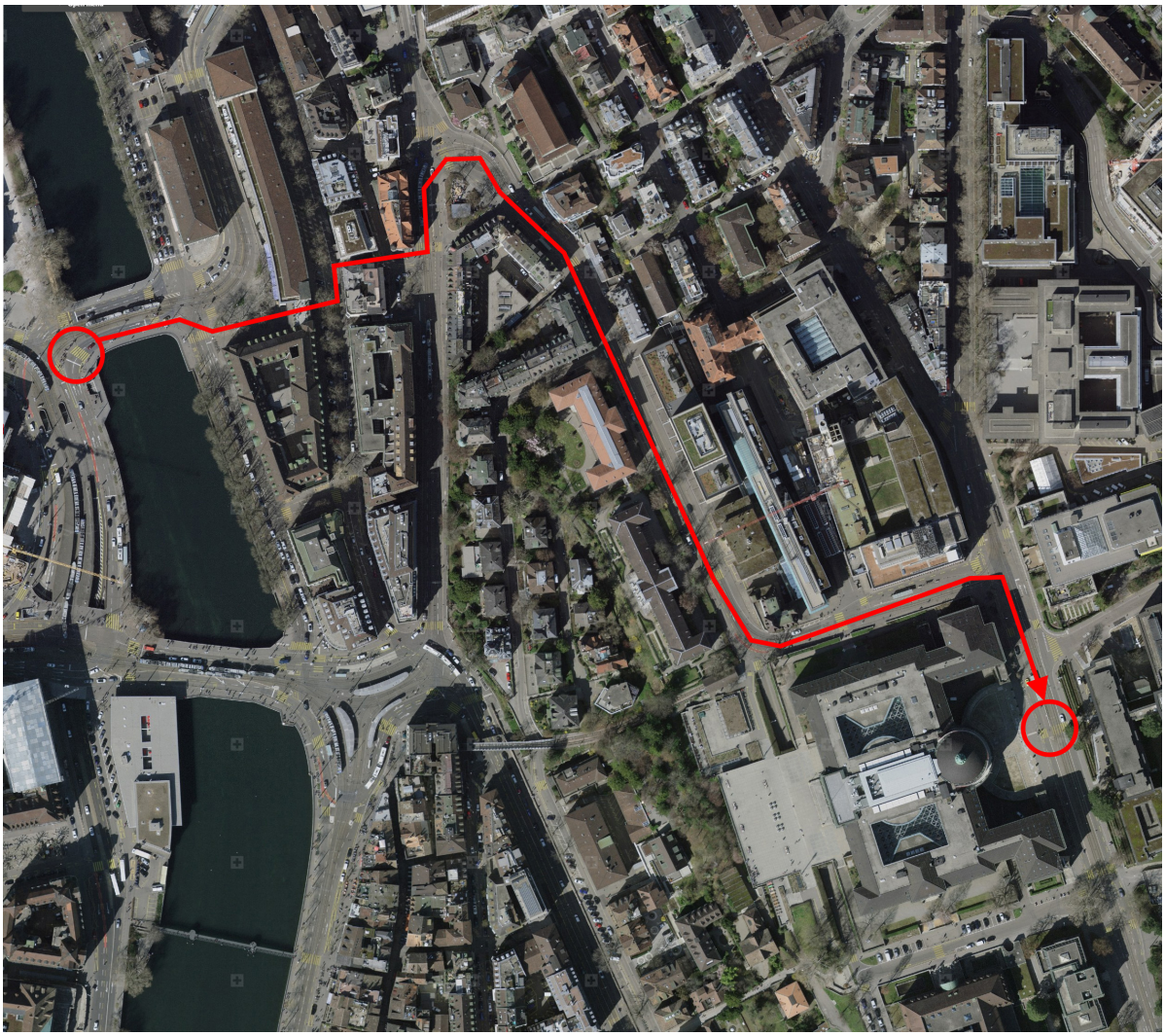
- **Start:** Central ($r_{tolerance} = 25m$)
- **End:** Main building, center in front on Rämistrasse side ($r_{tolerance} = 15m$)
- **Route:** Weinbergstrasse – Leonhardstrasse – Rämistrasse
- **Distance:** 830 m
- **Elevation gain:** 44 m
- **Special notes/constraints:**
  - Do not take the shortcut to cut the corner from Weinbergstrasse to Leonhardstrasse.

3.2 Path 1: Central to Main Building, Clausiusstrasse Variant



- **Start:** Central ($r_{tolerance} = 25m$)
- **End:** Main building, center in front on Rämistrasse side ($r_{tolerance} = 15m$)
- **Route:** Weinbergstrasse – Leonhardstrasse – Clausiusstrasse – Rämistrasse
- **Distance:** 840 m
- **Elevation gain:** 44 m
- **Special notes/constraints:**
  - Do not take the shortcut to cut the corner from Weinbergstrasse to Leonhardstrasse.

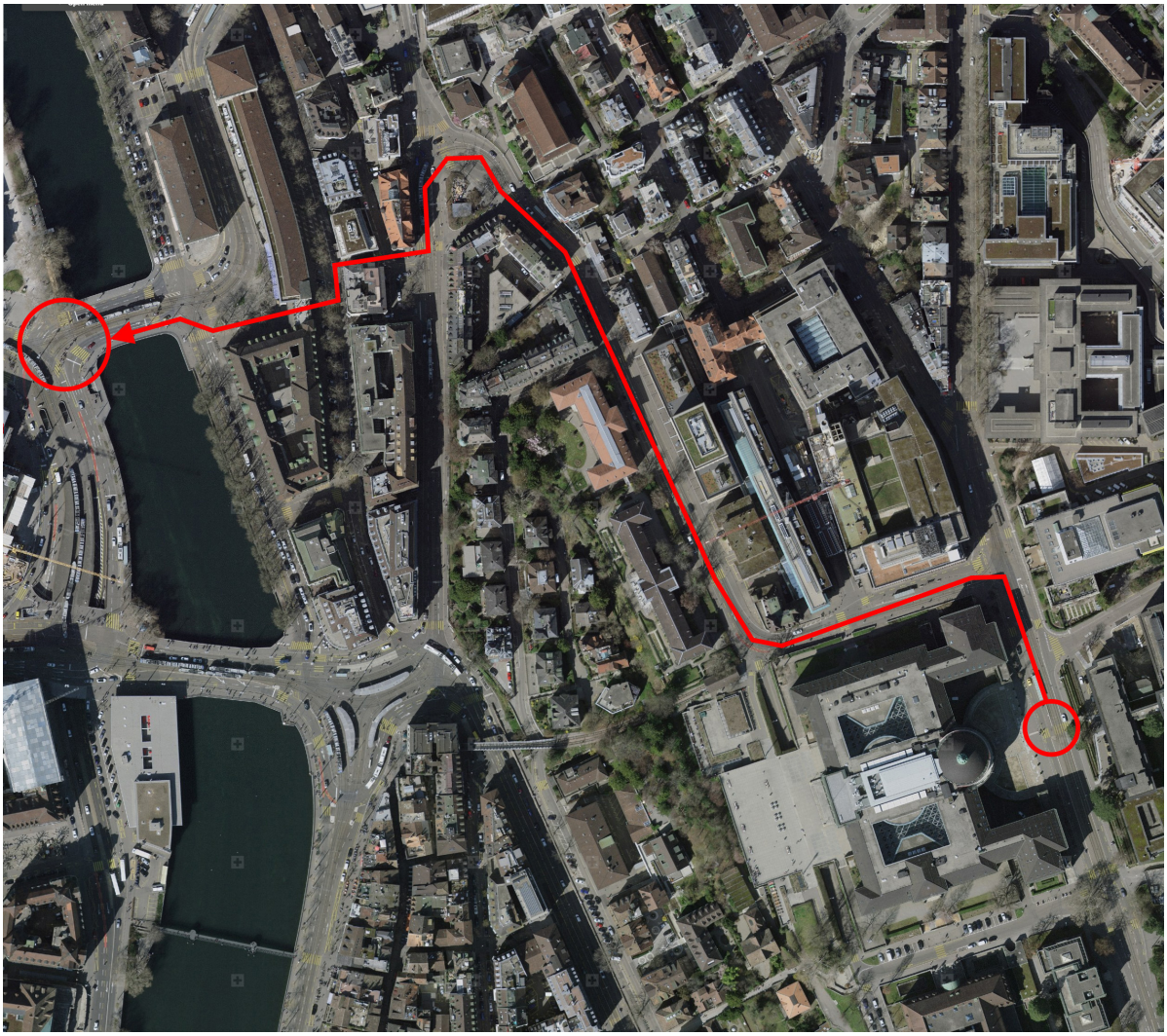
3.3 Path 2: Walchebrücke to Main Building



- **Start:** Walchenbrücke, east edge ($r_{tolerance} = 15m$)
- **End:** Main building, center in front on Rämistrasse side ($r_{tolerance} = 15m$)
- **Route:** Walchebrücke – Walchetor (pedestrian passage with a short staircase) – across Stampfenbachstrasse, up the stairs to Leonhardstrasse – Rämistrasse
- **Distance:** 830 m
- **Elevation gain:** 47 m
- **Special notes/constraints:**
  - Do not take the shortcut to cut the corner from Weinbergstrasse to Leonhardstrasse.

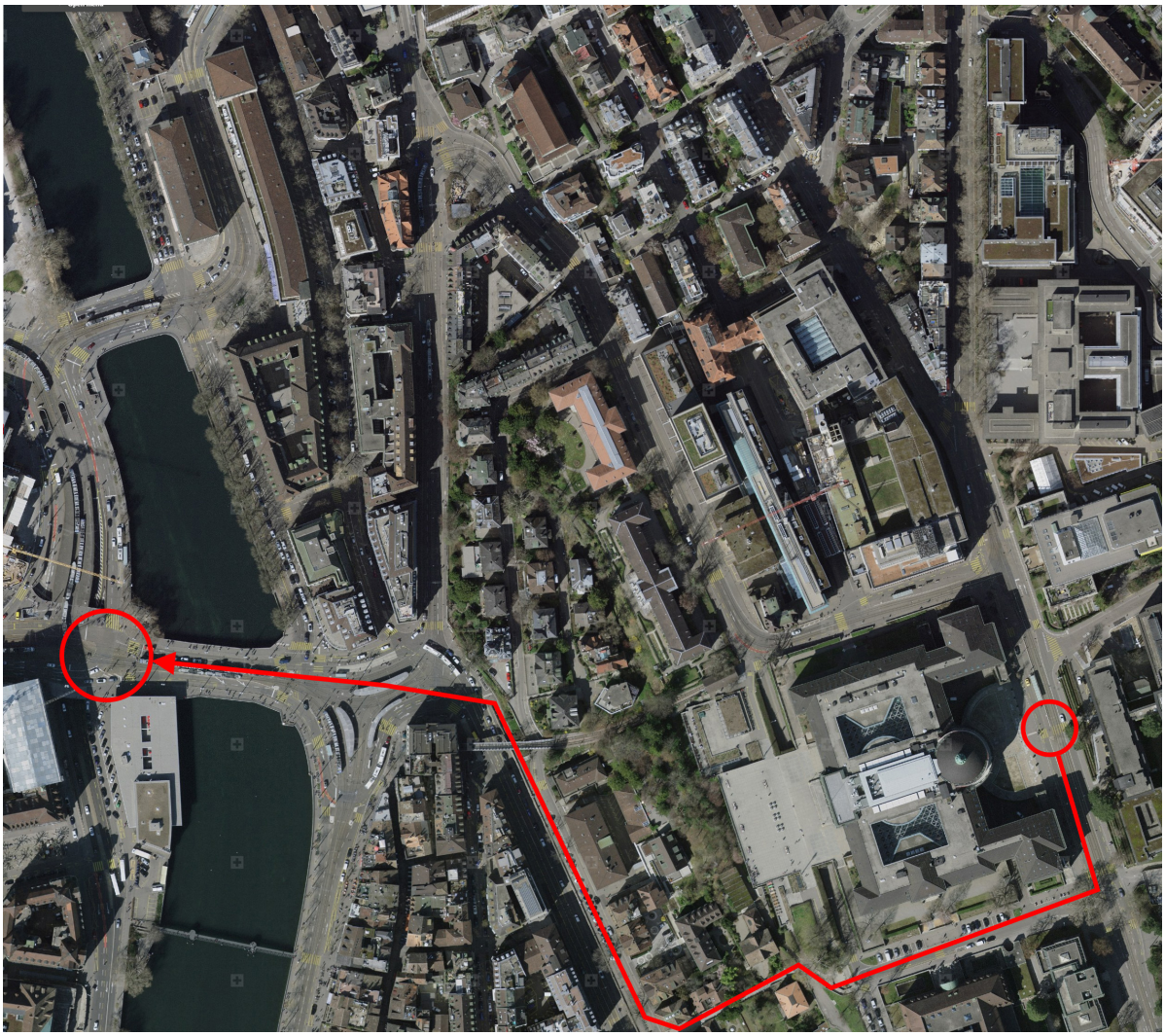
3.4 Path 3: Main Building to Walchebrücke

This path is the exact reverse of path 2.



- **Start:** Main building, center in front on Rämistrasse side ($r_{tolerance} = 15m$)
- **End:** Walchenbrücke, east edge ($r_{tolerance} = 25m$)
- **Route:** Rämistrasse – Leonhardstrasse – stairs down to Stampfenbachstrasse – Walchetor (pedestrian passage with a short staircase) – Walchebrücke
- **Distance:** 830 m
- **Elevation gain:** -47 m
- **Special notes/constraints:**
  - Do not take the shortcut to cut the corner from Leonhardstrasse to Weinbergstrasse.

3.5 Path 4: Main Building to Bahnhofquai



- **Start:** Main building, center in front on Rämistrasse side ($r_{tolerance} = 15m$)
- **End:** Junction Bahnhofquai, Bahnhofbrücke ($r_{tolerance} = 25m$)
- **Route:** Rämistrasse – Karl-Schmid-Strasse – Schienhutgasse – Hirschengraben – Central – Bahnhofbrücke
- **Distance:** 780 m
- **Elevation gain:** -45 m
- **Special notes/constraints:**
  - Make sure to use the Hirschengraben street and not Seilergraben street before Central (they run in parallel, Hirschengraben is elevated above Seilergraben)

4 Python Specifications

If you are not familiar with clean, portable Python package management, you may take this project as a chance to learn it or use Kaggle to code. Since we have to test your algorithms it is essential to the exercise that you work in an appropriate Python environment. If you choose to work locally, please follow the guidelines below.

4.1 External Code and Algorithms

You may use any Python package and algorithm that is included in the 'packagesCPU.txt' or 'packagesGPU.txt files' uploaded on Moodle (i.e., any package that you can use without installing in on Kaggle). Ultimately, it is your responsibility to ensure that your code is Kaggle-compatible and produces the outputs you intend it to when executed in a notebook that you created in this exercise's Kaggle competition without throwing any errors. If you use algorithms that are not completely your own development, you must label your source with a comment in the code including a link to the resource. Collaboration across groups is not permitted. This includes sharing code, results, data or anything that would represent an advantage to other groups.

4.2 Setup to code locally

You will need a way to create virtual Python environments. Think about them as independent Python installations you can easily create, switch between, and transfer to other machines. There are multiple tools to do this, we recommend Anaconda which is available for Windows, Linux, and macOS. We created a small sufficient environment with some basic packages that can be created using the 'mhealth24.yml' file. If you want to add packages, check the 'packagesCPU.txt' or 'packagesGPU.txt' files (depending if you're only using CPU or also GPU) to check if the package is Kaggle-compatible and what version is installed on Kaggle. Especially when using a Mac, there might be some unavoidable clashes depending on what packages you intend to use. When coding locally, the responsibility to ensure that the code is Kaggle-compatible ultimately lies with you. The Kaggle kernel is rather complex and includes hundreds of different packages. You'll likely only ever use a small fraction of those. Still, there might be unavoidable clashes. In case you're struggling to install the exact package versions installed on Kaggle that's unlikely to be a huge issue. However, please ensure that the Code runs as intended on Kaggle regularly to avoid surprises on deadline-day.

4.2.1 Anaconda

Setting things up and basic usage:

1. Install Anaconda
2. Place the mhealth24.yml file in the folder where you want to create your virtual environment
3. Open Anaconda prompt, type `conda env create -f mhealth24.yml` to create an environment named mhealth24 with Python 3.10.13 (this may take a while)

4. Activate the environment: `conda activate mhealth24`
5. To run e.g., jupyterlab, just type the command in the anaconda prompt while the desired environment is active: `jupyter lab` (depending on what version of Anaconda you installed, you might first have to run `conda install -c conda-forge jupyterlab`)
6. If you want to use interactive plots in your notebooks, you might have to use jupyter notebook (run: `jupyter notebook`) due to issues with jupyterlab and the version of ipympl installed on Kaggle
7. Happy coding!

4.2.2 Jupyter Lab

We recommend using JupyterLab (see here). In this environment, it is easy to visualize data and test algorithms. To submit your project, copy your algorithms into the provided .ipynb template files. Please test these files before submitting.

4.2.3 Interactive Data Visualization

Data visualization is an important aspect of data analysis. We recommend that you thoroughly visualize and inspect the data before implementing any algorithms.

Especially with long data traces, interactive plots that allow responsive zooming can be very helpful. We recommend using Matplotlib in a Jupyter notebook with the interactive features provided through ipympl.

This sample code will provide you with an interactive plot with two subplots on a shared x-axis. You may use this as a starting point for data visualization.

```
import matplotlib.pyplot as plt
%matplotlib widget
fig,ax=plt.subplots(nrows=2,ncols=1,figsize=(10, 6), sharex=True)

# plotting two lines in the top subplot
ax[0].plot(xvals, yvals, label="somelabel")
ax[0].plot(xvals2, yvals2, label="somelabel2")

# plotting a line in the bottom subplot
ax[1].plot(xvals3, yvals3, label="somelabel3")

ax[0].legend(loc='lower right')

# you can plot to the other subplot using this pattern
#ax[1].plot(...)

#programmatic zoom options
#ax[0].set_xlim([123, 543])
#ax[0].set_ylim([9000000, 9500000])

plt.tight_layout()
plt.show()
```

5 Available Sensor Data

The following is a list of available sensor traces when loading a recording in Python. The name can be used as dictionary key. **Bold** sensor names will always be contained when testing algorithms. All other sensors may or may not be included depending on the smartphone that was used for the recording. You may use them in your algorithm but it is not guaranteed that the data will be included in the test set. Please note that *latitude*, *longitude*, *bearing*, *speed*, and *phone_steps* data must not be used as input for your inference algorithm.

Meta data

- **timestamp**: UNIX timestamp in ms when packet was received
- **packetNumber**: Sent with BLE packet from wristband, should count up 1 per packet and wrap around 1 byte
- lostPackets: Number of lost packets since last received packet
- note: String of user note during recording

Smartwatch data

- **ax**: Wristband accelerometer X-axis [g]
- **ay**: Wristband accelerometer Y-axis [g]
- **az**: Wristband accelerometer Z-axis [g]
- **gx**: Wristband gyroscope X-axis [deg/s]
- **gy**: Wristband gyroscope Y-axis [deg/s]
- **gz**: Wristband gyroscope Z-axis [deg/s]
- **mx**: Wristband magnetometer X-axis [μT]
- **my**: Wristband magnetometer Y-axis [μT]
- **mz**: Wristband magnetometer Z-axis [μT]
- **temperature**: Wristband IMU die temperature [°C]

Phone data

- **longitude**[1]: Phone GPS longitude [degrees]
- **latitude**[1]: Phone GPS latitude [degrees]
- **altitude**: Phone GPS altitude [m]. *Please note that altitude measurements might be sparse. They should thus be treated with caution. Despite changes in altitude, smartphones might just duplicate a previous measurement.*
- bearing[1]: Phone GPS bearing [degrees]
- speed[1]: Phone GPS speed [m/s]
- **phone_ax**: Phone accelerometer X-axis [m/s^2]
- **phone_ay**: Phone accelerometer Y-axis [m/s^2]
- **phone_az**: Phone accelerometer Z-axis [m/s^2]
- **phone_gx**: Phone gyroscope X-axis [rad/s]
- **phone_gy**: Phone gyroscope Y-axis [rad/s]
- **phone_gz**: Phone gyroscope Z-axis [rad/s]
- **phone_mx**: Phone magnetometer X-axis [μT]
- **phone_my**: Phone magnetometer Y-axis [μT]
- **phone_mz**: Phone magnetometer Z-axis [μT]
- phone_gravx: Phone gravity X-axis [m/s^2]

[1]Must not be used in classification

- phone_gravy: Phone gravity Y-axis $[m/s^2]$
- phone_gravz: Phone gravity Z-axis $[m/s^2]$
- phone_lax: Phone linear acceleration X-axis $[m/s^2]$
- phone_lay: Phone linear acceleration Y-axis $[m/s^2]$
- phone_laz: Phone linear acceleration Z-axis $[m/s^2]$
- phone_rotx: Phone rotation vector X-axis
- phone_roty: Phone rotation vector Y-axis
- phone_rotz: Phone rotation vector Z-axis
- phone_rotm: Phone rotation vector $\cos(\theta/2)$
- phone_magrotx: Phone geomagnetic rotation vector X-axis
- phone_magroty: Phone geomagnetic rotation vector Y-axis
- phone_magrotz: Phone geomagnetic rotation vector Z-axis
- phone_orientationx: Phone orientation azimuth [degrees]
- phone_orientationy: Phone orientation pitch [degrees]
- phone_orientationz: Phone orientation roll [degrees]
- phone_steps[1]: Phone step counter (might not be accurate)
- phone_temp: Phone temperature sensor [°C]
- phone_light: Phone light sensor [lx]
- phone_pressure: Phone pressure sensor [hPa or mbar]
- phone_humidity: Phone humidity sensor [%]

Find more info about the Android sensors here:

- sensors overview
- position sensors
- location
- environment sensors

6 Any Other Questions?

Please ask any questions regarding this work in the Moodle forum for this exercise. Make sure to not give away parts of your solution when asking a question. If you have a question that does include parts of a solution or include specific ideas on how to solve a task, you may ask through e-mail. Send your e-mail to max.moebus@inf.ethz.ch, the subject must start with *[mhealth24 exercise]*.

7 Changelog

Here we will keep a changelog of this document. Corrections may be released during the exercise. Please check Moodle to stay up-to-date.

2024-04-09: original release

2024-04-15: longitude and altitude confusion in Section 5

2024-04-17: computational limit in Section 2