Software Documentation

Milena Bajic, Postdoc at DTU Compute
Tommy Sonne Alstrøm, Assoc. professor at DTU Compute

March 21, 2022
Version 0.9

Contents

1 Introduction

This report describes the software developed for road condition estimation using in-vehicle car sensor data. The software in employed in processing and analyses of data collected using Green Mobility (GM)[**GM**] and Danish Road Directorate (DRD) vehicles [**drd**].

The software is utilized in obtaining International Road Roughness index (IRI), Key Performance Index (KPI) and Damage Index (DI), using the machine learning pipeline, illustrated in Figure 1.

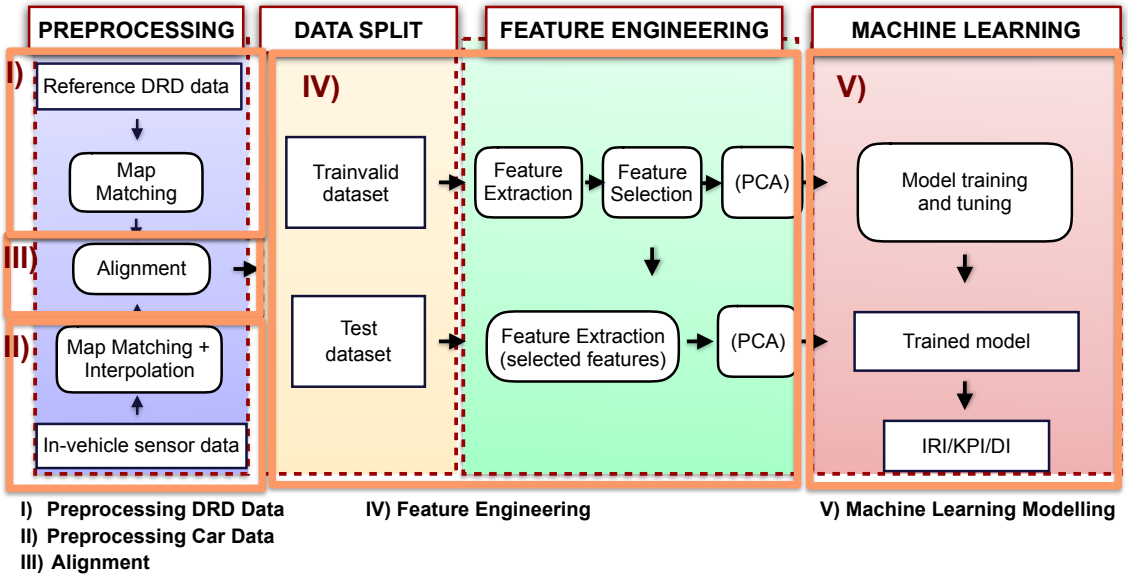


Figure 1: Framework for road condition estimation using machine learning. Illustrated are preprocessing, data split, feature engineering and machine learning phase, split into five processing units, each implemented in the corresponding software package. The preprocessing phase is implemented in `Preprocessing DRD Data`, `Preprocessing Car Data` and `Alignement` packages; the data split and feature engineering in `Feature Engineering` and machine learning in `Machine Learning Modelling` package.

The pipeline consists of four components: (i) preprocessing, (ii) data split, (iii) feature engineering, and (iv) supervised machine learning, implemented in five different packages. The packages are available at: lab.compute.dtu.dk/lira, specifically at (click for links):

- (I) `Preprocessing DRD Data`
- (II) `Preprocessing Car Data`
- (III) `Alignment`
- (IV) `Feature Engineering`
- (V) `Machine Learning Modelling`

The preprocessing phase involves map matching of reference measurements, implemented in the `Preprocessing DRD Data` package and separately in-vehicle sensor recordings which are further split into passes and interpolated to assign GPS coordinates to all

sensor readings, implemented in `Preprocessing Car Data` package. Further, reference values are defined for roads segments of fixed length and the map-matched in-vehicle dataset is aligned with the start and end of each segment, as implemented in `Alignement` package.

In the following phase, the aligned dataset is split into a 'trainvalid dataset' – utilized to selected an optimal subset of relevant ML input features and 'test dataset' – utilized for estimation of the generalization performance of machine learning models. An extensive set of features is defined and extracted from each aligned segment. An optimal subset of relevant features is selected using the Sequential Feature Selection (SFS) algorithm [**aha1996comparative**], applied on the 'trainvalid dataset' while the 'test dataset' is filtered to contain only the selected features. This phase is implemented in `Feature Engineering` package.

Lastly, the machine learning phase deals with the training and tuning of the set of models on the 'trainvalid dataset', and the application of the trained models to the 'test dataset' for road condition prediction. The machine learning phase is implemented in the `Machine Learning Modelling` package.

The prerequisites and the setup required to run the packages in described in Section 2, followed by the description of individual packages, with run examples to facilitate further usage. Overall, the results obtained using the developed software pipeline provide confidence in the proposed methodology, and demonstrate that the described approach is suited to meet road condition evaluation needs.

2 Prerequisites and Setup

2.1 General Setup

Two prerequisites for executing the software are the `Python` version and the installation of package dependencies. It is required to `Python` version $\geq$ 3.8. The development and testing was done using `Python 3.8.5` version on `Linux` with `GCC 7.5.0` and Python `3.8.11` on `macOS` with `Clang 10.0.0` compiler.

The package dependencies are listed in `requirements.txt` file, available for each package in its home directory. The dependencies can be installed from package home directory using:

```
cd <pkg_dir> (change to the package home directory)
pip install -r /path/to/requirements.txt  (install dependencies)
```

Finally, each package can be executed from its home directory using the main package script with user selected arguments.

It is strongly advisable to set up a virtual environment with Python version $\geq$ 3.8 and with installed dependencies. The software was developed and tested using the `pyenv` Python Version Management tool[**pyenv**].

If instead, the code is executed using a Docker image, there is no requirement for Python version setup nor for installation of dependencies. Instead, the code can be executing using the following command:

```
docker run -v $(pwd):/<docker_img_name> <docker_cont_name>
python3 -u <pkg_script>
```

where <docker_img_name> is the Docker image name, <docker_cont_name> is the Docker container name and <pkg_script> is the main package script with user selected arguments.

2.2 Database Connection

The database connection information is accessible through an additional `json` package, available here (click for the link):

- `json`

The database connection details are stored in a hidden `json/.connection.json` file.

There are two packages which require database connection, namely the `Preprocessing DRD Data` and `Preprocessing Car Data` packages. Hence if using those packages, it is required to check out the `json` package. It is advisable to place the `json` package on the same level with the referred packages - in this case, the json file will be automatically found. Alternatively, the user can provide the absolute path to the connection file using the script argument --conn_file.

The `json/.connection.json` file structure is as following:

```
"prod": {"database": "postgres",
         "user":guest,
         "password":<user_password>, (add guest password)
         "host":"liradb.compute.dtu.dk",
         "port":"5435"},

"dev" : {"database":"postgres",
         "user":guest,
         "password":<user_password>,
         "host":"liradbdev.compute.dtu.dk",
         "port":"5432"},

"osrm": {"host":"http://liradbdev.compute.dtu.dk:5000"}
```

The `"prod" and "dev"` refer to production and development database, while `"osrm"` refers to the `OSRM` service host. The database user and password refer to the credentials obtained by the user administrator while the database name, host and port refer to the the specific database.

In addition, if the packages requiring the access to the database are run outside of the DTU intranet, the user needs to set up a VPN connection. The VPN connection should be made using DTU credentials and a VPN software such as `Cisco`[**cisco**] or `OpenVPN`[**openvpn**]

2.3 Data Information

The packages can process either all available trips on one route or one selected trip. Those options can be controlled by `--route` and `--trip` arguments. The route information is set in `json/routes.json` file. Hence, to use the route and trip data, it is required to checkout the previously referred `json` package.

Currently available routes are: CPH1_VH, CPH1_HH,CPH1_HH,CPH6_VH, CPH6_HH, M3_VH, M3_HH, M13_VH and M13_HH. If the user requires a new route, the route data should be added to the json file.

The route information should be structured in the following way, as illustrated using CPH1_VH route as an example:

```
"CPH1_VH": {
            "title": "CPH1 VH",
            "geometry":"circle",


            "GM_trips": [7792,7805,8040,8042,8227,9289,10204,10218],
            "GM_validated_passes": {"7792":[0,1],"7805":[1,2],"8040":[1],
                                    "8227":[2],"9289":[0],"10204":[1],
                                    "10218":[0]},
            "GM_additional_info": {"7792":{"date":"April2021",
                                   "start":"markokirken_towards_kongensnytorv"}},

            "P79_trips":["7a11eeac-a010-4e35-90e5-45e0e54703fb"],
            "P79_additional_info": {"72454ca1-b7b9-4d1c-a22a-f7ee409010eb":
                                    {"date":"June2020"},
                                    "d6b1cf27-41ed-43b6-8050-2068ef941a0a":
                                    {"date":"Sep2020"},
                                    "7a11eeac-a010-4e35-90e5-45e0e54703fb":
                                    {"date":"May2021"}},

            "VIAFRIK_trips":["32158db7-f63a-4396-9bba-65101cecd75f"],

            "ARAN_trips":["9ab9ce62-e197-4672-9c85-c9b37b1c88cd"],
            "ARAN_additional_info": {"6ea35d27-b61d-4363-95fd-389679bc9a3f":
                                    {"date":"June2020"},
                                    "6d89136a-64cf-446a-81e4-4e67bccc0bb9":
                                    {"date":"August2020"},
                                    "9ab9ce62-e197-4672-9c85-c9b37b1c88cd":
                                    {"date":"June2021"}}
            }
```

The `title` refers to the route title, utilized in plot titles, output directory name and output filenames. The GM and DRD trips recorded on the given route are referred using their identification numbers (as given in the database) and set using `"GM_trips"` (GM), `"P79_trips"`, `"VIAFRIK_trips"` and `"ARAN_trips"` (DRD) variables. If multiple DRD id's are set for one route, the first one will be utilized. The `"GM_validated_passes"` refers to validated GM passes per each trip; it is strongly advisable be set it after the GM map matching step, as performed in `GM_preproccesing` package. The `"GM_validated_passes"` are interpolated and prepared for further analysis.

The `"GM_additional_info"`, `"P79_additional_info"` and `"ARAN_additional_info"` variables are optional and set for keeping track about the dataset information.

3 Preprocessing DRD Data

3.1 Package Description

The `Preprocessing DRD Data` package is developed to process and prepare standard data i.e. data recorded using the Danish Road Directorate (DRD) vehicles. The package can be employed in preprocessing P79, ARAN and VIAFRIK data, on multiple routes and trips. The package is run using the script `prepare_DRD.py` with user selected arguments.

The package performs initial cleaning of the selected trips, such as dropping duplicate columns in certain trips. The data is further map matched in slices of 100 GPS points using the `OSRM`[**osrm**] routing service with a radius of 50 m. The OSRM routing service performs map matching based on Hidden Markov Model (HMM). In the initial stage, the service finds a set of nearest Opeen Street Map (OSM)[**OpenStreetMap**] candidate points at given radius for each for each raw GPS point. Subsequently, it constructs possible routes and returns the most probable route on the OSM map.

The output contains cleaned and map matched `pickle` file per each trip. In addition, the package provides options to plot the raw and map matched trajectory on the OSM, further stored in `plots` subdirectory. A trip trajectory obtained using the map matched output data, projected on the OSM is shown in Figure 2.

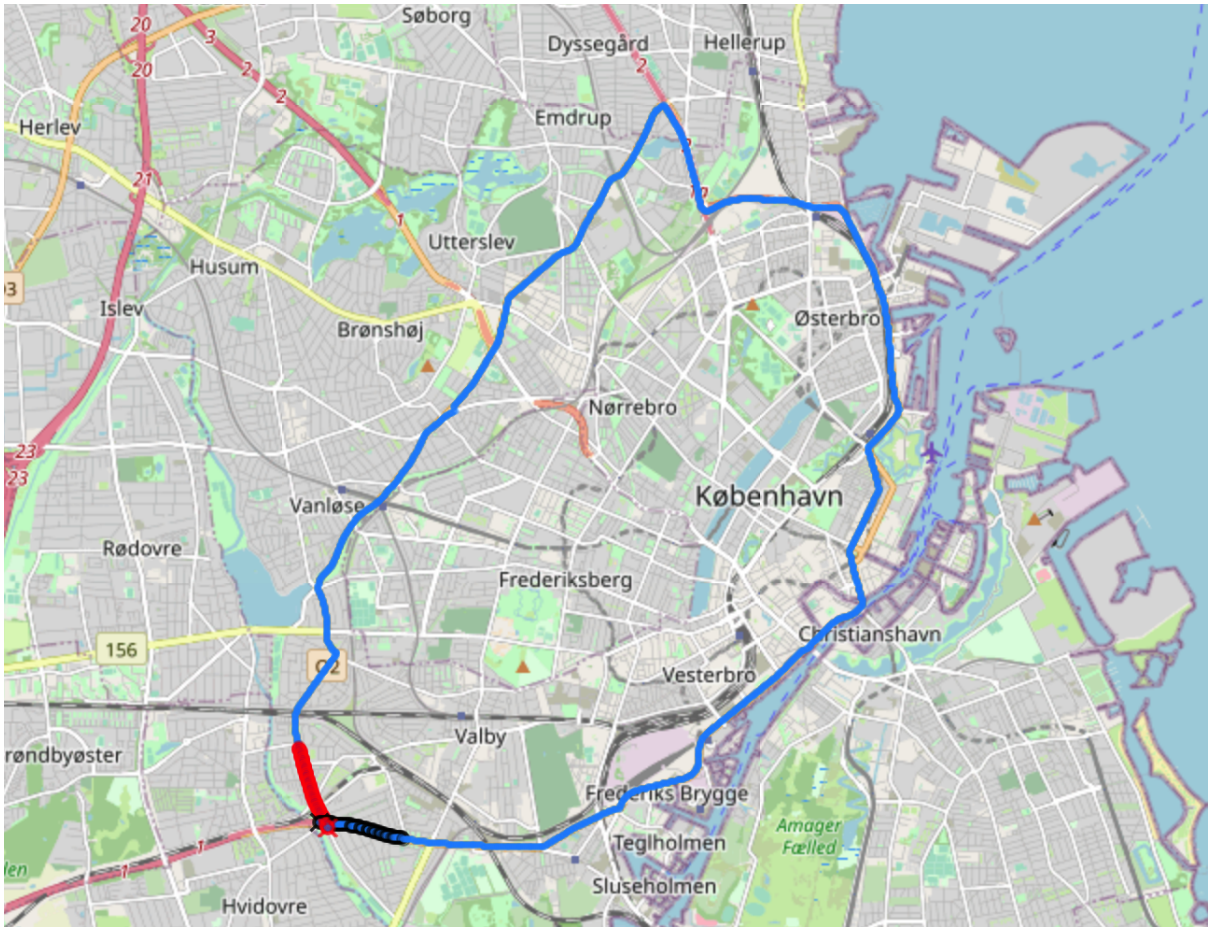


Figure 2: The output figure showing the map matched data, projected on OSM. The initial 100 points are colored in red (start of the trip) and the last 100 points in black (end of the trip).

The package contains `prepare_DRD.py` script, serving as the main script which can be utilized to run the package with multiple options which can be passed as user arguments. The package further contains `utils` directory with analysis and plotting modules; `json` directory with route and hidden database connection json files; `requirements.txt` file containing the sub-dependencies and a `Dockerfile` utilized to build the `Docker` image. In addition, the `mplleaflet` package with a fix required for compatibility with the newest `matplotlib` package is available.

The map matching is done using the `map_match_gps_data` function, available in `utils/matching.py` module. The function input should be a `Pandas` dataframe, containing latitude and longitude data in `lat_name` and `lon_name` columns and returns the dataframe with additional `lat_map` and `lon_map` columns, containing the map matched data.

The plotting of the trajectory is done using the `plot_geolocation function`, available in `utils/plotting.py` module. It has two required arguments: latitudes and longitudes array-type objects, utilized to plot the trip trajectory and several optional arguments related to the output directory, filename and plotting options. The function creates and saves two plots: a `matplotlib` scatter plot and its projection on OpenStreetMap (OSM) network. The latter employs `mplleaflet` package and requires access to a web browser; if the browser is not available, the OSM figure will not be created.

- It is required to pass the vehicle type, by specifying either: --p79, --viafrik or --aran.
- It is required to pass either:

  --route <route_name>: If route information is available, all trips from this route will be loaded from the routes json file. If this is a new route, please update the routes json file with the new new route information.

  --trip <trip_id>: This trip will be loaded and processed.
- To do map matching, pass:

  --map_match: To map match GPS coordinates, pass true. The default is False.
- Route and connection files:

  --routes_file: Json file with route information.

  --conn_file: Json file with connection information.
- Additional options:

  --plot: plot and save trip trajectory on Open Street Map if a web-browser is available, else only the trip geometry. If the code is run locally and a web-browser is open, the trajectory will be ploted on Open Street Map and the plot will be saved. If running with docker, the web-browser is not available, hence a plot of trajectory (geometry) without the Open Street Map will be plotted and saved. (default: False)

  --out_dir: set base output directory (default is 'data' and will be created in the current directory is not already present)

  --preload_map_matched: if passed with --map_match, the map matched output will be loaded but not recreated. Primarily used for debugging. (default: False)

  --preload_plots: load map matched OSM plots. Primarily used for debugging. (default: False)

--dev_mode: run the pipeline on selected lines of database entries. The number of lines can be changed by calling load_DRD_data(Primarily used for debugging. (default: False)

--dev_mode_n_lines: number of lines to load in

--dev_mode (default=500)

The output pickle files with map mached data will be located in:

- P79 data: <data>/P79_processesed_data/<route>
- ARAN data: <data>/ARAN_processesed_data/<route>
- VIAFRIK data: <data>/VIAFRIK_processesed_data/<route>

An additional subdirectory `plots`, containg the route plots, will be created in the output directory.

If only a trip is passed, the route name containing this trip, as given in the route json file, will be set. If this is a new trip for which the route name is not set, the route name will be set to 'unknown'.

3.2 Run Examples

- Process, map match and plot p79 data on CPH1_VH route:

  ```
  python -i prepare_DRD.py --p79 --route CPH1_VH --map_match --plot
  ```

  The output will be stored in: data/P79_processesed_data/CPH1_VH.

- Process, map match and plot ARAN data on M3_HH route and store output in /dtu-compute/lira/ml_data/data:

  ```
  python -i prepare_DRD.py --aran --route M3_HH --map_match --plot
  --out_dir /dtu-compute/lira/ml_data/data
  ```

  The output will be stored in: /dtu-compute/lira/ml_data/data/ARAN_processesed_data/M3_HH.

- Process, map match and plot a P79 trip, for which the information is available in the json routes file:

  ```
  python -i prepare_DRD.py --p79 --trip <trip_id> --map_match --plot
  ```

  The output will be stored in: data/P79_processesed_data/<route>, where the route name containg this trip is found in the routes json file.

- Process, map match and plot a new P79 trip, for which the information is not available in the json routes file:

  ```
  python -i prepare_DRD.py --p79 --trip <new_trip_id> --map_match --plot
  ```

  The output will be stored in: data/P79_processesed_data/unknown, if this is a new trip and the route name is not found in the route json file.

4 Preprocessing Car Data

4.1 Package Description

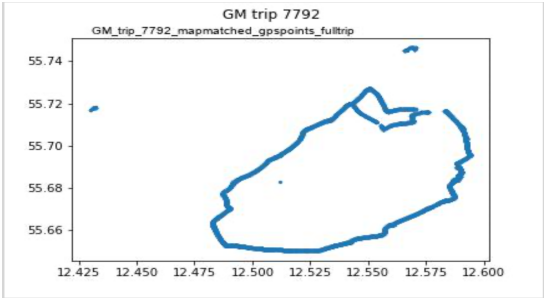
The `Preprocessing Car Data` package is developed to process and prepare in-vehicle sensor data i.e. data recorded using the GM cars. The package can be employed in preprocessing GM data, on multiple routes and trips. The package is run using the script `prepare_GM.py` with user selected arguments.

The package performs map matching of raw GPS coordinates using `map_match_gps_data` function, applied on 100 point data slices with a radius of 50 m. Non-matched GPS points are discarded, resulting in map matching coordinates `lat_map` and `lon_map` stored for successfully matched points.

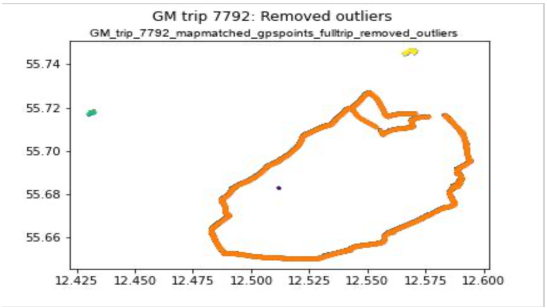
It was observed that certain GPS recordings were measured at distant locations, possibly due to sensor malfunction. Hence, the map matched points are further analyzed in order to detect and remove GPS outliers. The outlier removal procedure is implemented using the `DBScan` algorithm [**schubert2017dbscan**] which is an unsupervised machine learning algorithm capable of detecting and differentiating high density data clusters. The `DBScan` is employed using the maximum distance between two samples for one to be considered as in the neighborhood of the other, ϵ set at 0.01 and the number of samples in a neighborhood for a point to be considered as a core point, `min_samples` set at 20:

```
model = DBSCAN(eps=0.01, min_samples=20)
```

The `DBScan` detects a variable number of clusters, depending on point density. The points associated to clusters which contain $< 1\%$ of all GPS points recorded per trip are defined as outliers and discarded. The algorithm performance is illustrated in Figure 5. Shown are the raw GPS recordings for trip 7792 and four identified clusters, illustrated in different colors. After the outlier removal procedure, all points not associated to the orange cluster are discarded, resulting in a cleaned trip, without GPS outliers.



(a) GM trip 7792: Raw GPS points, containing multiple outliers.



(b) GM trip 7792: Identified clusters, shown in different colors (4 resulting clusters).

Figure 3: GPS clustering and outlier removal of GM trip 7792, using the `DBScan` algorithm. All clusters containing $< 1\%$ of all points are removed, resulting in points associated to the orange cluster being kept.

Since majority of trips consists of multiple passes, each trip is further split into multiple passes. This is done by performing a polynomial fit between the distance computed with respect to the initial point and the dataset index i.e. index difference with respect to the initial point index. The polynomial fit degree is tuned to the degree which results in the minimum discrepancy between the data and the fitted points, measured using the

minimum root mean square error. The different pass regions are defined using the valleys of the fitted function.

The function valleys are found using:

```
pred_inv = -1*pred.reshape(-1)
minima_indices_cand = find_peaks(pred_inv, prominence=p, distance=500)[0]
```

where prominence refers to valley prominence and distance to required minimal horizontal distance in points between neighbouring valleys. Tuning their values affects the algorithm sensitivity to different valley sizes. The algorithm performs well for both circular and non-circular trajectories.

Its performance is illustrated in Figure 4 for trip 7792 which is automatically split into 3 passes. The prominence and distance were tuned to detect prominent valleys, hence ignoring small valleys such as the one at index ≈ 6000.

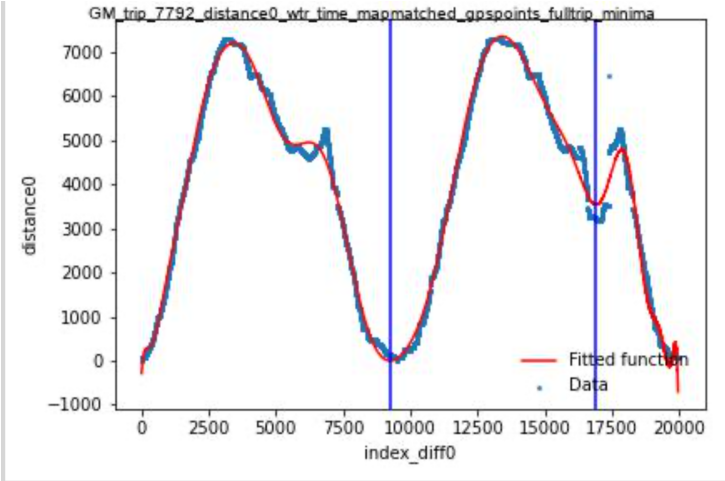
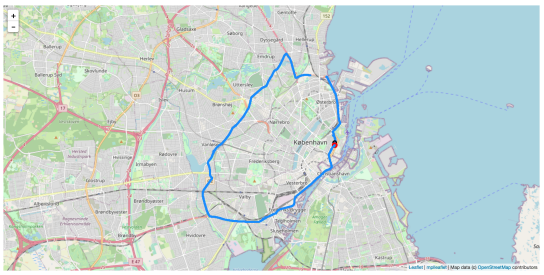


Figure 4: The passes are detected by performing a fit between the distance with respect to the initial point and the index. The algorithm is tuned to detect prominent valleys, used to define the regions (passes) into which a trip is split. The different passes are visualized with vertical lines.

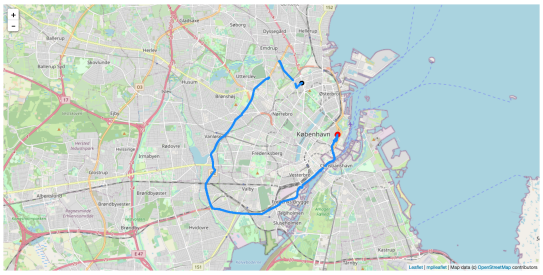
After the automatic pass detection, the code creates a `pickle` file containing data for each pass and outputs a set of plots visualizing trajectories of different passes. Each pass will be further interpolated which is a slow procedure. Since only the passes passing along the same route sections as the reference DRD trip can be aligned and exploited in further analysis - to avoid the slow interpolation step for passes without reference data, the plots should be initially manually checked and the passes along the reference DRD route should be added to the `json/routes.json` file under `"GM_validated_passes"` identifier.

Technically, it is suggested to initially run the package using the `--skip_interp` argument. The output plots should be checked and validated passes added to the json file. The user can decide to use all passes, but this will slow down the following interpolation step and will not add new data for the analysis.

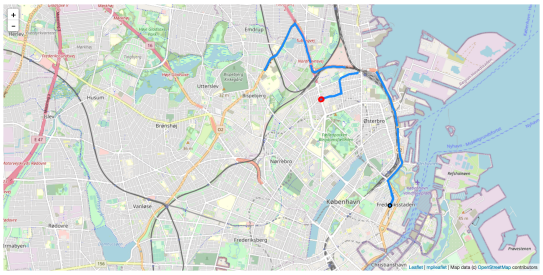
After the validated passes are defined, the package should be rerun without the `--skip_interp` argument. The map matching, outlier detection and automatic pass detection steps will be reloaded (hence not recomputed) unless the `--recreate` argument is passed.



(a) GM trip 7792, pass 0



(b) GM trip 7792, pass 0



(c) GM trip 7792, pass 3

Figure 5: The automatic pass detection result: a trip is split into multiple passes, visualized on OSM. The starting points are visualized in red and ending points in black.

Lastly, the interpolation procedure is applied on every pass, listed in the `json/routes.json` file and found in the input directory. In the interpolation stage, the GPS location (lat^{int}, lon^{int}) is assigned to all in-vehicle sensors by performing interpolation between the adjacent GPS measurements, assuming the constant vehicle speed between them:

The interpolation is done for acceleration and speed sensors as default. If `--user_add_sensors` is passed, the interpolation will be performed for additional steerng, wheel pressure, yaw rate and traction consumption sensors.

$$\text{lat}^{\text{int}} = \text{lat}^{\text{GPS}} + \Delta\text{lat}^{\text{GPS}} \cdot \frac{\Delta t}{\Delta t^{\text{GPS}}} \tag{1}$$

$$\text{lon}^{\text{int}} = \text{lon}^{\text{GPS}} + \Delta\text{lon}^{\text{GPS}} \cdot \frac{\Delta t}{\Delta t^{\text{GPS}}} \tag{2}$$

- It is required to pass either:

  route: Process all trips on this route, given in json file.

  trip: Process this trip only. The route name will be loaded from the json file, else set to 'unknown'.

- Route and connection files:

  --routes_file: Json file with route information.

  --conn_file: Json file with connection information.

- Additional options:

  --skip_interp: Do not interpolate.

  --load_add_sensors: Load additional sensors.

  --plot: Pass to plot data on Open Streep Map.

--only_load_pass_plots: Only load GM pass plots.

--recreate: If recreate is False and the files are present, the data will be loaded from files.

--recreate_interp: Recreate only interpolation files. If recreate is false and the files are present, the data will be loaded from files.

--out_dir: Output directory.

--dev_mode: Development mode. Will process a limited number of lines.

--dev_mode_n_lines: Process this number of lines in development mode.

--only_load_trips_data: Only load GM trips from the database to explore. Skip the rest.

4.2 Run Examples

It is advisable to initially run the code without interpolation to cross check the split into passes, in the following way:

```
python -i prepare_GM.py --route M3_VH --load_add_sensors --recreate
        --out_dir /dtu-compute/lira/ml_data/data --skip_interp
```

Then run the full preprocessing pipeline, including the interpolation for validated passes:

```
python -i prepare_GM.py --route M3_VH --load_add_sensors --recreate
        --out_dir /dtu-compute/lira/ml_data/data
```

The interpolated `pickle` files for each pass will be created in:

```
/dtu-compute/lira/ml_data/data/GM_processesed_data/M3_VH_add_sensors/passes
```

The output plots will be created in:

```
/dtu-compute/lira/ml_data/data/GM_processesed_data/M3_VH_add_sensors/plots
```

5 Alignement

5.1 Package Description

The `Alignement` package is developed to segment DRD data and align it with the in-vehicle sensor data i.e. data recorded using the GM cars. The package can be employed in aligning GM data with P79, ARAN, VIAFRIK data or their combination, on multiple routes and trips. The package is run using the script `align_data.py` with user selected arguments.

The package initially creates sliding window segments with the selected DRD data type. The window size and step can be set using the `--window_size` and `--step` arguments. In the default mode, the code will create 100 m segments with 10 m step size. The target variables such as IRI and defects types are averaged over the segment while variables such as GPS location and timestamp have their starting and ending segment points saved.

If several DRD vehicle types are required for alignment, the segmented and aligned DRD dataset is filtered to discard the segments in which the discrepancy between the starting or ending aligned points of different vehicle types is larger than 5 m.

In the following step, the DRD segmented data is aligned with each validated GM pass. The alignment is done by performing a nearest neighbor search for starting and ending segment points of each DRD segment. The nearest neighbor search is efficiently implemented using the `KDTree` algorithm with the Haversine distance. The GM aligned data is saved in the array form associated with each sensor. Similarly to the previous procedure, all segments in which the discrepancy between the starting or ending GM and DRD aligned points is larger than 5 m, are discarded.

The output creates aligned `pickle` files and a set of plots, examining the discrepancies between the aligned points. The discrepancies are evaluated by computing the Haversine distance between the aligned starting and ending segment points between each aligned data type i.e. for combinations of GM and required DRD vehicle types.

The alignment performance is illustrated using discrepancy errors between the aligned points in Figures 6 (M3 highway) and 7 (CPH1 Municipality road).

The alignment can be employed using the `align_data.py` script with user arguments. The default sensors are acceleration and speed. If `--user_add_sensors` is passed, the GM pass with the additional sensors will be loaded and the additional sensor data aligned with the DRD segments.

5.2 Run Examples

To align GM with additional sensors, P79 and ARAN on CPH1_VH route and recreate all files, run:

```
python -i align_data.py --p79 --aran --route CPH1_VH --load_add_sensors
--dir_base /dtu-compute/lira/ml_data/data --recreate
```

The output will be in:

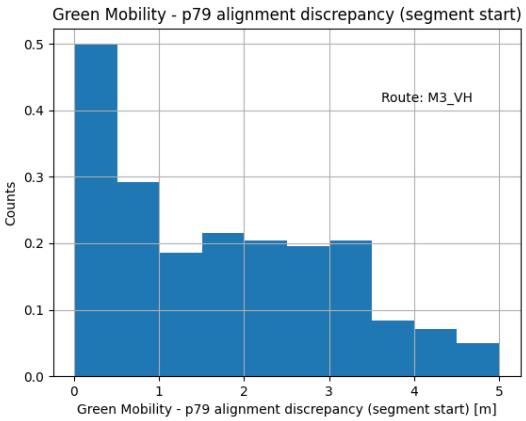
```
/dtu-compute/lira/ml_data/data/aligned_GM_P79_ARAN_data_window-100-step-10/
CPH1_VH_add_sensors
```

To align GM with default sensors with p79 and ARAN on M3_VH route and recreating all files:

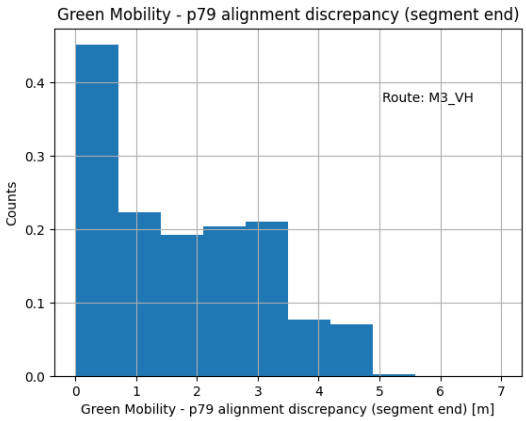
```
python -i align_data.py --p79 --route M3_VH
--dir_base /dtu-compute/lira/ml_data/data --recreate
```

The output will be in:

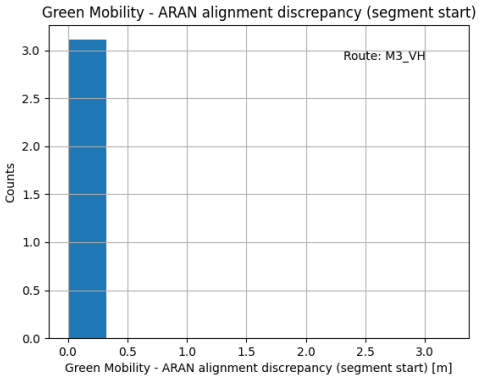
```
/dtu-compute/lira/ml_data/data/aligned_GM_P79_data_window-100-step-10/M3_VH
```



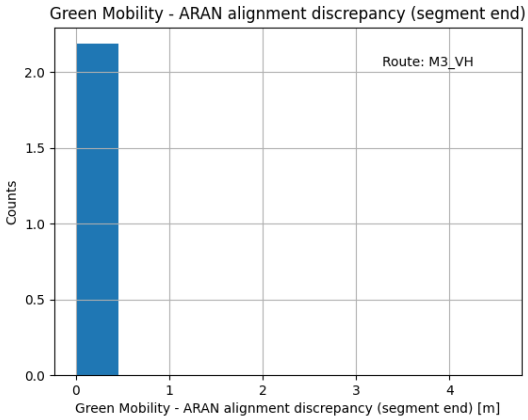
(a) GM-P79 segment (start points)



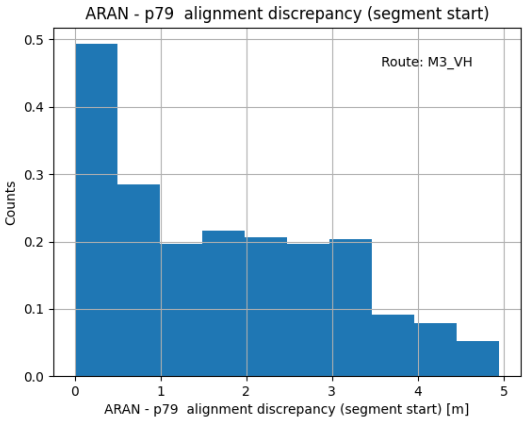
(b) GM-P79 segment (end points)



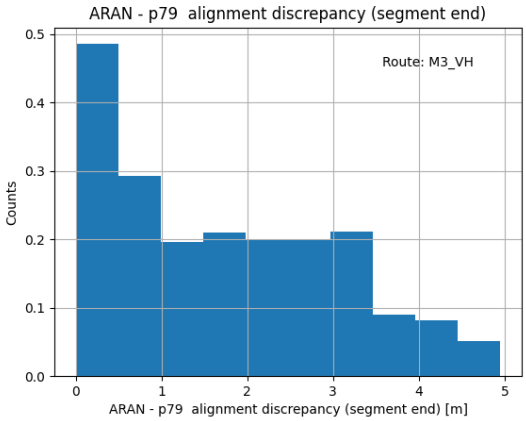
(c) GM-ARAN segment (start points)



(d) GM - ARAN segment (end points)

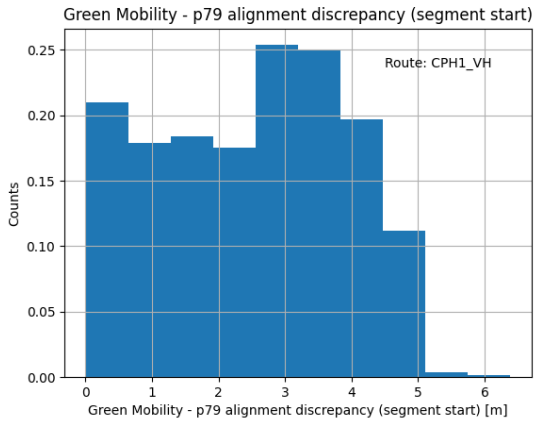


(e) P79-ARAN segment (start points)

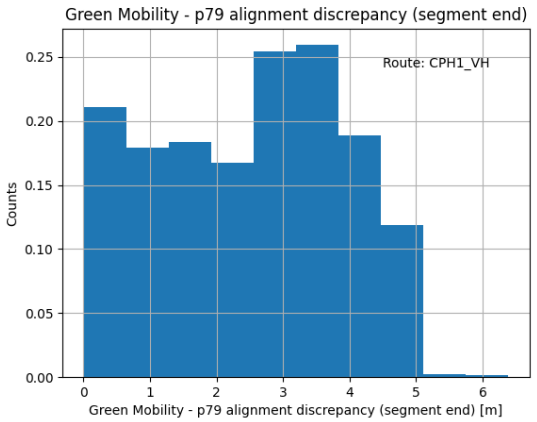


(f) P79-ARAN segment (end points)

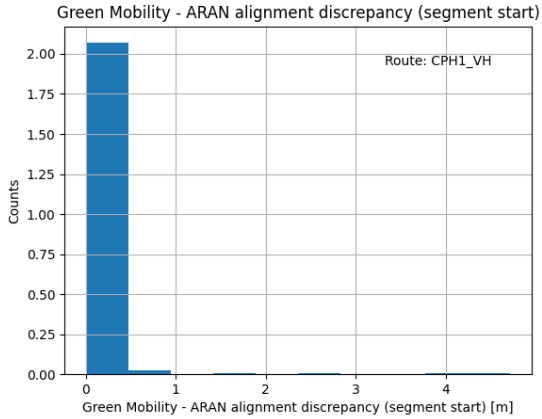
Figure 6: The discrepancy between the aligned GM, P79 and ARAN points on M3 route.



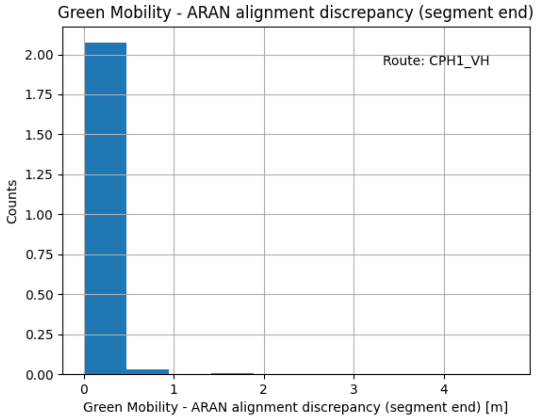
(a) GM-P79 segment (start points)



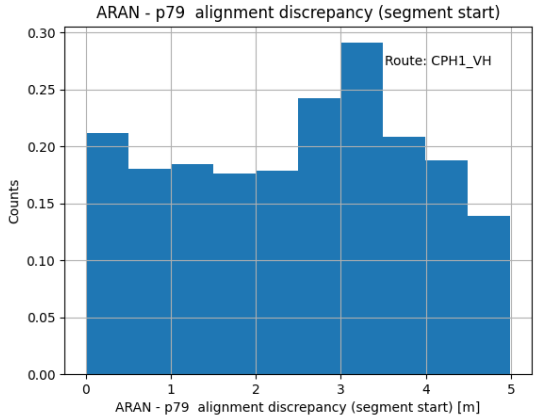
(b) GM-P79 segment (end points)



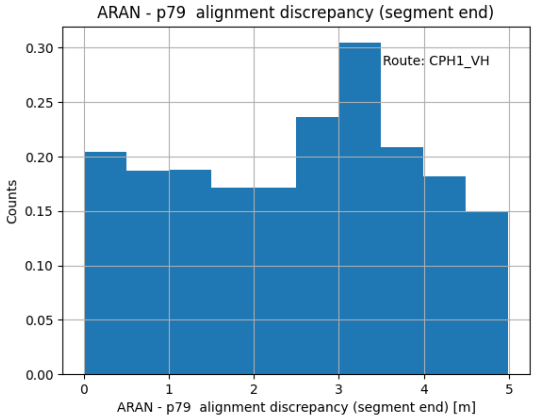
(c) GM-ARAN segment (start points)



(d) GM - ARAN segment (end points)



(e) P79-ARAN segment (start points)



(f) P79-ARAN segment (end points)

Figure 7: The discrepancy between the aligned GM, P79 and ARAN points on CPH1 municipality route.

6 Feature Engineering

6.1 Package Description

The `Feature Engineering` package is developed to compute relevant feature candidates and select an optimal subset of relevant features for modelling different targets, namely IRI, KPI, DI and various defect types. The package is run using the `get_features.py` script with user selected arguments.

In the first stage, the package extracts a number of features in the statistical and time domains, using the `tsfel`[**barandas2020tsfel**]. The features are computed per each aligned segment, for selected input sensors. Computed are statistical features such as mean, variance, quartile ranges and ECDF percentiles and temporal features such as autocorrelation, total energy, entropy and various signal peaks features. The extensive list of statistical and temporal features with their description is detailed in [**barandas2020tsfel**]. Prior to extraction, the sensors can be resampled using linear interpolation between the closest points to the length of the 90 % quartile. In the default mode, the segments with at least 1 point with speed below 20 km are discarded.

In the following, feature selection phase, an optimal subset of predictive irredundant features is constructed. The feature selection is employed in two stages: i)removal of constant features and ii) subset selection using the Sequential Feature Selection (SFS) algorithm.

The SFS is an iterative algorithm which starts by evaluation of a set of models, constructed using a single feature and selects the feature utilized in the model with the best performance. In the next iteration, subsets constructed using the selected and one additional feature are evaluated and the additional feature is selected in the same manner. The iteration process ends when performance starts decreasing. The SFS is employed using the `mlxtend` library [**raschkas˙2018˙mlxtend**], in the forward search mode, using the Random Forest model(RF)[**breiman2001random**]. The performance is evaluated in a 5-fold cross-validation applied on the 'trainvalid' dataset, using the R_2 metric.

The model employed in the SFS with IRI as the target is defined as:

```
model = RandomForestRegressor(n_trees, min_impurity_decrease=1.5e-2,
min_samples_leaf = 2, max_depth=5)
```

It was observed that the SFS is prone to overfitting resulting in the selection of features relevant to modelling the noise in the 'trainvalid' dataset which lead to poor performance in the modelling stage. Hence, the `min_impurity_decreases`, `min_samples_leaf` and `max_depth` might have to be tuned in the SFS stage for different target to combat overfitting.

The main script is `get_features.py`; the feature extraction phase is implemented using `feature_extraction` function while the feature selection phase is implemented using the `find_optimal_subset` function. The trip and routes can be passed as user arguments. By default, the 80 % is used to form the 'trainvalid' dataset used in the SFS procedure while the last 20 % is used to form the 'test' dataset on which the features are not tuned but only selected.

The default input sensors are vertical acceleration and speed, while additional sensors can be employed by passing the `--use_3daccm` to use 3D acceleraion and `--use_add_sensors` to use additional steering, wheel pressure and traction consumption sensors. The `--load_add_sensors` controls the input data; if passed, the inputs are loaded from the dataset con-

taining additional aligned sensors but this flag does not control the sensors employed in the feature engineering phase.

In addition to prepared 'trainvalid' and 'test' `pickle` files contaning optimal extracted features, the package produces additional plots with histograms of selected features and LaTeX tables.

The modelling of different targets is done using the same feature extraction files while the feature selection step is done for each target. In the default mode, the feature extraction and feature selection files are reloaded, if present and if recreate flags are not passed. Hence, if implementing changes in the feature selection step, only this step should be recreated, but not the feature extraction step.

6.2 Run Examples

To load the aligned P79-ARAN dataset on route M13_VH which contains additional sensors and process it to model IRI using vertical acceleration and speed as input sensors, run:

```
python -i get_features.py --in_dir /dtu-compute/lira/ml_data/data
--target IRI_mean --load_add_sensors --p79 --aran  --route M13_VH
```

To load the aligned P79-ARAN dataset on route M13_VH which contains additional sensors and process it to model KPI using 3D acceleration and speed as input sensors, run:

```
python -i get_features.py --in_dir /dtu-compute/lira/ml_data/data
--target KPI --load_add_sensors --p79 --aran  --route M13_VH --use_3dacc
```

To load the aligned P79-ARAN dataset on route CPH1_VH which contains additional sensors and process it to model DI using 3D acceleration, speed and additional nput sensors, run:

```
python -i get_features.py --in_dir /dtu-compute/lira/ml_data/data
--target DI --load_add_sensors --p79 --aran  --route CPH1_VH --use_3dacc
--use_add_sensors
```

To recreate all files and use 3D acceleration, speed and additional sensors on routes M3_VH, M3_HH, M13_VH and M13_HH to model IRI, run:

```
python -i get_features.py --in_dir /dtu-compute/lira/ml_data/data
--target IRI_mean --load_add_sensors --p79 --aran --target IRI_mean
--use_add_sensors --route M3_VH M3_HH --use_3dacc --recreate_fe
--recreate_fs  --route M3_VH M3_HH M13_VH M13_HH
```

The output files will be created in:

- Feature extraction output files:

  ```
  /dtu-compute/lira/ml_data/data/
  aligned_fe_fs_GM_P79_ARAN_data_window-100-step-10/
  (then <route> and <sensor> subdirectory)
  ```

For example, the output with the default [M3_VH, M3_HH] routes and default input sensors will be in:

```
/dtu-compute/lira/ml_data/data/
aligned_fe_fs_GM_P79_ARAN_data_window-100-step-10/
M3_VH_M3_HH_filter_speed_accspeed/
```

- Feature selection output files:

```
/dtu-compute/lira/ml_data/data/
aligned_fe_fs_GM_P79_ARAN_data_window-100-step-10/
M3_VH_M3_HH_filter_speed_accspeed/feature_selection_<target_name>/
```

7 Machine Learning Modelling

7.1 Package Description

The `Machine Learning Modelling` package is developed to utilize a set of regression and classification models to model the road roughness condition such as IRI, KPI, DI or specific defect types. The package utilizes data prepared in the feature selection step i.e. data with the optimal subset of extracted features per segment per selected sensors, trains a set of machine learning models and assesses their performance. The package is run using the `run_modelling.py` script with user selected arguments.

In the initial phase, each feature is scaled to the standard normal distribution i.e. to the Gaussian distribution with the mean value $\mu = 0$ and the standard deviation $\sigma = 1$. The scaler parameters are fitted using the 'trainvalid' dataset and applied on the 'test dataset'.

Further, two datasets types are defined: i) the scaled dataset ii) the PCA transformed scaled dataset where the PCA is fitted on the scaled 'trainvalid' dataset and utilized to transform the 'test' dataset. The number of PCA components is chosen to explain the 99 % of variance in the 'trainvalid' dataset.

A set of models defined using the `--rm` (regression) and `--cm` (classification) models is further fitted to the 'trainvalid' dataset and the hyperparameters are tuned in grid search manner with 5-fold cross validation. The hyperparameter search regions are defined in `get_regression_model` and `get_classification_model` functions available in `utils/analysis.py` module. The final performance is assesed on 'test' dataset and a set of plots and LaTeX tables is created as the output. The training and assesment pipeline is independently applied on the scaled and the PCA transformed scaled datasets.

Available regression models are: 'dummy', 'linear','lasso','ridge', 'elastic_net', 'kNN', 'random_forest', 'SVR_rbf','ANN', while available classification models are: 'dummy', 'naive_bayes', 'kNN','logistic_regresion', 'random_forest','SVC_rbf' and 'ANN'. The 'dummy' defines the baseline model which predicts the mean target value (regression) and the most frequent class (classification).

Similarly to the previous packages, the default input sensors are vertical acceleration and speed. The usage of additional sensors can be controlled by passing the `--use_3daccm` to use the 3D acceleraion and `--use_add_sensors` to use additional steering, wheel pressure and traction consumption sensors.

7.2 Run Examples

To run the regression pipeline using the set of default models to model IRI on p79 and aran input datasets to model IRI (aran target is not used if IRI is selected as the target, it refers to the combined input dataset) using vertical acceleration and speed as input sensors, run:

```
python -i run_modelling.py --in_dir /dtu-compute/lira/ml_data/data --do_reg
--aran --p79 --target IRI_mean
```

To utilize 3D acceleration, speed and additional input sensors, run:

```
python -i run_modelling.py --in_dir /dtu-compute/lira/ml_data/data --do_reg
--aran --p79 --target IRI_mean
--use_3dac --use_add_sensors
```

To model KPI, using vertical acceleration and speed as input sensors, run:

```
python -i run_modelling.py --in_dir /dtu-compute/lira/ml_data/data --do_reg
--aran --p79 --target KPI
```

To model KPI, using vertical acceleration and speed as input sensors on routes M3_VH and M3_HH, run:

```
python -i run_modelling.py --in_dir /dtu-compute/lira/ml_data/data --do_reg
--aran --p79 --target KPI --route M3_VH M3_HH
```

To model DI, using 3D acceleration, speed and additional input sensors using only 'Lasso' and 'Ridge' models and to recreate the grid search step, run:

```
python -i run_modelling.py --in_dir /dtu-compute/lira/ml_data/data --do_reg
--aran --p79 --target DI --rm lasso ridge --recreate_gs
--use_3dac --use_add_sensors
```

The output files will be created in:

- General output directory:

  `<feature_extraction_directory>/modelling_<target>/`

- Specifically for IRI target, using default sensors and M3_VH and M3_HH routes, the output will be created in:

  `/dtu-compute/lira/ml_data/data/`
  `aligned_fe_fs_GM_P79_ARAN_data_window-100-step-10/`
  `M3_VH_M3_HH_filter_speed_accspeed/modelling_IRI_mean/`

- For KPI target, using 3D acceleration and additional sensors on M3_VH, M3_HH, M13_VH and M13_HH routes, the output will be created in:

  `/dtu-compute/lira/ml_data/data/`
  `aligned_fe_fs_GM_P79_ARAN_data_window-100-step-10/`
  `M3_VH_M3_HH_M13_VH_M13_HH_filter_speed_3daccspeed_add_sensors//modelling_KPI/`