



Labs for prototyping future mobility data sharing solutions in the cloud

D4.8 Data Enrichment Processors (V2)

30/11/2023

Author(s): Mohamed KARAMI (AKKODIS) - Francesco LETTICH (CNR)



MobiDataLab is funded by the EU under the H2020 Research and Innovation Programme (grant agreement No 101006879).

Summary sheet

Deliverable Number	D4.8
Deliverable Name	Data Enrichment Processors (V2)
Full Project Title	MobiDataLab, Labs for prototyping future Mobility Data sharing cloud solutions
Responsible Author(s)	Mohamed KARAMI (AKKODIS) - Francesco LETTICH (CNR)
Contributing Partner(s)	CNR
Peer Review	Alberto BLANCO JUSTICIA (URV) - Aliko BENMAYOR (KUL)
Contractual Delivery Date	30-09-2023 (extended to 30-11-2023)
Actual Delivery Date	30-11-2023
Status	Final
Dissemination level	Public
Version	V1.0
No. of Pages	53
WP/Task related to the deliverable	WP4/T4.4
WP/Task responsible	AKKODIS/AKKODIS
Document ID	MobiDataLab-D4.8-DataEnrichmentProcessorsV2_v1.0
Abstract	This deliverable is a report to provide an overview of the Task 4.4 version 2 demonstrator, which encompasses the geographical and semantic enrichment demonstrators.

Legal Disclaimer

MOBIDATALAB (Grant Agreement No 101006879) is a Research and Innovation Actions project funded by the EU Framework Programme for Research and Innovation Horizon 2020. This document contains information on MOBIDATALAB's core activities, findings, and outcomes. The content of this publication is the sole responsibility of the MOBIDATALAB consortium and cannot be considered to reflect the views of the European Commission.

Project partners

Organization	Country	Abbreviation
AKKODIS	France	AKKODIS
CONSORZIO INTERUNIVERSITARIO PER L'OTTIMIZZAZIONE E LA RICERCA OPERATIVA	Italy	ICOOR
AETHON SYMVOULI MICHANIKI MONOPROSOPI IKE	Greece	AETHON
CONSIGLIO NAZIONALE DELLE RICERCHE	Italy	CNR
HOVE	France	HOVE
HERE GLOBAL B.V.	Netherlands	HERE
KATHOLIEKE UNIVERSITEIT LEUVEN	Belgium	KUL
UNIVERSITAT ROVIRA I VIRGILI	Spain	URV
POLIS - PROMOTION OF OPERATIONAL LINKS WITH INTEGRATED SERVICES	Belgium	POLIS
F6S NETWORK IRELAND LIMITED	Ireland	F6S

Document history

Version	Date	Organization	Main area of changes	Comments
0.1	05/08/2023	AKKODIS	Deliverable outline	Draft
0.2	10/10/2023	CNR	Updates to the Semantic enrichment processor part	Draft
0.3	15/10/2023	AKKODIS	Updates to the Geographical enrichment processor part	Draft
0.4	10/11/2023	AKKODIS, CNR	Integrated feedback from the Hackathon	Draft
0.5	23/11/2023	URV, KUL	--	Peer review
0.6	24-29/11/2023	CNR/AKKODIS	All	TL & Coordinator Quality check
1.0	30/11/2023	AKKODIS	All	Final Version Submission

Executive Summary

The deliverable 4.8 is a comprehensive report that outlines the advancements in the second version of the MobiDataLab data enrichment processors. This report offers an in-depth overview of the updated versions of the geographical and semantic enrichment demonstrators. Building upon the previous Deliverable 4.7, which introduced the initial versions of these enrichment processors, the current document highlights significant updates, i.e.:

- We report on the backend changes to the geographical enrichment processor, which have been done to improve the processing algorithm and the interoperability with the API services catalogue.
- We present the newly developed Semantic Enrichment Processor Web API, a critical component designed to expose the functionalities of the three modules comprising the semantic enrichment processor's back-end to remote users. This development transforms the semantic enrichment processor into a service integrated within the Transport Cloud platform. Furthermore, informed by feedback received during the MOBIDATALAB Hackathon, we provide potential applications of the semantic enrichment processor in addressing various aspects of the walkability issue presented in the Paris Challenge, offering interesting ideas for its utilization.

Table of contents

1. INTRODUCTION.....	8
1.1. PROJECT OVERVIEW.....	8
1.2. PURPOSE OF THE DELIVERABLE.....	8
1.3. INTENDED AUDIENCE & REVIEW PROCESS.....	8
1.4. STRUCTURE OF THE DELIVERABLE.....	8
2. SEMANTIC ENRICHMENT OF MOBILITY DATA (V2 DEMONSTRATOR)	9
2.1. HOW TO SET UP THE SEMANTIC ENRICHMENT DEMONSTRATOR	10
2.2. HOW TO USE THE INTERACTIVE USER INTERFACE	11
2.2.1. Pre-processing module.....	12
2.2.2. Trajectory segmentation module	13
2.2.3. Enrichment module	14
2.3. HOW TO USE THE WEBAPI.....	17
2.4. EXAMPLE OF CONTENT WITHIN AN RDF GRAPH	21
2.5. HOW TO TACKLE THE PARIS CHALLENGE WITH THE AID OF THE SEMANTIC ENRICHMENT PROCESSOR.....	23
3. GEOGRAPHICAL ENRICHMENT OF MOBILITY DATA (V2 DEMONSTRATOR)	25
3.1. HOW TO BUILD THE GEOGRAPHICAL ENRICHMENT DEMONSTRATOR.....	25
3.1.1. Manual build and dependencies installation	25
3.1.2. Build and run as a docker image through Gitlab-CI.....	27
3.1.3. Build and run as a docker image through Travis-CI	28
3.1.4. Pull and run MDL-Geo-Enrichment docker image.....	29
3.2. HOW TO USE THE GEOGRAPHICAL ENRICHMENT DEMONSTRATOR.....	31
3.2.1. Here API enrichment:	32
3.2.2. Navitia API enrichment:	34
3.2.3. OSM API enrichment:	34
3.2.4. GTFS API enrichment:.....	34
3.2.5. GeoJson API enrichment:.....	35
3.2.6. Generic Json API enrichment:	36
3.2.7. Enrich Open Street Map lines:.....	39
3.2.8. API demonstration:	40
3.3. MIGRATION OF THE API DOCUMENTATION.....	40
4. CONCLUSIONS	43
5. ANNEXES	45
5.1. SEMANTIC ENRICHMENT OF MOBILITY DATA.....	45
5.1.1. Specifications of the Pandas dataframe containing the raw trajectories	45
5.1.2. Specifications of the Pandas dataframe containing the output of the pre-processing module	45
5.1.3. Specifications of the Pandas dataframes containing the output of the segmentation module	46
5.1.4. Specifications of a file containing a dataset of points of interest.	47
5.1.5. Specifications of a file containing weather information.....	47

5.1.6. Specifications of a file containing social media posts.....	47
5.1.7. Details on the ontology used to structure the information within the RDF graph	48
6. REFERENCES.....	51

| List of Figures

Figure 1: Execution of the script mat_builder_ui.py	11
Figure 2: Demonstrator backend.....	12
Figure 3: pre-processing module, as shown in the UI.....	13
Figure 4: Trajectory segmentation module, as shown in the UI.	14
Figure 5: Enrichment module, as shown in the UI.	15
Figure 6: initial view of the subgraph associated with the user ID 402.....	21
Figure 7: Social Media Post aspect.....	22
Figure 8: The subgraph rooted in the node related to the trajectory 224641 of the user 402.	23
Figure 9: The Mobility data enrichment architecture.....	25
Figure 10: MDL-Geo-Enrichment pipeline on Gitlab-CI	27
Figure 11: Travis-CI's environment variables	28
Figure 12: MDL-Geo-Enrichment pipeline on Travis CI	28
Figure 13: Swagger UI – MDL-Geo-Enrichment API list.....	30
Figure 14: Sequence diagram of the enrichment process	31
Figure 15: Sequence diagram of the enrichment process	32
Figure 16: Here stations API enrichment.....	33
Figure 17: GeoJSON Api enrichment.....	35
Figure 18: Generic JSON Api enrichment	38
Figure 19: OSM lines enrichment.....	40
Figure 20: SpringDoc-openapi modules	41
Figure 21: Geographical enrichment API documentation	42
Figure 22: Overview of the Move class with its subclasses	49
Figure 23: Overview of the Stop class with its subclasses.....	49
Figure 24: Overview of the Weather and Social Media Post classes.	50

| List of tables

Table 1: Here API parameters.....	33
Table 2: GeoJson API parameters	36
Table 3: Generic Json API parameters	39

| Abbreviation and acronyms

Abbreviation	Meaning
API	Application programming interface
CI	Continuous integration
CD	Continuous delivery / deployment
GeoJSON	Geographical JSON representation
GTFS	General transit feed specification
JSON	JavaScript object notation
OSM	Open Street Map format
GIS	Geographic information system
MDL-Geo-Enrichment	MobiDataLab geographical enrichment

1. Introduction

1.1. Project overview

There has been an explosion of mobility services and data sharing in recent years. Building on this, the EU-funded MobiDataLab project works to foster the sharing of data amongst transport authorities, operators, and other mobility stakeholders in Europe. MobiDataLab develops knowledge as well as a cloud solution aimed at easing the sharing of data. Specifically, the project is based on a continuous co-development of knowledge and technical solutions. It collects and analyses the advice and recommendations of experts and supporting cities, regions, clusters, and associations. These actions are assisted by the incremental construction of a cross-thematic knowledge base and a cloud-based service platform, which will improve access and usage of data-sharing resources.

1.2. Purpose of the deliverable

The Reference Data enrichment processor v2 is the second version of a prototype that contains a set of open tools allowing data semantic and geographical enrichment.

1.3. Intended Audience & Review process

The dissemination level of the D4.8 deliverable is 'public' (PU).

1.4. Structure of the deliverable

This deliverable is organized as follows. Section 2 gives an overview of the Semantic enrichment processor, highlighting the major updates that have been done with respect to the deliverable 4.7. Similarly, Section 3 gives an overview of the Geographical enrichment processor, and highlights the major changes done with respect to the deliverable 4.7. Finally, Section 4 provides the conclusions, while Section 5 (i.e., Annexes) provides additional information and documentation concerning the enrichment processors.

2. Semantic enrichment of mobility data (v2 demonstrator)

According to the Grant agreement specifications, the objective of Task 4.4 is to “[...] contribute to the development of open tools allowing the enrichment of data [...] by combining the data with other datasets and gathering additional results”. Different data enrichment techniques will be provided as open tools. More specifically, this section focuses on: “Enrich data semantically (combining with the Linked Open Data cloud, RDF/SPARQL)”.

Accordingly, for the semantic enrichment of trajectories, we use the notion of multiple aspect trajectory (MAT) (dos Santos Mello 2019). A *multiple aspect trajectory* expresses movement data that is heavily semantically enriched with dimensions (*i.e.*, aspects) representing various types of semantic information (*e.g.*, stops, moves, weather, traffic, events, and points of interest). Aspects may be derived from different data sources and can be large in number, heterogeneous, or structurally complex.

One important point of novelty is that enrichment aspects might be associated with individual trajectory points, segments (*i.e.*, sub-trajectories which are part of a larger trajectory), a whole trajectory, or the moving objects that are generating the trajectories (*e.g.*, an individual). For instance, weather conditions might get associated with trajectory segments, thus indicating the weather conditions that the objects associated with those segments encountered. Some *Points of Interest* might get associated with a *trajectory stop segment*, thus indicating the point of interest that the object associated with such segment visited during their stay. A *move segment*, *i.e.*, part of a trajectory during which its object moved, might be enriched with the transportation means used to move. Personal data can be associated with the moving object (*e.g.*, gender, social media profile, etc.).

The semantic enrichment demonstrator is therefore focused on building such semantically enriched trajectories in the context of the MobiDataLab project. The demonstrator has been derived from the MAT-Builder system, which in turn is the result of several research works:

- Pugliese, C., Lettich, F., Renso, C. and Pinelli, F., 2022, June. Mat-builder: a system to build semantically enriched trajectories. In *2022 23rd IEEE International Conference on Mobile Data Management (MDM)* (pp. 274-277). IEEE.
- Lettich, F., Pugliese, C., Renso, C. and Pinelli, F., 2023, March. A general methodology for building multiple aspect trajectories. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing* (pp. 515-517).
- F. Lettich, C. Pugliese, C. Renso and F. Pinelli, "Semantic Enrichment of Mobility Data: A Comprehensive Methodology and the MAT-BUILDER System," in *IEEE Access*, vol. 11, pp. 90857-90875, 2023

MAT-builder serves the purpose of showing how a semantic enrichment processor (1) can be implemented purely on top of open-source libraries and tools, (2) is effectively able to semantically enrich raw trajectories via the use of a variety of external data sources, and (3) can generate knowledge graphs that can be subsequently imported in some RDF triple store of preference (e.g., GraphDB) for later analysis and querying, also in combination with Linked Open Data.

The second version of the demonstrator introduces a significant new component: the semantic enrichment processor **webAPI**. The processor can now function both as an interactive user interface and as an API server executed by a virtual machine within the Transport Cloud. With this webAPI, users can remotely access the functionalities of the semantic enrichment processor using POST and GET HTTP requests. Additionally, the webAPI enables users to combine the semantic enrichment processor with other processors, hence enabling them to build their customized mobility data processing pipelines. The semantic enrichment processor webAPI has been documented in Section 2.3.

The second version of the demonstrator also showcases a graphically revamped user interface, and the use of a more advanced algorithm for detecting systematic stops. All these updates are presented in Section 2.2. Finally, a video showcasing the second version of the semantic enrichment demonstrator has been provided in the semantic enrichment processor's GitHub repository¹.

2.1. How to set up the semantic enrichment demonstrator

The semantic enrichment demonstrator consists of a set of Python scripts that make exclusively use of open-source libraries. In the following, we illustrate the installation procedure needed to execute the semantic enrichment demonstrator. The installation procedure has been tested on Windows 10, Ubuntu (version > 20.x), and macOS.

1. The first step requires installing a Python distribution that includes a package manager. To this end, we recommend installing Anaconda^{2 3}, a cross-platform Python package manager and environment-management system which satisfies the above criteria.
2. Once Anaconda has been installed, the next step requires setting *up a virtual environment* containing the open-source libraries that our demonstrator requires during its execution. To this end we provide a YAML file, `mat_builder.yml`, that can be used to set the environment up.

The user must first open an Anaconda PowerShell prompt. Then, the user must type in the prompt:

```
conda env create -f path\mat_builder.yml -n name_environment
```

¹ <https://github.com/MobiDataLab/mdl-semantic-enrichment/blob/main/misc/videos/Semantic%20enrichment%20processor%20demonstrator%20v2%20demo.mkv>

² Anaconda installers are available at <https://www.anaconda.com/products/distribution>

³ The user can alternatively install Miniconda, i.e., a minimal version of Anaconda. Miniconda is available at <https://docs.conda.io/en/latest/miniconda.html>

where `path` represents the path in which `mat_builder.yml` is located, while `name_environment` represents the name the user wants to assign to the virtual environment.

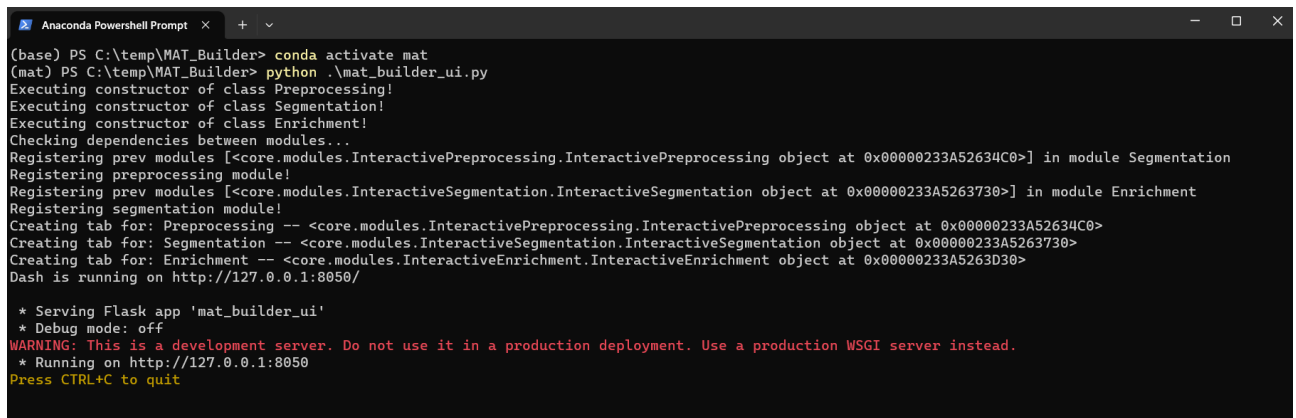
We report that the open-source libraries relevant to the demonstrator will also be relevant to the semantic enrichment processor. These libraries are *Pandas*⁴, *Geopandas*⁵, *scikit-learn*⁶, *scikit-mobility*⁷, *rdflib*⁸, *PTRAIL*⁹, and *Dash*¹⁰.

3. Once the environment has been created, the user must activate it in the prompt by typing `conda activate name_environment`.

The user is now able to execute and use the demonstrator.

2.2. How to use the interactive user interface

To run the semantic enrichment processor's interactive user interface, the user must first open an Anaconda PowerShell prompt and then, within the virtual environment created during the installation procedure, execute the Python script `mat_builder_ui.py`. Once executed, the script will tell the user the address at which the interactive user interface can be accessed through some web browser of preference (Figure 1).



```
(base) PS C:\temp\MAT_Builder> conda activate mat
(mat) PS C:\temp\MAT_Builder> python .\mat_builder_ui.py
Executing constructor of class Preprocessing!
Executing constructor of class Segmentation!
Executing constructor of class Enrichment!
Checking dependencies between modules...
Registering prev modules [<core.modules.InteractivePreprocessing.InteractivePreprocessing object at 0x00000233A52634C0>] in module Segmentation
Registering preprocessing module!
Registering prev modules [<core.modules.InteractiveSegmentation.InteractiveSegmentation object at 0x00000233A5263730>] in module Enrichment
Registering segmentation module!
Creating tab for: Preprocessing -- <core.modules.InteractivePreprocessing.InteractivePreprocessing object at 0x00000233A52634C0>
Creating tab for: Segmentation -- <core.modules.InteractiveSegmentation.InteractiveSegmentation object at 0x00000233A5263730>
Creating tab for: Enrichment -- <core.modules.InteractiveEnrichment.InteractiveEnrichment object at 0x00000233A5263D30>
Dash is running on http://127.0.0.1:8050/

* Serving Flask app 'mat_builder_ui'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:8050
Press CTRL+C to quit
```

Figure 1: Execution of the script `mat_builder_ui.py`

⁴ <https://pandas.pydata.org/>
⁵ <https://geopandas.org/en/stable/>
⁶ <https://scikit-learn.org/stable/>
⁷ <https://github.com/scikit-mobility/scikit-mobility>
⁸ <https://rdflib.readthedocs.io/en/stable/>
⁹ <https://github.com/YakshHaranwala/PTRAIL>
¹⁰ <https://plotly.com/dash/>

The user can then proceed to open a web browser and input the provided address: once this is done, the user interface of the demonstrator will appear.

The backend of the demonstrator is organized in three distinct modules, *i.e.*, *pre-processing*, *segmentation*, and *enrichment* (Figure 2). These modules must be executed in this order to compute the final dataset of multiple aspect trajectories. In the following, we illustrate what each of these modules does, and how the user interface guides the user during the enrichment process.

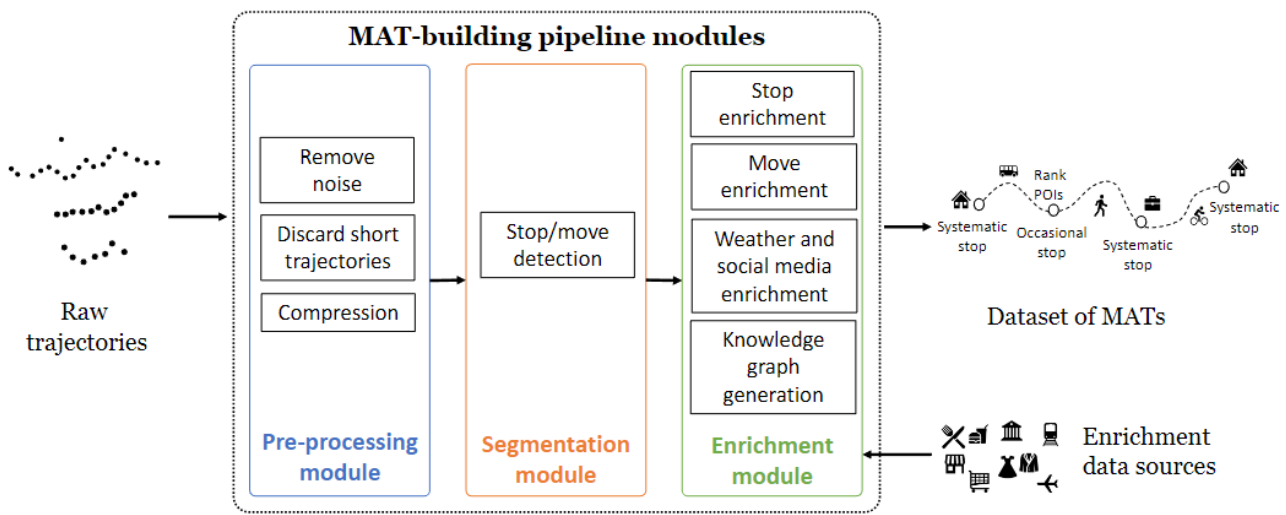


Figure 2: Demonstrator backend.

2.2.1. Pre-processing module

The first tab allows the user to access the functionalities offered by the *pre-processing* module (blue block in Figure 2, while Figure 3 shows the related tab shown by the UI). The goal of this module is to take into input a dataset of raw trajectories and produce a dataset of pre-processed trajectories that can be subsequently enriched.

The operations conducted by the pre-processing module are the following: (1) filtering out noisy or unusable data, (2) discarding trajectories that have an insufficient sampling rate, (3) filtering out outliers in the trajectories by analyzing their spatio-temporal characteristics, and finally (4) compressing the trajectories.

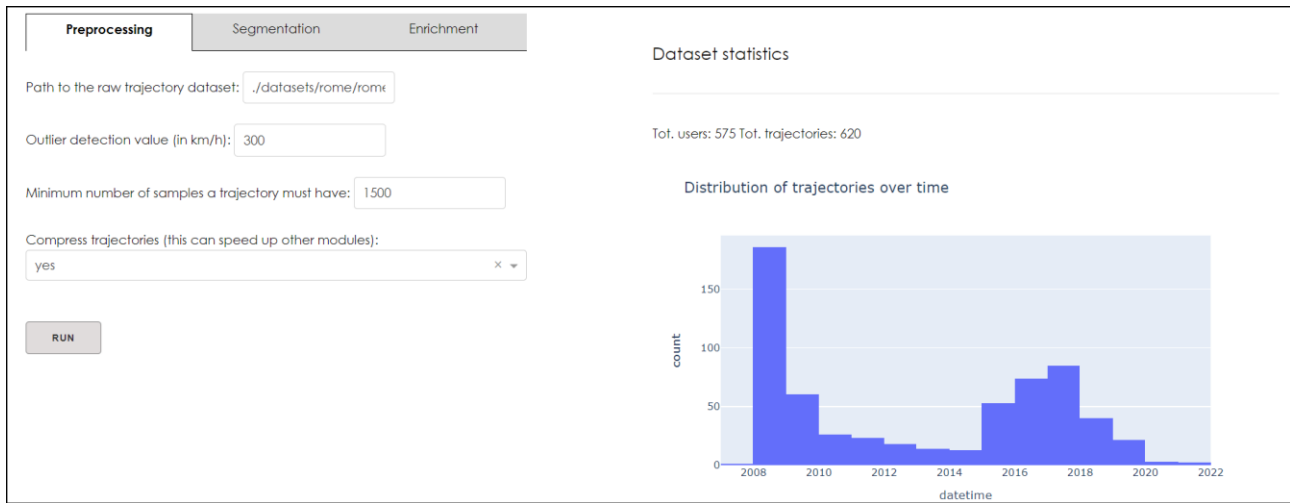


Figure 3: pre-processing module, as shown in the UI.

From Figure 3, left side, we see how the pre-processing tab allows the user to *input* the *raw trajectory dataset* they want to enrich. In Section 5.1.1 we provide the specifications that the file must follow to be recognized and used by this module.

The tab lets the user customize some of the pre-processing operations, *i.e.*, the user can specify the *minimum number* of points a trajectory should have and a *km/h threshold* between two consecutive points that the pre-processing module uses to filter out outliers. Once the raw trajectories have been pre-processed, the user interface presents some statistics gathered during this step (right side of Figure 3). The demonstrator also outputs a file containing the pre-processed trajectories named `traj_cleaned.parquet`. Such a file follows the format specified in Section 5.1.2.

2.2.2. Trajectory segmentation module

Once the raw trajectories are pre-processed, the user interface activates the segmentation module tab (orange block in Figure 2, while Figure 4 shows the related tab in the UI). The goal of the segmentation module to take in input a dataset of pre-processed trajectories and partition each one of them into sub-trajectories (*i.e.*, *segments*). The segmentation module uses a well-known and widely used segmentation criterion, *i.e.*, that of the stop and move (Spaccapietra *et al.* 2008) made available by the *scikit-mobility library*, one of the fundamental open-source libraries that our demonstrator uses.

The final output of the segmentation module consists of a set of segmented trajectories, which can then be processed by the enrichment module. Such output is saved in two distinct files named `stops.parquet` and `moves.parquet`. These files follow the specification provided in Section 5.1.3.

Preprocessing **Segmentation** Enrichment

Minimum duration of a stop (in minutes):

Maximum spatial radius (in meters) of a stop:

RUN

Overall number of moves found: 435

Overall number of stops found: 629

Select a user for more specific information:

Number of trajectories found for this user: 2

Number of moves found for this user: 3

Move average duration: 14.62 minutes

Number of stops found for this user: 3

Stop average duration: 12.49 minutes

Figure 4: Trajectory segmentation module, as shown in the UI.

Going back to the user interface, the segmentation tab lets the user specify the *minimum duration* and the *spatial radius* the demonstrator will use to identify the stop segments (and, indirectly, the move ones). Once the trajectories have been segmented, the user interface will activate a drop-down menu (right side of Figure 4) which can be used to summarily examine information concerning the stops found for each user.

2.2.3. Enrichment module

Once the trajectories are segmented, the user interface activates the enrichment module tab (green block in Figure 2, while Figure 5 shows the related tab shown by the UI). This module takes as input the output of the segmentation module and identifies the different segments to enrich, the aspects to consider, the datasets to be used to enrich the segments with different aspects, and the enrichment criteria.

Preprocessing

Segmentation

Enrichment

Move enrichment

Add transportation means to moves?

yes

Regularity aspect: systematic and occasional stops

Distance below which a stop can be included in a systematic stop (DBSCAN epsilon parameter): 50

Minimum size of a cluster of systematic stops (DBSCAN minPts parameter): 5

Stop augmentation with POIs

Download from OpenStreetMap the POIs of location: Rome, Italy

Poi categories (considered only when downloading from OpenStreetMap)

amenity

... or provide a path to a POI dataset (write 'no' to use OSM)

./datasets/rome/poi/

Maximum distance from the centroid of the stops (in meters): 50

Enrich trajectory users with social media posts:

Path to file containing the posts (write 'no' if no enrichment should be done):

./datasets/rome/tweet

Enrich trajectories with weather information:

Path to file containing the posts (write 'no' if no enrichment should be done):

./datasets/rome/wea

Save the enriched trajectories into an RDF graph:

yes

RUN

User: 417

General characteristics of the user's trajectories:

Number of trajectories: 2

Temporal interval spanned: [2008-02-05 12:46:57, 2008-02-05 14:18:24]

Average duration trajectories: 0 days 00:35:50

Average sampling rate trajectories: 0 days 00:03:22.481818181

Average gap between trajectories: 0 days 00:19:47

Regularity aspect

Number of systematic stops: 0

Number of occasional stops: 3

Move aspect (transportation means and duration):

Walk: 0

Bike: 0

Bus: 0 days 00:41:37

Car: 0

Train: 0

Subway: 0 days 00:19:28

Taxi: 0

Overall distribution of the systematic and occasional stops:

Types of stops

● occasional stops

Trajectory plotter (choose one from the dropdown menu):

Select...

Figure 5: Enrichment module, as shown in the UI.

The first aspect the segment enrichment module adds is the *transportation means* associated with each move. To this end, the segment enrichment module leverages a random-forest classifier that has been pre-built with the scikit-learn library and trained on the GeoLife dataset (Zheng et al 2010). The classifier recognizes the following transportation means: *walk*, *car*, *bike*, *bus*, *subway*, and *train*.

We report that the code of the demonstrator is general enough to include different transportation means inference methods, and it will be the object of future works to include more methods and transportation means.

The segment enrichment module then goes on to enrich each stop segment with the *regularity* aspect, i.e., whether a stop belongs to a *systematic*¹¹ stop or an *occasional* one. The module distinguishes between these two types of stops by leveraging the DBSCAN algorithm (Birant, D. et al, 2007): essentially, stops that tend to repeatedly gather around specific locations are considered systematic, while the other ones are considered occasional.

Here, the user must provide two parameter values, i.e., (1) the maximum distance (in meters) within which two stops are deemed neighbours and (2) the minimum count of neighbouring stops needed for a stop to being tagged as a core point, thereby forming an initial cluster. Once clusters are found, the module considers the stops falling within them as *systematic*, and proceeds to augment each cluster with an *activity*, which is established according to a pre-determined set of temporal criteria. In the current version of the demonstrator, such activities are *home* (a cluster of stops that tend to occur during the weekend or outside working hours), *work* (a cluster of stops that tend to occur during the working days and in working hours), or *other*. Again, the demonstrator is general enough to be extended to different activities and this is indeed the object for future work.

Next, the enrichment module can associate POIs to stops. This is reflected in the user interface since the user can retrieve a dataset of POIs either from OpenStreetMap¹² – in this case, the user must specify (1) the name of the city where the trajectories are located and (2) the POI types the user wants to use to enrich the trajectories – or from a file containing the POI dataset to be used. In Section 5.1.4 we provide the specifications that such file must follow to be recognized and used by the demonstrator.

Once a POI dataset is provided, the enrichment module decides which POIs should be used to enrich the stops by ranking them according to distance and temporal overlap criteria.

The module can also enrich trajectories with the *weather information* aspect, and the moving objects generating the trajectories with the *social media* aspect. In this case, the user must provide the paths to the files containing the respective datasets. In Sections 5.1.5 and 5.1.6 we provide the specifications that said files must follow to be recognized and used by the module.

Finally, the enrichment module allows the user to save the output of the whole enrichment process in an RDF graph. The content within the graph follows the schema defined by CNR's customized version of the STEP ontology (Nogueira *et al.* 2018) and it is saved to disk following the Turtle format. Note that, by using this format, the graph can be easily imported into popular triple stores (e.g., GraphDB) for further analysis and query processing. Section 5.1.7 provides the details on the customizations we did to the original STEP ontology for this demonstrator, while Section 2.4 provides an example of the content that can be found inside an RDF knowledge graph generated by the enrichment module.

¹¹ A systematic stop represents a set of stops that fall within the same area more than a given number of times. A few examples of systematic stops can be a person's home, work, gym, and so on.

¹² OpenStreetMap: <https://www.openstreetmap.org/>

2.3. How to use the webAPI

The semantic enrichment processor webAPI has been developed with FastAPI¹³, a modern, fast, open-source web framework for building APIs with Python 3.6+ based on standard Python type hints. To use the processor's webAPI, one must first execute the webAPI server via the python script `mat_builder_api.py`: once running, the server will accept HTTP POST and GET requests from users. The server exposes three different endpoints, providing functionalities of the preprocessing, segmentation, and enrichment modules shown in Section 2.2. If `baseaddress` is the server's base address, then the endpoints are reachable at:

- **Preprocessing endpoint** => `baseaddress/Preprocessing`
- **Segmentation endpoint** => `baseaddress/Segmentation`
- **Enrichment endpoint** => `baseaddress/Enrichment`

The server listens for HTTP POST and GET requests sent by users to these endpoints. POST requests enable users to initiate preprocessing, segmentation, enrichment tasks, while GET requests permit users to actively monitor the tasks execution status and retrieve the tasks results once they are available. The documentation concerning the use of the endpoints can be accessed at `baseaddress/docs`.

Finally, the Python script `examples_api_request.py` provides a complete and extensively commented example that shows how to make requests to the demonstrator webAPI to enrich a dataset of raw trajectories. In the following we give an overview on how to use the endpoints via HTTP POST and GET requests.

Preprocessing endpoint. As in the case of the preprocessing module exposed by the interactive user interface, the preprocessing endpoint exposes the module's functionalities needed to preprocess a dataset of raw trajectories.

The user can initiate a preprocessing task by sending an HTTP POST request. In this request, the user must send:

- the raw trajectory dataset, *file_trajectories*, which must be appropriately formatted (as specified in Section 5.1.1) and stored in a Parquet file;
- the query parameter *min_num_samples*, which represents the minimum number of samples a trajectory must have;
- the query parameter *max_speed*, which represents the maximum speed allowed in any trajectory (in km/h);

¹³ <https://fastapi.tiangolo.com/>

- the query parameter *compress_trajectories*, which is a Boolean value indicating whether the trajectories should be compressed or not.

The server can reply to such a request in two different ways:

- If the request can be processed, the server will send a reply with the HTTP code 200 and a JSON body containing two parameters, i.e., the *message* parameter containing a message from the server and the *task_id* parameter. The *task_id* parameter contains the unique identifier the user must use to monitor their preprocessing task execution and, when the task has ended, retrieve the results.
- If the request cannot be processed, for example in the case the user sent a misformatted request, the server will reply with the HTTP code 422.

The user can then monitor the status of their preprocessing task via HTTP GET requests. In these requests, the user must provide in input the *task_id* query parameter, which should contain the task identifier previously sent by the server in response to their (successful) POST request. The server can reply in three different ways:

- If the preprocessing task has terminated, the server will answer with a reply with HTTP code 200, and a parquet file containing the preprocessed trajectory dataset, as per the specifications outlined in Section 5.1.2.
- If the request cannot be processed, for example in the case the user sent a misformatted request, the server will reply with the HTTP code 422.
- If some internal error occurred during the preprocessing task, the server will answer with a reply with HTTP code 500.

Segmentation endpoint. As in the case of the segmentation module exposed by the interactive user interface, the segmentation endpoint exposes the module's functionalities needed to segment a dataset of trajectories.

The user can initiate a segmentation task by sending an HTTP POST request. In this request, the user must send:

- the trajectory dataset, *file_trajectories*, which can be preprocessed or not, must be appropriately formatted (as specified in Section 5.1.2), and must be stored in a Parquet file;
- the query parameter *min_duration_stop*, which provides the minimum duration a stop must have (in minutes);
- the query parameter *max_stop_radius*, which provides the maximum radius a stop can have (in km).

The server can reply to such a request in two different ways:

- If the request can be processed, the server will send a reply with the HTTP code 200 and a JSON body containing two parameters, i.e., the *message* parameter containing a message from the server and the *task_id* parameter. The *task_id* parameter contains the unique identifier the user must use to monitor their segmentation task execution and, when the task has ended, retrieve the results.
- If the request cannot be processed, for example in the case the user sent a malformed request, the server will reply with the HTTP code 422.

The user can then monitor the status of their segmentation task via HTTP GET requests. In these requests, the user must provide in input the *task_id* parameter previously sent by the server in response to their (successful) POST request. The server can reply in three different ways:

- If the segmentation task has terminated, the server will answer with a reply with HTTP code 200, and a JSON body containing the set of stop and move segments detected for the trajectory dataset. The JSON body consists of a dictionary containing two dictionaries: one containing the set of stop segments, and the other containing the set of move segments. Both dictionaries are internally structured according to the specifications provided in Section 5.1.3.
- If the request cannot be processed, for example in the case the user sent a misformatted request, the server will reply with the HTTP code 422.
- If some internal error occurred during the segmentation task, the server will answer with a reply with HTTP code 500.

Enrichment endpoint. As in the case of the enrichment module exposed by the interactive user interface, the enrichment endpoint exposes the modules' functionalities needed to enrich a dataset of segmented trajectories. The user can initiate an enrichment task by sending an HTTP POST request. In this request, the user must send:

- The trajectory dataset, *file_trajectories*, which must be appropriately formatted (as specified in Section 5.1.2) and stored in a Parquet file;
- The move segment dataset, *file_moves*, generated by the segmentation module;
- The stop segment datasets, *file_stops*, generated by the segmentation module;
- The POI dataset, *file_poi*, containing the POIs to possibly associate with the stop segments. Note that the dataset must be structured according to the specifications provided in Section 5.1.4, and must be stored in a Parquet file;
- The social media post dataset, *file_social*, containing the social media post that can be used to possibly enrich the moving objects that generated the trajectories. Note that the dataset must be structured according to the specifications provided in Section 5.1.6, and must be stored in a Parquet file;

- The weather conditions dataset, *file_weather*, containing weather data that can be used to enrich the trajectories. Note that the dataset must be structured according to the specifications provided in Section 5.1.5, and must be stored in a Parquet file;
- The query parameter *move_enrichment*, representing a Boolean value indicating whether the move segments should be enriched or not with the estimated transportation means;
- The query parameter *max_dist*, representing the maximum distance beyond which a POI won't be associated with a stop segment;
- The query parameter *epsilon_distance*, which is used by the DBSCAN algorithm to cluster stop segments (and thus find systematic stops). It represents the distance (in meters) below which a stop can be included in an existing cluster;
- The query parameter *systematic_threshold*, which represents the minimum size a cluster of stops must have to be considered a cluster of systematic stops.

The server can reply to such a request in two different ways:

- If the request can be processed, the server will send a reply with the HTTP code 200 and a JSON body containing two parameters, i.e., the *message* parameter containing a message from the server and the *task_id* parameter. The *task_id* parameter contains the unique identifier the user must use to monitor their enrichment task execution and, when the task has ended, retrieve the results.
- If the request cannot be processed, for example in the case the user sent a misformatted request, the server will reply with the HTTP code 422.

The user can then monitor the status of their enrichment task via HTTP GET requests. In these requests, the user must provide in input the *task_id* parameter previously sent by the server in response to their (successful) POST request. The server can reply in three different ways:

- If the enrichment task has terminated, the server will answer with a reply with HTTP code 200, and a parquet file containing the RDF knowledge graph, stored in a Turtle-formatted file, containing the dataset of semantically enriched trajectories. An example of content within an RDF graph is provided in Section 2.4, while Section 5.1.7 provides the details on the STEPv2 ontology used to structure the information within the graphs.
- If the request cannot be processed, for example in the case the user sent a misformatted request, the server will reply with the HTTP code 422.
- If some internal error occurred during the enrichment task, the server will answer with a reply with HTTP code 500.

2.4. Example of content within an RDF graph

Let us conclude by providing a visual example of what can be found in an RDF graph generated by the demonstrator, either via the interactive user interface or the webAPI. To this end, we use GraphDB, a very well-known and established triple store. We first import into the store an RDF graph that has been generated from trajectories and data covering Rome and then use GraphDB's visual inspection functionality to navigate the graph.

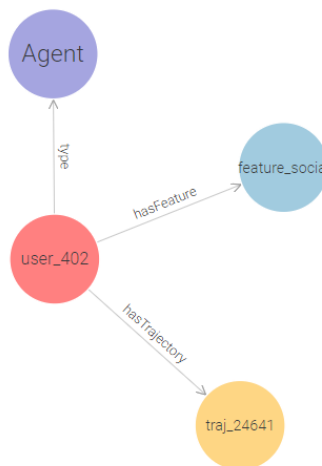


Figure 6: initial view of the subgraph associated with the user ID 402

In Figure 6 we see a (collapsed) subgraph within the RDF graph related to the user having ID 402. From the figure, we see that the user has associated a social media aspect (the cyan “feature_social” node) and a trajectory with ID 24641 (the yellow node). Let us expand the social media aspect node first.

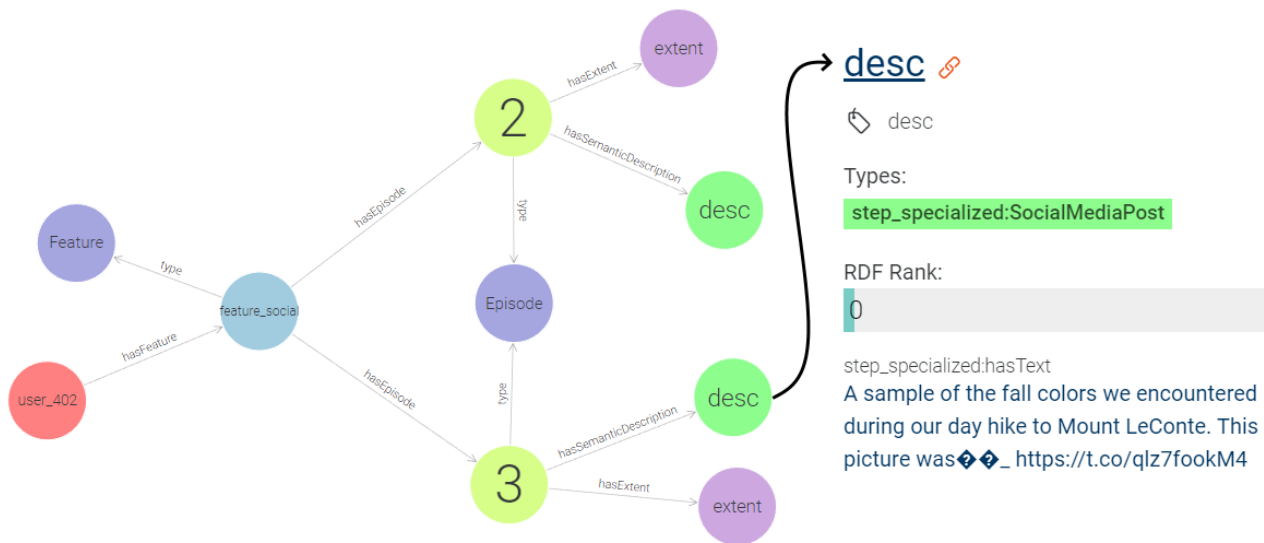


Figure 7: Social Media Post aspect.

From Figure 7 we see that the user is associated with two distinct episodes of the *social media post* aspect – in other words, the demonstrator found from the information that has been provided that the user has published two different tweets. If we then look at the content of one of their tweets (this is done by clicking on one of the “desc” nodes) we can see the text they have posted in one of their tweets. Note that every episode node is also in a relationship with a node of type “TemporalExtent” (the violet nodes in the Figure), each representing the time instant at which the post has been published.

Let us now focus on the trajectories that user 402 possesses. From Figure 6 we see that the user is associated with a single trajectory with ID 24641. Let us then focus on such a trajectory by expanding the subgraph rooted in that node. From Figure 8 we see that the trajectory node is in relationship with several nodes. First, observe that the trajectory is always in relationship with a “RawTrajectory” node, which in turn is in relationship with a set of nodes (i.e., the *fixes*) representing the samples (i.e., pairs (*position*, *time*)) making up the trajectory.

The trajectory has also been associated by the demonstrator with two different semantic aspects, i.e., the *moves* and the *occasional stops*. For what concerns the move aspect, we observe that the demonstrator has detected 10 different move episodes, each having a specific spatiotemporal extent and a semantic descriptor providing information on the transportation means that have been used during the move. For what concerns the occasional stops we observe that the demonstrator has detected 9 different episodes, with each episode having again a specific spatiotemporal extent and a semantic descriptor providing information on potential points of interest that the user may have visited during the stop.

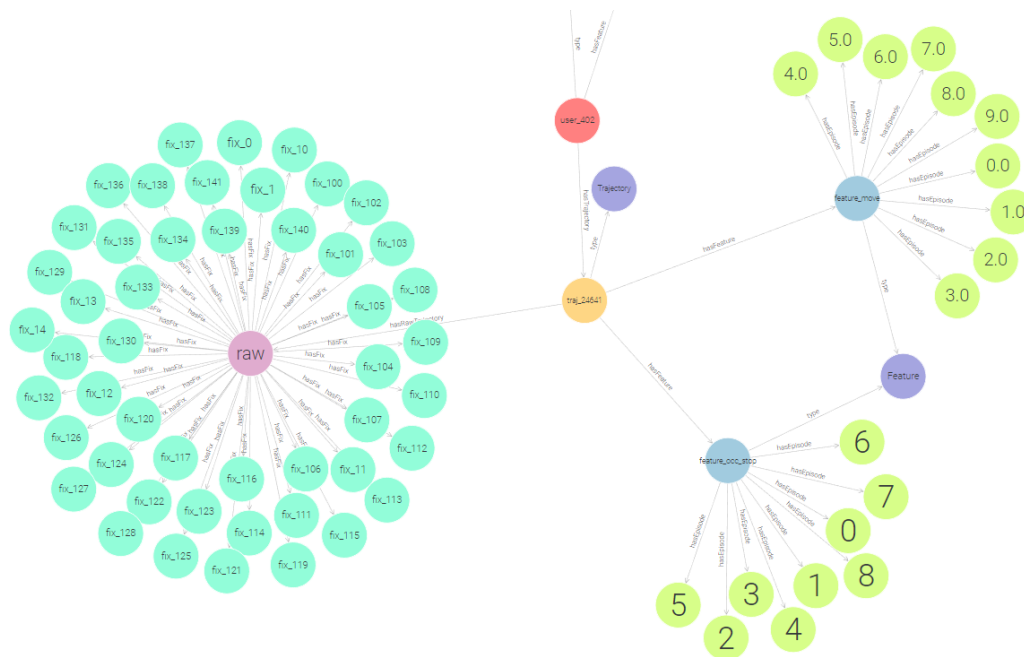


Figure 8: The subgraph rooted in the node related to the trajectory 224641 of the user 402.

2.5. How to tackle the Paris Challenge with the aid of the semantic enrichment processor

The MOBIDATALAB Hackathon, which was held in Paris, France, between the 15th and 16th September 2023, introduced a challenge proposed by the city of Paris' municipality. The challenge originates from relevant issues concerning the city's walkability that the municipality intends to address.

More specifically, the challenge focuses “[...] around the creation of a tool that facilitates the improvement of the walking environment in Paris, utilizing data-driven strategies without the need for on-site visits. With limited walking data available, participants are expected to employ data enrichment methods to model pedestrian behavior. The tool should then leverage this enriched data to analyze and enhance the walking experience by identifying and addressing issues such as identifying pedestrianization opportunities, sidewalk extensions, and potential conflicts with other users.”

Moreover, we report that in the scope of this challenge the MOBIDATALAB project was able to provide access to a dataset of anonymized raw GPS trajectories generated by individuals moving in the city of Paris; of particular interest is that part of these trajectories have been generated by individuals who were walking by foot, using public transportation, or both, thus potentially providing insights on the pedestrians' movement behaviours and possibly enabling to reason on how the walkability of the city can be improved.

Recall that the semantic enrichment processor has been designed to semantically enrich raw GPS trajectories: as such, we argue that the processor can be used to facilitate the implementation of the tool required to solve part of the issues characterizing the challenge.

First, observe that the semantic enrichment process can detect *stops*, i.e., when objects remain stationary at some location for some time, and *moves*, i.e., when objects are moving between stops. Moreover, recall that the semantic enrichment processor can augment stops with the POIs located close to the stops' centroids, and it can augment moves with the transportation means that have been likely used. Finally, recall that the semantic enrichment processor can enrich trajectories with weather conditions (i.e., the *weather* aspect), and trajectory users with the social media post they have published (i.e., the *social media* aspect).

With such information at hand, one can then develop a tool that analyzes where **stops** occur to try to identify destinations or points of interest that are popular among pedestrians. Knowing the POIs around which stops are frequently made can help identify key areas where pedestrians prefer to spend time. This can inform decisions about where to expand pedestrian zones or improve amenities. Moreover, by examining the duration and location of stops, one can help pinpoint areas where pedestrians experience delays or obstacles, which could indicate the need for sidewalk extensions or improved pedestrian facilities. Conversely, understanding the **moves** can reveal the most popular walking routes and connections between different areas of the city. This information is vital in crafting a walking strategy as it tells the city where to focus efforts on improving walkability, e.g., where to focus on improving walkways or creating new pedestrian paths.

Incorporating **weather data** can enrich pedestrian behaviour models by highlighting how weather affects walking habits. For instance, one might find that certain routes are less travelled during poor weather, indicating a need for weather-protected walking paths or improved drainage to reduce puddling on sidewalks. Furthermore, knowing how weather impacts pedestrian flow can assist in planning for seasonal changes or in the design of spaces that can adapt to different weather conditions, thereby improving the overall walking experience in the city.

Social media information can provide real-time data on pedestrian opinions and experiences. By analyzing geotagged posts, one can gauge public sentiment about certain walking areas and identify spots that are either problematic or well-liked. Moreover, social media data can also offer insights into cultural events or spontaneous gatherings that affect pedestrian traffic. This helps in aligning urban planning objectives with the ways people naturally use and enjoy the city's spaces.

Overall, integrating all the various semantic dimensions – stops augmented with POIs, moves, weather conditions, and social media posts – one can provide a holistic view of pedestrian behaviour. This analysis can help identify the most impactful improvements, such as better lighting around popular POIs or more benches where there are frequent stops. Moreover, the enriched data might be used to simulate changes to the walking environment and predict how these changes might affect pedestrian behaviour. For example, one might forecast the impact of a new pedestrian zone on foot traffic patterns and POI visits.

3. Geographical enrichment of mobility data (v2 demonstrator)

3.1. How to build the geographical enrichment demonstrator

MDL-Geo-Enrichment is a web application providing many enrichment APIs, it was created using Spring Boot framework and some other Java and JavaScript libraries.

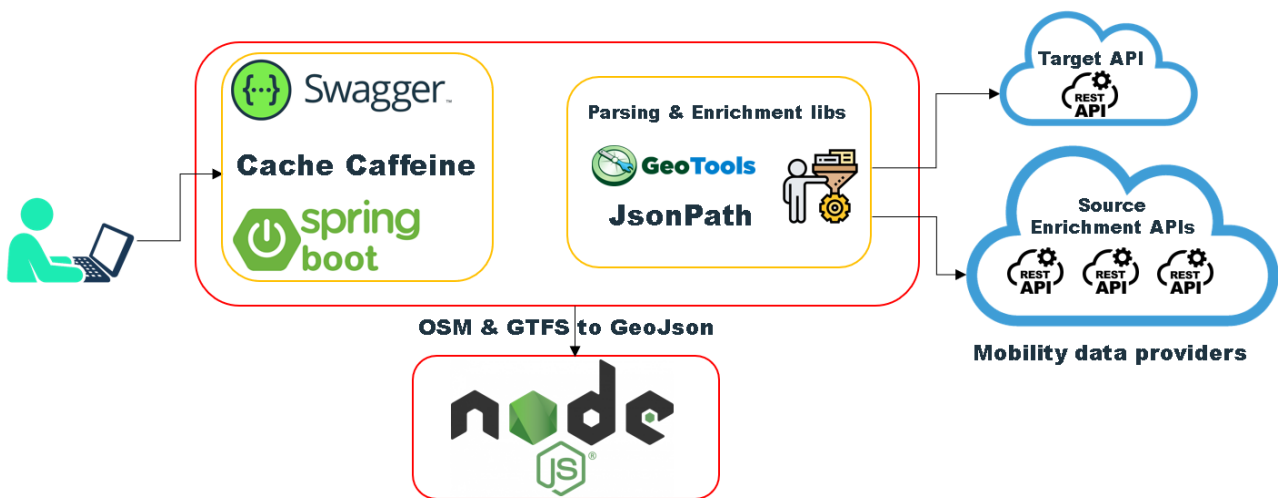


Figure 9: The Mobility data enrichment architecture

There are 3 ways to get the application built and deployed:

- Manual build
- Build and deploy through Docker
- Build and deploy through Travis

3.1.1. Manual build and dependencies installation

✓ Prerequisites:

OpenJDK 11:

You can get OpenJDK from <https://adoptopenjdk.net/>

Maven:

You can get Maven using the following guide <https://maven.apache.org/install.html>

Node.js:

You can get Node.js from <https://nodejs.org/en/download/>

OsmToGeoJson module:

After installing NodeJS, you can install OsmToGeoJson as a global module with the following command:

```
$ npm install -g osmtogeojson
```

GtfsToGeoJson module:

You need to install also Gtfs-to-GeoJson as a global module with the following command:

```
$ npm install -g gtfs-to-geojson
```

✓ Download source code:

You can download the repository as an archive file using the download menu on:

<https://github.com/MobiDataLab/mdl-geo-enrichment>

Or you can use Git (if installed) to clone the repository:

```
$ git clone https://github.com/MobiDataLab/mdl-geo-enrichment.git
```

✓ Build and package application:

You can build and package the application using maven:

```
$ mvn package -DskipTests
```

Then you can run the application with the built-in web server:

Using maven:

```
$ mvn spring-boot:run
```

Or by running the standalone java archive file:

```
$ java -jar target/mdl-geo-enrichment-0.0.1-SNAPSHOT.jar
```

You can specify the profile you want by adding the parameter to the above commands:

```
-Dspring.profiles.active=prod
```

There are 3 profiles: dev, integration-test, and prod.

You can customize them on the “resources/ application.yml” file.

3.1.2. Build and run as a docker image through Gitlab-CI

You can build a docker image bundled with all the application dependencies out of the box, this is done through Gitlab-CI:

✓ Prerequisites:

Gitlab Runner having Docker installed and running

You can follow the instructions on Gitlab's project: Settings / CI-CD à Runners

Gitlab-CI and docker configuration are available on the files: ".gitlab-ci.yml" and "Dockerfile".

Before creating a docker image, you need to:

- Set docker's registry account credentials on Gitlab CI/CD variables settings: CI_REGISTRY_USER and CI_REGISTRY_PASSWORD
- Put your Gitlab runner name on the "tags" attribute of ".gitlab-ci.yml"

To build an image based on a tag or the main branch, you can execute "Run pipeline" on the side menu "Pipelines" and choose the branch or tag you want to build the image upon.

Pipeline

Needs

Jobs 4

Tests 0

Status	Job	Stage	Name	Duration	Coverage
<div>passed</div>	<div>#2587159381</div> <div>main → eed03eed</div> <div>ekniba-local</div>	deploy	deploy-job	<div>00:02:08</div> <div>2 days ago</div>	<div></div>
<div>passed</div>	<div>#2587159379</div> <div>main → eed03eed</div> <div>ekniba-local</div>	package	package-job	<div>00:01:20</div> <div>2 days ago</div>	<div></div>
<div>passed</div>	<div>#2587159378</div> <div>main → eed03eed</div> <div>ekniba-local</div>	test	unit-test-job	<div>00:01:47</div> <div>2 days ago</div>	<div></div>
<div>passed</div>	<div>#2587159376</div> <div>main → eed03eed</div> <div>ekniba-local</div>	build	build-job	<div>00:01:10</div> <div>2 days ago</div>	<div></div>

Figure 10: MDL-Geo-Enrichment pipeline on Gitlab-CI

4 stages are executed:

- **Build:** it compiles the application
- **Test:** it runs integration tests, an artifact containing the binaries and test reports is saved and can be downloaded for further analysis when the non-regression tests fail.
- **Package:** it builds the package of the application to be deployed
- **Deploy:** it creates a minimal docker image based Alpine with the required dependencies (OpenJDK11-JRE, Nodejs, Osm2GeoJson module) and uploads it to the docker registry.

3.1.3. Build and run as a docker image through Travis-CI

You can use Github with Travis-CI to build a docker image the same way it's done with Gitlab-CI:

✓ Prerequisites:

Github + Travis CI account

You can log in to Travis-CI with your Github account and grant the repository access to Travis so it can automatically trigger build jobs and fetch the source code, otherwise, you will need to use [Travis-ci Api](#) to manage job scheduling.

Travis configuration is available on the file: ".travis.yml" and Docker configuration remains on the same file as for Gitlab-CI: "Dockerfile".

Before creating a docker image, you need to set the official or corporate docker hub registry account credentials on Travis-CI settings/environments variables: CI_REGISTRY_USER and CI_REGISTRY_PASSWORD and make it visible only for the main branch

Environment Variables

Customize your build using environment variables. For secure tips on generating private keys [read our documentation](#)

CI_REGISTRY_PASSWORD	••••••••••	Only available to the main branch	🗑
CI_REGISTRY_USER	••••••••••	Only available to the main branch	🗑

Figure 11: Travis-CI's environment variables

To build an image based on a tag or the main branch, you can click on "Trigger build" on the side menu "More options" and choose the branch or tag you want to build upon.

Figure 12: MDL-Geo-Enrichment pipeline on Travis CI

4 stages are executed:

- **Build:** it compiles the application
- **Test:** it runs integration tests, an artifact containing the binaries and test reports is saved and can be downloaded for further analysis when the non-regression tests fail.
- **Package:** it builds the package of the application to be deployed
- **Deploy:** it creates a minimal docker image based Alpine with the required dependencies (OpenJDK11-JRE, Nodejs, Osm2GeoJson module) and uploads it to the docker registry.

3.1.4. *Pull and run MDL-Geo-Enrichment docker image*

You must have docker installed and running, if you don't have it installed, you can follow [this guide](#) to install docker.

Then you can get the docker image by running the following command as an admin/root user:

```
$ docker login DOCKER_REGISTRY -u USER_NAME --password-stdin
$ docker pull DOCKER_REGISTRY/PROJECT/mdl-geo-enrichment:TAG
$ docker run -d -p 80:80 -p 443:443 registry.gitlab.com/PROJECT/mdl-geo-enrichment:TAG
```

- ✓ USER_NAME is the username of your GitLab's registry credentials, you will be prompted to enter your credentials password.
- ✓ PROJECT is the project name where the mdl-geo-enrichment repository is hosted
- ✓ TAG is the tag version or by default "latest"
- ✓ DOCKER_REGISTRY (ex: registry.gitlab.com) is the docker registry used to upload docker images (keep it empty if the image you are pulling is hosted on the official docker hub)

The later command exposes both HTTP and HTTPS ports on the docker container, a self-signed certificate is included for the TLS layer, but you may still need to manually accept the certificate on your browser since it is not signed by a known authority.

Once the server is up, you can browse Swagger UI through <http://SERVER/swagger-ui/>

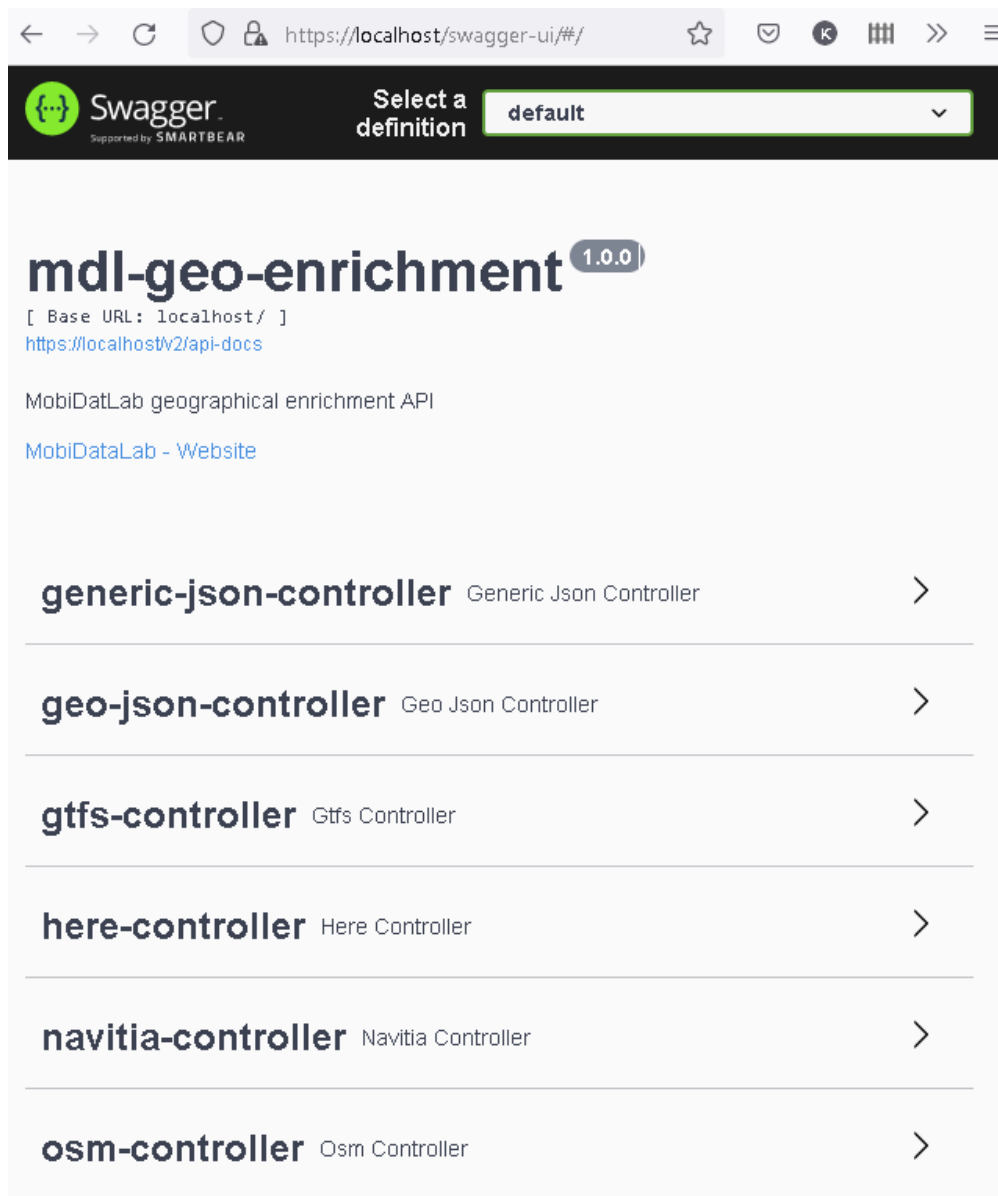


Figure 13: Swagger UI – MDL-Geo-Enrichment API list

3.2. How to use the geographical enrichment demonstrator

The mobility data mashup API is a Rest API that consumes a target API and enriches it with additional attributes extracted from a source API and produces the same format as the target API.

Here is a sequence diagram illustrating the enrichment process:

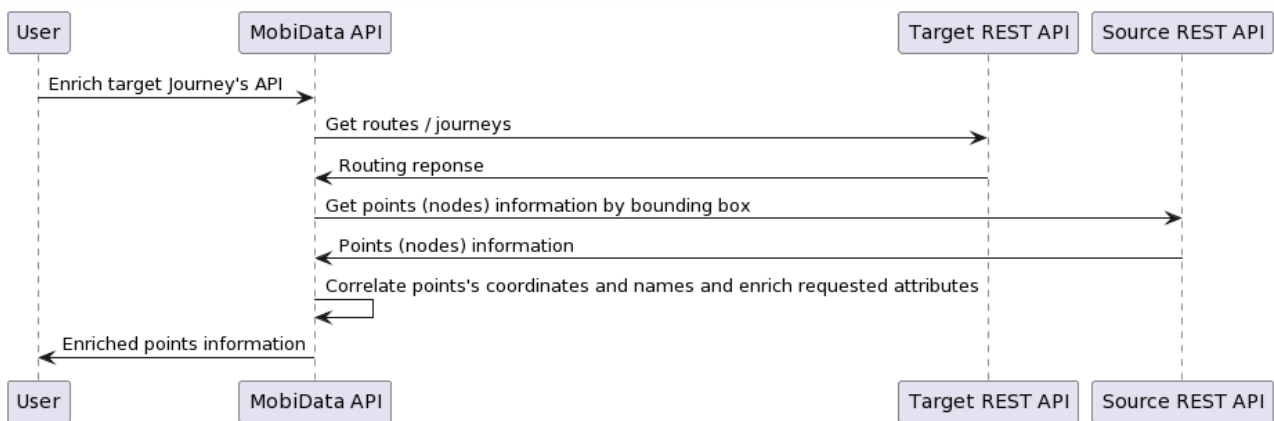


Figure 14: Sequence diagram of the enrichment process

The geographical enrichment demonstrator provides 6 examples of API enrichment, we use Navitia and Here's APIs as a target APIs to be enriched, unfortunately, those providers and many others use their proprietary format.

So, as an example of proprietary data format, we use Navitia and Here API as target REST APIs to be enriched, and we enrich them with any provider supporting one of the 3 following standards formats:

- **OSM:** OpenStreetMap, [Overpass output format](#)
- **GeoJson:** [Geospatial data interchange format](#)
- **GTFS:** [General transit feed specification format](#)

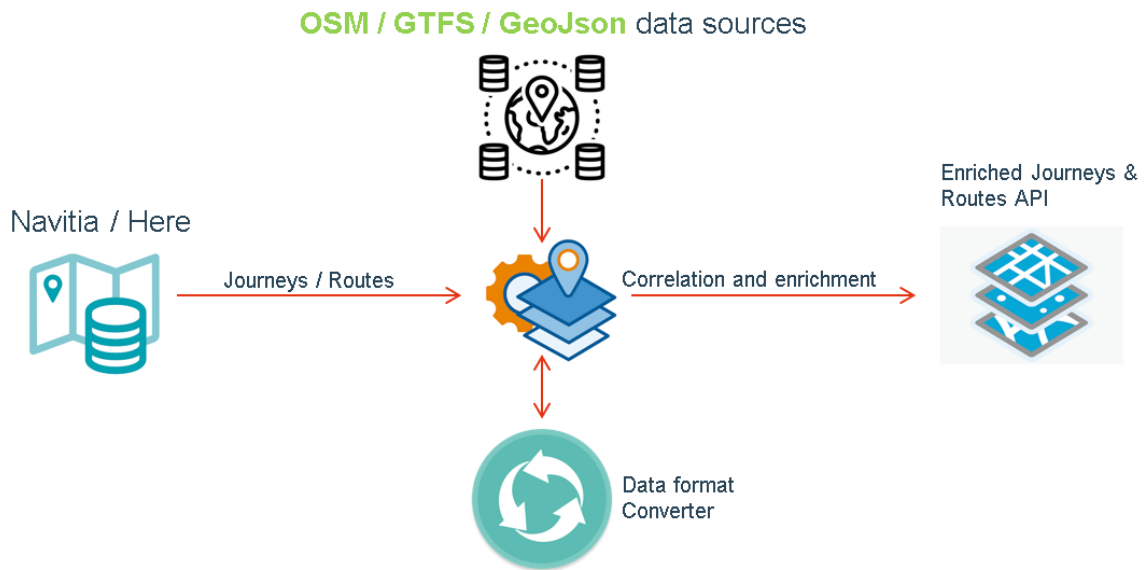


Figure 15: Sequence diagram of the enrichment process

The correlation of the nodes (stop point) is done using the open-source GIS toolkit [GeoTools](#), based on the coordinates and the name of the nodes.

Here is the list of the implemented APIs.

3.2.1. Here API enrichment:

We implemented 2 endpoints to demonstrate enrichment of Here stations and routes services:

- /api/v1/here/getNearStations
- /api/v1/here/getRoutes

The 3 open standard data types OSM, GeoJson, and GTFS are used to enrich the stop points with additional information such as accessibility, weather, air quality, etc.

Here is an example of how to the API through Swagger UI:

GET /api/v1/here/getNearStations getNearStations

Parameters Cancel

Name	Description
apiFormat * required string (query)	API format GTFS
apiKey string (query)	Here API authorization key 0PMpb1W_5iihYGu7UrBWsr8f6Utopf52hFB
apiUrl * required string (query)	API full url https://overpass.kumi.systems/api/interpreter?
coordinates * required string (query)	Coordinates of starting point: latitude, longitude 48.876892,2.352623
enrichAttributes string (query)	Attributes to be enriched on the target api, separated with commas wheelchair, shelter, tactile_paving, bench, bin,
sourceToken string (query)	Source API authorization token sourceToken - Source API authorization token

Execute

Figure 16: Here stations API enrichment

Following are the input parameters required by this API:

Table 1: Here API parameters

Parameter	Mandatory	Meaning
apiFormat	true	Provider data format: GTFS, OSM, GeoJSON
apiKey	false	Authorization key for Here API
apiUrl	true	API URL of the source API
coordinates	true	Coordinates of the location: latitude, longitude
enrichAttributes	false	List of the attributes name to be enriched (separated with commas)
sourceToken	true	Header's authorization token for the source API that will be used for enrichment, to be filled only if a token is required for the source API

The API can be also used with a curl request, here is an example of this request on a local installation of the demonstrator:

```
curl -X GET
"https://localhost/api/v1/here/getNearStations?apiFormat=GeoJson&apiKey=0PMpb1W_5iihYGu7UrBWsr8f
l6Utopf52hFBKOWl7Xc&apiUrl=https%3A%2F%2Foverpass.kumi.systems%2Fapi%2Finterpreter%3Fdata%
3D%5Bout%3Ajson%5D%3Bnode%5Bhighway%5D(48.856892%2C%202.332623%2C48.896892%2C%20
2.372623)%3Bnode%5Bbrailway%5D(48.856892%2C%202.332623%2C48.896892%2C%202.372623)%3B
out%2520meta%3B&coordinates=48.876892%2C2.352623&enrichAttributes=wheelchair%2C%20shelter%2C
%20tactile_paving%2C%20bench%2C%20bin%2C%20lit" -H "accept: application/json"
```

3.2.2. Navitia API enrichment:

We implemented 2 endpoints to enrich Navitia journeys and lines API:

- /api/v1/navitia/getJourneys
- /api/v1/navitia/getLines

You use both APIs to enrich Navitia's journey or lines API with an OpenStreetMap API or any other data provider API supporting one of the 3 standards data formats OSM, GeoJson, or GTFS.

Both endpoints use the same parameter list as previously listed for Here API enrichment.

3.2.3. OSM API enrichment:

We implemented 3 endpoints to handle this mobility data format:

- /api/v1/osm/convertOsmApiToGeoJson

This API is used as a proxy to call and convert the output of an OpenStreetMap data format API to GeoJson format

- /api/v1/osm/convertOsmDataToGeoJson

This API takes OSM data as a parameter and converts it to GeoJson format

- /api/v1/osm/enrichOsmApi

This API can be used to enrich any target OSM format API with a source mobility data API that supports one of the 3 standards of data format OSM, GeoJson, or GTFS.

3.2.4. GTFS API enrichment:

We implemented 3 endpoints to handle this mobility data format:

- /api/v1/gtfs/convertGtfsApiToGeoJson

This API is used as a proxy to call and convert the output of a GTFS data format API to GeoJson format

- `/api/v1/gtfs/convertGtfsDataToGeoJson`

This API takes GTFS data as a parameter and converts it to GeoJson format

- `/api/v1/gtfs/enrichGtfsApi`

This API can be used to enrich any target GTFS format API with a source mobility data API that supports one of the 3 standards data formats OSM, GeoJson, and GTFS.

3.2.5. GeoJson API enrichment:

We implemented 3 endpoints to handle this mobility data format:

- `/api/v1/geojson/enrichGeoJsonApi`

This API can be used to enrich any target mobility data API that produces GeoJson format, with a source mobility data API that supports one of the 3 standards data formats OSM, GeoJson, or GTFS.

The screenshot shows a web interface for the `/api/v1/geojson/enrichGeoJsonApi` endpoint. It features a 'Parameters' section with a 'Cancel' button. The parameters are listed in a table with columns 'Name' and 'Description'.

Name	Description
apiFormat * required string (query)	source API format GTFS
enrichAttributes * required string (query)	Attributes to be enriched on the target api, separated with commas wheelchair, shelter, tactile_paving, bench, bin,
sourceApiUri * required string (query)	Url of the source API to be used for enrichment https://overpass.kumi.systems/api/interpreter?
sourceToken string (query)	Source API authorization token sourceToken - Source API authorization token
targetApiUri * required string (query)	Url of the target API to be enriched https://overpass.kumi.systems/api/interpreter?
targetToken string (query)	Target API authorization token targetToken - Target API authorization token

At the bottom of the form is an 'Execute' button.

Figure 17: GeoJSON Api enrichment

Following is the input parameter list required by this API:

Table 2: *GeoJson API parameters*

Parameter	Mandatory	Meaning
apiFormat	true	The provider data format of the source API to be used for enrichment: GTFS, OSM, GeoJSON
enrichAttributes	false	List of the attributes name to be enriched (separated with commas)
sourceApiUrl	true	API URL of the source API
sourceToken	false	Header's authorization token for the source API
targetApiUrl	true	API URL of the target API to be enriched
targetToken	false	Header's authorization token for the target API

The API can be also used with a curl request, here is an example of this request on local a installation of the demonstrator:

```
curl -X GET
"https://localhost/api/v1/geojson/enrichGeoJsonApi?apiFormat=GTFS&enrichAttributes=wheelchair%2C%20shelter%2C%20tactile_paving%2C%20bench%2C%20bin%2C%20lit&sourceApiUrl=https%3A%2F%2Foverpass.kumi.systems%2Fapi%2Finterpreter%3Fdata%3D%5Bout%3Ajson%5D%3Bnode%5Bhighway%3Dbus_stop%5D(48.8345631%2C2.2433581%2C48.8775033%2C2.4400646)%3Bout%2520meta%3B&targetApiUrl=https%3A%2F%2Foverpass.kumi.systems%2Fapi%2Finterpreter%3Fdata%3D%5Bout%3Ajson%5D%3Bnode%5Bhighway%3Dbus_stop%5D(48.8345631%2C2.2433581%2C48.8775033%2C2.4400646)%3Bout%2520meta%3B" -H "accept: application/json"
```

3.2.6. Generic Json API enrichment:

The main goal of this demonstrator is to make it easy to collect and consolidate mobility data from different sources, unfortunately, this is not an easy task, because of the heterogeneity of the providers' services, the providers expose their data in different formats, some of them are open standards such as GTFS, GeoJson and OpenStreetMap, others are proprietary formats like Navitia's format (NTFS) and Here's format.

With this generic API, we try to improve the interoperability of those services using the only common part of their APIs output, JSON format.

The main goal of this API will be to enrich any Rest mobility data API with any other data format of another Rest API, using JSONPath expression, here is the user manual of how to use JSONPath expressions: <https://goessner.net/articles/JsonPath/index.html#e2>

The geographical enrichment demonstrator implements one generic API:

- /api/v1/json/enrichJsonApi

This API can be used to enrich any target mobility data API that produces JSON format, with any other source of mobility data that produces JSON format.

Authentication to the source and target APIs supports 2 ways:

- The header tokens
- The API key through request parameters, you can put it on the API's Url

generic-json-controller
Generic Json Controller

GET
/api/v1/json/enrichJsonApi
enrichJsonApi

Parameters
Try it out

Name	Description
enrichAttributes * required string (query)	Attributes to be enriched on the target api, separated with commas <input type="text" value="wheelchair, shelter, tactile_paving, bench, b"/>
sourceApiUri * required string (query)	Url of the source API to be used for enrichment <input type="text" value="https://overpass.kumi.systems/api/Interprete"/>
sourceAttributesParentPath * required string (query)	Source attributes parent's path using Jsonpath expressions <input type="text" value="\$..elements.tags"/>
sourceCoordsPath * required string (query)	The path of the source point coordinate's path using Jsonpath expressions <input type="text" value="\$..elements.coords"/>
sourceNamePath * required string (query)	The path of the source point name's path using Jsonpath expressions <input type="text" value="\$..elements.name"/>
sourceToken string (query)	Source API authorization token <input type="text" value="sourceToken - Source API authorization token"/>
targetApiUri * required string (query)	Url of the target API to be enriched <input type="text" value="https://overpass.kumi.systems/api/Interprete"/>
targetAttributesParentPath * required string (query)	Target attributes parent's path using Jsonpath expressions <input type="text" value="\$..stop_point.equipments"/>
targetCoordsPath * required string (query)	The path of the target point coordinate's path using Jsonpath expressions <input type="text" value="\$..stop_point.coordinates"/>
targetNamePath * required string (query)	The path of the target point name's path using Jsonpath expressions <input type="text" value="\$..stop_point.name"/>
targetToken string (query)	Target API authorization token <input type="text" value="targetToken - Target API authorization token"/>

Figure 18: Generic JSON Api enrichment

Following are the inputs parameters of this API:

Table 3: Generic Json API parameters

Parameter	Mandatory	Meaning
enrichAttributes	true	List of the attributes name to be enriched (separated with commas)
sourceApiUrl	true	API URL of the source API
sourceAttributesParentPath	true	The path of the parent node of the attributes on the source API response (using JSONPath)
sourceCoordsPath	true	The path of the “coordinates” attribute on the source API response (using JSONPath)
sourceNamePath	true	The path of the “name” attribute on the source API response (using JSONPath)
sourceToken	false	Header’s authorization token for the source API
targetApiUrl	true	API URL of the target API to be enriched
targetAttributesParentPath	true	The path of the parent node of the attributes on the source API response (using JSONPath)
targetCoordsPath	true	The path of the coordinates attribute on the target API response (using JSONPath)
targetNamePath	true	The path of the “name” attribute on the target API response (using JSONPath)
targetToken	false	Header’s authorizations token the target API

The API can be also used with a curl request, here is an example of this request on a local installation of the demonstrator:

```
curl -X GET
"https://localhost/api/v1/json/enrichJsonApi?enrichAttributes=wheelchair%2C%20shelter%2C%20tactile_p
aving%2C%20bench%2C%20bin%2C%20lit&sourceApiUrl=https%3A%2F%2Foverpass.kumi.systems%2
Fapi%2Finterpreter%3Fdata%3D%5Bout%3Ajson%5D%3Bnode%5Bhighway%3Dbus_stop%5D(48.8345
631%2C2.2433581%2C48.8775033%2C2.4400646)%3Bout%2520meta%3B&sourceAttributesParentPath
=%24..elements.tags&sourceNamePath=%24..elements.coords&targetApiUrl=https%3A%2F%2Foverpass
.kumi.systems%2Fapi%2Finterpreter%3Fdata%3D%5Bout%3Ajson%5D%3Bnode%5Bhighway%3Dbus_s
top%5D(48.8345631%2C2.2433581%2C48.8775033%2C2.4400646)%3Bout%2520meta%3B&targetAttrib
utesParentPath=%24..stop_point.equipments&targetNamePath=%24..stop_point.coordinates" -H "accept:
application/json"
```

3.2.7. Enrich Open Street Map lines:

This is a new feature introduced on the version 2 of the geographical enrichment demonstrator to let us enrich a line between 2 stop points with additional data such as accessibility and air quality index or any other useful information...

osm-controller

GET `/api/v1/osm/enrichOsmLineApi`

Parameters Try it out

Name	Description
enrichAttributes * required string (query)	Attributes to be enriched on the target api, separated with commas <i>Example</i> : wheelchair, shelter, tactile_paving, bench, bin, lit <input type="text" value="wheelchair, shelter, tactile_paving, bench, bin,"/>
targetApiUri * required string (query)	Url of the target API to be enriched <i>Example</i> : https://overpass.kumi.systems/api/interpreter?data=[out:json][timeout:25].(node[public_transport=stop_position]subway=yes;way[railway=subway]);out%20meta; <input type="text" value="https://overpass.kumi.systems/api/interpreter?"/>
sourceApiUri * required string (query)	Url of the source API to be used for enrichment <i>Example</i> : https://overpass.kumi.systems/api/interpreter?data=[out:json];node[highway=bus_stop];out%20meta; <input type="text" value="https://overpass.kumi.systems/api/interpreter?"/>
apiFormat * required	

Figure 19: OSM lines enrichment

This API takes as input an OSM data that describes for example a Bus line stops, and it uses another API to get useful information and put it inside the according lines based on the coordinates and stop points names.

Technically this is done by parsing the OSM input data, converting it to GeoJson, then we convert also the enrichment data (OSM/GTFS) to GeoJson data, if it is not already on a GeoJson format, and after that we correlate the data based on coordinates and names to enrich the line with a new Json attribute “enriched_properties”.

3.2.8. API demonstration:

A short demonstration video has been made available on the project repository ¹⁴, it describes how to build and use the API with some simple data enrichment use cases.

3.3. Migration of the API documentation

On the version 2 of the enrichment demonstrator, we migrated the Swagger UI framework from SpringFox to SpringDoc.

¹⁴ <https://github.com/MobiDataLab/mdl-geo-enrichment/raw/main/demo/T4.7-mdl-geo-enrichment-demo-audio-26092022.mp4>

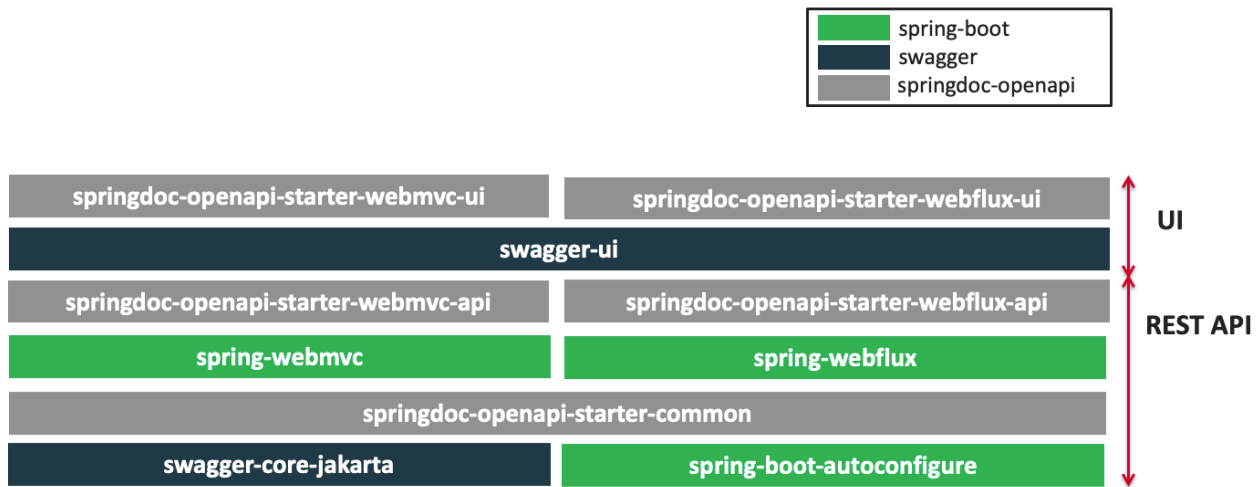


Figure 20: SpringDoc-openapi modules

SpringDoc-OpenAPI¹⁵ java library helps to automate the generation of API documentation using spring boot projects. springdoc-openapi works by examining an application at runtime to infer API semantics based on spring configurations, class structure and various annotations.

SpringDoc-OpenAPI is relatively a new library that is much easier to integrate with our demonstrator, this library has also the advantage of being actively maintained and it provides practically the same Swagger UI as on SpringFox that lacks technical support.

This framework simplifies also adding the demonstrator API to the services catalogue¹⁶, because it exposes the api documentation in an OpenAPI format.

¹⁵ <https://springdoc.org/>

¹⁶ <https://mobidatalab.github.io/mdl-catalog-ui/>

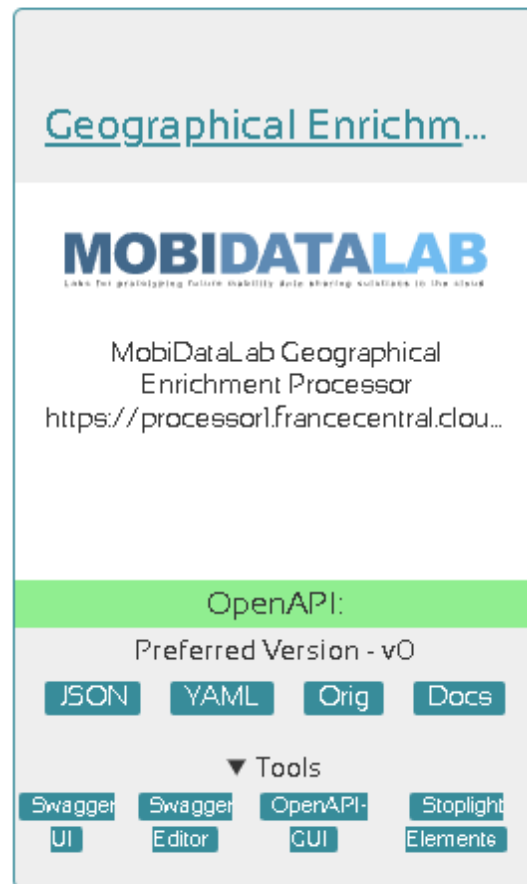


Figure 21: Geographical enrichment API documentation

The API documentation through the services catalog is available through 2 formats: YAML and JSON

Many tools are used to export the API documentation through the services catalog:

- Redocly¹⁷
- StopLight¹⁸
- OpenAPI-GUI¹⁹ This investigation introduced the Mobility Data Geographical and Semantic Enrichment demonstrators, which are both open-source solutions that enrich heterogeneous data providers. The effort we spent developing these demonstrators allowed us to quantitatively evaluate how much we can enrich trajectories. The results discussed here may also serve as a basis for further exploration of new research ideas.

¹⁷ <https://redocly.github.io/redoc/>

¹⁸ <https://stoplight.io/>

¹⁹ <https://mermade.github.io/openapi-gui/>

4. Conclusions

Enriching mobility data is pivotal in multiple domains, because it allows for a more nuanced understanding of movement patterns. By adding context like weather conditions, stop and move segments, various types of points of interest, social media, and more, we gain insights essential for effective urban and transportation planning. This enriched data not only aids in optimizing traffic flow and public transport routes but also supports sustainable transport initiatives, crucial in today's environmentally conscious world. Additionally, it enables personalized travel recommendations, improving user experiences, and can be instrumental in enhancing safety by identifying high-risk areas. Furthermore, enriched mobility data has significant economic benefits, reducing costs associated with congestion and inefficiencies. In research and development, it opens new avenues for innovation, especially in smart city technologies and autonomous vehicles. Finally, in emergency situations, such enriched data has the potential to make crisis management and response effective, underscoring its multifaceted importance.

The need of a semantic enrichment processor, which we detailed in Section 2, emerges from the widespread adoption of personal location devices, the Internet of Mobile Things, and Location-Based Social Networks, which collectively facilitate the collection of extensive movement data. This data often requires enrichment with various semantic dimensions to provide rich, contextual, and diverse information about the environment, thereby leading to the need of having semantically enriched trajectories. The processor's backend is structured as a three-step pipeline. It begins with a dataset of trajectories and a set of enrichment data sources, including linked open data, and ultimately outputs datasets of semantically enriched trajectories adhering to the open Resource Description Format (RDF) standard and to a customized version of the STEPv2 ontology. Such a format allows for the integration of these enriched datasets into a chosen triple store, enabling the extraction of insightful movement behaviours and patterns through SPARQL and federated SPARQL queries. The semantic enrichment processor is usable through (1) an interactive user interface (which can be installed and executed in a machine) has been introduced in the first version of the demonstrator) and (2) a webAPI server that brings the processor's backend functionalities to remote users. This latter component effectively positions the semantic enrichment processor as a service within the transport cloud platform.

Finally, we highlight that the design and development of the semantic enrichment processor are grounded in the principles of modularity, configurability, and extensibility. This approach allows future developers the flexibility to potentially create any semantic enrichment pipeline to suit their needs and application scenarios, and to extend the functionalities provided by the semantic enrichment processor's current version.

The geographical enrichment of mobility data stands as a pivotal instrument in understanding and optimizing various facets of human movement in our increasingly interconnected world. By integrating geographic information into mobility datasets, we gain valuable insights into spatial patterns, route preferences, and urban dynamics.

This enrichment not only enhances the accuracy of location-based services but also fuels advancements in urban planning, transportation management, and environmental sustainability. The synergy between mobility data and geographical context empowers decision-makers to formulate informed policies, design efficient transportation systems, and respond effectively to evolving

societal needs. As we navigate the complexities and heterogeneity of modern mobility formats (GTFS, OSM, GeoJSON...), the geographical enrichment of data emerges as an indispensable tool, fostering a more resilient and responsive framework for addressing the challenges and opportunities of our dynamic and interconnected societies.

5. Annexes

5.1. Semantic enrichment of mobility data

5.1.1. *Specifications of the Pandas dataframe containing the raw trajectories*

The user is requested to input a file containing the raw trajectory dataset to the demonstrator. The file must contain a Pandas dataframe saved in the Parquet²⁰ format. Each record of the dataframe represents a sample of some trajectory and must have the following fields:

- `traj_id`, a string that represents the identifier of the trajectory the sample is associated with.
- `user`, an integer representing the identifier of the entity (e.g., user, vehicle) with which the sample is associated with. Note that a user may have multiple trajectories.
- `lat`, a floating-point value representing the latitude associated with the sample.
- `lon`, a floating-point value representing the longitude associated with the sample.
- `time`, a `datetime64` value representing the timestamp associated with the sample.

5.1.2. *Specifications of the Pandas dataframe containing the output of the pre-processing module*

The output of the pre-processing module consists of a file named `traj_cleaned.parquet` containing the dataset of the pre-processed trajectories. The file contains a Pandas dataframe saved in the Parquet²¹ format. Each record of the dataframe represents a sample of some trajectory and has the following fields:

- `tid`, a string that represents the identifier of the trajectory the sample is associated with.
- `uid`, an integer representing the identifier of the entity (e.g., user, vehicle) with which the sample is associated with. Note that a user may have multiple trajectories.
- `lat`, a floating-point value representing the latitude associated with the sample.
- `lng`, a floating-point value representing the longitude associated with the sample.
- `datetime`, a `datetime64` value representing the timestamp associated with the sample.

²⁰ For more information on how to save a Pandas dataframe in the Parquet binary format, please refer to https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html

²¹ For more information on how to save a Pandas dataframe in the Parquet binary format, please refer to https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html

5.1.3. Specifications of the Pandas dataframes containing the output of the segmentation module

The output of the segmentation module consists of two files: `stops.parquet` and `moves.parquet`, which respectively contain the stops and the moves detected for a given set of pre-processed trajectories. Both files contain a Pandas dataframe saved in the Parquet²² format.

Each record in `stops.parquet` (or internal dictionary, containing the stop segments, in the JSON body of a response to a GET request to the segmentation endpoint) represents a stop that has been detected for some trajectory of some user, and has the following fields:

- `tid`, a string that represents the identifier of the trajectory the stop is associated with.
- `uid`, an integer representing the identifier of the entity (e.g., user, vehicle) the stop is associated with.
- `lat`, a floating-point value representing the latitude associated with the stop centroid.
- `lng`, a floating-point value representing the longitude associated with the stop centroid.
- `datetime`, a *datetime64* value representing the timestamp associated with the instant the stop begins.
- `leaving_datetime`, a *datetime64* value representing the timestamp associated with the instant the stop ends.

Each record in `moves.parquet` (or internal dictionary, containing the move segments, in the JSON body of a response to a GET request to the segmentation endpoint) represents a move that has been detected for some trajectory of some user, and has the following fields:

- `tid`, a string that represents the identifier of the trajectory the move is associated with.
- `uid`, an integer representing the identifier of the entity (e.g., user, vehicle) the move is associated with.
- `lat`, a floating-point value representing the latitude associated with the location where the move begins.
- `lng`, a floating-point value representing the longitude associated with the location where the move begins.
- `datetime`, a *datetime64* value representing the timestamp associated with the instant the move begins.
- `move_id`, a float value representing the identifier associated with the move.

²² For more information on how to save a Pandas dataframe in the Parquet binary format, please refer to https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html

5.1.4. *Specifications of a file containing a dataset of points of interest.*

The user is allowed to input a file containing the POIs to be used to enrich the occasional stops to the demonstrator. The file must contain a Pandas dataframe saved in the Parquet format. Each record of the dataframe must have the following fields:

- **osmid**: a string representing the identifier that OpenStreetMap associates to a specific POI.
- **category**: a string representing the OpenStreetMap category to which the POI belongs. The current version of the demonstrator supports the following categories: `amenity`, `aeroway`, `building`, `historic`, `healthcare`, `landuse`, `office`, `public_transport`, `shop`, and `tourism`.
- **wikidata**: a string representing the identifier that has been assigned by WikiData to the POI (note: this field can contain a missing value in case a POI is not present in WikiData).
- **geometry**: Python object describing the shape (e.g., point, polygon, etc.) associated with the POI.

5.1.5. *Specifications of a file containing weather information*

The user may pass to the demonstrator a file containing the dataset with weather information to be used to enrich the trajectories. The file must contain a Pandas dataframe saved in the Parquet format. Each record of the dataframe must have the following fields:

- **DATE**: a string representing a date in `yyyy-mm-dd` format. Such date represents the day covered by the record.
- **TAVG_C**: float value representing the average temperature associated with DATE.
- **DESCRIPTION**: a string representing the weather condition associated with DATE (e.g., *sunny*, *rainy*).

Note that in each record there is no association between the weather information and the location it refers to. In other words, the demonstrator assumes that the weather information provided within the weather dataframe covers the geographical area in which the trajectories are known to be located.

5.1.6. *Specifications of a file containing social media posts*

The user can pass to the demonstrator a file containing the dataset with tweets (i.e., social media posts) that can be used to enrich the trajectories. The file must contain a Pandas dataframe saved in the Parquet format. Each record of the dataframe must have the following fields:

- **tweet_id**: a string representing the identifier of the tweet.
- **text**: a string representing the content of the tweet.
- **tweet_created**: a string representing the date on which the tweet was made. The date is in `yyyy-mm-dd` format.

- **uid**: identifier of the user that made the tweet. Must correspond to the user field in the specifications of the Pandas dataframe containing the raw trajectories.

5.1.7. Details on the ontology used to structure the information within the RDF graph

The specifications of the original STEP ontology can be found at <http://talespaiva.github.io/step/>.

In the following, we introduce the main customizations CNR did to STEP for the demonstrator.

We introduced a class, **Point of Interest**, representing instances of points of interest. Each instance of this class possesses (via the *hasOSMValue* data property) the identifier that OpenStreetMap associates to its POI, and (if present) the URI (via the *hasWDValue* data property) to the WikiData page associated with the POI.

We introduced *several subclasses* of the class **Qualitative Description**. We recall that the authors of the STEP ontology introduced the Qualitative Description class to enable individual episodes²³ of semantic aspects to have complex representations. Moreover, we recall that this class represents a fundamental building block that *must be extended* according to one's specific needs.

Accordingly, the **subclasses** of Qualitative Description we introduce are:

Move: this class (Figure 22) models instances of qualitative descriptions associated with episodes of aspects concerning move segments of trajectories. We also extended the Move class with several subclasses representing different transportation means, i.e., **Bike**, **Bus**, **Car**, **Subway**, **Train**, **Taxi**, and **Walk**.

²³ *By episode of a semantic aspect* here we mean a specific occurrence of such aspect in space and/or time. For instance, an episode of an Occasional Stop occurs during some time interval in some spatial region. Another example can be an episode of a Move, which occurs during some time interval along the path associated with the move segment.

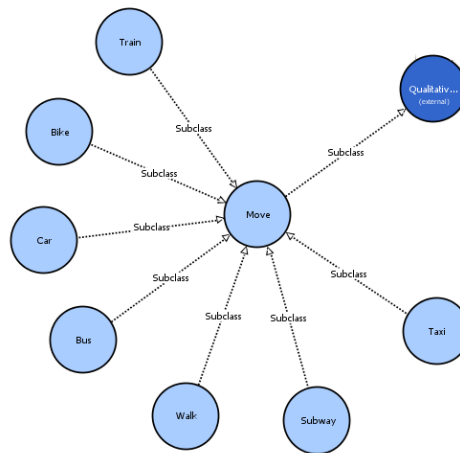


Figure 22: Overview of the Move class with its subclasses

Stop: this class (Figure 23) models instances of qualitative descriptions associated with episodes of aspects concerning stop segments of trajectories. We also extended the Stop class with two subclasses representing the two different types of stops the demonstrator detects during the enrichment process, i.e., **Occasional Stop** and **Systematic Stop**.

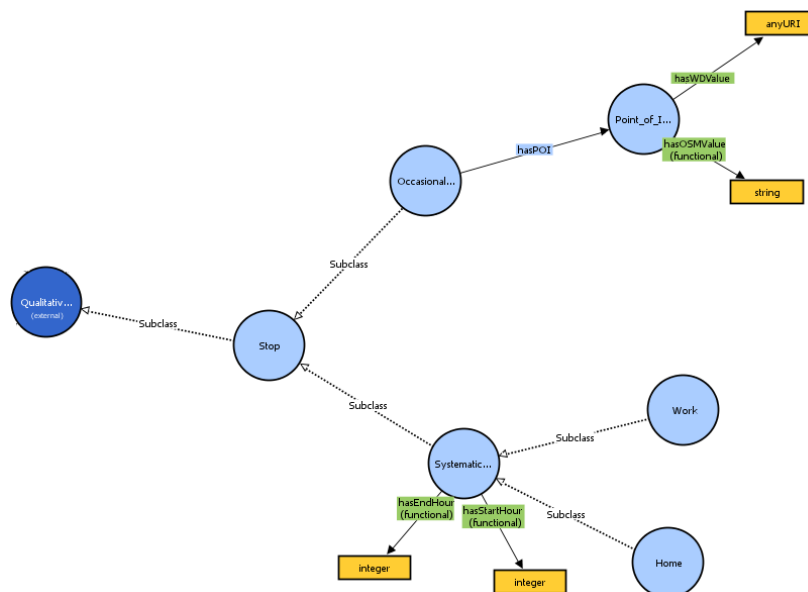


Figure 23: Overview of the Stop class with its subclasses.

Each instance of **Occasional Stop** can be associated with one or more instances of Point of Interest via the *hasPOI* property. Each instance of **Systematic Stop** is associated with a pair of values (via the *hasStartHour* and *hasEndHour* datatype properties) indicating the hours at which the systematic stop begins and ends.

We also provide two further subclasses derived from the subclass Systematic Stop, i.e., **Home** and **Work**, which conceptually represent the two types of systematic stops the demonstrator currently attempts to recognize.

Weather: this class (Figure 24) models instances of qualitative descriptions associated with episodes of the weather aspect. Each instance of this class possesses two values, i.e., the average temperature (via the *hasTemperature* datatype property) and the weather conditions (via the *hasWeatherCondition* datatype property) observed during the episode the instance is associated with.

Social Media Post: this class (Figure 24) models instances of qualitative descriptions associated with episodes of the social media post aspect. Each instance of this class possesses two values, the first one being a string representing the text of a post (via the *hasText* datatype property), and the second one being a timestamp indicating the publication time of the post (via the *hasPublicationTime* datatype property).

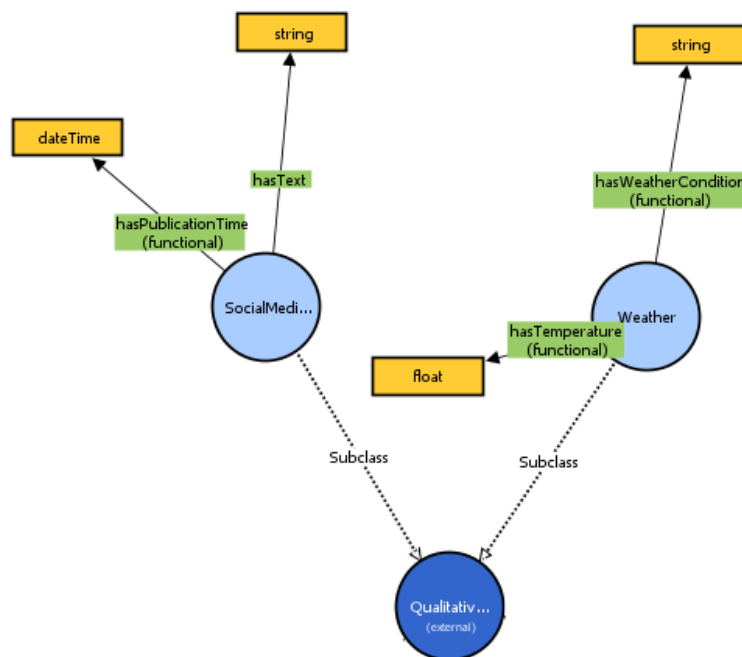


Figure 24: Overview of the Weather and Social Media Post classes.

We let instances of the **Agent** class (i.e., the users producing the trajectories) have semantic aspects. This is achieved by modifying the domain of the *hasFeature* property, which is now the **union** of the classes Spatiotemporal Element and Agent.

We provide the OWL files containing the customized version of the STEP ontology in the GitHub repository associated with the demonstrator: <https://github.com/MobiDataLab/mdl-semantic-enrichment>

6. References

Dos Santos Mello Ronaldo, Bogorny Vania, Alvares Luis Otávio, Zambom Santana Luiz Henrique, Ferrero Carlos Andres, Frozza Angelo Augusto, Schreiner Geomar Andre, Renso Chiara MASTER: A multiple aspect views on trajectories. Trans. GIS 23(4): 805-822 (2019).

Nogueira, Tales P., Reinaldo B. Braga, Carina T. de Oliveira, and Hervé Martin. "FrameSTEP: A framework for annotating semantic trajectories based on episodes." Expert Systems with Applications 92 (2018): 533-545.

Spaccapietra, Stefano, Christine Parent, Maria Luisa Damiani, Jose Antonio de Macedo, Fabio Porto, and Christelle Vangenot. "A conceptual view on trajectories." Data & knowledge engineering 65, no. 1 (2008): 126-146.),

Zheng, Yu, Xing Xie, and Wei-Ying Ma. "GeoLife: A collaborative social networking service among user, location and trajectory." IEEE Data Eng. Bull. 33, no. 2 (2010): 32-39.)).

Birant, D. and Kut, A., 2007. ST-DBSCAN: An algorithm for clustering spatial-temporal data. Data & knowledge engineering, 60(1), pp.208-221.

MobiDataLab consortium

The consortium of MobiDataLab consists of 10 partners with multidisciplinary and complementary competencies. This includes leading universities, networks, and industry sector specialists.



[@MobiDataLab](https://twitter.com/MobiDataLab)
[#MobiDataLab](https://twitter.com/MobiDataLab)



<https://www.linkedin.com/company/mobidatalab>

For further information please visit www.mobidatalab.eu



MobiDataLab is co-funded by the EU under the H2020 Research and Innovation Programme (grant agreement No 101006879).

The content of this document reflects solely the views of its authors. The European Commission is not liable for any use that may be made of the information contained therein. The MobiDataLab consortium members shall have no liability for damages of any kind that may result from the use of these materials.