MOBIDATALAB

Labs for prototyping future mobility data sharing solutions in the cloud

D4.7 Data enrichment processors (V1)

23/01/2023
Author(s): Mohamed KARAMI (AKKA) - Francesco LETTICH (CNR)

Summary sheet

Deliverable Number	D4.7
Deliverable Name	Data enrichment processors (V1)
Full Project Title	MobiDataLab, Labs for prototyping future Mobility Data sharing cloud solutions
Responsible Author(s)	Mohamed KARAMI (AKKA) - Francesco LETTICH (CNR)
Contributing Partner(s)	
Peer Review	Alberto BLANCO JUSTICIA (URV) - Thierry-Xavier CHEVALLIER (AKKA)
Contractual Delivery Date	31-07-2022
Actual Delivery Date	28-07-2022
Status	Final
Dissemination level	Public
Version	V1.0
No. of Pages	**40**
WP/Task related to the deliverable	WP4/T4.2
WP/Task responsible	AKKA/AKKA
Document ID	MobiDataLab-D4.7-DataEnrichmentProcessors_V1_v1.0
Abstract	This deliverable is a report to provide an overview of the Task 4.4 demonstrator

Legal Disclaimer

Funded by the European Union

Project partners

Organization	Country	Abbreviation
AKKA I&S	France	AKKA
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	02/06/2022	AKKA	outline	Draft
0.2	07/07/2022	AKKA	Geographical enrichment	Draft
0.3	15/07/2022	CNR	Semantic enrichment	Draft
0.4	26/07/2022	AKKA	All	Peer review
0.5	26-28/07/2022	AKKA	All	Quality check
1.0	28/07/2022	AKKA	All	Final version

Executive Summary

The deliverable D4.7 is a report providing an overview of version 1 of the MobiDataLab data enrichment processors: Geographical and semantic enrichment demonstrators

Table of contents

List of figures

List of tables

MOBIDATALAB

Funded by the European Union

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

MOBIDATALAB

Funded by the European Union

Introduction

1.1. Project overview

1.

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 v1 is 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 D4.7 deliverable is 'public' (PU).

Funded by the
European Union

Semantic enrichment of mobility data (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 MAT-Builder (Pugliese *et al.* 2022) a tool developed by CNR to semantically enrich trajectories.

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.

2.1. How to build 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 Anaconda1,2, 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*

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*[3], *Geopandas*[4], *scikit-learn*[5], *scikit-mobility*[6], *rdflib*[7], *PTRAIL*[8], and *Dash*[9].

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.

---

[1] Anaconda installers are available at https://www.anaconda.com/products/distribution
[2] 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
[3] https://pandas.pydata.org/
[4] https://geopandas.org/en/stable/
[5] https://scikit-learn.org/stable/
[6] https://github.com/scikit-mobility/scikit-mobility
[7] https://rdflib.readthedocs.io/en/stable/
[8] https://github.com/YakshHaranwala/PTRAIL
[9] https://plotly.com/dash/
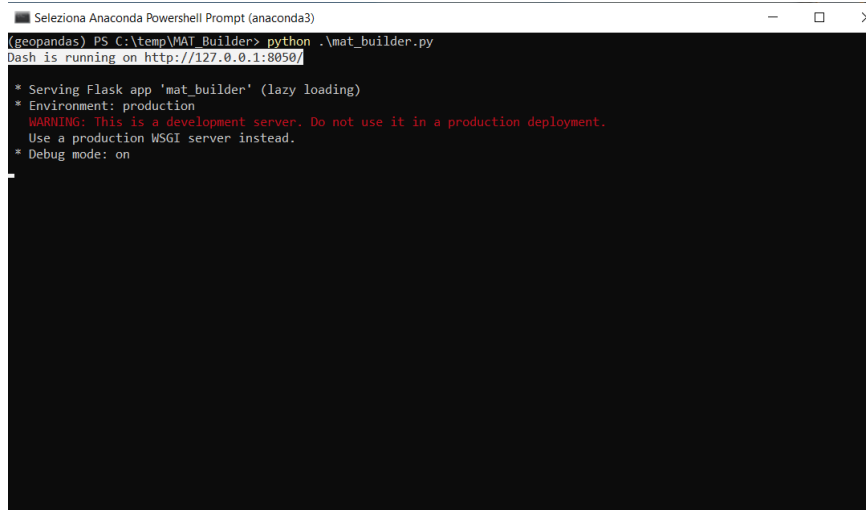
MOBIDATALAB

Funded by the European Union

2.2. How to use the semantic enrichment demonstrator

To run the semantic enrichment demonstrator the user must first execute from the Anaconda PowerShell prompt, and within the virtual environment created during the installation procedure, the Python script mat_builder.py. Once executed, the script will tell the user the address at which the demonstrator can be accessed through some web browser of preference (Figure 1).

Figure 1: Execution of the script mat_builder.py

The user must then open a web browser and input the above 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 exactly 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.
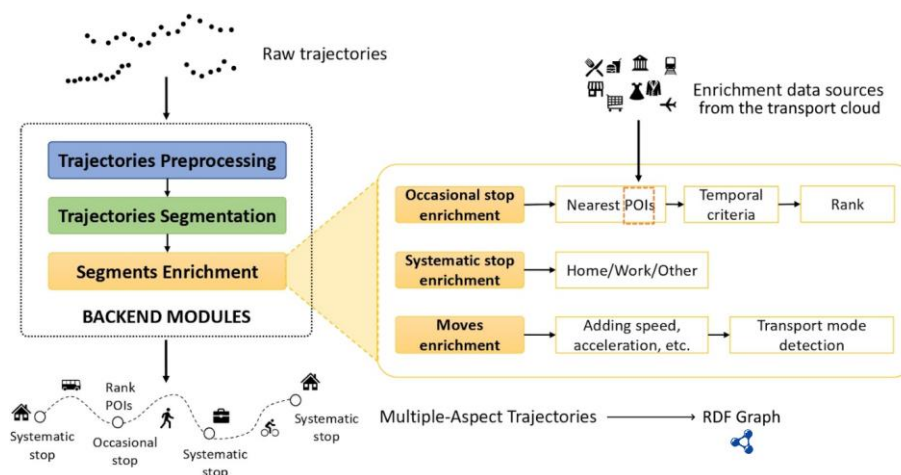

Figure 2: Demonstrator backend.

MOBIDATALAB
MOBIDATALAB – H2020 G.A. No. 101006879

Funded by the European Union

2.2.1. Pre-processing module

The first tab allows the user to access the functionalities offered by the *pre-processing* module (Figure 3). 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.
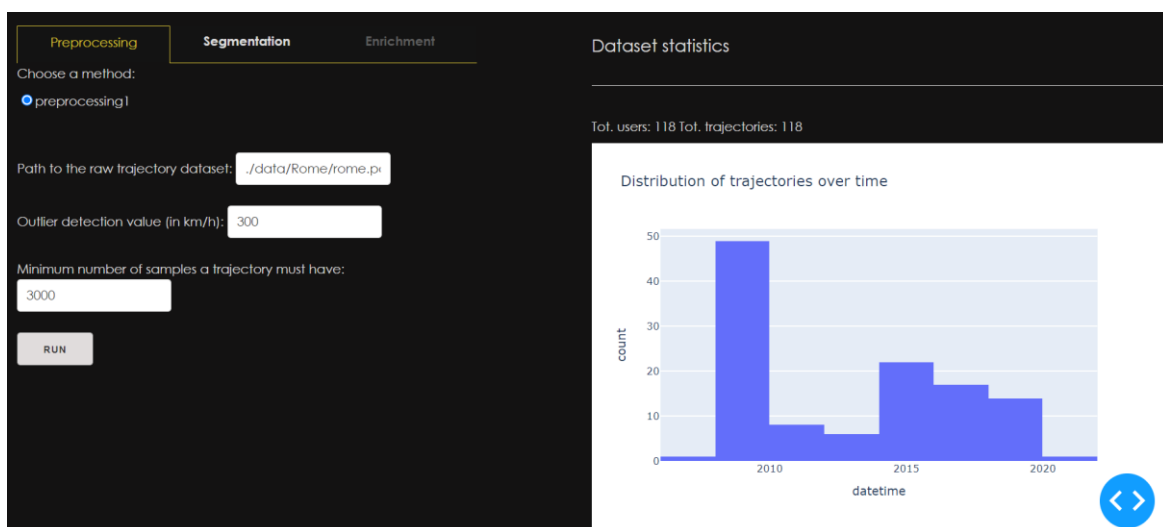
Figure 3: pre-processing module.

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 2.2.5 we provide the specifications that the file must follow to be recognized and used by the demonstrator.

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 2.2.6.

2.2.2. Trajectory segmentation module

Once the raw trajectories are pre-processed, the user interface activates the segmentation module tab (Figure 4). The goal of the segmentation module (green block) is to take an input consisting of a set of pre-processed trajectories and partition 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 2.2.7.

Figure 4: Trajectory segmentation module.

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. Segment enrichment module

Once the trajectories are segmented, the user interface activates the segment enrichment module tab (Figure 5). The segment enrichment module (yellow block in Figure 2) takes 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.
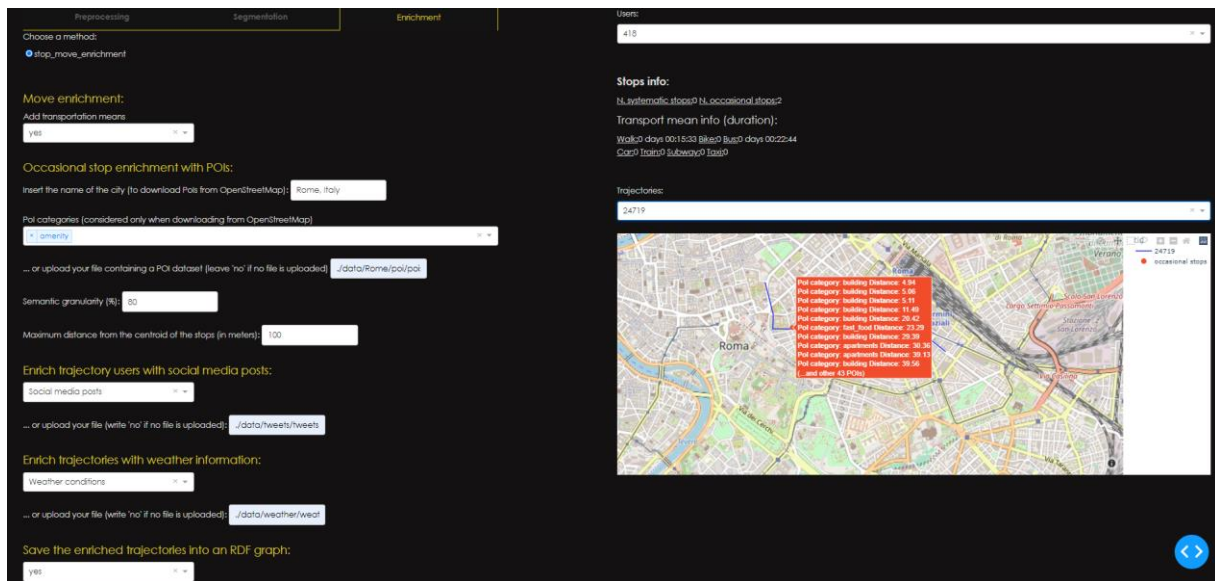
Figure 5: Segment enrichment module.

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 an aspect that concerns its *regularity*, *i.e.*, whether a stop belongs to a *systematic*[10] stop or an *occasional* one.

Once the enrichment module enriched the stop segments with the regularity aspect, the module associates each systematic stop to an *activity*, which is established according to a pre-determined set of criteria. In the current version of the demonstrator, such activities are *home*, *work*, or *other*. Again, the demonstrator is general enough to be extended to different activities and this is indeed the object for future work.

---

[10] 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.

The demonstrator can also associate POIs with occasional stops. This is reflected in the user interface since the user can retrieve a dataset of POIs either from OpenStreetMap[11] – 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 2.2.8 we provide the specifications that such file must follow to be recognized and used by the demonstrator.

We observe that, in some cases, the POI datasets downloaded from OpenStreetMap might have an extremely large number of attributes, and many of these may be missing or null values. To deal with this issue, the interface lets the user specify a value – that we name *semantic granularity* - that the demonstrator uses to discard attributes with too many missing values. Note that such value represents a percentage: for instance, a value equal to 80 means we discard those attributes with several missing values equal to or greater than 80%. Once the POI dataset is loaded, the segment enrichment module decides which POIs should be used to enrich the occasional stops by ranking them according to the distance and temporal overlap criteria.

The demonstrator can also enrich trajectories with weather information and social media posts. In this case, the user must provide the paths to the files containing the respective datasets. In Sections 2.2.9 and 2.2.10 we provide the specifications that said files must follow to be recognized and used by the demonstrator.

Finally, the demonstrator 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. In Section 2.2.11 we provide details on the customizations we did to the original STEP ontology for this demonstrator.

2.2.4. *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. 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.

---

[11] OpenStreetMap: https://www.openstreetmap.org/

MOBIDATALAB
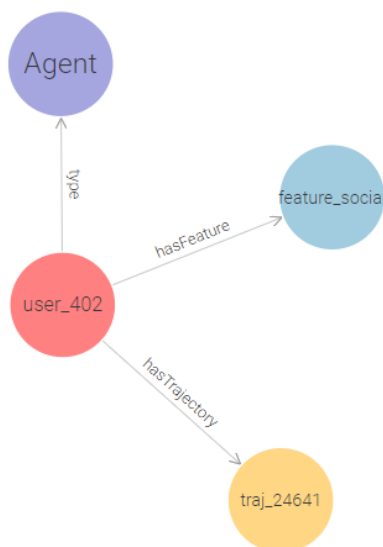
Funded by the European Union

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

In Figure 5 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.
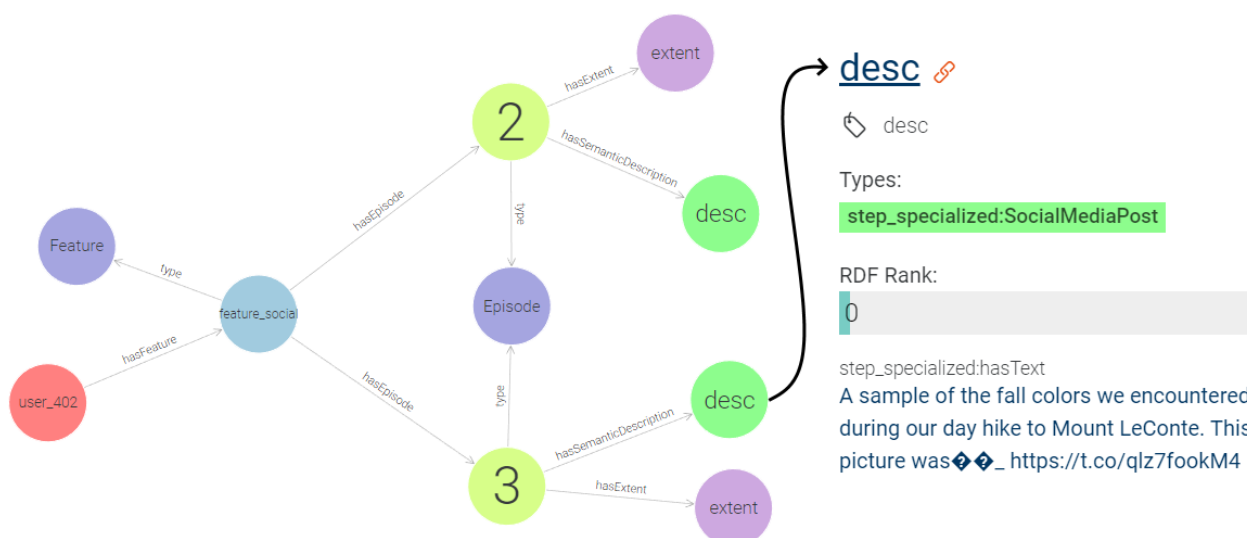

Figure 7: Social Media Post aspect.

From Figure 6 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 5 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 6 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.
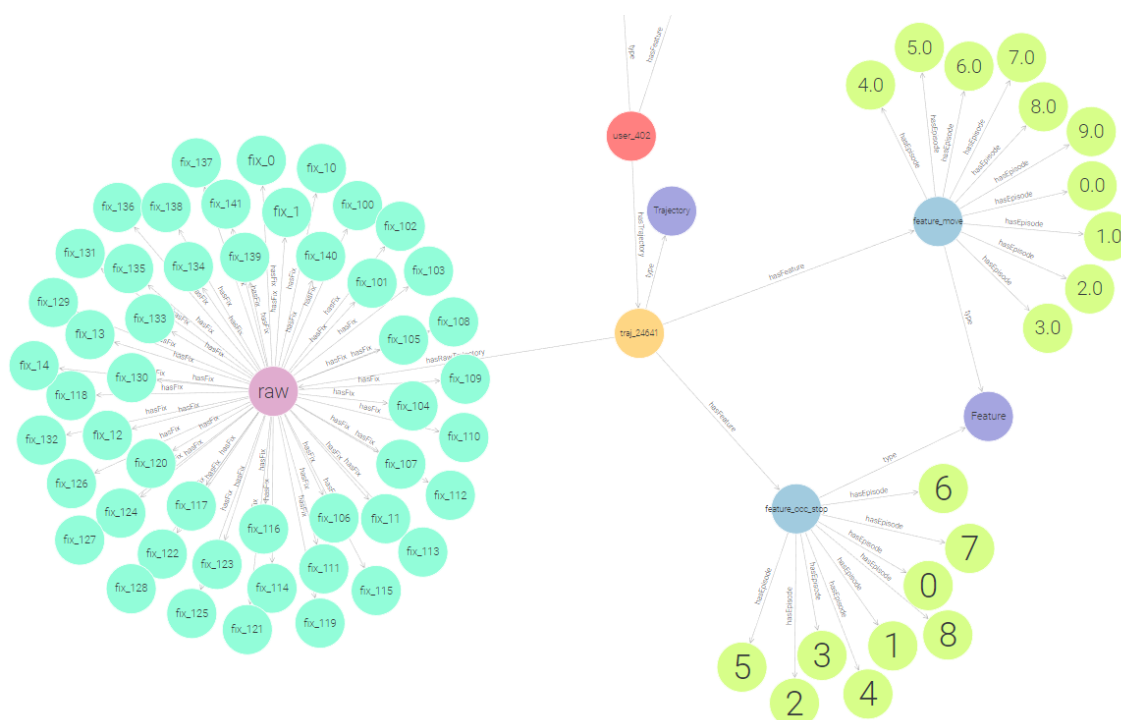

Figure 8: The subgraph rooted in the node related to the trajectory 224641 of the user 402.

2.2.5.   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 Parquet[12] 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.

2.2.6.   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 Parquet[13] 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.

---

[12] 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
[13] 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

2.2.7. *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 Parquet[14] format.

Each record in `stops.parquet` 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` 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.

---

[14] 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

2.2.8.   Specifications of the file containing the 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.

2.2.9.   Specifications of the 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.

2.2.10. Specifications of the Pandas dataframe 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.

2.2.11. *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 episodes[15] 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 8) 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**.

---

[15] *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.
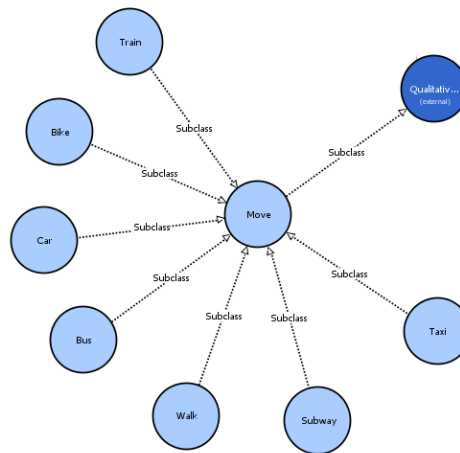
Figure 9: Overview of the Move class with its subclasses

Stop: this class (Figure 9) 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**.
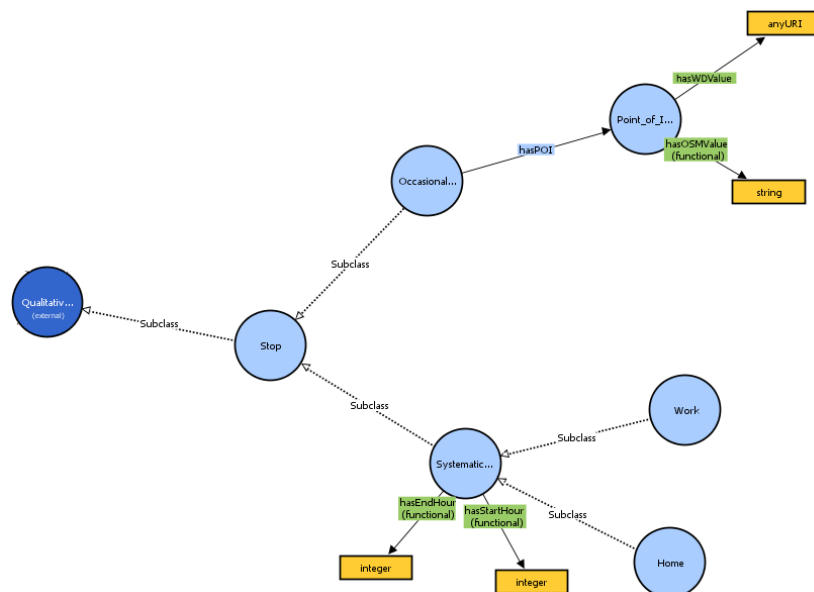

Figure 10: Overview of the Weather and Social Media Post classes.

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 10) 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 10) 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).
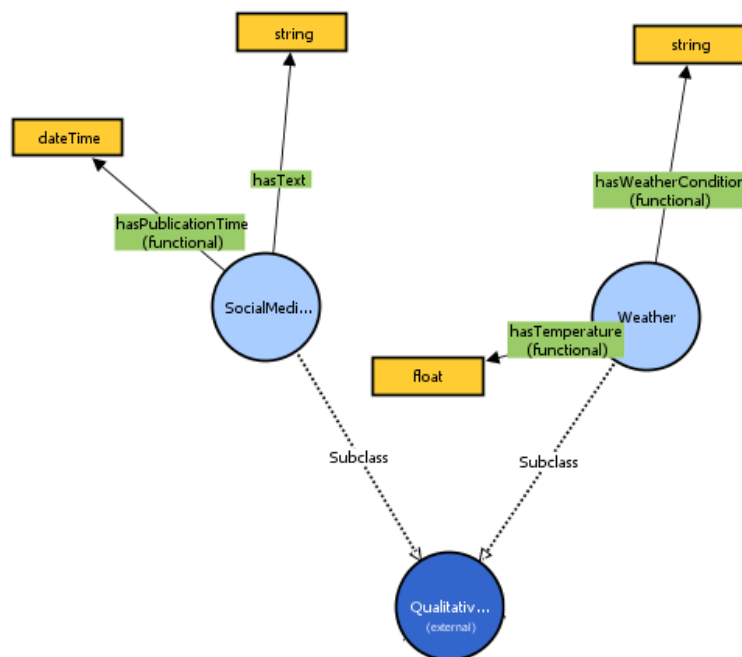
Figure 11: an overview of the Stop class with its subclasses.

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

MOBIDATALAB

Funded by the European Union

Geographical enrichment of mobility data (demonstrator)

3. 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 12: 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.

MOBIDATALAB

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 è 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.
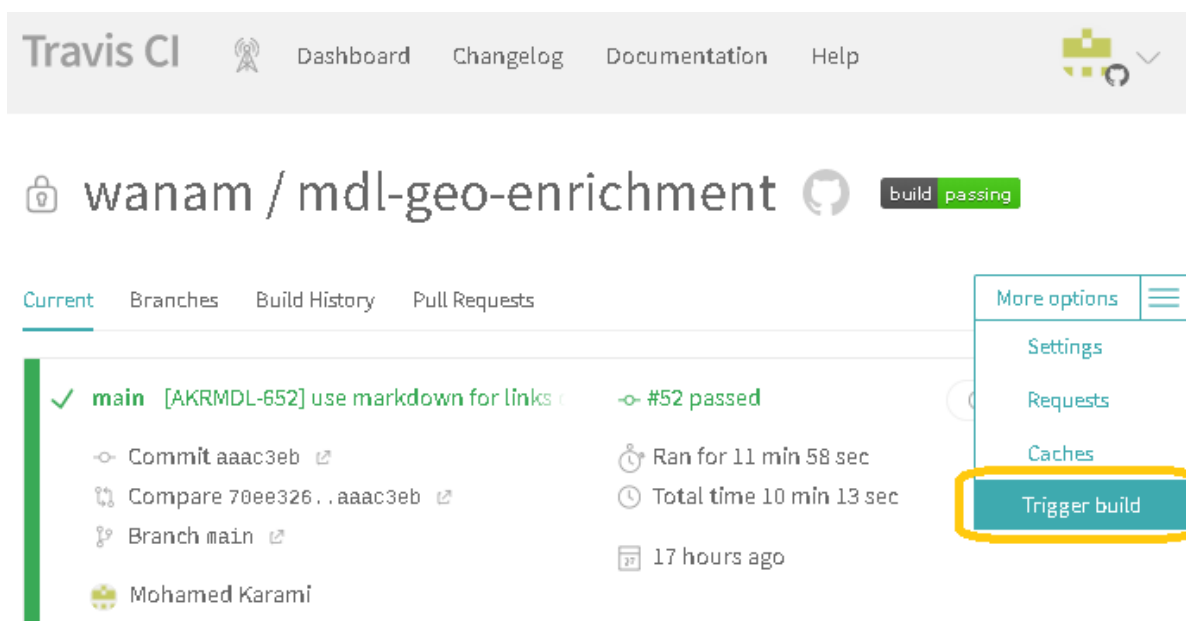

Figure 13: MDL-Geo-Enrichment pipeline on Gitlab-CI

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

Figure 14: 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 15: 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.

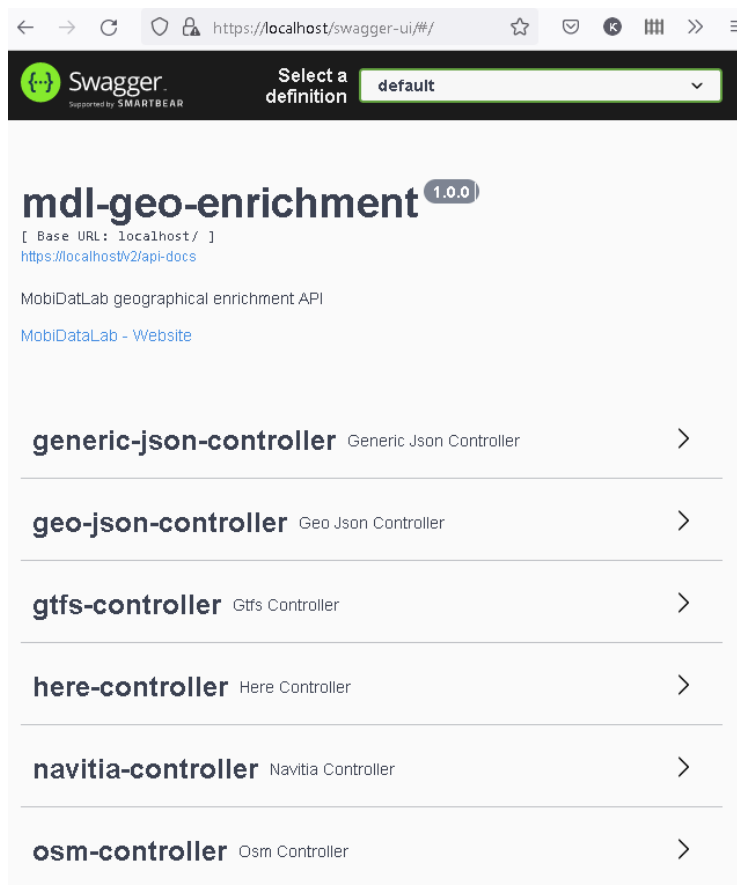Once the server is up, you can browse Swagger UI through http://SERVER/swagger-ui/

MOBIDATALAB

Funded by the European Union

Figure 16: 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.

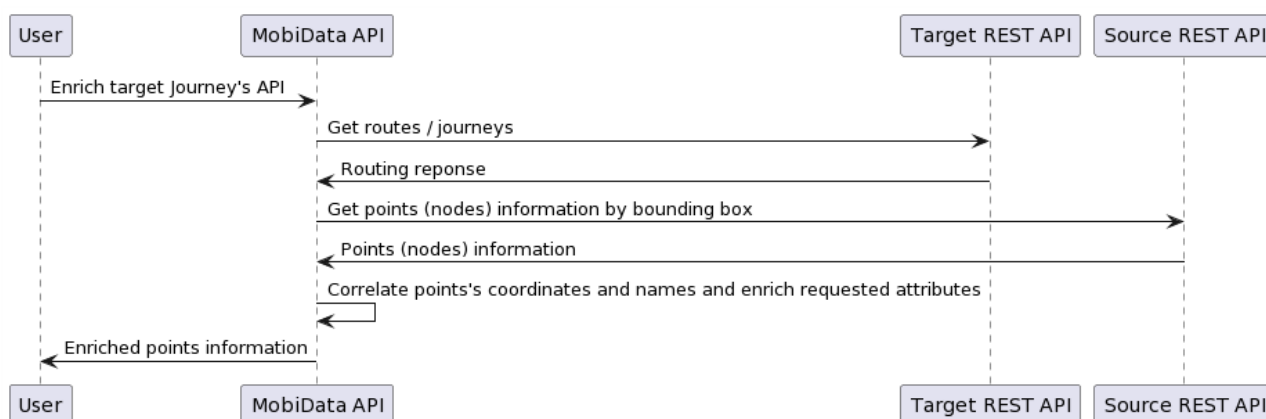Here is a sequence diagram illustrating the enrichment process:

Figure 17: 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
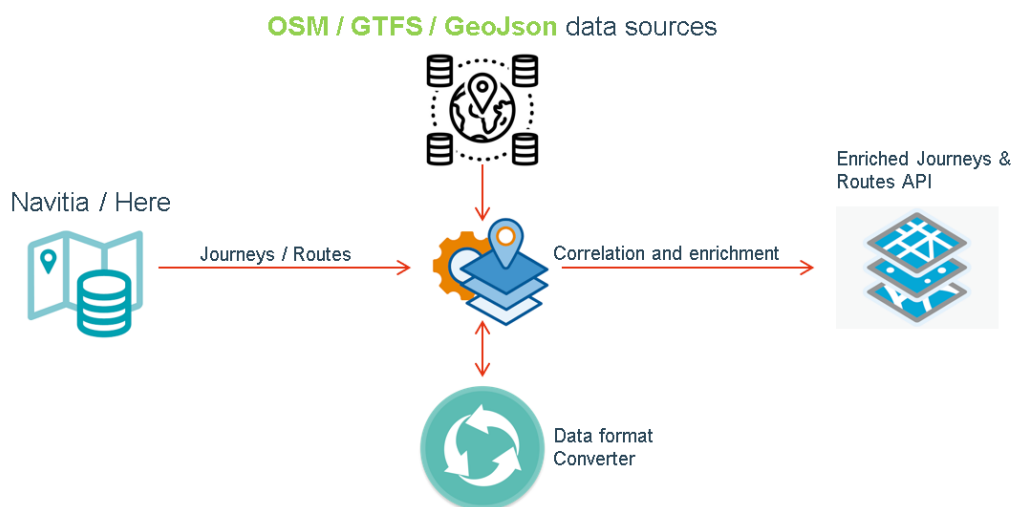
Figure 18: Sequence diagram of the enrichment process

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:

Figure 19: Here stations API enrichment

Following are the input parameters required by this API:

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
I6Utopf52hFBKOwl7Xc&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%5Brailway%5D(48.856892%2C%202.332623%2C48.896892%2C%202.372623)%3Bo
ut%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.

MOBIDATALAB

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.

Figure 20: 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%20
shelter%2C%20tactile_paving%2C%20bench%2C%20bin%2C%20lit&sourceApiUrl=https%3A%2F%2Fover
pass.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&targetApiU
rl=https%3A%2F%2Foverpass.kumi.systems%2Fapi%2Finterpreter%3Fdata%3D%5Bout%3Ajson%5D%3B
node%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 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

Figure 21: Generic JSON Api enrichment

Following are the input parameter 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"
```

**Funded by the
European Union**

4. Conclusions

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.

As a future line of work, we plan to improve the quality of the enriched data. For what concerns the semantic demonstrator we also plan to improve the tool to better integrate it into the transport cloud architecture, and since the tool has been designed to be flexible and extensible, we plan to include more semantic enrichment datasets and enrichment methods.

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.

Pugliese Chiara, Lettich Francesco, Renso Chiara, Pinelli Fabio. MAT-Builder: a System to Build Semantically Enriched Trajectories. The 23rd IEEE International Conference on Mobile Data Management, June 2022, Cyprus

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.)).

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](#)
#MobiDataLab

in https://www.linkedin.com/company/mobidatalab

For further information please visit **www.mobidatalab.eu**

MOBIDATALAB
MOBIDATALAB – H2020 G.A. No. 101006879

Funded by the European Union