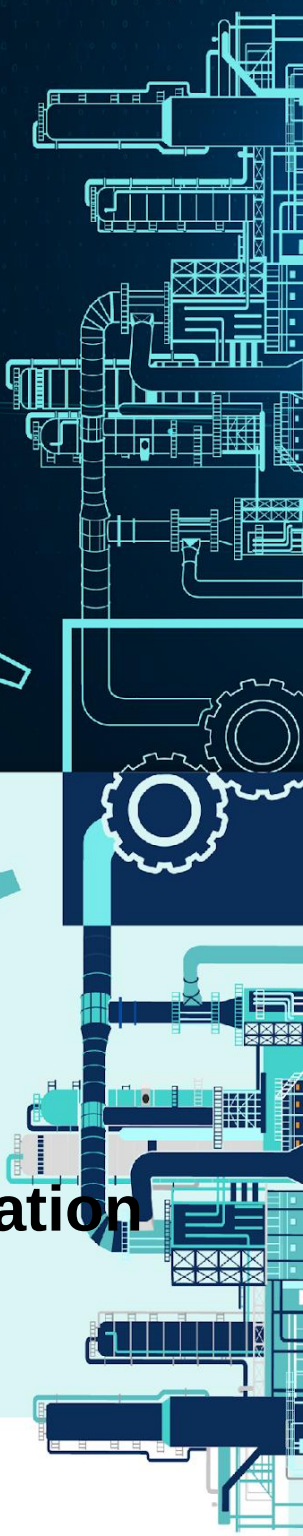
FACTLOG
www.factlog.eu

ENERGY-AWARE FACTORY ANALYTICS FOR PROCESS INDUSTRIES

Deliverable D4.4

KG-based analytics for process optimization



Version 1.0	**Lead Partner** JSI
Date 30/06/2022	**Project Name** FACTLOG – Energy-aware Factory Analytics for Process Industries

FACTLOG has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 869951

Call Identifier H2020-NMBP-SPIRE-2019	**Topic** DT-SPIRE-06-2019 - Digital technologies for improved performance in cognitive production plants
Project Reference 869951	**Start date** November 1st, 2019
Type of Action IA – Innovation Action	**Duration** 42 Months

Dissemination Level

X	**PU**	Public
	CO	Confidential, restricted under conditions set out in the Grant Agreement
	CI	Classified, information as referred in the Commission Decision 2001/844/EC

Disclaimer

This document reflects the opinion of the authors only.

While the information contained herein is believed to be accurate, neither the FACTLOG consortium as a whole, nor any of its members, their officers, employees or agents make no warranty that this material is capable of use, or that use of the information is free from risk and accept no liability for loss or damage suffered by any person in respect of any inaccuracy or omission.ca

Executive Summary

This document presents the work done regarding the design and development of knowledge graph (KG) operated cognitive services. The development work builds on initial services designed under D3.2 Data Analytics as a Cognitive Services and ontology-based KG designed in deliverable D4.2.

Initially, the technology background is presented with description of services architecture, conceptual design and approach in integrating KG into cognitive API analytical workflow. The redesign and upgrade of the initial API is presented as well as its functional design.

Furthermore, in continuation, the ontology model used and KG extraction methods are explained, including the use of domain specific concepts for data analytics in FACTLOG Ontology for cognitive API automation. The final solution is presented on two main use cases – using a predefined AI model and an ontology managed AI model pipeline (feature vectors and data pipeline setup).

In the final chapter, a short consolidation and interpretation of the results in the light of project's main objective were included. More importantly, the KG-based cognitive API enables to use initial analytical tools developed in FACTLOG in a more generic fashion. The results show how using an advanced approach such as ontology driven process definitions can enable new approaches in utilizing analytical technologies for faster deployment, easier scalability and more effective maintainability.

Revision History

Revision	Date	Description	Organisation
0.1	10/05/2022	Table of contents	JSI
0.2	20/05/2022	Initial inputs	JSI
0.3	30/05/2022	Architecture, ontology schema and schema generation	JSI, EPFL
0.4	06/06/0222	Pilot implementation, conclusion	JSI, EPFL
0.5	21/06/2022	Final draft ready for review	JSI
0.6	28/06/2022	Peer review	TUC, SIMAVI
1.0	30/06/2022	Final version ready for submission	JSI

Contributors

Organisation	Author	E-Mail
JSI	Miha Cimperman	miha.cimperman@ijs.com
JSI	Jože Rožanec	joze.rozanec@ijs.si
JSI	Beno Šircelj	beno.sircelj@ijs.si
JSI	Bor Brecelj	bor.brecelj@gmail.com
EPFL	Lu Jinzhi	jinzhi.lu@epfl.ch

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose and Scope

The purpose of this deliverable is to present KG -based analytics for process optimization services. We present the basic requirements and goals, as well as a methodology for building a Knowledge Graph-based analytics pipeline and analytics services operation. We have implemented a Knowledge Core module that leverages semantic domain knowledge and automatically runs the pipeline for analytics services. The designed interfaces allow the models to be used and integrated into the FACTLOG infrastructure as distributed microservices components in a generic (automated configuration) manner.

This document explains the conceptual background of KG-based analytics and provides a description of the supporting ontologies, the developed tools, the selected technologies, the architectural approach, and the final implementation demonstrated in specific pilots of FACTLOG project.

1.2 Relation with other Deliverables

This work builds on the developments presented in Deliverables D4.1, D4.2, and D4.3. The KG-based analytics for process optimization provides the means to extend the existing processes with ontology-related knowledge to build machine learning models for advanced use cases. This provides baseline ontology models that can be used to facilitate the development of final models for specific use cases.

This deliverable also builds on the basic cognitive API architecture, developed, and presented in Deliverable D3.2 -Data Analytics as a Cognitive Service, in which all the analytical tools from WP2 are consolidated. The initial API enables processing analytical services developed based on predefined use case requirements. The initial API has been extended with an ontological engine that enables automation of service configuration while building internal data structures essential for processing analytic services (e.g., automatic design of feature vector structures).

The services listed in this deliverable will be used in the pilots and therefore will also have some relation to all the deliverables describing the final deployments.

1.3 Structure of the Document

Following the introduction in Section 1, Section 2 explains the architectural approach in augmenting cognitive services with ontological model. This includes services designed based on the developed analytic components, the configuration of services for the development of feature vectors, and the querying of model services. Section 3 describes the created ontology and ontological model, as well as the ontology augmentation used in API design. Section 4 presents two main examples of using ontology-based services and feature vector configurations. Section 5 presents the interpretation of the final results, key results and achievements in the light of the initial project goals and objectives.

2 Architecture and Concept design

The Knowledge Graph-based analytics process optimization was designed as a service backed with a knowledge graph and providing its functionality through an HTTP Application Programming Interface (API). The main goals of the service were to provide application endpoints where: (a) given a description of a use case and a dataset, a baseline model can be created that leverages domain knowledge about that specific use case; and (b) given a description of a machine learning task, a model can be trained. In both cases, the machine learning model would be made available for future use.

To build the knowledge graph and establish a common vocabulary, we first defined an ontology (more details in Section 3.1). Based on the ontology concepts, we then defined the HTTP API and the required JSON format to perform application queries (more details in Section 2.1). A knowledge graph was initialized to record relevant information about the queries to the application and domain information about the datasets, features, and algorithms used to train the models, along with the performance of the machine learning models, the binaries of the trained models, and other relevant metadata. The binaries and the performance of the models could be used to decide whether to use the models in production and how to use them. In addition, the stored knowledge could later be used to learn common patterns and identify best practices that would lead to better performance of the machine learning models.

The architecture is depicted in the following figure.

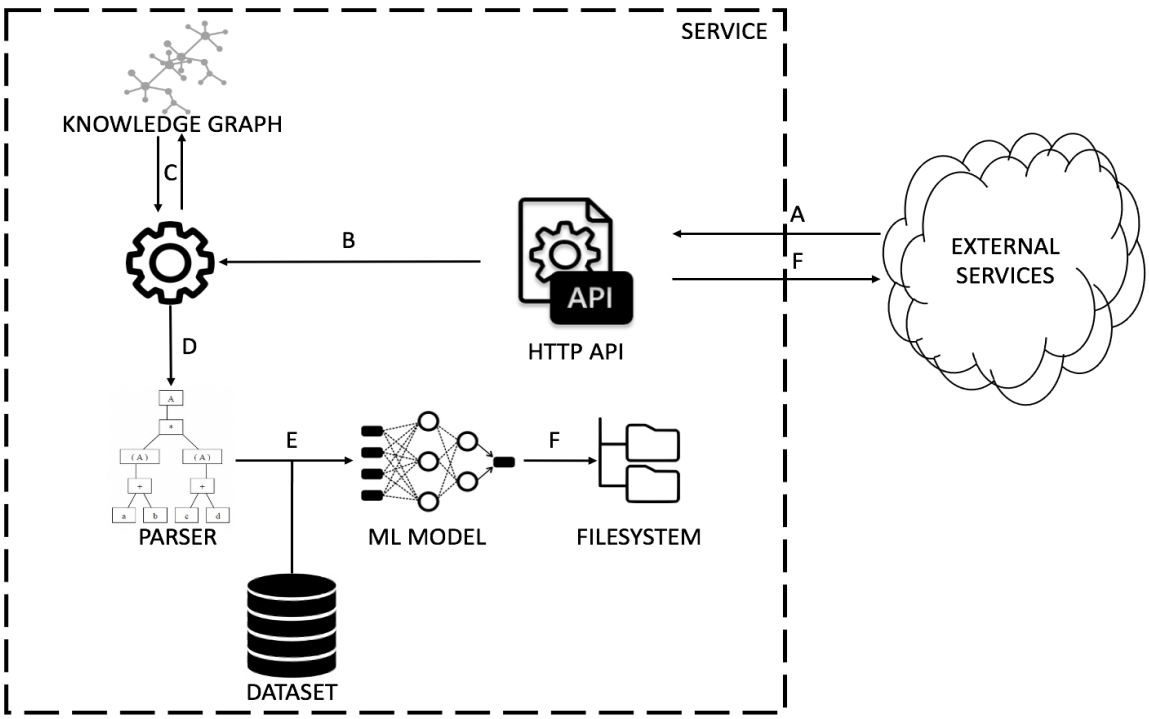


Figure 1. KG-based analytics conceptual design.

Figure 1 shows the sequence of actions performed for each request. The requests are triggered by an external service (A) via an HTTP API, and most of the information received via JSON is parsed and used to determine the details of the training process (B). In addition, the specific process is assigned a Universally Unique Identifier (UUID) that is stored in the Knowledge Graph (C) along with the data received in the incoming request and JSON snippet. If the creation of specific features is

needed, a grammar is used to parse the description with the variables and functions used to create such features (D). Then, a machine learning model is trained (E) with the corresponding dataset (E), taking into account the reference machine learning algorithm (either specified by the user or used by default for a given use case). Finally, the model is stored in the file system (F). The binary file of the model and its associated metadata can be retrieved from HTTP API through a specified endpoint. The architecture was implemented in an application using the Flask framework.

2.1 KG-based analytics API architecture

2.1.1 New model training request

A submission can be sent to train a new model, identified with its own universally unique identifier (UUID). Since model building can take a long time, this is done asynchronously. The model can be retrained by using model retrieval endpoints, with the assigned UUID of the model, which triggers the progress of model training, logs, and retrieval of the model itself.

2.1.2 Feature vector construction commands

Custom features can be constructed from a set of commands. These commands are a list of assignments, where each assignment can create a variable for each row of the dataset in order to be later used as features. The commands are specified in a json structure. If for any reason, the value of a feature vector for a certain row is None, that row is excluded from training.

2.1.3 Command structure (grammar)

Root element starts with the key "`feature_commands`". The value is one or more Assignments.

```
Commands -> {"feature_commands": [Assignment*]}
```

Assignment initializes a new variable. These variables can be used as features or as parts of later commands.

```
Assignment -> {"type": "assignment"
               "to": String
               "from": Value}
```

Value holds either the result of an operator or data. Data can come from a variable previously assigned, sensor data or a constant. The constants are the same for all rows, sensor data can be preprocessed before usage. Sources can be previously defined variables, sensor names, control parameters or laboratory measurements. The types of preprocessing include:

- AggregateData
  - window: specifies for how many elements in the past the operation is executed
  - mean: calculates mean of all the elements in the time window
  - median: calculates median of all the elements in the time window
  - slope: computes percentage change from the current element versus the element at the beginning of the time window
- SingleData
  - The data with a single operator is just point data that can be shifted by an offset. Offset of -1 means the data in the previous time instance is used.
- BooleanData

- If the value in the row is strictly above lower_bound or strictly below upper_bound the value is 1 and 0 otherwise. If any of the bounds is missing it is automatically set to – infinite (-∞) and infinite (-∞), respectively.

```
Value -> Data | Operator

Data -> Sensor | Constant | Variable

Sensor -> AggregateData | SingleData | BooleanData

AggregateData -> {"type": "data",
                  "operator": "mean" | "median" | "slope",
                  "source": String,
                  "window": Integer,
                 }

SingleData -> {"type": "data",
               "operator": "single",
               "source": String,
               "offset": Integer
               }

BooleanData -> {"type": "data",
               "operator": "boolean",
               "source": String,
               "lower_bound": Float,
               "upper_bound": Float
               }
```

Constants are a single number or None. Variables have to be defined in previously executed commands.

```
Constant -> {"type": "constant",
             "value": Float | None}

Variable -> {"type": "variable",
             "name": String}
```

Operators execute the operation between all the arguments and return a value.

sqrt: square root of the value

not: If the value is non-zero it sets if to 0. If the value is 0 it sets it to 1.

log: Computes the natural logarithm of Value.

"+" | "-" | "*" | "/" : Calculates Value1 [operator] Value2

"and" | "or" | "<=" | ">=" | "<" | ">" | "==": Calculates Value1 [operator] Value2. All the logical operators return 1 for true and 0 for false.

pow: Calculates Value1 to the power of Value2.

```
Operator -> UnaryOperator | BinaryOperator | If
```

```
UnaryOperator -> {"type": "operator",
                  "operator": "sqrt" | "not" | "log",
                  "arguments" : [Value]}

BinaryOperator -> {"type": "operator",
                "operator": "+" | "-" | "*" | "/" | "pow" | "and" | "or"
                          | "<=" | ">=" | "<" | ">" | "==",
                  "Arguments": [Value, Value]}
```

If is a special type of an operator. First the condition is evaluated for each row. For rows with elements of 1 the true Value is chosen, for rows with 0, the false Value is chosen.

```
If -> {"type": "operator",
        "operator": "if",
        "condition" : Value,
        "true": Value,
        "false": Value}
```

2.1.4 Feature vector

Feature vector is a list of variables, sensor names, control parameters or laboratory measurements that belong to a single unit.

Example:

The command constructs a custom feature named “median2” which is later used as a first element of the feature vector.

```
"feature_commands": {
      "type": "assignment",
      "to": "median2",
      "from": {
        "type": "data",
        "operator": "median",
        "source": "5FIC366.PV",
        "window": 2
      }
    },
  "feature_vector":["median2", "median3","5FI370.PV","5FIC386.PV"]
```

2.1.5 Custom model

POST http://goat.ijs.si:6780/new_custom_model

This endpoint creates a custom model. The request contains the following structure:

- unit: on which unit the feature is constructed, and the model is trained
- feature_commands: a set of commands described in Section 2.1.3 that construct variables which can be used as a features
- feature_vector: list of features described in Section 2.1.4
- target: The target value which the model is predicting. Only a single target is available.
- test: Optional parameter. If set to “True”, the dummy model will be returned. This is used for testing the APIs input and output without waiting for the model to finish training.
- model: Selects which model is used. The options are:
    - CatBoostRegressor
    - XGBRegressor

- LinearRegression
- SVR

2.1.6 Forecasting model

This endpoint constructs a new model for forecasting the state of the unit. The model predicts the state of all sensors and uses as input all sensor values and all control parameters. How much of the past values will be used as input and for what horizon in the future the predictions will be made, can be specified in the parameters at the time of the request. Since the model uses a fixed set of inputs, the feature vector will be a list of SingleData instances that contains either shifted sensor values from encoder time up to present and control parameters for the time of the prediction.

The request for new sensor state forecasting model can be done through

POST http://goat.ijs.si:6780/new_forecasting_model

Parameters:

- unit: Name of the unit on which the model is done. Example: `CrudeDistillationUnit-2`
- encoder: number of past inputs which model will take. Example: 20. In this case, the model needs all sensor values from 20 minutes in the past up to the present.
- horizon: For how many minutes in the future the forecast will be made. Example: 60. In this case, 60 minutes into the future will be the forecast time. The control parameters which are provided need to be chosen for this time.

2.1.7 Model

Once the model is trained, the model can be obtained with the following request:

GET http://goat.ijs.si:6780/get_model/<uuid>

The <uuid> is the identifier that was returned when the request was made. The specifics for how to load and use each model can be found in the metadata.

2.1.8 Model metadata

Metadata can be retrieved with the following request:

GET http://goat.ijs.si:6780/get_metadata/<uuid>

The metadata includes information about the feature vector construction, training metrics and model parameters. The structure is a json with the following fields:

- UUID: the universally unique identifier which was assigned to this model
- config: the configuration file which was sent along with the request for this model
- metrics: shows metrics such as Mean Average Error (MAE) or Root Mean Squared Error (RMSE). In case of quantile results, these metrics were calculated only on the most likely prediction (quantile 0.5)
- feature_vector: exact order of features that the model requires. Features are variable names in the case of the custom model and SingleData instances in the case of forecasting model. The specific of the feature construction can be found in the config parameter which shows all the options the request was called with.

- model_type: model type
- model_usage: Instruction on how to load and use the model
- status: Status of the model training. Possible statuses:
  - processing – model is still being trained
  - successful – training finished successfully, the model can be retrieved
  - error – something went wrong, and the training is not done

2.1.9 logs

Logs can be retrieved with the following request:

GET http://goat.ijs.si:6780/get_logs/<uuid>

The logs show the parameters of the query calls. The process of the feature vector constructor and the training of the model can be seen. The most important feature of the logs is the display of errors that occurred during the processing of the query.

2.1.10 New data upload

New data can be uploaded at the following endpoint:

POST http://goat.ijs.si:6780/datasets/upload

The structure of the request is the following:

```
"file": <unit_name>.csv

"data_schema_entry": {{<unit_name>: {{sensors: []*, targets: []*,
controls: []*}} }}
```

Dataset file along with the appropriate information needs to be uploaded. Example data file content:

```
datetime,sensor1,sensor2,target1,target2,control1
1514764860,1,2,3,4,5
1514764920,0,0,0,0,0
1514764980,1,5,2,5,4
```

The required file structure is .csv with the first row being the header. Necessary column is datetime which needs to have timestamp values in unix time. Each column needs to be classified in one of the three categories:

- sensors: sensor data available for the unit;
- targets: sensor or laboratory measurements data which can be used as target for models;
- controls: these are the control parameters which are set by the user.

In this case the data_schema_entry json needs to be the following:

```
{
  "test_unit": {
    "sensors": [
      "sensor1",
      "sensor2"
    ],
    "targets": [
      "target1",
      "target2"
```

```
    ],
    "controls": [
      "control1"
    ]
  }
}
```

2.1.11 Ontology retrieval

Each successful query updates the ontology with data about new models. The current ontology can be retrieved via the endpoint:

GET http://goat.ijs.si:6780/get_ontology

The endpoint returns a file named FACTLOG-ontology-models-representation.owl. All relevant information about each operation performed on the above endpoints is stored in the ontology-based Knowledge Base. Figure 2 shows an example of individuals obtained after multiple endpoint executions.

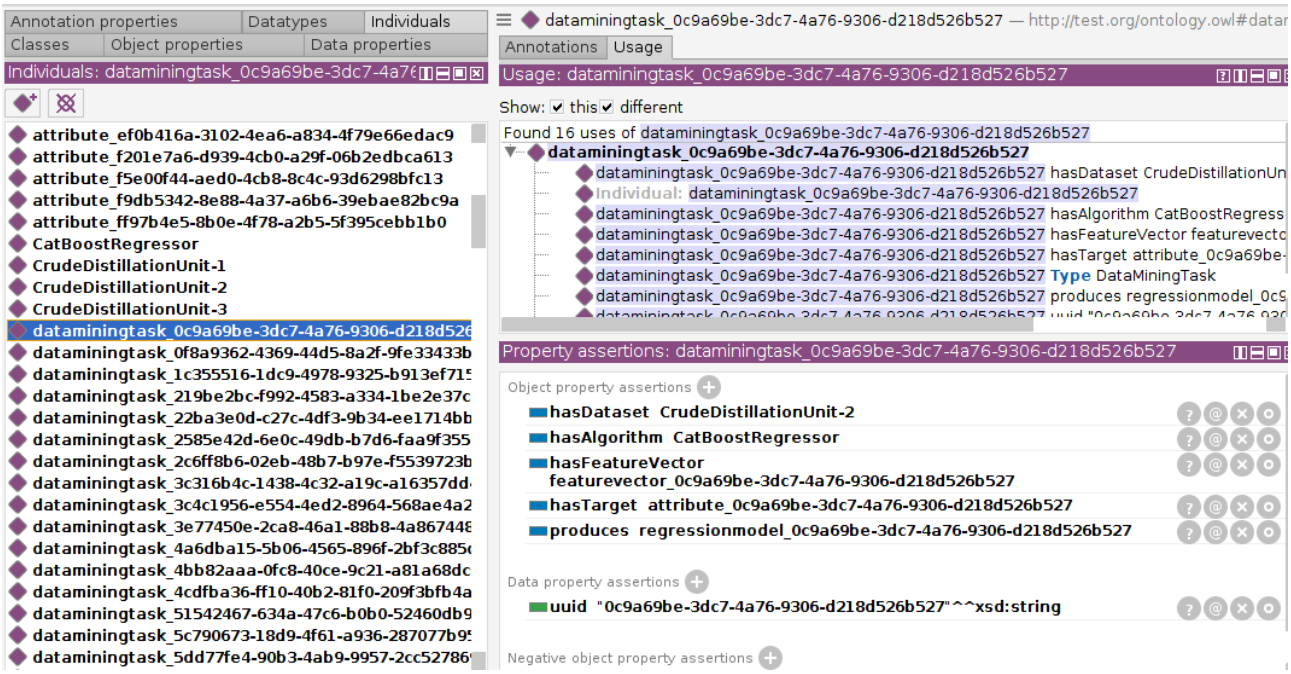


Figure 2. Ontology augmentation based on multiple API calls.

3 Knowledge graph model

3.1 Ontology definition

The need to encode domain knowledge in the context of data mining processes is not new and has been addressed by several researchers in the past. Such efforts need to be considered in order to build a unified ontology that can handle multiple use cases and provide intelligence for building machine learning models that solve the challenges associated with each use case. Following the MIREOT principle [1], we analyzed ontologies related to the artificial intelligence domain. In paper [2], DAMON (Data Mining Ontology for Grid Programming) was developed to provide a reference model for data mining tasks, methodologies, and available software. A heavyweight ontology was developed by [3,4], which provides means for representing data mining entities, inductive queries, and data mining scenarios. In paper [5] KDDONTO was developed, which focuses on the discovery of data mining algorithms.

One of the possible approaches for developing machine learning models is the standard CRISP-DM, which specifies six phases of model development: business understanding, data understanding, data preparation, modeling, evaluation and deployment. With KG-based analytics for process optimization, we aim to capture some business understanding so that baseline models can be created for a given use case, regardless of the amount of domain knowledge of the person requesting such a model. In addition, the service provides means for data preparation, modeling, and evaluation. If there is insufficient knowledge about a particular use case in the knowledge base, a more detailed specification of the data mining task is required, indicating how the features should be built.

The ontologies mentioned above provide a solid foundation for building a unified ontology that can be used to encode domain knowledge related to multiple use cases and automatically building baseline artificial intelligence models for them. In particular, we used the Basic Formal Ontology to define the upper classes and sought concepts and terminology defined in the Information Artifact Ontology (IAO), Ontology of Data Mining (OntoDM), DAta Mining ONtology (DAMON), and the Data Mining OPtimization Ontology (DMOP). This particular ontology was incorporated into the broader ontology developed for the FACTLOG project and presented in D4.2. We developed and persisted the ontology using the Web Ontology Language (OWL). Figure 3 shows a screenshot of the ontology we developed, while Table 1 describes all ontology concepts and the reference ontologies from which they were taken.

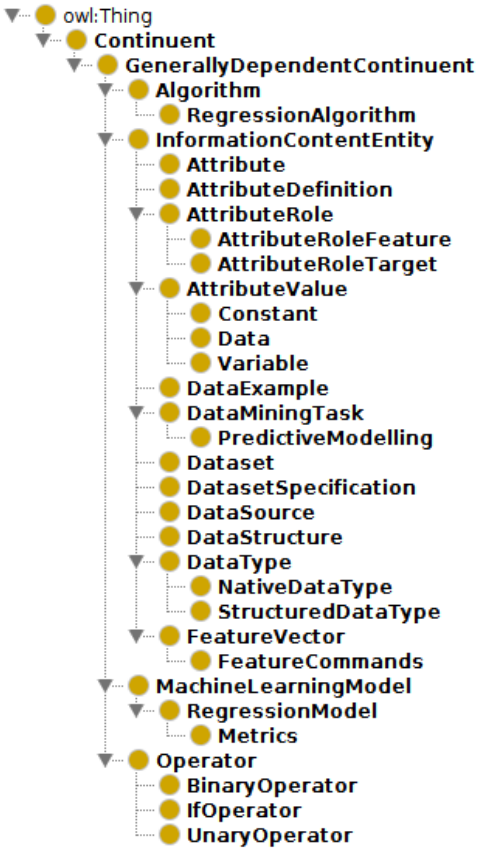


Figure 3. Hierarchy of new concepts included in Ontology

Upper ontology	**Concept**	**Description**
BFO	Thing	Thing is considered the highest-level concept, from which the rest of the concepts descend.
BFO	Continuant	A continuant is an entity that persists, endures, or continues to exist through time while maintaining its identity.
	Generally Dependent Continuant	Dependent continuants are related to their bearers by inherence. Inherence is defined as a one-sided, existential dependence relation. Generally Dependent Continuants can exist in a multiplicity of bearers.
OntoDM, DMOP	Algorithm	A data mining algorithm is an algorithm that solves a data mining task and as a result outputs a generalization, which is realized in a *Machine Learning Model*. It is usually published in some document (journal publication or technical report). Aligns with DMOP *DM-Algorithm*, and the OntoDM *Data Mining Algorithm* concepts.

Upper ontology	Concept	Description
OntoDM	Regression Algorithm	Regression algorithm is a data mining algorithm that solves a regression task and as a result produces a regression model.
IAO	Information Content Entity	An entity which is generically dependent on some material entity and which stands in a relation of aboutness to some entity.
OntoDM	Attribute	Qualities of a dataset so the relation to the class Dataset is expressed via the property has_quality Attribute.
	Attribute Definition	Definition of a given attribute.
OntoDM	Attribute Role	Role of the attribute in a given Dataset.
	Attribute Role Target	Signals Target role in a particular Dataset.
	Attribute Value	Value of a given Attribute.
	Constant	Invariable or unchanging value.
	Data	Facts and statistics collected together for reference or analysis.
	Variable	A quantity that may assume any one of a set of values.
OntoDM	Data Example	A data example is a data item that represents one unit of data, and it is a part of a dataset. It is a synonym with a case or example or observation in statistics.
OntoDM	Data Mining Task	A data mining task is an objective specification that specifies the objective that a data mining algorithm needs to achieve when executed on a dataset to produce as output a generalization.
OntoDM	Predictive Modeling	In the task of predictive modeling, we are given a dataset that consists of examples of the form (d,o), where d is of type Td and each o is of type To. To learn a predictive model means to find a mapping from description to output m::Td--->To that fits data closely. Aligns with OntoDM *Predictive Modeling Task* concept.
OntoDM	Dataset	A collection of data.
OntoDM	Labeled Dataset	Labeled dataset is a dataset specification for datasets that have both a descriptive and output specification of the data examples contained in the dataset.

Upper ontology	Concept	Description
OntoDM	Dataset Specification	A dataset specification is a data item specification about a dataset defined with a data type specification of the data examples aggregated in the dataset.
DAMON	Data Source	The input on which data mining algorithms work to extract new knowledge.
	Data Structure	Represents a data organization, management, and storage format that enables efficient access and modification.
	Data Type	Specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error.
	Native Data Type	Data types that cannot be divided into other Data Types. Common Native Data Types are booleans, integers, or characters.
	Structured Data Type	User-defined data type containing one or more named attributes, each of which has a data type.
	Feature Vector	Vector of features used to issue a prediction with a machine learning model.
	Feature Commands	The commands are a list of assignments, each assignment can create a variable used by following commands or a feature used for a specific data mining task.
OntoDM	Predictive Model	A predictive model M for types Td and Tc is a function that takes an object of type Td and returns an object of type Tc, i.e. has the signature m:Td-->Tc.
OntoDM	Regression Model	A regression model is a feature-based predictive model for primitive output and denotes predictive models that are built on data having a real datatype on the output part and set of features (represented by a descriptive tuple of primitives) on the descriptive part.
	Metrics	Metrics calculated for the model evaluated on a validation dataset.
DMOP	Operator	A mapping or function that acts on elements of a space to produce elements of another space. Aligns with the DMOP *DM-Operator* concept.
	Binary Operator	An operator that operates on two operands and

Upper ontology	Concept	Description
		manipulates them to return a result
	If Operator	Evaluates a condition and assigns either value chosen for true result or value chosen for false result to the chosen target.
	Unary Operator	An operator used to operate on a single operand to return a new value.

Table 1. Ontology concepts included in building our ontology

3.2 Data analytics ontology integrating into the FACTLOG ontology

3.2.1 Principles

Entities in the FACTLOG semantic framework have been arranged based on the Basic Formal Ontology (BFO) which is a formal ontology framework developed by Barry Smith and his associates [6]. In BFO, there are two varieties which are continuants comprehending continuant entities such as three-dimensional enduring objects and occurrents comprehending processes conceived as extended through (or as spanning) time. To adopt BFO framework will provide availability to merge the other Cognitive Twin domain ontology structured by BFO. The method of ontology design is presented on Figure 4 below.

Originated from BFO, ontology design principles of FACTLOG are as follows:

- use single nouns (except data) and avoid acronyms
- ensure unicity of terms and relational expressions
- distinguish the general from particular
- provide all non-root terms with definitions
- use essential features in defining terms and avoid circularity
- start with the most general terms in the domain
- use simpler terms than the term you are defining (to ensure intelligibility)
- do not create terms for universals through logical combination
- structure ontology around is_a hierarchy and ensure is_a completeness
- single inheritance

3.2.2 Methodology

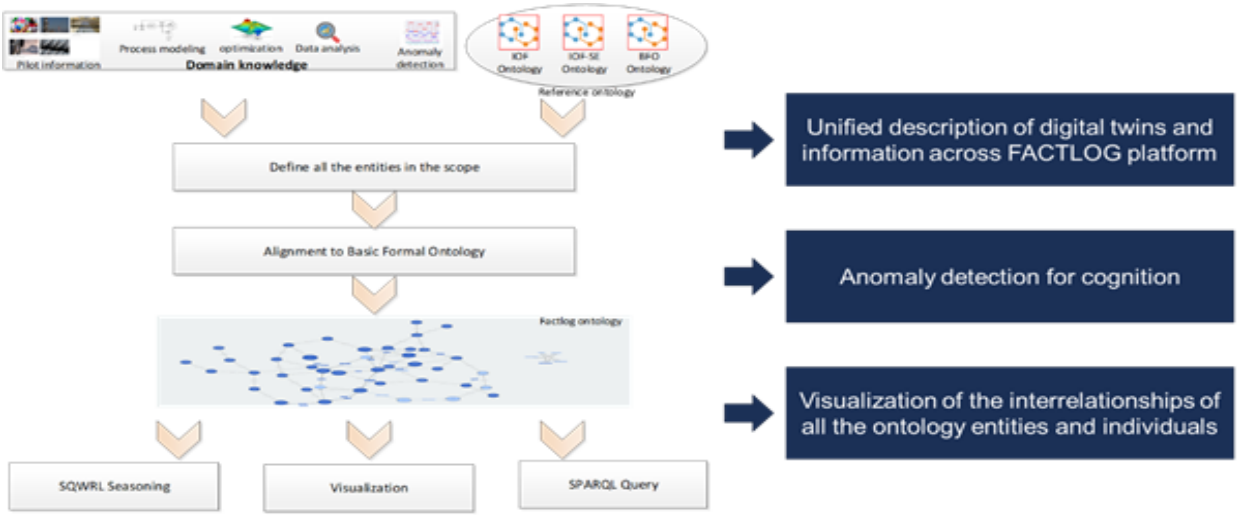


Figure 4. Ontology development workflow

In order to design the unified ontology for developing knowledge graph models supporting cognitive capabilities, one of the most well-known systems thinking development methodology (D8.3) through domain knowledge has been applied to define the domain knowledge including: (i) pilot information; (ii) process modeling; (iii) optimization; (iv) data analysis; (v) anomaly detection. IoF ontology (industrial Ontology Framework), BFO ontology and IoF SE ontology are three main reference ontologies. By composing a top-level overview, abstract concepts form domain specific knowledge from FACTLOG pilots and technical views. After the extraction of entities from FACTLOG pilots, the list of classes was updated in a comparison with existing ontology such as IOF-SE ontology, and IOF ontology. And then, all the entities were rearranged in the BFO structure. Finally, the SQWRL and SPARQL are used to support reasoning and query of the OWL models.

All the ontology concepts are mainly used for three aspects:

1. Unified description of digital twin and information across the FACTLOG platform.
2. Ontology reasoning for anomaly detection.
3. Visualization of the interrelationships of all the ontology entities and individuals.

3.2.3 Ontology Framework

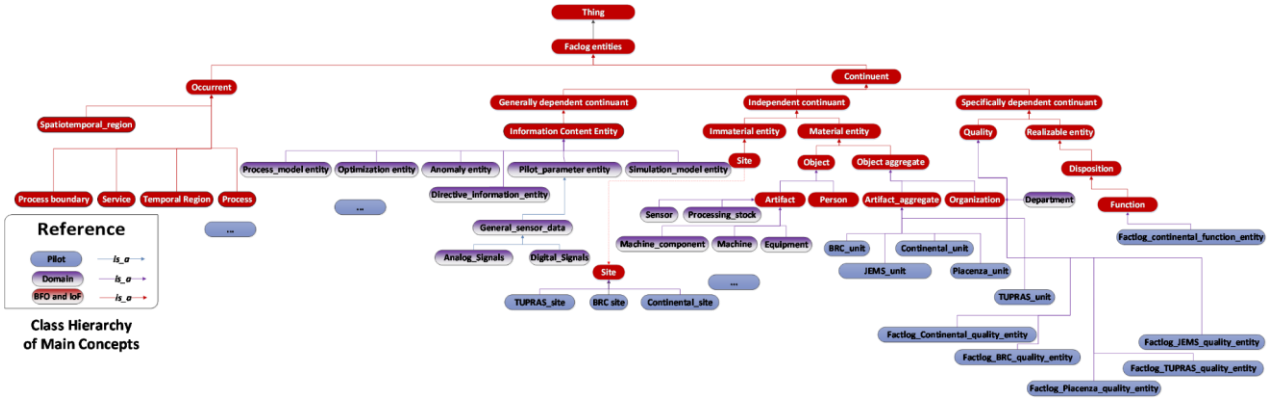


Figure 5. Ontology framework based on BFO.

As shown in Figure 5 the FACTLOG ontology is developed based on basic formal ontology. The red blocks refer to BFO and IoF concepts. The purple blocks refer to domain concepts. The blue ones which are defined under BFO and domain concepts are used to define FACTLOG concepts. The whole FACTLOG entities include occurrent and continuant entities.

- A **continuant** is an entity that persists, endures, or continues to exist through time while maintaining its identity.
- An **occurrent** is an entity that unfolds itself in time or it is the instantaneous boundary of such an entity (for example a beginning or an ending) or it is a temporal or spatiotemporal region which such an entity occupies_temporal_region or occupies_spatiotemporal_region.

Under the occurrent entity, several concepts are defined:

- **Process**: an occurrent that has temporal proper parts and for some time t, p s-depends_on some material entity at t.
- **Process_boundary**: a temporal part of a process and the process has no proper temporal parts.
- **Service**: Service is delivered when the service implements the system function.
- **Spatiotemporal_region**: an occurrent entity that is part of spacetime.
- **Temporal_region**: an occurrent entity that is part of time as defined relative to some reference frame.

Under the continuent entity, several concepts are defined:

- **generically_dependent_continuant** is a continuant that generally depends on one or more other entities.
- **independent_continuant**, a continuant which is such that there is no c and no t such that b s-depends_on c at t.
- **specifically_dependent_continuant**, a continuant and there is some independent continuant c which is not a spatial region and which is such that b s-depends_on c at every time t during the course of b's existence

Under **generically_dependent_continuant** entity, several concepts are defined:

- **Information_content_entity**, a generically dependent continuant that is about something.
    - **Process_model_entity**, a virtual concept used to define process model.
    - **Optimization_entity**, a virtual concept used to define optimization concept.
    - **Simulation_model_entity**, a virtual concept used to define simulation model concepts.
    - **Pilot_parameter_concept**, a virtual entity to define FACTLOG pilot parameters.
        - General sensor data, sensor data used for all the FACTLOG pilot
    - **Directive_information_entity**, a plan specification which describes the inputs and output of mathematical functions as well as workflow of execution for achieving a predefined objective. Algorithms are realized usually by means of implementation as computer programs for execution by automata.
    - **Anomaly_entity**, a virtual entity to support anomaly detection.
    - **Data_analysis_entity**, a virtual entity used for data analysis.
- **Specifically_dependent_continuant**, is a continuant and there is some independent continuant c which is not a spatial region and which is such that b s-depends_on c at every time t during the course of b's existence
    - **Quality**, a specifically dependent continuant that, in contrast to roles and dispositions, does not require any further process in order to be realized.
        - FACTLOG_BRC_quality_entity, quality used in the BRC pilot.
        - FACTLOG_Continental_quality_entity, quality used in the Continental pilot.
        - FACTLOG_JEMS_quality_entity, quality used in the JEMS pilot.
        - FACTLOG_Piacenza_quality_entity, quality used in the Piacenza pilot.
        - FACTLOG_TUPRAS_quality_entity, quality used in the TUPRAS pilot.
    - **realizable_entity**, a specifically dependent continuant that inheres in some independent continuant which is not a spatial region and is of a type instance of which are realized in processes of a correlated type.
        - **Disposition**, a realizable entity and b's bearer is some material entity and b is such that if it ceases to exist, then its bearer is physically changed, and b's realization occurs when and because this bearer is in some special physical circumstances, and this realization occurs in virtue of the bearer's physical make-up.
            - **Function**, a disposition that exists in virtue of the bearer's physical make-up and this physical make-up is something the bearer possesses because it came into being, either through evolution (in the case of natural biological entities) or through intentional design (in the case of artifacts), in order to realize processes of a certain sort.
                - FACTLOG_continental_function_entity, functions used in Continental pilot.

        - **Role**, a realizable entity and b exists because there is some single bearer that is in some special physical, social, or institutional set of circumstances in which this bearer does not have to be and b is not such that, if it ceases to exist, then the physical make-up of the bearer is thereby changed.

- **Independent_continuant**, a continuant which is such that there is no c and not such that b s-depends_on c at t.
    - **immaterial_entity**, which are divided into two subgroups: boundaries and sites, which bound, or are demarcated in relation, to material entities, and which can thus change location, shape and size and as their material hosts, move or change shape or size (for example: your nasal passage; the hold of a ship; the boundary of Wales).
        - **Site**, three-dimensional immaterial entity that is (partially or wholly) bounded by a material entity or it is a three-dimensional immaterial part thereof.
            - BRC_site, site used in BRC.
            - Continental_site, site used in Continental.
            - TUPRAS_site, site used in TUPRAS.
    - **material_entity**, which can preserve their identity even while gaining and losing material parts. Continuants are contrasted with occurrents, which unfold themselves in successive temporal parts or phases.
        - **Object**, a material entity which manifests causal unity of one or other of the types listed above and is of a type (a material universal) instance of which are maximal relative to this criterion of causal unity.
            - **Artifact**, an Object that was designed by some Agent to realize a certain Function.
                - **Sensor**, a device that produces an output signal for the purpose of sensing a physical phenomenon.
                - **Processing stock**, is an artifact in an industrial site corresponding to any material in the process of producing or manufacturing finished product.
                - **Machine component**, compositions for constructing machines.
                - **Machine**, a physical system using power to apply forces and control movement to perform an action.
                - **Equipment**, the set of physical resources serving to equip a person or thing implementing used in an operation or activity
        - **Person**, an object that is a human being.
            - **Object_aggregate**, an object aggregates if and only if there is a mutually exhaustive and pairwise disjoint partition of that object into other objects.
            - **Artifact_aggregate**, a collection of artifacts that are arranged by some Agent to realize a certain Function.

- ❖ BRC_unit, a company group generally equivalent in size and character to implement BRC services.
- ❖ Continental_unit, a company group generally equivalent in size and character to implement Continental services.
- ❖ JEMS_unit, a company group generally equivalent in size and character to implement JEMS services.
- ❖ Piacenza_unit, a company group generally equivalent in size and character to implement Piacenza services.
- ❖ TUPRAS_unit, a company group generally equivalent in size and character to implement TUPRAS services.

- **Organization**, an object aggregate that corresponds to social institutions such as companies, societies etc. that does something.
  - ➢ **Department**, an organizational unit in FACTLOG.

3.2.4 Data analysis Ontology under BFO ontology

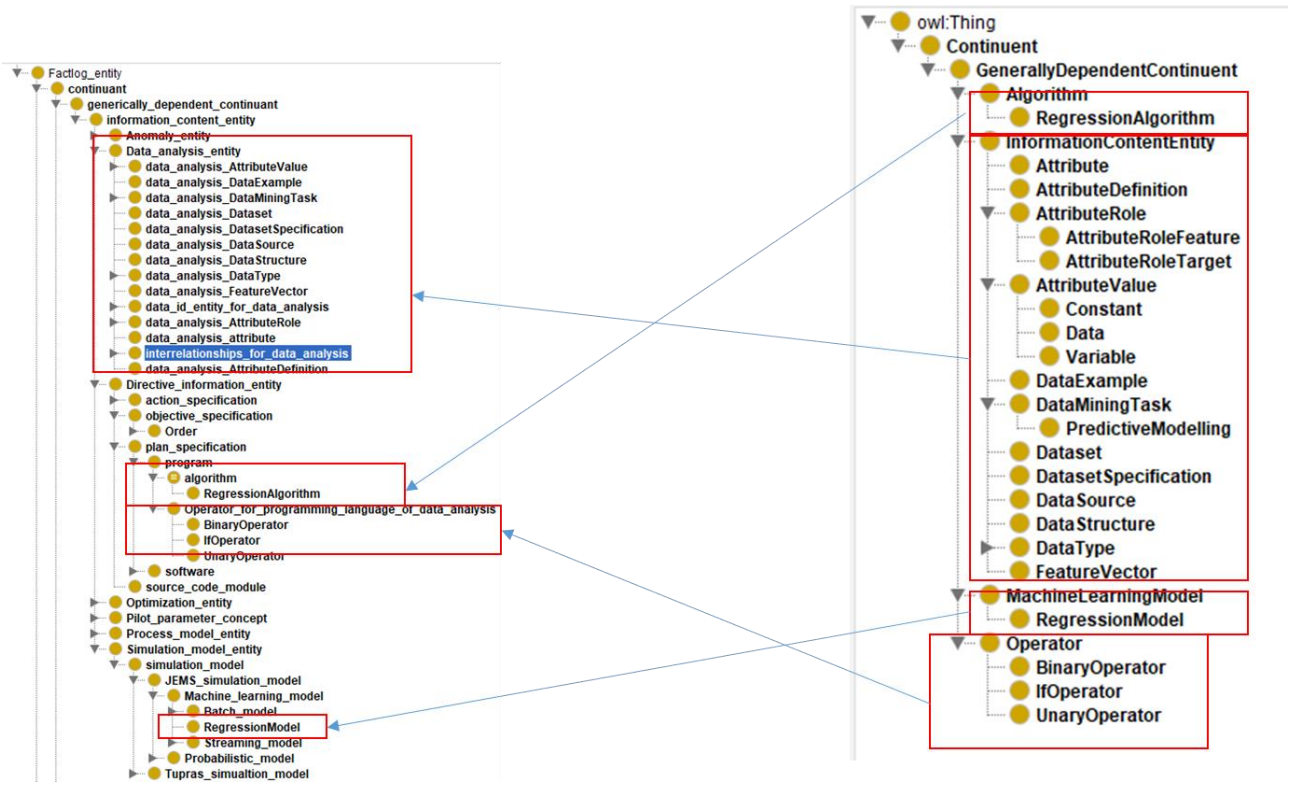


Figure 6. Data analytics concepts integrated with BFO ontology.

As shown in Figure 6, the data analysis ontology is integrated into BFO ontology.

Data_analysis_entity includes several classes:

…

- data_analysis_attribute, attributes used for data analysis
- data_analysis_AttributeDefinition, attribute definition used for data analysis

- data_analysis_AttributeRole, the role of the attribute used for data analysis
  - data_analysis_AttributeRoleFeature, role feature of attribute for data analysis
  - data_analysis_AttributeRoleTarget, role feature of attribute for data analysis
  - data_analysis_AttributeValue, attribute value of data analysis
  - data_analysis_Constant, constant value of data analysis
  - data_analysis_Data, data for data analysis
  - data_analysis_Variable, variable for data analysis
- data_analysis_DataExample, data example for data analysis
- data_analysis_DataMiningTask, data mining task for data analysis
  - data_analysis_PredictiveModelling, Predictive Modeling entity for data analysis
- data_analysis_Dataset, data set for data analysis
- data_analysis_DatasetSpecification, Data set specification of data analysis
- data_analysis_DataSource, Data resource of data analysis
- data_analysis_DataStructure, Data structure of data analysis
- data_analysis_DataType, Data type of data analysis
  - data_analysis_NativeDataType, Native Data Type for data analysis
  - data_analysis_StructuredDataType, Structured Data Type for data analysis
- data_analysis_FeatureVector, Feature Vector for data analysis
- data_id_entity_for_data_analysis, data id entity for data analysis
  - controller_id_for_data_analysis
  - sensor_id_for_data_analysis
  - target_Id_for_data_analysis
  - unit_Id_for_data_analysis
- interelationships_for_data_analysis, interrelationships among data analysis entities
  - sequence_from_unit_to_unit, a directional relationship from one unit to another unit

…

- algorithm
  - RegressionAlgorithm, regression algorithm for data analysis
- Operator_for_programming_language_of_data_analysis, operator used in the programming language of data analysis
  - BinaryOperator, Binary operator used for programming data analysis algorithm
  - IfOperator, If operator used for programming data analysis algorithm
  - UnaryOperator, Unary operator used for programming data analysis algorithm

…

- simulation_model
  - JEMS_simulation_model
    - RegressionModel, regression model for data analysis

4 KG – driven model services operation

The initial ontology engine was integrated into the cognitive services architecture. The initial cognitive API has been extended to allow both the use of existing cognitive services and the creation of AI models on-the-fly with fully custom model configuration. The predefined/configured models can be used to enable "simple/light use" of services for some of the main use cases, such as streaming sensor data prediction. The custom endpoints, on the other hand, enable the implementation of any specific use case scenario for smart factory operations. In the following two sections, we present a pilot case that leverages both main use cases for ontology-based cognitive API usage.

4.1 Forecasting model example

The forecasting model endpoint supports the construction and retrieval of models built for the use case of sensor data prediction from Deliverable D3.2. The use case which is supported uses all available data from one unit and predicts all sensor data for the same unit.

The parameters for forecasting model call can be seen in Section 2.1.6. For this example, we want to make a model that predicts all sensors for 60 minutes into the future and uses the last 20 minutes of data. The service needs to be called with only three parameters: encoder, horizon and unit. All the other parameters such as control parameters and sensor list are fetched from ontology and don't need to be provided. The service always uses a full sensor and control parameter list when building the models. The data about the model inputs and usage can be found in model metadata. In this example we would call the /new_forecasting_model with the following parameters:

```
{
        "unit": "CrudeDistillationUnit-2",
        "encoder": 20,
        "horizon": 60
}
```

The server returns a response:

```
200 OK :
{
    "UUID": "64aeba8a-9314-480e-95b2-6629e8a90cf6"
}
```

If we immediately try to get the model at /get_model/<uuid> we get:

```
400 BAD REQUEST :
The model is still in training. Please try again later.
```

The model training can take quite a while, especially if the encoder length is big and if the unit has many sensors. We can check the status of the training either in

- The model metadata under tag status. Now it should be set at processing and should change to either successful or error.
- Send a request for the model again. Once done, the model should return with a HTTP status code of 200 if successful or 400 otherwise.
- Check the /get_logs endpoint. In logs, the training progress is displayed. Example for our case:

```
2022-06-08 13:17:17,642 [INFO] Start of CrudeDistillationUnit-2
2022-06-08 13:17:17,656 [INFO] Train and validation pair construction
2022-06-08 13:17:22,661 [INFO] Available device: cuda
2022-06-08 13:17:23,009 [INFO] Model fitting
2022-06-08 13:17:29,146 [INFO] Optimal learning rate found:
0.0009999999999999994
2022-06-08 13:17:29,398 [INFO] Saving model to:
data/users/64aeba8a-9314-480e-95b2-6629e8a90cf6/model.pt
2022-06-08 13:17:37,451 [INFO] Model training finished successfully
```

Once we know the training is finished, we can save the model locally.

```
response =
requests.get(url=f"http://goat.ijs.si:6780/get_model/64aeba8a-9314-480e-
95b2-6629e8a90cf6")

with open('model.pt', 'wb') as f:
      f.write(response.content)
```

We can check the metadata for details about the model. The metadata is available at:

GET http://goat.ijs.si:6780/get_metadata/64aeba8a-9314-480e-95b2-6629e8a90cf6

The important thing here is the feature vector. The feature vector defines what kind and in what order the input data should be used for using the model.

```
{
    "UUID": "64aeba8a-9314-480e-95b2-6629e8a90cf6",
    "config": { },
    "feature_vector": [
        {
            "offset": -19,
            "operator": "single",
            "source": "5FIC365.PV",
            "type": "data"
        },
        {
            "offset": -19,
            "operator": "single",
            "source": "5TI469.PV",
            "type": "data"
        },
…
         {
            "offset": 0,
            "operator": "single",
            "source": "5TI485.PV",
            "type": "data"
        },
              {
            "offset": 60,
            "operator": "single",
            "source": "5FIC365.SV",
            "type": "data"
        },
        {
```

```
                "offset": 60,
                "operator": "single",
                "source": "5FIC369.SV",
                "type": "data"
            },
…
            ],
        "metrics": {
            "MAE": 2.8333,
            "RMSE": 3.5609
        },
        "model_type": "TorchNetworkQuantile",
        "model_usage": "model = TorchNetworkQuantile()

model.load('model.pt')
Example of model loading and usage can be found on
https://github.com/JozefStefanInstitute/factlog-data-
pipeline/tree/main/api/examples/new_forecasting_model_example.py",
        "status": "successful"
}
```


Metadata includes UUID, original request config, feature vector, metrics, model type, usage and status.

The feature vector is expressed using feature construction command notation for singular data described in Section 2.1.3. We can see that it uses all sensors from the past 20 minutes (offset -19) in the past up to current time (offset 0) and also requires the control parameters at the time of the forecast (offset 60).

Model can be loaded and called with appropriate feature vector:

```
    model = TorchNetworkQuantile()
    model.load('model.pt')

    with open('example_feature_vector_forecasting.json', 'rb') as f:
        fet = json.load(f)
        feature_vector = fet['feature_vector']

    result = model.predict(feature_vector)
```

Giving us a result:


```
[array([1114.70983887, 1148.42504883, 1171.24401855]),
array([79.30570984, 87.68345642, 91.94986725]), array([367.06661987,
382.61004639, 419.50704956]), …]
```


The model is a quantile model meaning it outputs 3 values for each prediction, the lower bound (quantile 0.1), the most likely prediction (quantile 0.5) and upper bound (quantile 0.9).

4.2 Custom model example

The functionality of the models constructed here can support the use cases of estimating energy consumption and C2/C5 impurity content regarding Tupras pilot case. The same models that were constructed manually in the D3.2 can also be constructed using this endpoint.

Parameters for the custom model include unit, target, feature commands, and feature vector. Unit is the entity from which the model will obtain data. Target is the name of a sensor, variable, or laboratory measurement that exists on the unit. Unlike with forecasting models from the previous section, for the custom model, all features must be defined when making the request. This must be done using feature construction commands. Its structure is described in Section 2.1.3.

The configuration example:

```
{
  "unit": "CrudeDistillationUnit-2",
  "target": "Energy Exchanger 5E-12 [kJ/h]",
  "feature_commands": [...],
  "feature_vector": [...],
}
```

Feature vector and its construction commands example:

```
    "feature_commands": [
            {
                "from": {
                    "operator": "median",
                    "source": "5FIC366.PV",
                    "type": "data",
                    "window": 2
                },
                "to": "median2",
                "type": "assignment"
            },
            {
                "from": {
                    "offset": -1,
                    "operator": "single",
                    "source": "5TI472.PV",
                    "type": "data"
                },
                "to": "Tc",
                "type": "assignment"
            },
            {
                "from": {
                    "condition": {
                        "lower_bound": 300,
                        "operator": "boolean",
                        "source": "Tc",
                        "type": "data",
                        "upper_bound": 800
                    },
                    "false": {
```

```
                        "type": "constant",
                        "value": 0
                    },
                    "operator": "if",
                    "true": {
                        "arguments": [
                            {
                                "offset": 0,
                                "operator": "single",
                                "source": "Tc",
                                "type": "data"
                            },
                            {
                                "type": "constant",
                                "value": 2
                            }
                        ],
                        "operator": "pow",
                        "type": "operator"
                    },
                    "type": "operator"
                },
                "to": "saturation_pressure_butane",
                "type": "assignment"
            }
        ]
"feature_vector":[
   "median2",
   "saturation_pressure_butane" ]
```

The response is the same as with the forecasting endpoint, giving us the UUID.

```
200 OK :
{
    "UUID": "4a6dba15-5b06-4565-896f-2bf3c885db2b"
}
```

The metadata can be retrieved from:

GET http://goat.ijs.si:6780/get_metadata/4a6dba15-5b06-4565-896f-2bf3c885db2b

```
{
   "UUID": "4a6dba15-5b06-4565-896f-2bf3c885db2b",
   "config": {},
   "metrics": {
           "RMSE": 78664.7641727221
       },
    "model_type": "CatBoostRegressor",
    "model_usage":
    "model = pickle.load(model.pickle)
result =    model.predict(input_vector)
https://github.com/JozefStefanInstitute/factlog-data-
pipeline/tree/main/api/examples/new_custom_model_example.py",
    "status": "successful"
}
```

In the case of the custom model, the configuration we used to invoke the endpoint is stored in the metadata. The configuration contains the feature vector, the feature commands, and any other parameters we used in the request. The main difference between the models is the loading, which here uses the pickle library. The example can be seen in model_usage. The metric used here is only the RMSE.

```
    response =
requests.get(url=f"http://goat.ijs.si:6780/get_model/4a6dba15-5b06-4565-
896f-2bf3c885db2b")
    filename = response.headers['Content-
Disposition'].split("filename=")[1]
    with open(filename, 'wb') as f:
        f.write(response.content)

    with open(filename, 'rb') as f:
        model = pickle.load(f)

    feature_vector = [1153.09, 87.39]

    result = model.predict(feature_vector)
```

Giving us the result:

```
4977539.088928778
```

5 Conclusion

We have developed an ontology that refers to the Basic Formal Ontology as the upper ontology and reuses concepts from well-known ontologies developed in the past to model concepts and processes related to artificial intelligence. In addition, based on the ontology, we have developed a custom JSON format to interface with the cognitive API services and a knowledge-based service that provides endpoints to train machine learning models based on either a model specification or a known use case (for which domain knowledge exists within the application to create the features and train a baseline model).

The main impact we report for this deliverable are the generalization of the cognitive services we have designed, which allows for greater scalability in terms of training models (models can be re-trained based on a model or use case specification), and the service can be further enriched to support more use cases and thus a larger number of baseline models. We anticipate that such functionality will enable training of sophisticated models by people with less specialized knowledge than required to create such a service, “democratizing” the use of machine learning in smart factories.

References

[1] Courtot, Mélanie, et al. "MIREOT: the minimum information to reference an external ontology term." ICBO (2009): 87.

[2] Cannataro, Mario, and Carmela Comito. "A data mining ontology for grid programming." Proc. 1st Int. Workshop on Semantics in Peer-to-Peer and Grid Computing. 2003.

[3] Panov, Pance, Sašo Džeroski, and Larisa Soldatova. "OntoDM: An ontology of data mining." 2008 IEEE International Conference on Data Mining Workshops. IEEE, 2008.

[4] Panov, Panče, Larisa Soldatova, and Sašo Džeroski. "Ontology of core data mining entities." Data Mining and Knowledge Discovery 28.5 (2014): 1222-1265.

[5] Diamantini, Claudia, Domenico Potena, and Emanuele Storti. "Kddonto: An ontology for discovery and composition of kdd algorithms." Third Generation Data Mining: Towards Service-Oriented Knowledge Discovery (SoKD'09) (2009): 13-24.

[6] Smith, Barry, Anand Kumar, and Thomas Bittner. "Basic formal ontology for bioinformatics." (2005).