DAIKIRI
Erklärbare Diagnostische KI für industrielle Daten

**Project Number**: 01IS19085B **Start Date of Project:** 01/01/2020 **Duration:** 24 months

# Deliverable 4.1
# Structured Machined Learning Library

| | |
|---|---|
| **Dissemination Level** | Public |
| **Due Date of Deliverable** | Month 15, 31.03.2021 |
| **Actual Submission Date** | Month 15, 31.03.2021 |
| **Work Package** | WP4 — Explainable Machine Learning |
| **Tasks** | T4.1, T4.2 |
| **Type** | Report |
| **Approval Status** | Final |
| **Version** | 1.0 |
| **Number of Pages** | 19 |

**Abstract**: This deliverable presents the Onto*learn* structured machine learning library, usage, design, architecture and experiments.

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

This project has received funding from the Federal Ministry of Education and Research (BMBF) within the project DAIKIRI under the grant no 01IS19085B.

## History

| Version | Date | Reason | Revised by |
|---|---|---|---|
| 1.0 | 28/03/2021 | Final version created | Simon Bin |

## Author List

| Organisation | Name | Contact Information |
|---|---|---|
| UPB | Simon Bin | simon.bin@uni-paderborn.de |
| UPB | Stefan Heindorf | heindorf@uni-paderborn.de |

## Executive Summary

We developed the library Onto*learn* for structured machine learning. In the following, we describe its architecture as well as its formal and empirical properties. The Onto*learn* library will be used to apply structured machine learning on OWL in the Python language. It is an excellent basis for explainable artificial intelligence and paves the way for combination with other machine learning libraries available. After introducing the deliverable, we will give some background information on the topic as well as a motivating example arising from the Use Cases in our project. Afterwards, the installation, build procedure and usage of the library will be described in depth. We will give some hands-on examples on working with Ontologies in our library, define the learning problem for the Structured Machine Learning algorithm and detail how to configure the learning algorithm. Furthermore, the design and architecture of the library will be presented, finished by several experiments demonstrating the current state and success of the library. Finally, we will have a short conclusion.

## Contents

## 1 Introduction

In this deliverable, we create a library for Structured Machine Learning. Structured Machine Learning is one of the building blocks in Explainable Artificial Intelligence. Our initial prototype focuses on the specification and proof of implementation as outlined in task description 4.1.

The results of a Structured Machine Learning approach will be representable in a structured logical formalism (Description Logics) and are thus inherently suited for manual inspection and verification by humans. In the context of this project, we aim to explore the suitability of the Structured Machine Learning approach on the Smart Logistics use case presented in work package 6. To that end, the use case data first needs to be transformed into graph data suitable for structured machine learning. This task is part of work package 3.

In the following, we expect data to be available as OWL Ontology Documents. The full API reference can be found on our website (references to the API are given in footnotes).

The source code will be available at `https://github.com/dice-group/ontopy`.

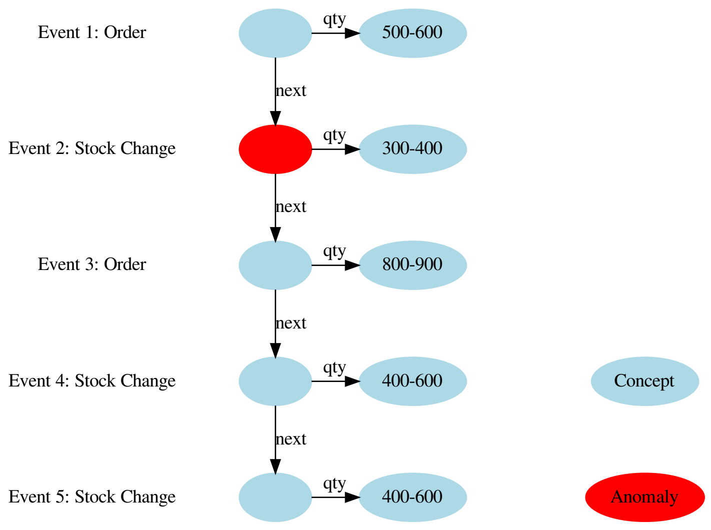
## 2 Background and Motivating Example

We are developing a library for Machine Learning in OWL. OWL is the Web Ontology Language[1], a W3C standard ontology language for the Semantic Web. Our intention is to combine the power of Description Logics, which OWL is based in, with supervised machine learning algorithms. The algorithms are based on the idea of Inductive Logic Programming by Muggleton [1999]. The resulting descriptions for our input data can be verified by humans more easily than pure black-box approaches. One such framework is the DL-Learner by Bühmann et al. [2018].

As a first step, a base library was created. For that, a number of supporting libraries had to be developed as well, since to date there is no OWL-API[2] library available for Python. Next, the library architecture could be defined (see section 7). For the initial version, the OWL/Description Logics language subset has been decided to be $\mathcal{ALC}$: the Attributive Concept Language with Complements as introduced by Schmidt-Schauß and Smolka (see Baader et al. [2003]). We hope to extend this to even more expressive languages during the project. The adaptors in the library are making use of Owlready2 by Lamy [2020], but there is a possibility for different adaptors to be added later.

To deal with the insufficient speed of reasoners for our application, a fast instance checker was developed as well. This is an approximation to reasoning by employing set semantics and structural reasoning when querying sub-class relations.

[1] `https://www.w3.org/TR/owl-overview/`
[2] `http://owlcs.github.io/owlapi/`



In above figure, we have drawn an example of structured data derived from a possible Use Case scenario from Use Case 2 (Work Package 6.2). There is a warehouse on display, and different events can change the amount of stock. For some reason, the event 2 was found to be anomalous as the stock quantity did not change in the expected amount. Here, the anomalies are marked in red and it would be desirable to find a description for it. One such a description, expressed in Description Logics notation, might be

$$Stock\_Change \sqcap \exists qty.300\text{-}400 \sqcap \exists next^{-}.\exists qty.500\text{-}600$$

Translated in a more human language, this formula might read: "(The event type was) Stock Change **and** the quantity was 300-400 **and** the predecessor (reverse of next) had a quantity of 500-600". For supervised machine learning, we require many such instances to classify. In the course of this project we want to evaluate if our machine learning approach can derive such descriptions.

In the next sections, we will describe how to install and use our library. At the end, we will evaluate the library on given data sets.

## 3 Installation

Since Onto*learn* is a Python library, you will need to have Python on your system. Python comes in various versions and with different, sometimes conflicting dependencies. Hence, most guides will recommend to set up a "virtual environment" to work in.

One such system for virtual python environments is Anaconda[1]. You can download miniconda from `https://docs.conda.io/en/latest/miniconda.html`.

We have good experience with it and make use of conda in the *Installation from source* step.

[1] `https://www.anaconda.com/`

## 3.1 Installation from source

To download the Onto*learn* source code, you will also need to have a copy of the Git[2] version control system.

Once you have `conda` and `git` installed, the following commands should be typed in your shell in order to download the Onto*learn* development sources, install the dependencies listened in the `environment.yml` into a conda environment and create the necessary installation links to get started with the library.

- Download (clone) the source code

```
git clone https://github.com/dice-group/OntoPy.git
cd OntoPy
```

- Load the dependencies into a new conda environment called "temp" (you can name it however you like)

```
conda create --name temp python=3.8
conda activate temp
conda env update --name temp
```

- Install the development links so that Python will find the library

```
python -c 'from setuptools import setup; setup()' develop
```

- Instead of the previous step there is also Possibility B, which is valid temporarily only in your current shell:

```
export PYTHONPATH=$PWD
```

Now you are ready to develop on Onto*learn* or use the library!

### 3.1.1 Verify installation

To test if the installation was successful, you can try this command: It will only try to load the main library file into Python:

```
python -c "import ontolearn"
```

[2] https://git-scm.com/

### 3.1.2 Tests

In order to run our test suite, pytest[3] has to be installed first if it is not yet installed. This can be done with conda (or with pip):

```
conda install pytest
```

Then you can run the tests using:

```
pytest
```

## 3.2 Installation via pip

Released versions of Onto*learn* can also be installed using `pip`, the Package Installer for Python. It comes as part of Python. Please research externally (or use above `conda create` command) on how to create virtual environments for Python programs.

```
pip install ontolearn
```

This will download and install the latest release version of Onto*learn* and all its dependencies from `https://pypi.org/project/ontolearn/`.

## 3.3 Building (sdist and bdist_wheel)

In order to create a *distribution* of the Onto*learn* source code, typically when creating a new release, it is necessary to use the `build` tool. It can be installed with:

```
pip install build
```

and invoked through:

```
python -m build
```

from the main source code folder. Packages created by `build` can then be uploaded as releases to the Python Package Index (PyPI)[4] using twine[5].

## 3.4 Contribution

Feel free to create a pull request.

[3] `https://pytest.org/`
[4] `https://pypi.org/`
[5] `https://pypi.org/project/twine/`

## 3.5 Questions

For any further questions, please contact: `caglar.demir@upb.de` or open an issue on our GitHub issues page[6].

# 4 Working with Ontologies

To get started with Structured Machine Learning, the first thing required is an Ontology[7] with Named Individuals[8]. We cannot provide such ontologies for you and depending on the use case, it may be necessary to first map existing data into an ontology.

However, some sample ontologies are included with Onto*learn* so that you can start using it right away. One such sample ontology is contained in the `KGs/father.owl` file. It contains six persons (individuals), of which four are male and two are female. We will use this tiny ontology as an example.

## 4.1 Loading an Ontology

To load an ontology, use the following Python code:

```python
from owlapy import IRI
from owlapy.owlready2 import OWLOntologyManager_Owlready2

mgr = OWLOntologyManager_Owlready2()
onto = mgr.load_ontology(IRI.create("file://KGs/father.owl"))
```

First, we import the IRI class and a suitable OWLOntologyManager. To load a file from our computer, we have to reference it with an IRI[9]. Secondly, we need an Ontology Manager, which is a component that can manage ontologies for us. Currently, Onto*learn* contains one such manager: The OWLOntologyManager_Owlready2.

Now, we can already inspect the contents of the ontology. For example, to list all individuals:

```python
for ind in onto.individuals_in_signature():
    print(ind)
```

Refer to the *OWLOntology*[a] documentation for more details.

[6] `https://github.com/dice-group/OntoPy/issues`
[7] `https://www.w3.org/TR/owl2-overview/`
[8] `https://www.w3.org/TR/owl-syntax/#Named_Individuals`
[9] `https://tools.ietf.org/html/rfc3987`

[a] `owlapy.model.OWLOntology`

## 4.2 Attaching a reasoner

In order to validate facts about statements in the ontology (and thus also for the Structured Machine Learning task), the help of a reasoner component is required.

In our Onto*learn* library, we provide several reasoners to choose from. Currently, there are the the *fast instance checker*[b], the *structural Owlready2 reasoner*[c], and the *class instantiation Owlready2 reasoner*[d] available to choose from.

To load any reasoner, follow this Python code:

```python
from owlapy.owlready2 import OWLReasoner_Owlready2
from owlapy.owlready2.temp_classes import OWLReasoner_Owlready2_TempClasses
from owlapy.fast_instance_checker import OWLReasoner_FastInstanceChecker

structural_reasoner = OWLReasoner_Owlready2(onto)
temp_classes_reasoner = OWLReasoner_Owlready2_TempClasses(onto)
fast_instance_checker = OWLReasoner_FastInstanceChecker(onto, temp_classes_reasoner)
```

The reasoner takes as its first argument the ontology to load. The fast instance checker requires a base reasoner to which any reasoning tasks not covered by the fast instance checking code are deferred to.

# 5 Defining a Learning Problem

The Structured Machine Learning implemented in our Onto*learn* library is working with a type of supervised learning[10]. One of the first things to do after loading the Ontology is thus to define the positive and negative examples whose description the learning algorithm should attempt to find.

## 5.1 Referencing Named Individuals

Let's assume we are working with the Ontology `father.owl` that was loaded in the previous chapter. Our positive examples (individuals to describe) are stefan, markus, and martin. And our negative examples (individuals to not describe) are heinz, anna, and michelle. Then we could write the following Python code:

```python
from owlapy import IRI
from owlapy.namespaces import Namespaces
from owlapy.model import OWLNamedIndividual

NS = Namespaces('ex', 'http://example.com/father#')

positive_examples = {OWLNamedIndividual(IRI.create(NS, 'stefan')),
```

(continues on next page)

[10] https://en.wikipedia.org/wiki/Supervised_learning

[b] owlapy.fast_instance_checker.OWLReasoner_FastInstanceChecker
[c] owlapy.owlready2.base.OWLReasoner_Owlready2
[d] owlapy.owlready2.temp_classes.OWLReasoner_Owlready2_TempClasses

(continued from previous page)

```
                             OWLNamedIndividual(IRI.create(NS, 'markus')),
                             OWLNamedIndividual(IRI.create(NS, 'martin'))}
negative_examples = {OWLNamedIndividual(IRI.create(NS, 'heinz')),
                             OWLNamedIndividual(IRI.create(NS, 'anna')),
                             OWLNamedIndividual(IRI.create(NS, 'michelle'))}
```

Note that the namespace has to match the Namespace/IRI that is defined in the Ontology document.

### 5.2 Knowledge Base over the Ontology

The knowledge base is the Onto*learn* representation of an Ontology and a Reasoner. It also contains a Hierarchy generator as well as other Ontology-related state required for the Structured Machine Learning library. It is required to run a learning algorithm. Creation is done like follows:

```
from ontolearn import KnowledgeBase

kb = KnowledgeBase(ontology=onto, reasoner=fast_instance_checker)
```

Further actions are possible on the knowledge base, for example the ignorance of specific classes. Refer to the documentation for the full details.

### 5.3 Creating the Learning Problem

Now the learning problem can be captured in its respective object, the *positive-negative standard learning problem*[e]:

```
from ontolearn.learning_problem import PosNegLPStandard

lp = PosNegLPStandard(kb, pos=positive_examples, neg=negative_examples)
```

# 6 Learning Algorithm

With a *learning problem* defined, it is now possible to configure the learning algorithm.

Currently, two *Base Concept Learners*[f] are provided in our Onto*learn* library. The *length base learner*[g] and the *modified CELOE algorithm*[h]. Each algorithm may have different available configuration. However at minimum they require a *knowledge base* and a learning problem.

[e] `ontolearn.learning_problem.PosNegLPStandard`
[f] `ontolearn.base_concept_learner.BaseConceptLearner`
[g] `ontolearn.concept_learner.LengthBaseLearner`
[h] `ontolearn.concept_learner.CELOE`

## 6.1 Configuring CELOE

Let us now configure the modified CELOE algorithm:

```
from ontolearn.concept_learner import CELOE

alg = CELOE(kb, learning_problem=lp)
```

There are further configuration choices of CELOE, such as the *Refinement Operator*[i] to use in the search process. The quality function (*Predictive Accuracy*[j], *F1 Score*[k], *Precision*[l], or *Recall*[m]) to evaluate the quality of the found expressions can be configured. There is the heuristic function to evaluate the quality of expressions during the search process. And some options limit the run-time, such as `max_runtime` (maximum run-time in seconds) or `max_num_of_concepts_tested` (maximum number of concepts that will be tested before giving up) or `iter_bound` (maximum number of refinement attempts).

### 6.1.1 Changing the quality function

To use another quality function, first create an instance of the function:

```
from ontolearn.metrics import Accuracy

pred_acc = Accuracy(learning_problem=lp)
```

### 6.1.2 Configuring the heuristic

```
from ontolearn.heuristics import CELOEHeuristic

heur = CELOEHeuristic(
    expansionPenaltyFactor=0.05,
    startNodeBonus=1.0,
    nodeRefinementPenalty=0.01)
```

Then, configure everything on the algorithm:

```
alg = CELOE(kb, learning_problem=lp, quality_func=pred_acc, heuristic_func=heur)
```

[i] `ontolearn.abstracts.BaseRefinement`
[j] `ontolearn.metrics.Accuracy`
[k] `ontolearn.metrics.F1`
[l] `ontolearn.metrics.Precision`
[m] `ontolearn.metrics.Recall`

## 6.2 Running the algorithm

To run the algorithm, you have to call the *fit*[n] method. Afterwards, you can fetch the result using the *best_hypotheses*[o] method:

```
from owlapy.render import DLSyntaxObjectRenderer

dlsr = DLSyntaxRenderer()

alg.fit()

for desc in alg.best_hypotheses(1):
    print('The result:', dlsr.render(desc.concept), 'has quality', desc.quality)
```

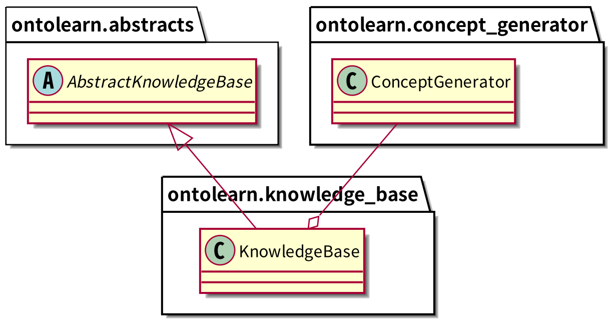
In this example code, we have created a DL-Syntax renderer in order to display the OWL Class Expression[11] in a Description Logics style. This is purely for aesthetic purposes.

# 7 Architecture

Here we present the class hierarchies of Onto*learn*.

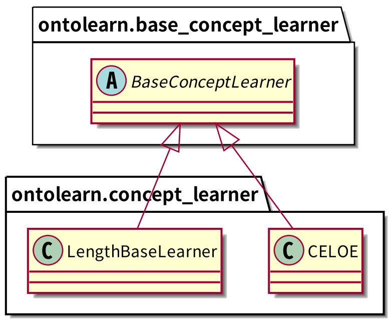
## 7.1 Knowledge Base organisation

The Knowledge Base keeps the OWL Ontology and Reasoner together and manages the OWL Hierarchies. It is also responsible for delegating and caching instance queries and encoding instances.



## 7.2 Learning Algorithms

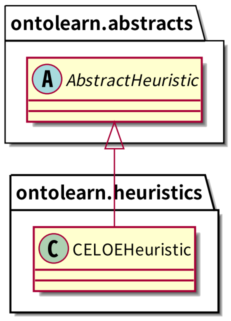
There may be several Concept Learning Algorithms to choose from. Each may be better suited for a specific kind of Learning Problem or desired result.



[11] https://www.w3.org/TR/owl-quick-reference/#Class_Expressions

[n] ontolearn.base_concept_learner.BaseConceptLearner.fit
[o] ontolearn.base_concept_learner.BaseConceptLearner.best_hypotheses

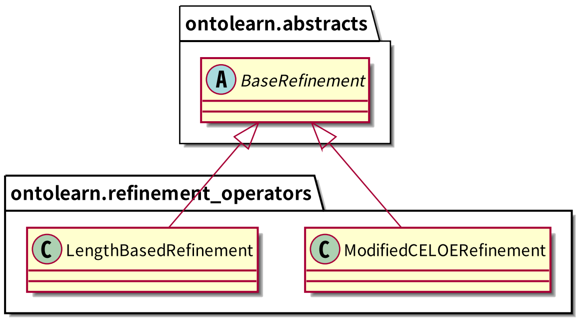
## 7.3 Heuristics



Heuristics are an abstraction to guide the concept search process. The included CELOE Heuristic takes as basis the *Quality function* and then adds some weights, like penalties for extending the length of a concept or for repeatedly searching the same position in the search space. Other heuristics can be invented and tested.

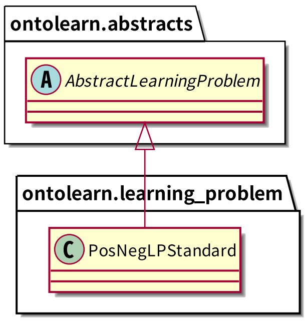
## 7.4 Refinement Operators

Refinement operators suggest new concepts to evaluate based on a given concept. Ultimately, the refinement operator decides what Description Logics language family the learning algorithm will be able to cover. The included modified CELOE refinement will iteratively generate $\mathcal{ALC}$ concepts by generating all available atomic classes and properties, and combine them with $\sqcap$, $\sqcup$, $\neg$, $\forall$ and $\exists$, and successively increase their length. The length based refinement will intuitively do the same but immediately go to a specific length instead of searching the concepts by increasing length.



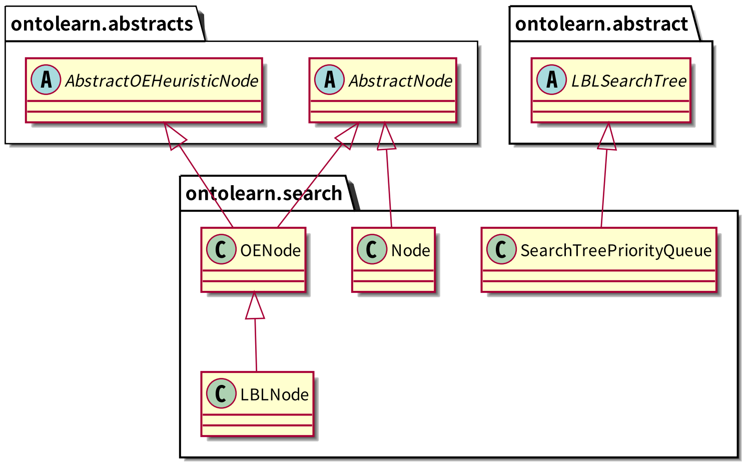
## 7.5 Learning Problem types

Learning Problems encode all information pertaining to the classification task, i.e. the positive examples (that should be covered by the learning result) and the negative examples (that should not be covered by the learning result).



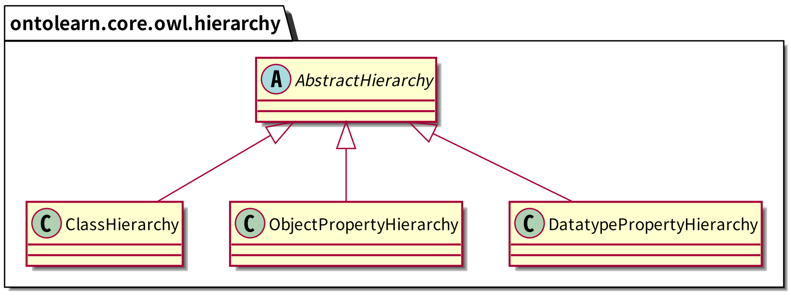
## 7.6 Search trees and nodes

Nodes are tuples of concept and (typically) quality and other related measures or information, and are used in the search process or possibly to present the algorithm results.



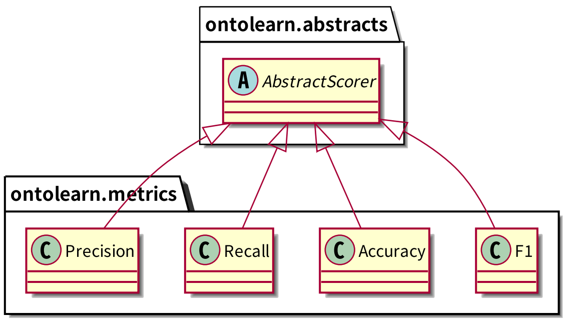
## 7.7 OWL Hierarchies

Hierarchies order their taxonomies into a sub- and super-kind relation, for example sub-classes and super-classes.


ontolearn.core.owl.hierarchy
A AbstractHierarchy
C ClassHierarchy
C ObjectPropertyHierarchy
C DatatypePropertyHierarchy

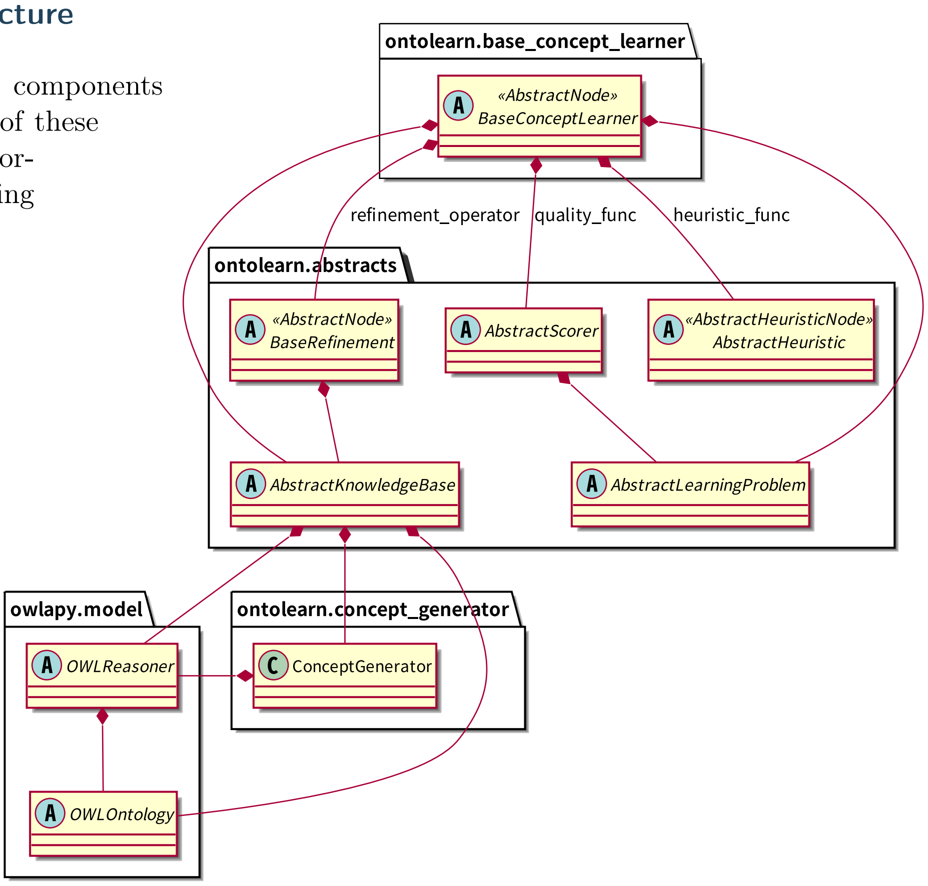
## 7.8 Quality Functions

You can choose one quality function which will be the quality of a Node in the search process. The way the quality is measured typically influences the heuristic and thus the direction of the search process.



## 7.9 Component architecture

Here you can see how the components depend on each other. All of these components are required in order to run the concept learning algorithm.


ontolearn.base_concept_learner
A «AbstractNode» BaseConceptLearner
refinement_operator
quality_func
heuristic_func
ontolearn.abstracts
A «AbstractNode» BaseRefinement
A AbstractScorer
A «AbstractHeuristicNode» AbstractHeuristic
A AbstractKnowledgeBase
A AbstractLearningProblem
owlapy.model
A OWLReasoner
A OWLOntology
ontolearn.concept_generator
C ConceptGenerator

# 8 Model Adaptor

To simplify the connection between all the *Components*, there is a model adaptor available that automatically constructs and connects them. Here is how to implement the previous example using the Model Adaptor:

```
from ontolearn.concept_learner import CELOE
from ontolearn.heuristics import CELOEHeuristic
from ontolearn.learning_problem import PosNegLPStandard
from ontolearn.metrics import Accuracy
from ontolearn.model_adapter import ModelAdapter
from owlapy import IRI
from owlapy.fast_instance_checker import OWLReasoner_FastInstanceChecker
from owlapy.model import OWLOntology, OWLNamedIndividual
from owlapy.namespaces import Namespaces
from owlapy.owlready2.base import OWLOntology_Owlready2, OWLOntologyManager_Owlready2
from owlapy.owlready2.temp_classes import OWLReasoner_Owlready2_TempClasses
from owlapy.render import DLSyntaxObjectRenderer


def my_reasoner_factory(onto: OWLOntology):
    assert isinstance(onto, OWLOntology_Owlready2)
    temp_classes_reasoner = OWLReasoner_Owlready2_TempClasses(onto)
    fast_instance_checker = OWLReasoner_FastInstanceChecker(
        onto,
        temp_classes_reasoner)
    return fast_instance_checker


NS = Namespaces('ex', 'http://example.com/father#')

positive_examples = {OWLNamedIndividual(IRI.create(NS, 'stefan')),
                     OWLNamedIndividual(IRI.create(NS, 'markus')),
                     OWLNamedIndividual(IRI.create(NS, 'martin'))}
negative_examples = {OWLNamedIndividual(IRI.create(NS, 'heinz')),
                     OWLNamedIndividual(IRI.create(NS, 'anna')),
                     OWLNamedIndividual(IRI.create(NS, 'michelle'))}

# Only the class of the learning algorithm is specified
model = ModelAdapter(learner_type=CELOE,
                     ontologymanager_factory=OWLOntologyManager_Owlready2,  # (*)
                     reasoner_factory=my_reasoner_factory,  # (*)
                     )

# no need to construct the IRI here ourselves
model.fit(path="../KGs/father.owl",
          learning_problem_type=PosNegLPStandard,  # (*)
          quality_type=Accuracy,
```

(continues on next page)

(continued from previous page)

```
          heuristic_type=CELOEHeuristic,
          expansionPenaltyFactor=0.05,
          startNodeBonus=1.0,
          nodeRefinementPenalty=0.01,
          pos=positive_examples,
          neg=negative_examples,
          )

dlsr = DLSyntaxRenderer()

for desc in model.best_hypotheses(1):
    print('The result:', dlsr.render(desc.concept), 'has quality', desc.quality)
```

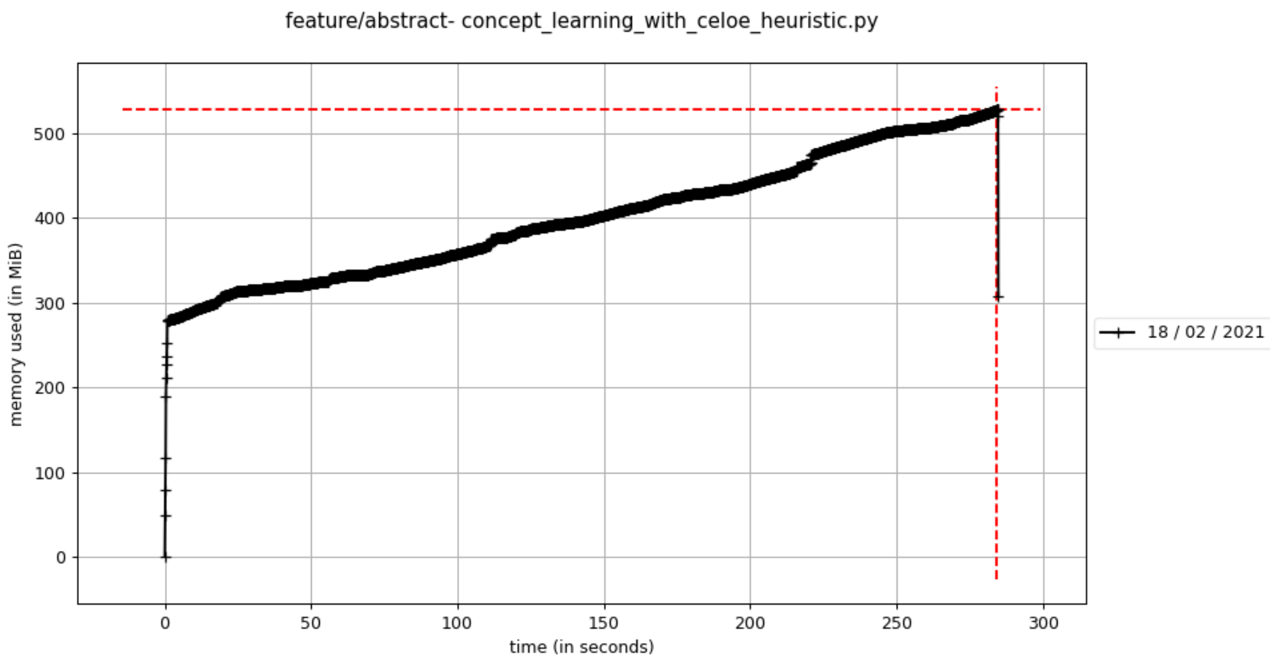
Lines marked with (*) are not strictly required as they happen to be the default choices.

## 9 Further reading

Further references and material can be found by inspecting the existing tests and the code in the examples folder.

## 10 Experiments

### 10.1 Scalability



To assess the scalability of Onto*learn*, we measured the memory consumption while running the learning algorithm, using the Python memory-profiler[3]. We tested the CELOE algorithm as implemented on Onto*learn* on the family benchmark data set (a copy is included in the library source). We can observe an approximately linear increase through the run-time of just below five minutes before the algorithm found a solution.

## 10.2 Predictive Performance on Benchmarking Datasets

We tested the operation of our library on benchmark data sets published in SML-Bench by Westphal et al. [2019]. SML-Bench is the largest central collection of data sets for structured machine learning on OWL. The settings used were 10 minutes run-time, default parameters, maximum 1 million expression tests. No cross-validation was done. The source code for this experiment is available in our repository[4]. The results of our library are reproduced below:

| Learning Problem | Accuracy | F-score | Learnt Description |
|---|---|---|---|
| animals/bird | 1.0 | 1.0 | $Homeothermic \sqcap (\neg HasMilk)$ |
| animals/fish | 1.0 | 1.0 | $HasGills$ |
| animals/mammal | 1.0 | 1.0 | $HasMilk$ |
| animals/reptile | 1.0 | 1.0 | $(\neg HasGills) \sqcap (\neg Homeothermic)$ |
| carcinogenesis/1 | 0.55705 | 0.71053 | $\exists hasAtom.Carbon$ |
| hepatitis/1 | 0.414 | 0.5844 | $Male \sqcup (\exists hasScreening.\top)$ |
| lymphography/1 | 0.84459 | 0.87006 | $CIN14\_Lac\_Margin \sqcup NON19\_n0\text{-}9$ |
| mammographic/1 | 0.46306 | 0.633 | $\top$ |
| mutagenesis/42 | 0.78571 | 0.66667 | $\exists hasAtom.Carbon\text{-}21$ |
| nctrer/1 | 0.60268 | 0.74644 | $\exists has\_bond.(\neg NonStereoBond)$ |
| pyrimidine/1 | 0.5 | 0.66667 | $\top$ |
| suramin/1 | 0.41176 | 0.58333 | $\top$ |

## 10.3 AI4BD Use Case: Smart Logistics

We have built a pipeline for the Smart Logistics Use Case data, as outlined in section 2. With the current state of the input data transformation and the modified CELOE algorithm as implemented in our library, the result is:

$$\exists delivery\_qty.(\neg delivery\_qty\_nan) \qquad \text{(F-score : 0.57143)}$$

That is rather unimpressive and alludes that there needs to be a lot more investigation into the use case, data modelling and transformation. In the future, extending the language support of our library from $\mathcal{ALC}$ to something more expressive may also help.

[3] `https://pypi.org/project/memory-profiler/`
[4] `https://github.com/dice-group/OntoPy/blob/D4.1/examples/sml_bench.py`

## 11 Conclusion

In this deliverable, we have presented our work on the structured machine learning library. We can report that the progress has been exceptional. We have implemented a structured machine learning library from scratch. We also had to implement the required parts of OWL-API, which is a great contribution as well. As a first step, we have implemented the $\mathcal{ALC}$ description logic in our refinement operator and fast instance checker. The spike solution shows that the code can be run on Use Case data as well. Nevertheless there remains work to be done, as we want to improve these results beyond the current state. One of these improvements may be the continued development of our library to a more powerful language such as $\mathcal{ALCOIN}(\mathcal{D})$. Project-wise, the next work package will deal with verbalising found descriptions to make them easier to read for humans.

## References

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, USA, 2003. ISBN 0521781760.

Lorenz Bühmann, Jens Lehmann, Patrick Westphal, and Simon Bin. DL-Learner – Structured Machine Learning on Semantic Web Data. In *The Web Conf (WWW) 2018 Journals Track*, 2018.

Jean-Baptiste Lamy. *Ontologies with Python: Programming OWL 2.0 Ontologies with Python and Owlready2*. Apress, 2020. ISBN 9781484265512.

Stephen Muggleton. Inductive logic programming. In *The MIT Encyclopedia of the Cognitive Sciences*, 1999.

Patrick Westphal, Lorenz Bühmann, Simon Bin, Hajira Jabeen, and Jens Lehmann. SML-Bench – A benchmarking framework for structured machine learning. *Semantic Web*, 10(2):231–245, 2019.