IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Télécom Évolution

Formation continue : Traitement automatique du langage naturel

Lab 3: An Ontology-Based Question-Answer System

Yannis Haralambous (IMT Atlantique) Derya Can (La mètis)

In this lab we will use OWL ontologies (under Python package Owlready2) and SPARQL queries (under Python package rdflib) to create a simple, but intelligent, question-answer system. We will also discover the Protégé software to better understand ontologies and practice populating them.

1 Description Logics Formalism

A small ontology (created a long time ago for educational purposes, cf. [1, 2, 3]) describes some animals and plants. Write the corresponding formulas, in description logic.

Animals and plants form classes.

Trees are a type of plant.

Branchs are parts of trees.

Carnivores are exactly those animals that also eat animals.

Leaves are parts of branches.

Giraffes are herbivores and eat only leaves.

Herbivores are animals that eat only plants or parts of plants.

Lions are animals that eat only herbivorous animals.

Plants are a disjoint class from animals.

Edible plants are plants that are eaten by both herbivores and carnivores.

Leo is a lion. Gigi is a giraffe. Giginou is a giraffe, different from Gigi.

We are in an "open world," so everything that is not explicitly stated can also be true. For example, there is nothing to prevent Leo from being a giraffe at the same time. How can we avoid this, by indicating that lions and giraffes are disjoint sets?

Can we show that lions eat giraffes?

2 Use of Protégé

Download Protégé from `https://protege.stanford.edu/products.php#desktop-protege`

Open the ontology `wildlife-en.xml`, check that it verifies the above description. If not, complete it.

Add the fact that Leo eats Gigi (Entities > Individuals, select Leo, then in Property Assertions click on the cross).

Activate the reasoner (menu Reasoner, choose “Start Reasoner”) and at each modification do control-S and also control-R to synchronize the reasoner (command-S and command-R on Macs).

Go to the tab "DL Query" and type "(eaten-by some lion) and giraffe" by clicking on "Subclasses" and "Instances", explain the result.

Add the fact that Gigi eats Giginou. What is happening and why?

Complete the information of the three object properties: is-part-of, eats, eaten-by. Are they functional, inverse functional, transitive, symmetric, asymmetric, reflexive, irreflexive?

Add individuals Arbrounet, Brabranche and Feufeuille for the classes Tree, Branch and Leaf. Enter the relations is-part-of(Leaf,Branch), is-part-of(Branch,Tree).

Can the is-part-of relation be transitive? Check in Protégé and explain.

Correct the ontology so that is-part-of becomes transitive. Check under DL Query.

Apart from the relations between individuals, we can also have attributes attached to them. We will define two attributes: weight and size. Select the individual (for example Gigi) in Entities > Individuals and click on the cross next to Data property assertions. In the window that appears select owl:topDataProperty and click on the first button. Call the new property "weight," choose as type “xsd:float” and as value[1]:

Individual	Weight (kg)	Size (m)
Gigi	800	5.80
Giginou	735	5
Léo	190	1.25

3 Using Owlready2

Install:

```
python3 -m pip install cython
python3 -m pip install owlready2
python3 -m pip install rdflib
python3 -m pip install stanza
```

3.1 Ontology Exploration

Here’s a little program that shows how you can access every corner of the ontology using Owlready2:

```
from owlready2 import *
onto = get_ontology("wildlife.owl").load()
print("Voici les classes de l'ontologie:")
print(list(onto.classes()))
print("###\nVoici les instances de chaque classe:")
for cl in onto.classes():
    print(cl.name,cl.instances())
print("###\nVoici les rôles:")
for pr in onto.object_properties():
    print(pr.name,pr.domain,pr.range)
print("###\nVoici les attributs:")
for da in onto.data_properties():
    print(da.name,da.domain,da.range)
print("###\nVoici les individus et leurs propriétés:")
for ind in onto.individuals():
    print(ind)
    for prop in ind.get_properties():
        for value in prop[ind]:
            print(".%s == %s" % (prop.python_name, value))
```

The code is available (`explore-ontology.py`). You will find more information on
`https://owlready2.readthedocs.io/en/latest/onto.html#accessing-the-content-of-an-ontology`

[1] Note that Giginou is an anorexic giraffe, she only weighs 735 kg.

3.2 Search in an Ontology

The method `search()` of the ontology object allows to perform searches using:

1. `iri`: the unique label of each entity (class, individual, role);
2. `type`: the individuals of a class;
3. `sublass_of`: the subclasses of a class;
4. the property names.

It is possible to add a wildcard * to write only a part of the string.

IRIs (Internationalized Resource Identifiers) are unique identifiers of given entities, and according to the principles of the W3C they consist in a protocol, a document identifier and an internal link (of the type `http://machine.domaine.ext/chemin/fichier#id`). In our case it will be `file:wildlife.owl#qqch`, but we can use the wildcard * to avoid this part.

Example (available, `search-ontology.py`):

```
from owlready2 import *
onto = get_ontology("wildlife-en.xml").load()
sync_reasoner([onto])
print("Here are classes and their IRIs:")
for cl in onto.classes():
    print(cl.name,cl.iri)
print("###\nFind class animal")
r = onto.search(iri="*#animal")
print(r)
print("###\nFind classes ending with plant")
r = onto.search(iri="*#*plant")
print(r)
print("###\nFind sublclasses of the class animal")
r = onto.search(subclass_of = onto.search(iri="*#animal")[0])
print(r)
print("###\nFind the individuals of the second animal subclass")
r = onto.search(type = onto.search(subclass_of = onto.search(iri="*#animal")[0])[1])
print(r)
print("###\nFind all animals")
r = onto.search(type = onto.search(iri="*#animal")[0])
print(r)
print("###\nFind all individuals weighting 800")
r = onto.search(weight = 800)
print(r)
print("###\nFind who eats Gigi")
r = onto.search(eats = onto.search(iri="*Gigi")[0])
print(r)
print("###\nFind who makes what with whom")
for ind in onto.individuals():
    for rol in onto.object_properties():
        if rol[ind]:
            for suc in rol[ind]:
                print(ind.iri,rol.name,suc.iri)
```

Note that the question "Find me all the animals" gave the result "wildlife.Léo, wildlife.Gigi, wildlife.Giginou", and yet in the ontology we only specified that the first one is a lion and the other two are giraffes. It is search() that made the necessary inferences.

Nevertheless this search does not allow us to go very far. What we need is a real query language, a kind of SQL adapted to the situation. Fortunately, this exists! This is the subject of the next section: the SPARQL language.

4 SPARQL

SPARQL is a knowledge base query language (in the same way that SQL is a database query language). It has been defined by the Web consortium and is currently at version 1.1 (`https://www.w3.org/TR/2013/REC-sparql11-query-20130321/`). We will only scratch the surface of the top layer of the iceberg here. SPARQL has a number of powerful features, which allow you to write very sophisticated queries.

A typical query is of the form:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?b rdf:type me:lion .
}
```

and if it is necessary to filter the results obtained, for example to obtain the giraffes that weigh more than 750 kg, one will write

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?b rdf:type me:giraffe .
?b me:weight ?weight .
FILTER (?weight >= 750 )
}
```

As these are Semantic Web technologies, all entities are IRIs, and so we need namespaces for whatever is not variable name. That's why before the `SELECT`, we have a certain number of `PREFIX`, which define prefixes to make the query more readable. The `SELECT` and the `WHERE` are like in SQL. The variables all start with a question mark `?`. In the above queries we are looking for individuals of the classes `me:lion` and `me:giraffe`.

4.1 Use SPARQL under Protégé

Warning: do not use the "SPARQL Query tab", as it does not use the reasoner. Instead, go to Window > Create New Tab... and then to Windows > Views > Query views > Snap SPARQL Query and assign this view to the tab you just created.

Once the tab is created, write a query in the top part and click Execute, the results will appear in the bottom part.

The proof that the reasoner works: type the following query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b
WHERE {?b rdf:type me:animal }
```

You will see that you get all the animals (Gigi, Leo, Giginou) despite the fact that they are not declared as instances of the animal class but rather as instances of the giraffe or lion classes.

4.2 Use SPARQL under Python

Here is the code to send a SPARQL query to the graph created out of our ontology:

```
from owlready2 import *
from rdflib import *
import rdflib.plugins.sparql as sparql
```

```
onto = get_ontology("wildlife.owl").load()
default_world.graph.dump()

graph = default_world.as_rdflib_graph()

query = sparql.prepareQuery("""
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?b rdf:type me:lion .
}
""")
qres = graph.query(query)
```

The result of `graph.query_owlready` is an iterator rdflib.plugins.sparql.processor.SPARQLResult, by browsing it we obtain rdflib.query.ResultRow which are tuples containing IRIs with type rdflib.term.URIRef. Write a function `into_owlready` that reads rdflib.term.URIRef and returns Owlready2 objects when they exist, and None otherwise.

Unfortunately, as much transitive closure works under Protégé, it doesn't work in rdflib and therefore you have to use some tricks to get the same results. One of the tricks is to use transitive relations: instead of asking for the subclasses of animal

```
?x rdfs:subClassOf me:animal
```

we ask for the subclasses, as well as the subclasses of the subclasses and so on, and thus for the closure of the subclass relation:

```
?x rdfs:subClassOf* me:animal
```

It suffices to add an asterisk * to take any number of relations between the left and the right part (for those who know neo4j there is an analogous property). Here are the results:

```
(rdflib.term.URIRef('file:wildlife.owl#animal'),)
(rdflib.term.URIRef('file:wildlife.owl#carnivore'),)
(rdflib.term.URIRef('file:wildlife.owl#lion'),)
(rdflib.term.URIRef('file:wildlife.owl#herbivore'),)
(rdflib.term.URIRef('file:wildlife.owl#giraffe'),)
```

Here are some examples (very close, if not identical, with those of the section 3.2, p. 5):

- Find the class animal

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?b rdf:type owl:Class .
FILTER ( regex(str(?b),"animal") )
}
```

- Find all classes that end with "plant"

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?b rdf:type owl:Class .
FILTER ( regex(str(?b),".*plant") )
}
```

- Find all animals

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?x rdfs:subClassOf* me:animal .
?b rdf:type ?x .
?b rdf:type owl:NamedIndividual .
}
```

Explanation: for any class ?x that is a subclass of me:animal, find the individuals named ?b of ?x.

- Find all individuals that weight more than 500 kg and have a size superior to 2 m:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?x rdfs:subClassOf* me:animal .
?b rdf:type ?x .
?b rdf:type owl:NamedIndividual .
?b me:weight ?weight; me:size ?sizz .
FILTER (?weight >= 500 && ?size >= 2 )
}
```

The semi-colon ; here means that we keep the same "subject" (?b) and combine it with two predicate+object pairs. There is also the possibility to put a comma when keeping both the subject and the predicate. The above code can therefore be written in a slightly shorter way:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?x rdfs:subClassOf* me:animal .
?b rdf:type ?x , owl:NamedIndividual; me:weight ?weight; me:size ?size .
FILTER (?weight >= 500 && ?size >= 2)
}
```

- Find me who eats Gigi

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX me: <file:wildlife.owl#>
SELECT ?b WHERE {
?b me:eats me:Gigi .
}
```

- Find me who does what with whom:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX me: <file:wildlife.owl#>
SELECT ?a ?b ?c WHERE {
?b a owl:ObjectProperty .
?a ?b ?c .
}
```

Attention, here we get three values, they must be passed separately by `owl_ready`.

5 Problem: A Question-Answer System

We are finally in a position to tackle the problem of the question-answer system.

Here is a problem that is reasonably feasible without being too simplistic:

> We want to be able to answer the following questions about our ontology:
>
> 1. What are the (put here a class name, in the plural)?
>
>   Expected answers: «There is no (class name).», «There is a single (class name). It is (name of individual).» or «There are (quantity) (class name in plural). They are (list of individuals).»
> 2. Is (name of individual) a (class name)?
>
>   Expecte answers: «Yes.» or «No.»
> 3. What do we know about (name of individual)?
>
>   Expected answers: «(name of individual) is a (name of class). It weights (value) kg and its size is (valeur) m. (As well as other pieces of information such that "It eats XXX")»

More generally we will identify sentences of the type

wh-pronoun (what) - verb to be - definite article - class name
verb to be (is) - name of individual - indefinite article - class name
wh-pronoun (what) - (do we know) - preposition (about) - name of individual

Tip: We will use a new package, called stanza, which is infinitely better than spaCy (although a little slower at startup).

Under stanza we will write the corresponding tests based on syntax dependencies, and from the class name or the individual's name we will write a SPARQL query that we will transmit to the knowledge base. Once the result is returned to us, we will included it in one (or more) sentence(s) corresponding to the desired answer.

Also foresee the case where the individual or the class are unknown (answer for a class: "I don't know anything about (class name in plural)", for an individual: "I don't know anything about (name of individual)").

With the following code

```
import stanza
stanza.download('en')
nlp = stanza.Pipeline('en')
for text in ["What are the giraffes?", "Is Leo a giraffe?",\
    "What do we know about Leo?"]:
    doc=nlp(text)
    for sent in doc.sentences:
        for token in sent.words:
            print(token.id, token.text, token.upos, token.feats,\
            token.lemma, token.head, token.deprel)
```

stanza gives the following:

```
1 What PRON PronType=Int what 0 root
2 are AUX Mood=Ind|Tense=Pres|VerbForm=Fin be 1 cop
3 the DET Definite=Def|PronType=Art the 4 det
4 giraffes NOUN Number=Plur giraffe 1 nsubj
5 ? PUNCT None ? 1 punct

1 Is AUX Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin be 4 cop
2 Leo PROPN Number=Sing Leo 4 nsubj
3 a DET Definite=Ind|PronType=Art a 4 det
4 giraffe NOUN Number=Sing giraffe 0 root
5 ? PUNCT None ? 4 punct

1 What PRON PronType=Int what 4 obj
2 do AUX Mood=Ind|Tense=Pres|VerbForm=Fin do 4 aux
3 we PRON Case=Nom|Number=Plur|Person=1|PronType=Prs we 4 nsubj
4 know VERB VerbForm=Inf know 0 root
5 about ADP None about 6 case
6 Leo PROPN Number=Sing Leo 4 obl
7 ? PUNCT None ? 4 punct
```

6 To go further...

One aspect of Owlready2 that we have not explored at all is ontology creation/feeding. An intelligent chatbot should be able to enrich its knowledge by adding new concepts, relations and individuals to the ontology. The book [4] is a very good introduction to ontology management in Python.

References

[1] Grigoris Antoniou and Frank van Harmelen, *Web Ontology Language: OWL*, https://www.cs.vu.nl/~frankh/postscript/OntoHandbook03OWL.pdf.

[2] C. Maria Keet, *The African Wildlife Ontology tutorial ontologies: requirements, design, and content*, preprint https://arxiv.org/abs/1905.09519, 2019.

[3] C. Maria Keet, *An Introduction to Ontology Engineering*, https://people.cs.uct.ac.za/~mkeet/files/OEbook.pdf, 2020.

[4] Jean-Baptiste Lamy, *Python et les ontologies*, Éditions ENI, 2019.