Projet Pensées Profondes

Midterm report

Fundamental Computer Science Master's Degree 1
September 2014 – December 2014

Adviser: Eddy CARON

Marc CHEVALIER
Raphaël CHARRONDIÈRE
Quentin CORMIER
Tom CORNEBIZE

Yassine HAMOUDI
Valentin LORENTZ
Thomas PELLISSIER TANON

Contents

Introduction

The *Projet Pensées Profondes* (Deep Thought Project) aims at providing a powerful software for answering questions written in natural language. To accomplish this, we developped an eponymous set of tools that accomplish different tasks and fit together thanks to a protocol we developped.

These various tasks include data querying (using the young and open knownledge base Wikidata), question parsing (using the CoreNLP software written by Stanford University and machine learning), requests routing, web user interface, and feedback reporting.

Given the young age of this projet, these pieces are only starting to emerge with their first features and mutual communications, so we will describe them separately in this document without much of a general overview of the project.

Chapter 1

Datamodel and communication

We describe the choices we did about representation of the data and communication between modules.

1.1 Data model

First, we present the data model. All normalized structures of the PPP are JSON-serializable, i.e. they are trees made of instances of the following types:

- `Object`
- `List`
- `String`
- `Number`
- `Boolean`
- `Null`

We chose to represent all normalized data as trees. To represent sentences, we have 4 kinds of nodes.

- `sentence`: a question in natural language like "Who is George Washington?".
- `resource`: a leaf containing any kind of data (string, integer…).
- `missing`: a leaf which marks missing values.
- `triple`: a 3-ary node:
  - `subject`: what the triple refers to
  - `predicate`: denotes the relationship between the subject and the object
  - `object`: what property of the subject the triple refers to

For example, the work of the NLP is to transform

```
{
    "type": "sentence",
    "value": "Who is George Washington?"
}
```

into

```
{
    "type":
        "triple",
    "subject":{
        "type": "resource",
        "value": "George Washington"
    },
    "predicate":{
        "type": "resource",
        "value": "identity"
    },
    "object":{
        "type": "missing"
    }
}
```

This structure has been chosen for its good adaptability. For instance, we can add other kind of nodes such as intersection, union, node for yes/no questions (triples without missing son), boolean operations etc..

1.2 Communication

Modules communicate with the core via HTTP requests.

The core sends them a JSON object, and they return another one.

The basic idea is that the core iterates requests to modules, which return a simplified tree, until the former gets a complete response.

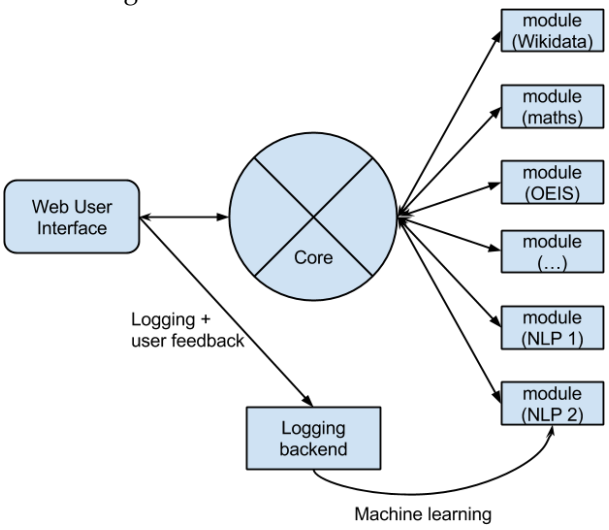
During these exchanges, we keep a trace of the different steps between the original request and the current tree. The structure of a trace is a list of such trace items:

```
{
    "module":
        "<name of the module>",
    "tree":{
        <answer tree>
    },
    "measures":{
        "relevance": <relevance of the answer>,
        "accuracy": <fiability of the answer>
    }
}
```

The measure field contains two values: relevance and accuracy.

- `accuracy` is a self-rating of how much the module may have correctly understood (ie. not misinterpreted) the request/question. It's float from 0 to 1.
- `relevance` is a self-rating of how much the tree has been improved (ie. made its way from a question to an useful answer). A positive float (not necessarily greater that 1; another module might use it to provide a much better answer).

Figure 1.1: Architecture of the PPP



This form allow each module to access to the previous results, particularly to the user's request. The objects for request and response contains few more information such that the language used.

The data model have been implemented in a nice set of objects in both Python and PHP in order to help the write of modules.

Chapter 2

Core

2.1 Communications

As its name suggests, the core is the central point of the PPP. It is connected to all other components — user interfaces and modules — through the protocol defined above.

The core communicates with the user interfaces and the modules *via* HTTP: each time the core receives a requests from an interface, it forwards it to modules, using a configurable list of URL where to reach modules.

An example configuration is the one we use on the production server:

```
{
    "debug": false,
    "modules": [
        {
            "name": "nlp_classical",
            "url": "http://localhost:9000/nlp_classical/",
            "coefficient": 1
        },
        {
            "name": "flower",
            "url": "http://localhost:9000/flower/",
            "coefficient": 1
        },
        {
            "name": "wikidata",
            "url": "http://wikidata.ppp.pony.ovh/",
            "coefficient": 1
        }
    ]
}
```

The current state is the the Core is successfully able with all modules that have been written: the Wikidata module, an example Python module (which answers the question “Who are you?”), and the Natural Language Processing module.

2.2 Routing

Besides from communicating with other pieces of the PPP, the Core will also route requests in such a way the power of the modules can be combined to give something greater.

For instance, when a user will input a question like “Who is the first president of the United States?”, the Core will send the question to all modules and get a parsed tree from the Natural Language Processing module. Then, it will forward this answer to all other modules, which the Wikidata module will be able to answer.

This part is not implemented, but will be next step in the implementation of the Core.

Chapter 3

User interface

We decided to implement first only a web user interface. This interface is only composed of one web-page developed in HTML 5 with some pieces of JavaScript and CSS. We have taken care of having an interface that fit nice on both huge screens of desktop computers and small screens of phones.

TODO: screenshot

It is composed of only one huge text input with a button to submit the query and an other one to get a random question. The text area allows both the input of questions in English or directly of triple using an easy notation like `(Douglas Adam, birth date,?)` to find the birth date of Douglas Adam. A small parser written in JavaScript convert this easy to use notation into the standard format.

In order to build this interface we have relied on some famous libraries ike jQuery and Bootstrap.

3.1 Logging

We decided to log all requests made to the PPP to improve our algorithms, and particularily to feed the results to Natural Language Processing modules that use Machine Learning. We may also use it to improve the way the Core routes/sorts answers from the different modules, either manually or with some basic Machine Learning.

The main idea is to log user feedback in addition to the requests themselves: after showing the user the way we interpreted their question alongside the answer to their request, we provide them a way to give us feedback. What we call feedback is actually a thumb up / thumb down pair of buttons, and, if the latter is pressed, a way to correct the requests parsing result so it can be fed to the Machine Learning algorithms.

Since Machine Learning algorithms are not ready yet, we did not focus on this feature of the user interface and thus it is not implemented; so far we only started implemented a backend that stores data (gathered via the user interface) to a SQLite database.

Chapter 4

Question parsing

The goal of this module is to transform questions into trees of triples, as described in section 1.1, which can be handled by backend modules.

The difficulty of this task can be illustrated on the following example:

What is the birth date of the president of the United States?

A first possible tree is: (?,birth date, president of the United States). However, this tree is difficult to handle by databases-querying modules. Indeed, the "president of the United States" occurence in a database probably does not contain the birth date of the current president.

On the other hand, the following tree is much more easy to process : (?,birth date, (?,president of, United States)). In this case, the president of United States is identified (Barack Obama), the triple becomes (?,birth date, Barack Obama), and finally the answer can be found easily in "Barack Obama" occurence.

Our goal is to product simplified and well structured trees, without losing relevant informations of the original question. We are developing three different approaches to tackle this problem. The first tries to analyze the grammatical structure of questions, the two other ones are based on machine learning.

4.1 Grammatical approach

Trees of triples can be producted after analyzing the grammatical structure of sentences. First, we present the tool we use to extract grammatical dependencies. Then, we expose chronologically our algorithm to product triples from grammatical structure.

We will detail throughout this section our algorithm on the example:

What is the birth date of the president of the United States?

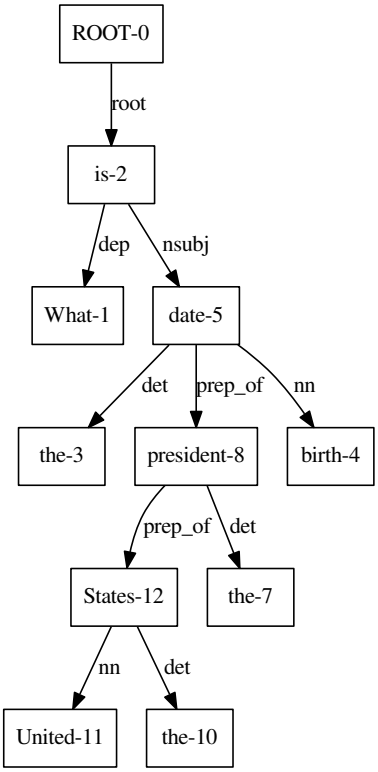
4.1.1 Stanford CoreNLP

The *Stanford CoreNLP* library [1] is a tool developed by the Stanford Natural Language Processing group, composed of linguists and computer scientists. This software is well-documented and considered as a "state of the art" tool. Moreover, it includes very efficient grammatical parsers.

Since this library is written in Java, and our module in Python, we use a Python wrapper[2] we first patched to support Python 3 and some features the wrapper did not implement.

[1] `http://nlp.stanford.edu/software/corenlp.shtml`
[2] `https://bitbucket.org/ProgVal/corenlp-python/overview`

Figure 4.1: Dependency tree



We use CoreNLP mostly to get grammatical dependency trees from input questions. It consists of trees which nodes are the words of the sentence, and edges reflect the grammatical relations between words.

Figure 4.1 provides an overview of such a tree on our question example *What is the birth date of the president of the United States?*. For instance, the edge:

$$\texttt{president} \xrightarrow{\texttt{det}} \texttt{the}$$

means that *the* is a determiner for *president*.

Some nodes of this tree are also endowed with tags. For example, *United* and *States* have the tag *location*.

The Stanford typed dependencies manual ([dMM08]) provides a full list and description of possible grammatical dependencies.

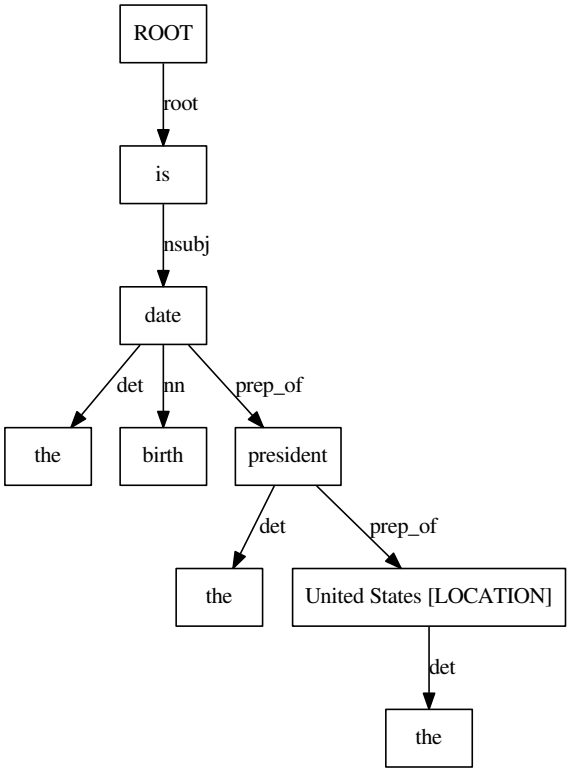
4.1.2 Preprocessing

The preprocessing consists in a sequence of operations executed on the tree outputs by the *Stanford CoreNLP* library. The aim is to simplify it, by merging the nodes which should belong together.

The current version of the module performs two sorts of merges:

- **Merge quotation nodes.** This operation merges all the nodes which are in a same quotation (delimited by quotation marks). It also adds the words of the quotation which were deleted by the *Stanford CoreNLP* library (e.g. *in*, *of*...). The final result is a node, containing the exact quotation, and placed at the appropriate position in the tree.

Figure 4.2: Dependency tree preprocessed



- **Merge named entities.** The *Stanford CoreNLP* library performs a *named entities recognition* (NER), which provides informative tags in some nodes. For instance, *United* and *States* are tagged *LOCATION* (see figure 4.2). In the preprocessing step, we merge all neighbour nodes with a same NER tag. In our example, we merge the two nodes *United States* into one single node.

The preprocessing also identifies the question word (Who, What, Where...) and removes it from the dependency tree.

Preprocessing is illustrated on figure 4.2. The question word is *What*.

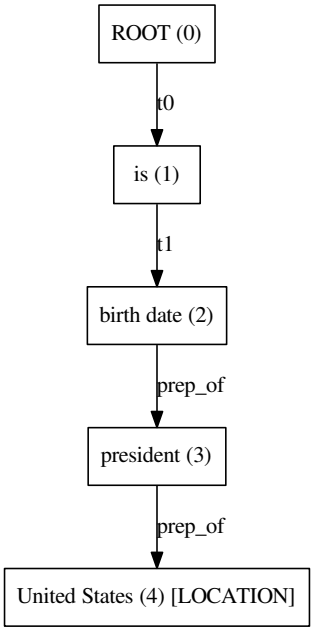
4.1.3 Grammatical dependencies analysis

The grammatical tree is simplified by applying one of the following rules to each edge:

- remove the edge and its endpoint node. For instance, a *dep* relation, such as *the* in our example, is often removed.
- merge the two nodes of the edge. Merge operations try to gather words of a same expression (e.g. phrasal verbs) that have not been merged during preprocessing.
- tag the edge with a "triple production rule".

The third operation is the most important. Dependencies relations are replaced by a restricted set of tags that will enable us to product a triples tree thereafter.

Figure 4.3: Dependency tree simplified



On our example, the edge:

$$\texttt{birth} \xrightarrow{\texttt{nn}} \texttt{date}$$

is merged into a single node : *birth date*.

One of the triple production rule tag is :

$$\texttt{is} \xrightarrow{\texttt{t1}} \texttt{birth date}$$

The simplified tree of our example is illustrated on figure 4.3.

4.1.4 Triple production

The triple production is the final step. It outputs the triples tree.

First, we assign a number to each remaining node. The root of the tree has always number *0*. We have directly print these numbers on figure 4.3.

Then, we associate to each subtree of root's number x an unknown denoted $?x$ that identifies the information the subtree refers to. On our example, the subtree of root *president* (number *3*) represents the name of the president of the United States. This unknown is denoted *?3*.

Unknowns are linked together into triples thanks to the triple production rules tagged previously. For instance, an edge tagged **t2**:

$$\texttt{a} \xrightarrow{\texttt{t2}} \texttt{b}$$

products the triples (?a,a,?b), or (?a,a,b) if b is a leaf (a and b are replaced by the words of the node they refer to).

The tag **t1** directly linked two unknowns ?a = ?b, instead of producting a triple.

The tag **t0** products nothing.

We obtain the following result on our example:

?1 = ?2
(?2 , birth date of , ?3)
(?3 , president of , United States)

Then, we link *?0* to *?1*, depending on the question word of the question. Here we have (question word *What*):

(?1,definition,?0)

The four previous rules are simplified in a set a triples:

(?1,definition,?0)
(?1 , birth date of , ?3)
(?3 , president of , United States)

Find an answer to the question is equivalent to build a model of the conjunctive formula: **(?1 , definition , ?0)∧(?1 , birth date of , ?3)∧(?3 , president of , United States)** and outputs the value of *?0*.

The triples tree is obtained by replacing each unknown *?x* by a triple containing *?x and not ?0*. The final result, taken from the PPP website, is printed on figure 4.4. Figure A.1 contains the formal representation of the triples tree of our example.

Figure 4.4: Triples tree

4.1.5 Future work

Grammatical rules analysis

Our analysis of grammatical rules, in order to product triples, is very basic. Currently, we only have about 5 rules. Although it is good enough to handle a lot of questions, we are not able to process conjonctions for example (e.g. *"Who wrote "Lucy in the Sky with Diamonds" and "Let It Be"?"*).

Preprocessing merging

There remains nodes which should stay together but are not merged by our module, for instance *prime minister* or *state of the art*. Recognizing such words is called *Multiword Expressions Processing*. This task is a whole part of Natural Language Processing theory.

We have several tracks to improve merging. Existing algorithms or softwares need to be tested. We could also use multiword expressions dictionaries.

Question type analysis

The current algorithm attaches great importance to the type of the input question. Sentences starting by a question word (Who, Where, How ...) are better processed than Yes/No questions for instance.

Triples tree improvement

The triples tree will be improved to take into account new types of nodes, adapted to databases queries. For example, a node could be tagged "FIRST" to pick the first occurence of a list of answers (e.g. FIRST(?,presidents of, United States)).

4.2 Machine Learning 1

TODO

4.3 Machine Learning 2

TODO

Chapter 5

Wikidata module

Wikidata module is our main proof of concept module which aim is to demo the ability of our framework to allow the easy creation of huge modules able to answer to thousand of questions. This module tries to answer to general knowledge using the data stored in Wikidata.

Wikidata is a free knowledge base hosted by the Wikimedia Foundation as a sister project of Wikipedia. Its aim is to build a free, collaborative, multilingual structured database of general knowledge (for more information see [VK14]). It provides a very good set of API that allows to consume and query Wikidata content easily. Wikidata is builds upon elements (called items) that are about a given subject. Each item has a label, a description and some aliases in order to describe it and statements that provides data about this subject.

The Wikidata module has been written in PHP in order to rely on good libraries that allow to easily interact with the Wikidata API. Some contributions to these libraries have been done in order to make them fit better with the module use case. This module works in tree steps:

1. It maps `resource` nodes of the question tree into Wikidata content: the subjects of `triple` nodes are mapped to Wikidata items, predicates to Wikidata properties and objects to the type of value that is the range of the Wikidata property of the predicate. If more than one match are possible, a tree per possible match is output.
2. It does queries against Wikidata content using the previously done mapping in order to reduce as much as possible trees. When we have a `triple` node where the object is missing the module gets the Wikidata item of the subject and looks for values for the predicate property and replace the `triple` node with a `resource` node for each value of the triple (and so builds as many trees as there are of values). When there is a `triple` node with a missing subject the module uses the WikidataQuery tool API with a standalone wrapper built for the project that returns all items with a given statement.
3. It adds clean text representation of `resource` nodes added by the previous phase.

The global architecture of the module has been quickly studied by one of the Wikidata developpers that found it fairly good.

Conclusion

Even though the Projet Pensées Profondes has not yet started to actually answer questions, we already made a huge progress in this direction by having the structure of the project already up and running.

Few pieces remain to implement before we get something working as we expect it and this means we do not see any incoming structural problem. Thus we are confident in the future of the project and that it will continue improving over time.

Bibliography

[dMM08] Marie-Catherine de Marneffe and Christopher D. Manning. Stanford typed dependencies manual, 2008. http://nlp.stanford.edu/software/dependencies_manual.pdf.

[VK14] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledge base. *Communications of the ACM*, 57:78–85, 2014. http://cacm.acm.org/magazines/2014/10/178785-wikidata/fulltext.

Appendix A

Question parsing – Triples tree

Figure A.1: Triples in tree form


```
{
    "subject": {
        "subject": {
            "type": "missing"
        },
        "type": "triple",
        "object": {
            "subject": {
                "type": "missing"
            },
            "type": "triple",
            "object": {
                "type": "resource",
                "value": "United States"
            },
            "predicate": {
                "type": "resource",
                "value": "president of"
            }
        },
        "predicate": {
            "type": "resource",
            "value": "birth date of"
        }
    },
    "type": "triple",
    "object": {
        "type": "missing"
    },
    "predicate": {
        "type": "resource",
        "value": "definition"
    }
}
```