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 developed an eponymous set of tools that accomplish different tasks and fit together thanks to a protocol we developed.

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

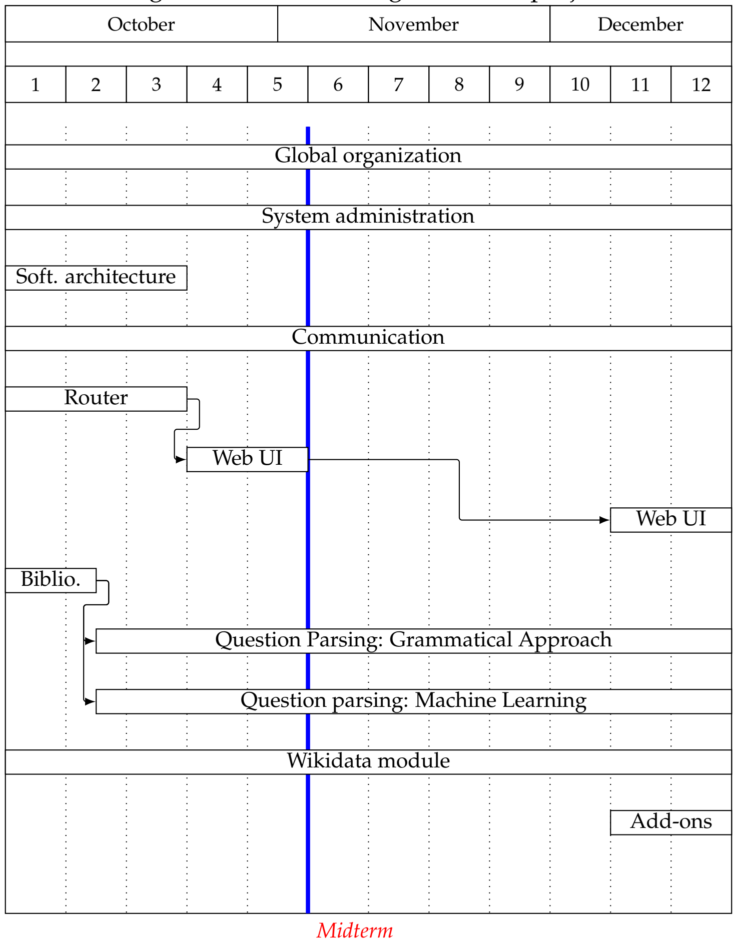
Given the young age of this project, these pieces are only starting to emerge with their first features and mutual communications, so we will describe them separately in this document without much of a general overview of the project.

Chapter 1

Overview

Figure 1.1 presents the initial schedule of the project.

Figure 1.1: GANTT diagram of the project



Arriving to the midterm, we can see that only two work packages are supposed to be finished. Indeed, the software architecture is well defined. On the other hand, the web user interface needs to have more features than we thought at the beginning.

A major reorganization is about machine learning. We split it into two different work packages which implement two different machine learning algorithms.

In the other work package, the progression matches with what we expected.

Most of them produced, at least, a partially working code, allowing us to deploy the current state of the PPP on line:

`http://ppp.pony.ovh/`

There is also a website, for communication purpose:

`http://projetpp.github.io/`

The whole project is hosted on GitHub:

`https://github.com/ProjetPP/`

Chapter 2

State of the art

The project is about *Question Answering*, a field of research included in *Natural Language Processing* (NLP) theory. NLP mixes linguistic and computer science and is made of all kinds of automated technics that have to deal with natural languages, such as French or English. For instance, automatic translation or text summarization are also parts of NLP.

In Natural Language Processing, sentences are often represented in a condensed and normalized form called *triple representation*. It distinguishes three types of units: subject, predicate and object. These units are gathered into triples to catch the meaning of the sentence. For example, the phrase "The turtle eats a salad." will be represented by the triple (the turtle,eats,the salad). Two triples can be associated to "The president was born in 1950 and died in 2000.": (the president, was born in, 1950) and (the president,died in, 2000). This representation has been formalized into the *Resource Description Framework* (RDF) model. It consists of a general framework used for describing any Internet resource by sets of triples. Our first goal is to parse questions to get their triples representation.

Many algorithms have been developed since fifty years with the objective of understanding the syntax and the semantic of sentences. Two popular graph representations are widely used:

- parse structure tree. It tries to split the sentence according to its grammatical structure.
- dependency tree. It reflects the grammatical relationships between words.

Existing libraries, such as *NLTK*[1] or *StanfordParser*[2] provide powerful tools to extract such representations.

We did not find a lot of article exposing procedures to get triples from the grammatical structure. For instance [RDD+07] tries to perform this from the parse structure tree. However we believe the dependency tree could be more appropriate. We intend to develop a new algorithm using it.

We have also observed a growing use of machine learning technics in Natural Language Processing, and especially in Question Answering. Some very interesting results have been obtained with this popular field of computer science. We are trying to apply two existing machine learning algorithms to our project.

Finally, some existing tools are very close to our goal[3,4]. They allow us to have a clear idea of the state of the art, and what performances in question answering we can expect. Moreover, some datasets of questions are available from two popular challenges in Question Answering (TREC and QUALD challenges). They will enable us to compare our performances to existing state of the art tools.

[1]`http://www.nltk.org/`
[2]`http://nlp.stanford.edu/software/lex-parser.shtml`
[3]`http://quepy.machinalis.com/`
[4]`http://www.ifi.uzh.ch/ddis/research/talking.html`

Chapter 3

Datamodel and communication

We describe the choices we did about representation of the data and communication between modules. These choices are described precisely in the documentation[1] of the project.

3.1 Data model

First, we present the data model. All normalised 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 normalised 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 question parsing module is to transform

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

[1] https://github.com/ProjetPP/Documentation/

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.

We do not plot explicitly the tree in the user interface, but we used a string representation defined recursively by:

- A missing node is symbolized by a "?", possibly followed by an id (an integer).
- A resource word is symbolized by the corresponding string.
- A triple node of subject `subj`, predicate `pred` and object `obj` is symbolized by `(SUBJ,PRED,OBJ)` (where `SUBJ`, `PRED` and `OBJ` are the string representations of `subj`, `pred` and `obj`).

For instance, the previous tree will be represented by the string:

```
(George Washington, identity, ?)
```

3.2 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 core gets a complete response, ie. a tree without any 'missing' node.

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": <accuracy of the answer>
    }
}
```

The measure field contains two values: relevance and accuracy.

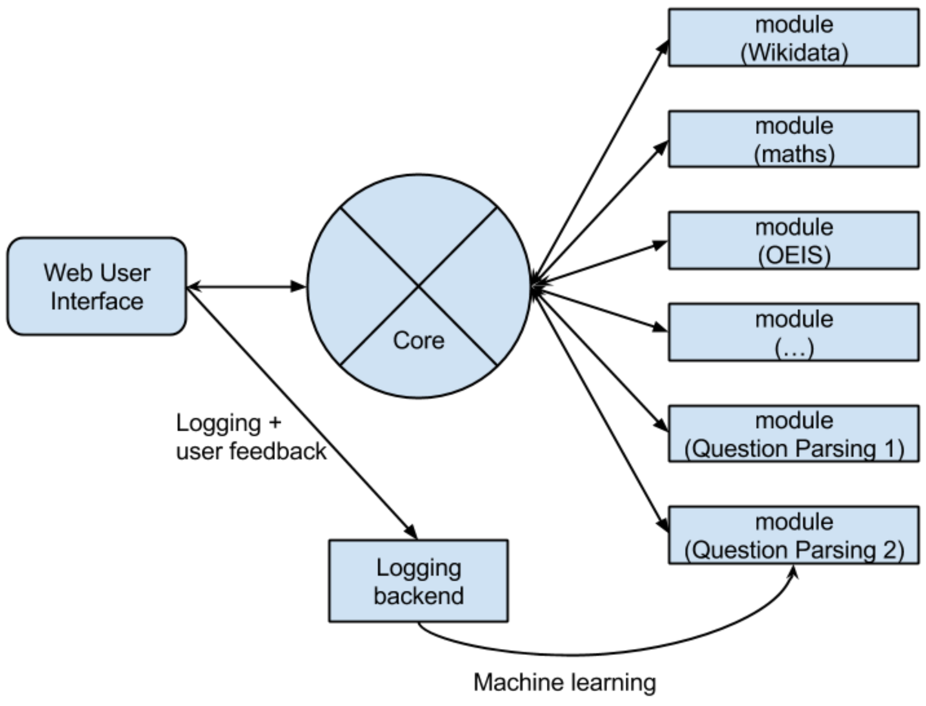
- `accuracy` is a self-rating of how much the module may have correctly understood (ie. not misinterpreted) the request/question. It is a float between 0 and 1.
- `relevance` is a self-rating of how much the tree has been improved (i.e. progressed on the path of becoming a useful answer). A positive float (not necessarily greater that 1; another module might use it to provide a much better answer).

This form allows each module to access to the previous results, particularly to the request of the user. The objects for request and response contain some extra data, such as the language used.

The data model have been implemented in a nice set of objects in both Python[2] and PHP[3] in order to help the writing of modules.

We could define a linear representation for the trace, using the representation of the datamodel, but it is not relevant. Indeed, this information will never be printed on the user interface.

Figure 3.1: Architecture of the PPP



[2]`http://github.com/ProjetPP/PPP-datamodel-Python/`
[3]`http://github.com/ProjetPP/PPP-datamodel-PHP/`

Chapter 4

Core

4.1 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, and is developed in Python.[1]

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 above configuration presents three modules: the Wikidata module, an example Python module (which answers the question "Who are you?"), and the Question parsing module.

The current state is that the Core is successfully able to communicate with all modules that have been written: the Wikidata module, an example Python module (which answers the question "Who are you?"), and the Question parsing module.

[1]https://github.com/ProjetPP/PPP-Core/

4.2 Libraries for modules

The core also exports its class in charge of handling and parsing HTTP requests following the format defined in the data model. This class is an abstraction over a Python HTTP library (python-requests), allowing module developpers to focus on developping their actual code instead of handling communication.

These has proven to be efficient for connecting the Grammatical Question Parsing with the Core: we only had to copy-paste the demo code (used for reading and printing in the console) into a function called by this library, and it was working as is.

We are also planning on exporting the configuration library too, since we notice modules are likely to share the same way of handling configuration (a JSON file, whose path is given *via* an environment variable.

4.3 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 Question parsing module. Then, it will forward this answer to all other modules, including the Wikidata module which will be able to answer.

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

Chapter 5

User interface

We decided to implement first only a web user interface. This interface is composed of one web-page developed in HTML 5 with some pieces of JavaScript and CSS .[1] We have taken care of having an interface that fits nice on both huge screens of desktop computers and small screens of phones.

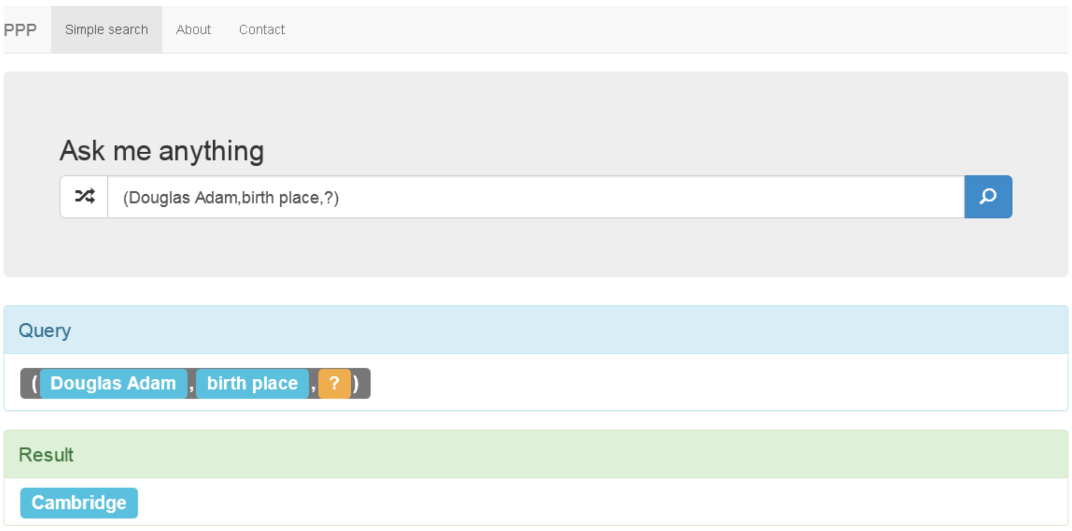


Figure 5.1: The user interface

It is composed of one huge text input with a button to submit the query and another 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 converts this easy to use notation into the standard format.

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

5.1 Logging

We decided to log all requests made to the PPP to improve our algorithms, and particularly to feed the results to Question parsing 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.

[1]`https://github.com/ProjetPP/PPP-WebUI/`

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 yet; so far we only started implemented a backend that stores data (gathered via the user interface) to a SQLite database.

Chapter 6

Question parsing

The goal of this module is to transform questions into trees of triples, as described in section 3.1, 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" occurrence 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" occurrence.

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

6.1 Grammatical approach

Trees of triples can be produced after analysing the grammatical structure of sentences. We developed a module in Python which produces triples using this grammatical approach.[1] 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:

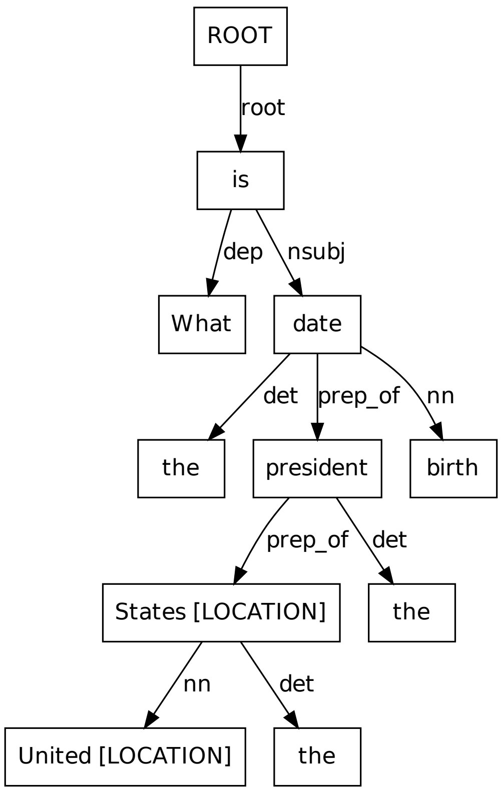
What is the birth date of the president of the United States?

6.1.1 *Stanford CoreNLP*

The *Stanford CoreNLP* library [2] 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.

[1]`https://github.com/ProjetPP/PPP-NLP-classical/`
[2]`http://nlp.stanford.edu/software/corenlp.shtml`

Figure 6.1: Dependency tree



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

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

Figure 6.1 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.

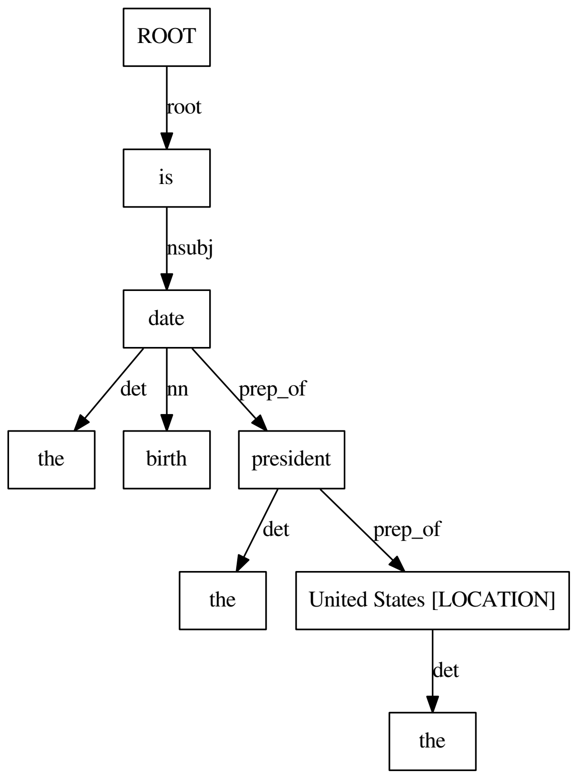
6.1.2 Preprocessing

The preprocessing consists in a sequence of operations executed on the tree output 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:

[3]`https://bitbucket.org/ProgVal/corenlp-python/overview`

Figure 6.2: Dependency tree preprocessed



- **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.
- **Merge named entities.** The *Stanford CoreNLP* library performs a *named entities recognition* (NER), which provides informative tags for some nodes. For instance, *United* and *States* are tagged *LOCATION* (see figure 6.2). 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.

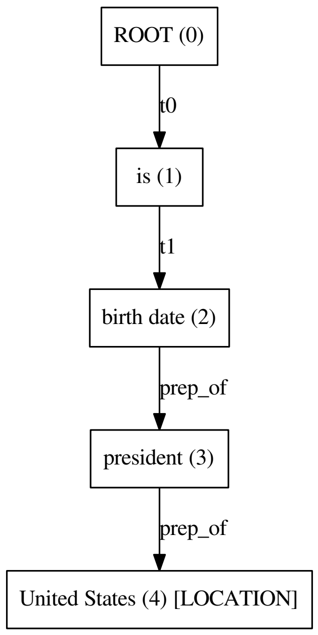
Preprocessing is illustrated on figure 6.2. The question word is *What*.

6.1.3 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.

Figure 6.3: Dependency tree simplified



- 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.

On our example, the edge:

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

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

One of the triples production rules tag is :

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

The simplified tree of our example is illustrated on figure 6.3.

6.1.4 Triples production

The triples 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 printed these numbers on figure 6.3.

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 triples production rules tagged previously. For instance, an edge tagged **t2**:

$$a \xrightarrow{t2} b$$

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 links two unknowns ?a = ?b, instead of producing 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)

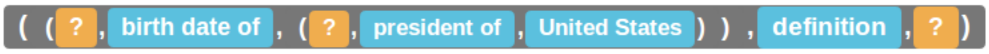
The four previous rules are simplified into a set of triples:

(?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)**$\wedge$**(?1 , birth date of , ?3)**$\wedge$**(?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 6.4. Figure A.1 contains the formal representation of the triples tree of our example.

Figure 6.4: Triples tree


((? , birth date of , (? , president of , United States)) , definition , ?)

Backend modules (such as Wikidata module) will have to fill intermediate unknowns : ((August 4 1961,birth date of, (Barack Obama, president of, United States)), definition, ?) and finally provide the final answer that replaced *?* (for example: a description of the August 4 1961 date).

6.1.5 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 conjunctions 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 occurrence of a list of answers (e.g. FIRST(?,presidents of, United States)).

6.2 Machine Learning: Reformulation

One approach with neural network is a reformulation approach. It can work at two levels:

Either after the Grammatical approach, it consists in taking the already formed tree, and modifies it in another tree. The idea is maybe a tree is correct for a human, but for the answering module it is not, because of a complex formulation for instance, so the aim of the module is to transform it in an adapted tree. For example "What color is the white horse of Henri IV?" is trivial for us, but is very complex for the modules.

Or we can use the same idea after a syntactic analysis.

We developed a Machine Learning module in C++.[4]

6.2.1 How it works

As we works with requests, we consider everything is a request, even a word.

Mathematical spaces

There are 2 generic spaces: the one of words which is a vector space of dimension 50 and the space of request which is the space of word triples. The first word of a triple represents the subject, the second represents the predicate and the last the object. To distinguish words which are vectors and words with letters, we will add the adjective English to the seconds. The choice of a 50-dimension is arbitrary and can be modified if necessary, but taking a higher dimension could slow the learning process, and with a lower space we could lost some expression power, which means lead to very poor results.

Dictionnary

There is a dictionary of correspondences between English words and requests which is the base of the process. As there are so many proper names (which can be arbitrary), we replace them in a sentence with a tag NAME, then we treat the problem with the tags. At the end, we replace tags with corresponding real names in the final tree. We do the same with numbers or math formula. Finally we add tag UNKNOWN to represent the "hole" in the request tree.

[4]`https://github.com/ProjetPP/PPP-NLP-ML/`

Transform a question into a word

The reader may wonder, a request is a triple of word, but that does not mean a request has only three elements, or only one level of recursion. Such an approach would be very poor. So we have two functions to deal with tree complexity: compact and uncompact. Compact takes a request and makes a word of it, uncompact does the reverse job. Both are matrices. It is important to notice that they are not bijective after restraining the request space to existent requests. Applying uncompact then compact should give the identity, with of course an error margin, but when compacting then decompacting we only have to find an equivalent request, i.e. the same question with another formulation. The reader will understand what it means with the algorithm for the tree reconstruction. We can recursively transform an arbitrary request tree in a word.

The second approach use directly syntactic tree, to reconstruct the request we use the fusion operation, it takes two requests in entry and give one at output. It is a matrix. First replace all English words of the tree with the corresponding request of the dictionary, and then take two leafs with same parent and use fusion operation to replace the parent node with a leaf well labeled and repeat this operation till root is a leaf. Finally we compact the only remaining request.

Transform a word into a request tree

In both case we have a word representing a question. Now let us construct a request tree. First we must take $\delta > 0$ a precision factor. Infinity is the lowest precision, small δ is good but can conduct to an infinite request tree, meaning the functions do not do their job well.

TREERECONSTRUCTION Entry e is a word

- (s,p,o)$\leftarrow$ uncompact(e)
- Find English word fp in the dictionary with nearest predicate to p
- Find English word fs in the dictionary with nearest subject to s
- If distance fs to s is greater than δ fs $\leftarrow$ TREERECONSTRUCTION(s)
- Find English word fo in the dictionary with nearest subject to o
- f distance fo to o is greater than δ fo $\leftarrow$ TREERECONSTRUCTION(o)
- Return (fs,fp,fo)

6.2.2 Advancement

The implementation of the reformulation is written in C++. However it is not finished yet. The dictionary has been generated using the clex .[5] The three functions are functional with a multithread approach to speed-up the computation time, also backpropagation is ready to be implemented. First approach of reformulation is implemented: that means taking a request tree in input, it returns another tree which should be equivalent if learning succeeded. However the learning process is not implemented yet.

6.2.3 Future work

Finding a way to learn everything with first or second approach is the most important, as the first answering module in functional, learning is possible. Then, learning and computation speed-up will be important, the search for nearest neighbor is long, maybe it is linear, but with near 100 000 words and high dimension it becomes consequent, use of heuristics could be a good idea, for example there exists distance sensitive hash. Kd-trees allow a search in log-time (with precomputation) ; but with dimension 50, the constant factor 2^{50} is too large.

[5]`https://github.com/Attempto/Clex`

6.3 Machine Learning: Window approach

We used machine learning algorithms in order to produce triples from scratch, i.e. without any grammatical library like *Stanford CoreNLP*.

Motivations come from two points:

- Triples are linked to the semantic of the sentence, and not directly from the grammar
- It has been shown that a machine learning approach can produce, for a large panel of different NLP problems, very good solutions, closed to *state of the art* algorithms [CWB⁺11].

This work is based mainly on the paper "Natural Language Processing (almost) from Scratch" [CWB⁺11]. For the baseline, we limited ourselves to one level of depth. For example, the sentence "What is the birth date of the president of the United States?" will be converted to the triple: (president of the United States, birth date, ?).

We used a look-up table and a window approach neural network. The complete package[6] was written in Python 3 and in Lua, with the Torch7 library.[7]

6.3.1 Data set

Because we used supervised algorithms, we need a data set of annotated questions. This data set has to be built manually, because we did not find on internet a data set that directly answers to the problem of triple extraction. Build this data set is a fastidious work. Currently our data set is composed of 168 questions. Because the mean of number of words in one question in the data set is around 7.5, it gives us 1264 entries to train/test the neural network.

6.3.2 Structure of the network

For each word w of the sentence, we want to classify w into four categories: *subject*, *predicate*, *object* and *to ignore*. The classification of each word of the sentence into these four categories produce the desired triple.

As described in [CWB⁺11], we used a window approach, and a look-up table.

Window approach

We used a window that focuses on the word to classify. For example, if the sentence is "What is the birth date of the president of France?", and the word to classify is "date", for a window size of 7, the window is: "is the birth **date** of the president".

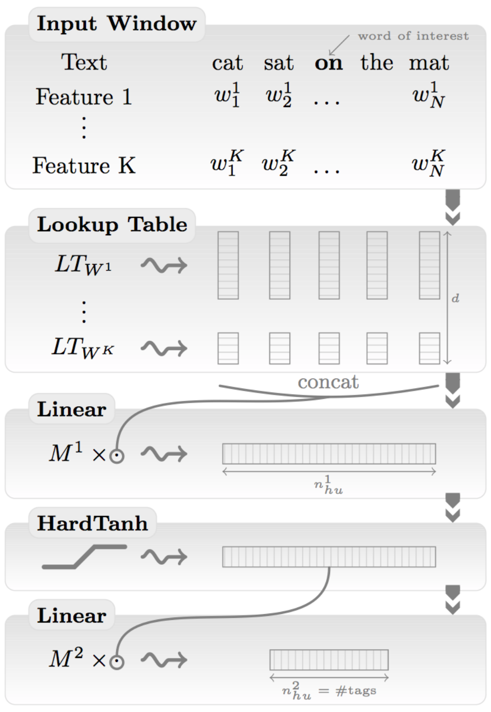
We used this window because neural networks works with a fixed number of input parameters. The window size is a meta parameter to choose. This window has to be large enough that we can decide in the context of the window in which category a word is. We used a window of size 7.

Look-up table

The look-up table is a dictionary that associates to each word w a vector $V_w \in \mathbb{R}^n$, where n is the number of parameters we used to encode a word. We used $n = 25$. If two English words w_1 and w_2 are synonymous, then $||w_1 - w_2||_2$ is small.

[6] `https://github.com/ProjetPP/PPP-NLP-ML-standalone/`
[7] `http://torch.ch/`

Figure 6.5: The neural network architecture, as described in [CWB+11]



The construction of look-up table is described in [CWB+11] and used unsupervised machine learning techniques. We used the precomputed look-up table found here: `http://metaoptimize.com/projects/wordreprs/`

We also add one parameter that give us if the word starts with a capitalize character or not. Finally words are embedded in vectors of dimension 26.

The neural network

We tried two different architectures: -A linear model, i.e. without any hidden layers. This gives us $26 \times 7 \times 4 = 728$ parameters to optimize. -A non linear model with one hidden layer of size 10. This gives us $26 \times 7 \times 10 \times 4 = 7280$ parameters to optimize.

The linear model has the advantage to have few parameters, so it can be learned with a small data set of annotated questions. However we found that this model is not enough powerful to catch the complexity of the problem we want to solve. The non linear model is more complex and can describe with more precision how the English language works. But because of the huge number of parameters to learn, we need a larger annotated data set that we currently have.

We add one regularization parameter to limit over-training. The neural network is implemented in Lua with the Torch7 framework. Few minutes of computation are needed to train successfully the model.

6.3.3 Results

This baseline algorithm, witch was the goal for the midterm, give us quite good results. The linear model has an accuracy of 80% on the training set, and the non linear model has an accuracy of 98% on the training set. On the test set, these two models have an accuracy of 60%, witch is much better than chance (a random method would give us 25% of accuracy), but it is not efficient enough to be used for tricky questions (e.g. that are not closed to one of the sentence in our annotated data set).

6.3.4 Future work

Unsupervised deep learning

We could use auto-encoders with Restrictive Boltzmann Machine (R.B.M) and an unsupervised data set of questions to learn a much more efficient representation of a question, as explained in [FI12] and in [HS06]

We can easily found large data set of non annotated questions. One advantage of doing this is to limit supervision (because our annotated data set is very small), and it should improve the capacity of our model to generalize to questions that are not in our data set.

Used a more efficient preprocessing

We could reuse a part of the work done in the grammatical approach to have a better input for the neural network. For example, we could use the "Merge quotation nodes" and the "Merge named entities" steps to simplify input questions.

Merge the work done with the grammatical approach

This ML approach gives us, for each word of the sentence, the probability to be in the *subject*, in the *predicate*, in the *object* or a word to ignore. Maybe we can use this information to improve the accuracy of the grammatical approach.

Chapter 7

Wikidata module

Wikidata module[1] is our main proof of concept module which aims to demonstrate 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*.[2]

Wikidata is a free knowledge base hosted by the *Wikimedia Foundation* as a sister project of *Wikipedia*. It aims 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 built upon elements (called items) that are about a given subject. Each item has a label, a description and some aliases 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 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 performs queries against *Wikidata* content using the previously done mapping 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, 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 values). When there is a `triple` node with a missing subject the module uses the WikidataQuery[3] tool API with a standalone wrapper[4] 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* developers that found it fairly good.

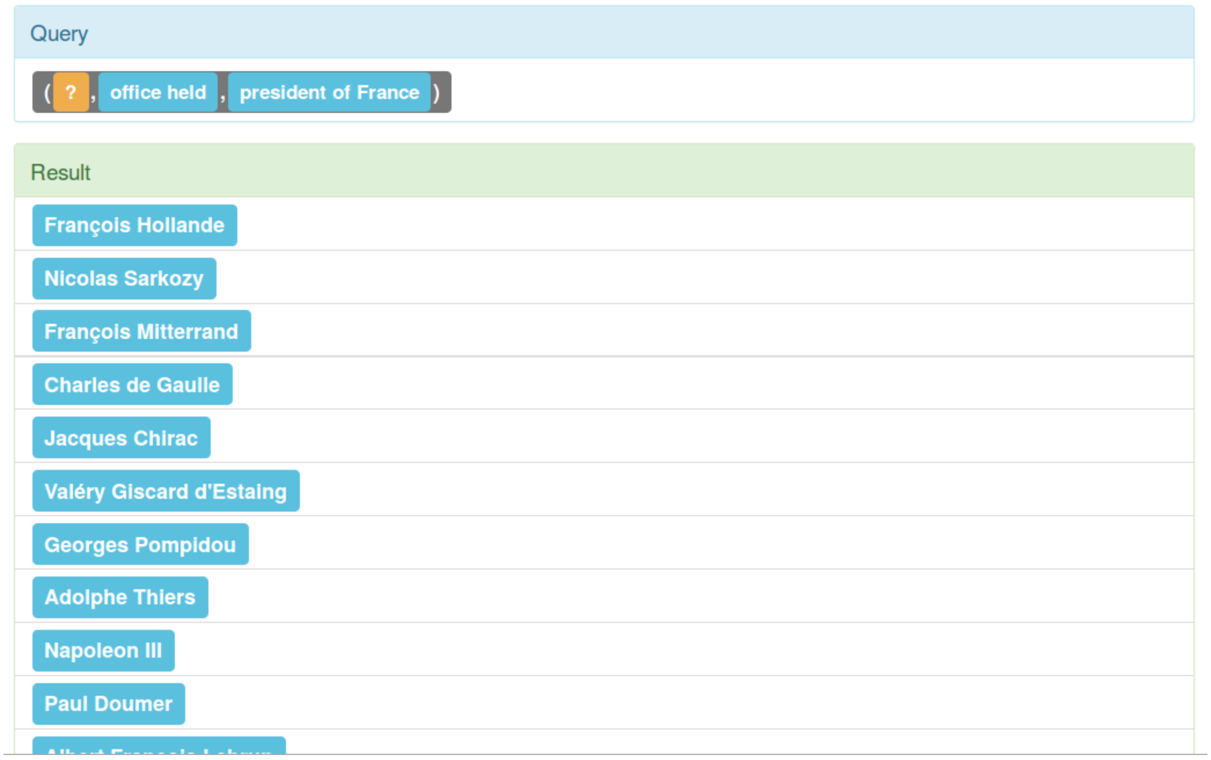
7.1 Future work

A lot of work remains to do in this module like:

[1]`https://github.com/ProjetPP/PPP-Wikidata/`
[2]`http://www.wikidata.org/`
[3]`http://wdq.wmflabs.org/`
[4]`https://github.com/ProjetPP/WikidataQueryApi/`

- Improve formatting of the answers and supports of Wikidata value types.
- Compute relevance and accuracy of the anwsers.
- Filter not relevant result: people that are looking for the list of the presidents of the United States are usually not looking for fictional ones.
- Handle triple that does not directly match to Wikidata model. If we looks for people born in France we are also looking for people born in Lyon or if we are looking for the American Secretary of State we are looking for the person that has as office "American Secretary of State".

Figure 7.1: An example output from the *Wikidata* module


Query

(? , office held , president of France)

Result

François Hollande

Nicolas Sarkozy

François Mitterrand

Charles de Gaulle

Jacques Chirac

Valéry Giscard d'Estaing

Georges Pompidou

Adolphe Thiers

Napoleon III

Paul Doumer

Conclusion

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

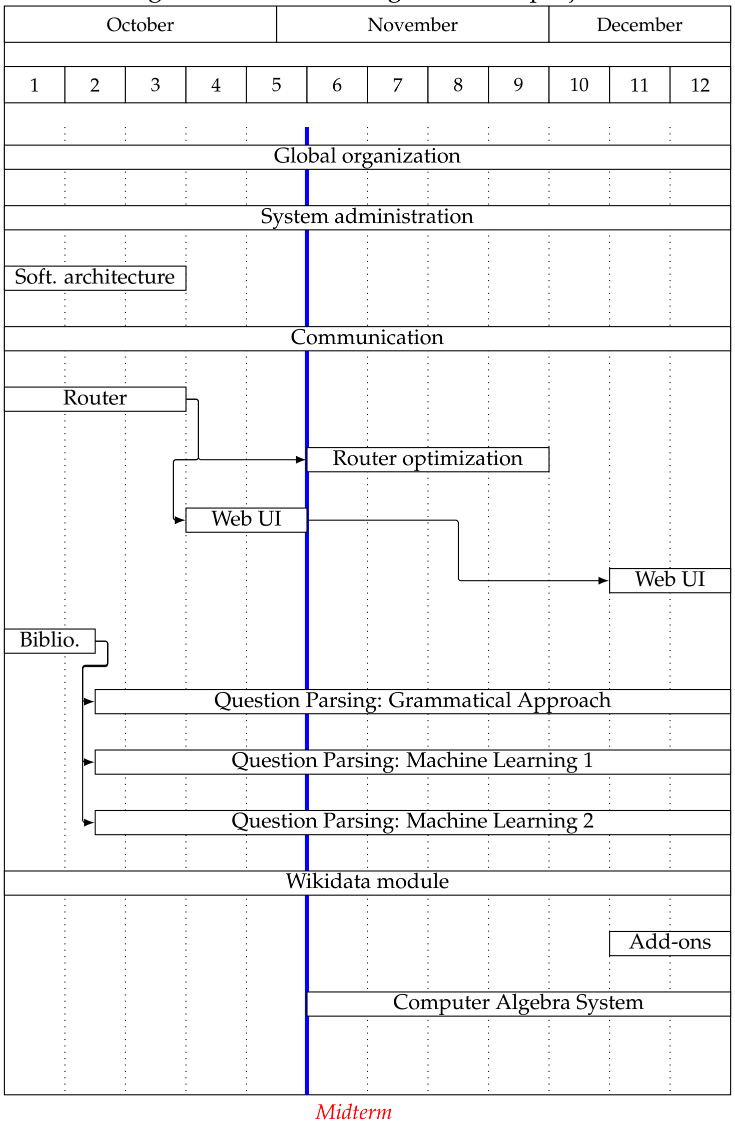
Few pieces remain to be implemented before we get something working as we expect, but we are confident in the future of the project and in its improvement over time.

We keep the same organisation with few modifications, as presented in figure 7.2.

Their are two new work packages:

- "Computer Algebra System", which will to implement such a system in the PPP.
- "Router optimization", which will consists in writing an algorithm for sorting requests by usefulness, and eventually dropping the less useful ones during the computation, in order to improve performance.

Figure 7.2: GANTT diagram of the project



Bibliography

[CWB$^+$11] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

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

[FI12] Asja Fischer and Christian Igel. An introduction to restricted boltzmann machines. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 14–36. Springer, 2012.

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[RDD$^+$07] Rusu, Delia, Dalia, Lorand, Fortuna, Blaž, Grobelnik, Marko, Mladenić, and Dunja. Triplet extraction from sentences. 2007. `http://ailab.ijs.si/delia_rusu/Papers/is_2007.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"
    }
}
```