French Journal Officiel Graph Construction

Alexis Thual
alexis.thual@polytechnique.edu

Joël Seytre
joel.seytre@student.ecp.fr

Abstract

The French *Journal Officiel* (JO) contains additions and amendments to the French law and is published almost every day. These modifications are reported in articles which sometimes refer to one another and therefore inherently present a graph structure where each article can be seen as a node and references are analogous to edges. This project's first goal was to parse data coming from the JO's official website and to build such a graph structure using an efficient database system. The second goal was to demonstrate what use can be made of such data and build a suggestion tool which would take advantage of the captured graph structure.

1 Scrapping data from the JO's official website

Almost every day, a new publication of the French Journal Officiel is published on `http://www.journal-officiel.gouv.fr/` and it's articles stored on `https://www.legifrance.gouv.fr/`. Such a publication consists of a categorized list of links to articles; articles themselves consist of pieces of text which contain additions or amendments to the French law. These pieces of text often contain links to other articles - especially in the case of amendments but not only.
Before explaining how one can parse these publications and build a graph structure out of it, we will expose some of the reasons why we believe such construction is important.

1.1 Motivations for scrapping the French JO

The first assessment to make is that it is currently impossible to use a text query to search publications of the JO - private solutions might exist but standard citizens do not have access to them. Google's search engine will eventually index the pages of this website after some time but complex searches remain be out of reach. Building an efficient search tool requires to first parse all available information coming from the website.
Moreover, every JO publication contains on average 100 articles, which in a year approximately sums up to 20,000 pages of text. Therefore, not only is it important to be able to search such database, it is also crucial to evaluate the relative importance of a given article.

Finally, we believe such results, as well as the algorithms which helped construct it, should be open-source. In that spirit, all our code can be found on the following public Github repository:

https://github.com/alexis-thual/parsing-journal-officiel

File names given in the rest of this report refer to files which can be found on this repository.

1.2 JO Publications' HTML structure

In order to scrap a website, one first has to study the HTML structure of pages that are to be parsed. The difficulty concerning the French JO is that some errors occasionally arise - probably due pieces of the publication process that we suspect are not automatic but human-made.

1.2.1 Summary structure

The JO's summaries present a recursive HTML structure: articles usually belong to a given category of articles and categories can be represented as a tree of concepts. An example of a cleaned HTML structure is given in code snippet 1.

```
<div class="sommaire">
  <ul>
    <li>
      <h3>Décrets, arrêtés, circulaires</h3>
      <ul>
        <li>
          <h4>Textes généraux</h4>
          <ul>
            <li>
              <h5>Premier ministre</h5>
              <ul>
                <li>
                  <a href="href">Link</a>
                </li>
                ...
              </ul>
            </li>
            ...
          </ul>
        </li>
        ...
      </ul>
    </li>
    ...
  </ul>
</div>
```

Code Snippet 1: Recursive structure example

The main HTML tag is easily identified thanks to it's `class` attribute and can be seen as the root of our concept tree. It contains a list of objects which are either lists of concepts themselves - that is to say a simple node of our graph - or HTML links to articles - which are the leaves of our graph. Let's name this type of recursive list `recursiveUL`. The overall structure is further illustrated in code snippets 2 and 3.
In order to tell if an instance of `recursiveUL` is a list or a link to an article, one checks the presence of a title tag - such as `<h3>`. In case one can't find a title, it means they have reached a leaf of the graph; otherwise, one treats the current node as a casual node.

```
<div class="sommaire">
  {recursiveUL}
</div>
```

Code Snippet 2: Main HTML tag structure description

```
<ul>
  [
    <li>
      (
        <h3 | h4 | h5>Title</h3 | h4 | h5>
        recursiveUL
      )
      OR
      articleLink
    </li>
  ]
</ul>
```

Code Snippet 3: recursiveUL HTML structure description

1.2.2 Article structure

Every link found in the summary is then explored, leading to article web-pages whose structure is similar to that of code snippet 4.
Once again, the main `div` tag is easily identified thanks to it's class attribute. Key pieces of information about the article are given in the `enteteTexte` tag, such as ELI and NOR number, which are normalized references used to classify articles (as decrees - *décrêts* - ordres - *arrêtés* - etc). The article's body is contained in the following `div` tag.
A unique identifier for each article is needed. NOR references are supposed to play this role but are not reliably available for every webpage. However, *legifrance* seems to have created it's own primary key which can be found in every url displayed in a JO's publication. We'll denote this primary key as *cid*.

```
<div class="data">
  <div>...</div>
  <div>
    <div class="enteteTexte">
      ...
    </div>
    <div>
      ...
    </div>
  </div>
</div>
```

Code Snippet 4: Article HTML structure

1.3 Text parsing

Previously presented structures were cleaned and simplified: the HTML code of the summaries and articles looks a lot messier and the abundance of useless HTML tags greatly obfuscates a general pattern. In order to get all needed information, eliminating all HTML tags in targeted divs was needed: that means, for instance, eliminating all the tags in the *enteteTexte* `div` tag and parse the remaining text using regular expressions so as to find important pieces of information.
If this works for some tags, some of the tags' inner HTML have a meaningful structure and cannot be blindly thrown away. Tables and links for instance are two frequent issues.
One therefore has to isolate such tags and parse them into a text-only format. We chose JSON in this implementation.

1.4 Page scrapping

Once the expected HTML structure is well known, one can start scrapping the source and copy it to local text documents. We used the library Scrapy in order to do so and implement the logic previously introduced. The pieces of code using Scrapy intended to scrap web-pages are called *spiders*.

1.4.1 Seeking summary web-pages

Once the URL of a summary web-page is known, feeding it to a spider is easy. The spider will then parse the given web-page, extract all links pointing to article web-pages, follow them and parse these pages as well before all results are stored locally. Our spider class implementation can be found in `spiders/JOPublicationSpider.py` and contains the aforementioned logic.
However, finding such URL isn't necessarily simple. Luckily for us, JO publications can be found using the following URL pattern:

https://www.legifrance.gouv.fr/eli/jo/year/month/day

As not all (year, month, day) tuples lead to actual JO publications, one tries and reach out to such URL and checks whether the yielded web-page follows the previously described HTML-structure referenced in code snippet 1.

1.4.2 Parallel spiders and query speed

Spiders can be parallelized and launched as independent processes. As they do not need a lot of memory, a casual laptop can easily launch up to 10 spiders. So as to make sure spiders parse different URLs, we first partitioned the range of dates from which URLs are to be generated such that the spiders had non-overlapping URLs as shown in algorithm 1.
Parallelizing spiders using a single IP address can usually lead to web session problems (the target server can for instance limit the number of requests allowed per minute from a single IP address). Fortunately for us, neither `journal-officiel.gouv.fr/` nor `legifrance.gouv.fr/` implement such restrictions. However, if too many requests are sent every second, both servers tend to yield errors and data comes missing for some articles. We therefore chose to limit the query speed and impose a delay of .1 second after every response from the server.
Overall, we used 5 parallel spiders and can send up to 50 requests per second at cruise speed. The full code can be found in `crawl.py`.

Algorithm 1: Parallelize scrapping

Input : $D = [d_1, ..., d_m]$ an array of dates
  $(p_1, ..., p_n) \leftarrow$ partition of D
  Instantiate spiders set $S = \big(\text{spider}(p_i)\big)_{1 \leq i \leq n}$
  for $s \in S$ **do**
    s.start()
  end for

1.4.3 Results

When using 5 parallel spiders, parsing 10 JO publications takes approximately 20 seconds, which is consistent with what one would expect.
In order to further test our implementation, we tried parsing 5 consecutive years (if we were to parse the entire JO, we would have access to up to 30 years of data). Results are consistent with what one should expect: it takes approximately 6 hours to parse 5 years of data, which results in 1.3Go of parsed data.

2 Database construction

Spiders parse data and store it in local JSON files. These files can then be stored in a file system dedicated to text search such as ElasticSearch.

2.1 Motivations for choosing ElasticSearch

2.1.1 Document-oriented file system

We chose to use ElasticSearch as the database to store our data. Contrary to SQL databases, ElasticSearch isn't relational: text documents are analyzed using algorithms which calculates word

frequency (in the given document and in the database overall) so as to speed up text queries afterwards. In particular, ElasticSearch makes it complicated to do nested-type queries: for instance, recovering all articles from a given summary isn't quite done the proper way. A more efficient system would be to store summaries and articles primary keys in a SQL database system and put articles' content an ElasticSearch-like system.

2.1.2 Storing an optimized version of the graph structure

However, in order to make the nested queries which we are interested in fast enough, we stored in every article all primary keys of linked articles, which also allows us to deal more easily with edges storing.

2.2 Populating our database

The algorithm which populates our database is quite straight-forward: one iterates through all generated documents and stores them as-is in the database. ElasticSearch is optimized to receive and store such field-oriented documents and indexes them so as to optimize text search over the database. The entire logic can be found in `populateES.py`.

2.3 Term Frequency - Inverse Term Frequency (TF-IDF)

2.3.1 Scoring

ElasticSearch itself uses the well-established TF-IDF method to score the elements of the database with respect to a query (more information here: https://www.elastic.co/guide/en/elasticsearch/guide/current/scoring-theory.html).
This inspired us to build connections on a k-nearest neighbor basis. To achieve that we query the database with the words of an article and compute a proximity score between that article and all the other ones as follows:

$$\text{score(JO article, query JO article)} = \sum_{\text{word} \in \text{query JO article}} \text{score (JO article, JO database, word)}$$

where we have:

$$\text{score (JO article, JO database, word)} = \frac{|\text{occurences in query}| + |\text{occurences in JO article}|}{|\text{occurences in JO database}|}$$

Note: to implement our TF-IDF scoring we made some use of this public GitHub library https://github.com/hrs/python-tf-idf.

2.3.2 Interpretation

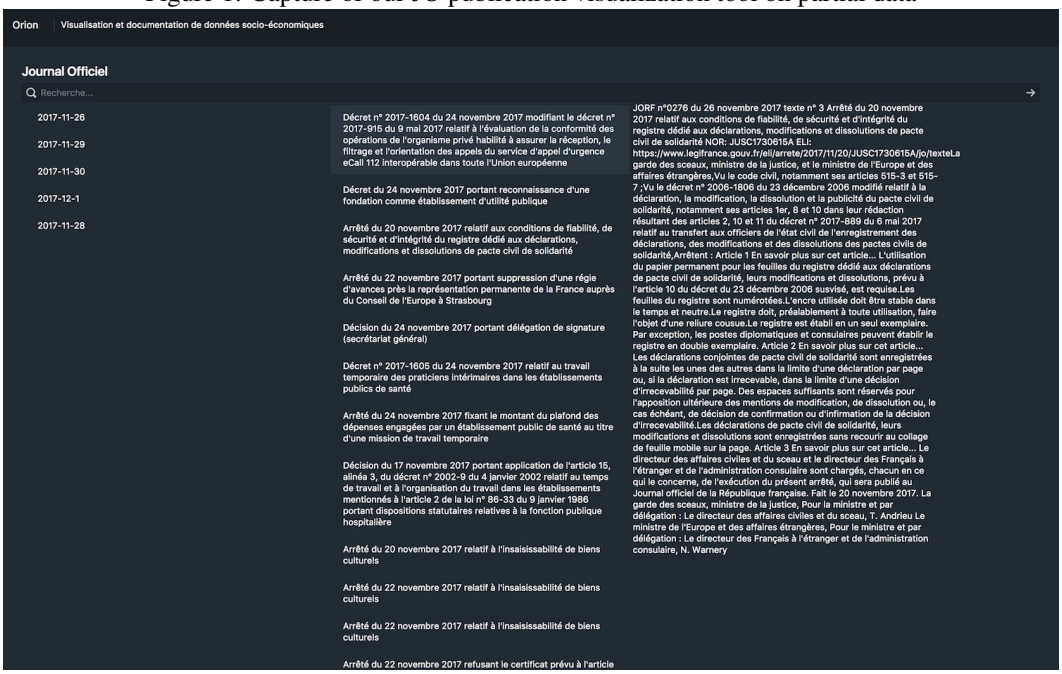
The goal of TF-IDF is to increase the similarity score of words that are characteristic of a given article. The more a word is frequent within an article, the more it is deemed relevant. On the other hand, the more a word is frequent overall, the less important it is to the given article as it will not discriminate one article versus another.

Note: a potential next step could be to explore an Article Embedding distance using a framework such as ParagraphVector, Word2Vec or FastText.

2.3.3 Pre-processing

In order to make the query as efficient and accurate as possible we did some pre-processing on the articles' content: after going through the entire database, we established a list of stopwords not to be included.
A first list of stopwords was extracted from this webpage. We then added other words that are deprived of meaning in this context (overall high frequency, for example words like 'executive order',

Figure 1: Capture of our JO publication visualization tool on partial data



'Mr.' etc.).
A potential next step to such a process would be to tokenize the words and treat words such as 'register' and 'registers' in a similar fashion.

3 Data visualization

As these results are meant to be shared and used (both with an API and a web-based interface), therefore we developed web-based visualization tools as part of an other open-source project so as to display results previously obtained.
In particular, Figure 1 show a more user-friendly interface to navigate through the various publications of the JO as well as their articles. It furthermore allows fast text-search on articles through the entire database.
Figure 2 displays a graph view of the stored data using d3 (which is a javascript library intending at making SVG drawing easier). It was made using only data from the last two weeks: that is to say most nodes are articles published in the last two weeks. The graph was then completed by adding nodes which are referenced by already existing nodes but do not yet exist in the graph. Edges represent the fact that one document makes a reference to the other.

4 Conclusion

The French *Journal Officiel* is a major point of French politics and French people should be more aware of its existence as well as able to examine its contents easier.
This project is a stepping stone in that direction: we established a framework to scrape every publication of the JO as well as parsing it and putting it in an ElasticSearch database.
For the end user, it is now possible to view a user-friendly data visualization of its articles and it is also possible to execute some search queries that return the most relevant publications and articles.
Additionally, we explored two separate ways to connect the articles and form graph edges: the natural one consisted in following the URL references that are mentioned in the articles, and the second one is based on a TF-IDF proximity score.

Figure 2: Capture of the graph visualization tool on partial data