BLEXTRA

A blog extraction application

Report – Web Engineering Mini-Project 2015/2016

Authors: Mojgan Madah
Amanda Karavolia
Maria Schmidt

Contents

1. *Introduction*

The number of blogs these days is growing rapidly. More and more people are expressing themselves in blogs - the stay at home mum, fashion professionals, journalists, sport professionals, IT-Nerds and so on.
The more and more newspaper have an own blog section where journalist of all genres are writing their opinions about different topics.
These blogs could be interesting for a wide range of user. The problem is that the particular search for blog articles is nothing which is supported by normal search engines like Google or Yahoo. For example if I want to read opinions about the Climate Change Conference 2015 I can search texts with Google. But most of them will be actually newspaper articles or knowledge sites like Wikipedia. Also the search inside the blogs is not the best. Say I like an author and want to search if he wrote about the Climate Change Conferene. In the above mentioned blogs I don't have the chance to do so.

In our Web Engineering mini project we addressed this problem and extracted data from blogs. Next to the normal content we also extracted metadata like the author and the time to get more information about the articles.
In a web application a user can search for articles.

We used the blogs of the newspapers "Tagesanzeiger", "Neue Züricher Zeitung" and "Tribune de Geneve". The first two are written in German and the third one is written in French.

This document is a report for our work. It contains the explanation of important terms we are using, and details about the implementation and logic behind our application. What program structure and databases we used, as well an installation guide and a user guide.

The program is handed in as an USB-Key. There are two version of our project on this USB-Key. One version has already around 1500 extracted articles inside the structure. The ontology and index is already build for this articles so that you can run the Web application directly after installing the required libraries listed in 8.1.2.
The second version of our project does not have any extracted articles in them.

2. *Terms*

2.1. *Blog*

A blog is a web page that serves as a publicly accessible personal journal for an individual. Typically updated daily, blogs often reflect the personality or opinions of the author.

2.2. *Article*

An article is a written work published in a print or electronic medium. It may be for the purpose of propagating the news, research results, academic analysis or debate. For us an article is a peace of text which has a title, author, date and belongs to a blog.

2.3. *Metadata*

Metadata describes how and when and by whom a particular set of data was collected, and how the data is formatted. Metadata is essential for understanding information stored in data warehouses and has become increasingly important in XML-based Web applications.

2.4. *Crawling*

A web crawler (also known as a web spider or web robot) is a program or automated script which browses the World Wide Web in a methodical, automated manner. This process is called Web crawling or spidering.

2.5. *Scraping of data*

Data Scraping is the act or process of retrieving data out of data sources for further data processing or data storage

2.6. *Ontology*

In computer science and information science, an ontology is a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse.

2.6.1. *SPARQL*

The Simple Protocol and RDF Query Language (SPARQL) defines a standard query language and data access protocol for use with the Resource Description Framework (RDF) data model. SPARQL works for any data source that can be mapped to RDF.

A SPARQL query comprises, in order:

- *Prefix declarations*, for abbreviating URIs
- *Dataset definition*, stating what RDF graph(s) are being queried
- A *result clause*, identifying what information to return from the query
- The *query pattern*, specifying what to query for in the underlying dataset
- *Query modifiers*, slicing, ordering, and otherwise rearranging query results

```
# prefix declarations
PREFIX foo: <http://example.com/resources/>
...
# dataset definition
FROM ...
# result clause
SELECT ...
# query pattern
WHERE {
   ...
}
# query modifiers
ORDER BY …
```

3. Implementation

3.1. Crawler

The crawler was build using an open source web crawler called Scrapy. Scrapy is the most popular tool for web crawling written in Python. It is simple and powerful, with lots of features and possible extensions. Scrapy uses selectors based on XPath mechanisms to extract data from a web page. Selectors are components that select parts of the source code of web pages and perform data extractions from the HTML source. It also uses Pipelines and once you retrieve the data you can pass them through various pipelines which are basically bunch of functions to modify the data.

The main advantages of Scrapy are the following:

- Requests are scheduled and processed asynchronously. This means that Scrapy doesn't need to wait for a request to be finished and processed, it can send another request or do other things in the meantime. This also means that other requests can keep going even if some request fails or an error happens while handling it. Thus, this enables you to do very fast crawls (sending multiple concurrent requests at the same time, in a fault-tolerant way)

- Scrapy also gives you control over the politeness of the crawl through a few settings. You can do things like setting a download delay between each request, limiting amount of concurrent requests per domain or per IP, and even using an auto-throttling extension that tries to figure out these automatically.

3.2. Data extraction

Once we found the Url of an article we extracted the data. We used Xml Language Path (XPath) to find the data in the HTML documents. XPath is a query language to select nodes from a XML document. It can also be used with HTML documents.

For example in the Tagesanzeiger blogs the format for the date is formatted like this:
* date [...] *

In order to find the date we have to use the following XPath query:
sel.xpath("//div[@class='article-date']/text()").extract()

Some of the data still has to be processed after extracting it. For example the actual content has to be cleaned from HTML formatting tags.

All the extracted metadata was saved in a json-file for further process. At the moment we are extracting the title, author, date and the name of the bloog.
The content was saved in a text file. IDs were used to make a connection between the items in the json-file and the text file.

3.3. Index building

3.3.1. Ontology

The schema of our blog ontology was first build with Protégé which is a free, open source ontology editor and a knowledge acquisition system. It provides a graphic user interface to define ontologies and also includes deductive classifiers to validate that models are consistent and to infer new information based on the analysis of an ontology.

3.3.2. Index

For the text search the content and title of the articles were indexed. For this we used the library Whoosh.
The library takes care about building the index and storing of terms. It can also be used to search the index later. Next to the normal terms we also saved the word

stems and set the character folding option to true. This means characters like á , â, a, ã are all treated the same

3.4. Search

3.4.1. Terms

The search for terms is done with Whoosh. The operators *and*, *or* and *not* can be used to combine different terms.
It was also implemented to just search in the titles or search for the word stems.
Another option is to expand the query with synonym. If this option is true a request to the webiste "https://www.openthesaurus.de/synonyme/" is done to get the synonym for a given word.
At the moment synonym expansion is just supported for the German language.

3.4.2. Metadata

For the search by metadata, we loaded them in triple stores for querying with the SPARQL query language. In the figure below, a SPARQL query is represented:

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX BlogsOntology: <http://www.semanticweb.org/amanda/ontologies/2015/10/BlogsOntology#>
SELECT ?title ?date
WHERE {
        ?a BlogsOntology:hasName ?hasName . FILTER ( hasName="Mariannne"^^xsd:string) .
        ?a BlogsOntology:hasArticles ?b .
        ?b BlogsOntology:hasTitle ?title .
        ?b BlogsOntology:hasDate ?date .
    }
```

Example: All articles for an author

In the graph **<http://www.semanticweb.org/amanda/ontologies/2015/10/BlogsOntology#>**, find all subjects (**?a**) and objects (**?hasname**) that have name "Marianne" and linked with the **BlogsOntology:hasName** predicate.
Secondly, find all subjects (**?a**) and objects (**?b**) linked with the **BlogsOntology:hasArticles** predicate and find all subjects (**?b**) and objects (**?title**) and (**?date**) linked with the **BlogsOntology:hasTitle** and **BlogsOntology:hasDate** predicates respectively. Then, return all the values of **?title** and **?date**.
In other words, find all articles of the author Marianne and return their title and date.
(By using **?b** as an object of one triple and the subject of another, we traverse multiple links in the graph.)

3.5. Web Application

The Web Application was implemented with the Flask framework.
Flask is a lightweight Python web framework based on *Werkzeug* and *Jinja 2*.

If the user calls the start Url a search form is presented. Here the user can type is query and search for all the relevant articles. A more detailed explanation about the search options is presented in the user guide.

4. Program structure

The program consists of two parts. One is the scraping part which does all the crawling and data extraction.
The second one is the Web Application which handles the user requests.

The scraping of data of a blog has to be manually started from the console.
After new data was extracted one has to run the script main.py. This script starts the indexing and building of the ontology and copies all the files for searching from the scraping folder to the web application.
Like this the two parts of the program can work independently.

5. Logic

5.1. Data Extraction

5.1.1. NZZ

On the website www.nzz.ch/meinung/blogs one can find 13 different blogs.
The Url of a specific blog is
http://www.nzz.ch/meinung/blogs/medienblog/?page=1...n

One page like this always holds 5 articles.
Every article has also its own Url but most of the time one has to be logged in to read the article like this. To not care about the logging in, the data was directly extracted from the overview blog pages, because the whole content was also written here.

The NZZ spider consists of two parts. One is a spider which get all the blog Urls from the website www.nzz.ch/meinung/blogs.

The other one goes directly to the Url of one blog and crawls to all the pages there are.

5.1.2. Tagesanzeiger

The Tagesanzeiger crawler starts at the Url 'http://blog.tagesanzeiger.ch/'
It than extracts all the articles which have an Url which matches the following regular expression:

[^http:\/\/blog\.tagesanzeiger\.ch\/((?!replytocom).)$]*

which means basically all Urls of the form http://blog.tagesanzeiger.ch/something but the word "reployтocom" shouldn't be inside, because this leads to the commentary page of this article.

The problem with the articles on the Tagesanzeiger website were that a lot of data was different formatted in different blogs.

5.1.3. Tribune de Genève

The website of the Tribune de Genève is http://blog.tdg.ch/ and so we started crawling from this Url.

Due to the fact that the domain was changing blog name for each blog i.e. http://causetoujours.blog.tdg.ch/ or http://boulevarddelislamisme.blog.tdg.ch/ etc. and all article Urls have the form: http://soyonsnet.blog.tdg.ch/archive/2015/12/14/les-pommes-et-l-arbalete-19-a-suivre-272575.html we created a rule in order to define the exact paths that the crawler is going to follow:

[r'blog\.tdg\.ch\/archive\/\d+\/\d+\/\d+\/.+html$']

This regular expression keeps the .blog.tdg.ch, in order to not the crawler get away from this domain and also it defines the exact syntax of all Urls that are going to be crawled.

As we mentioned before, the format of the data differed from blog to blog and so we have to deal with unstructured data with different formats.

5.2. Ontology Schema

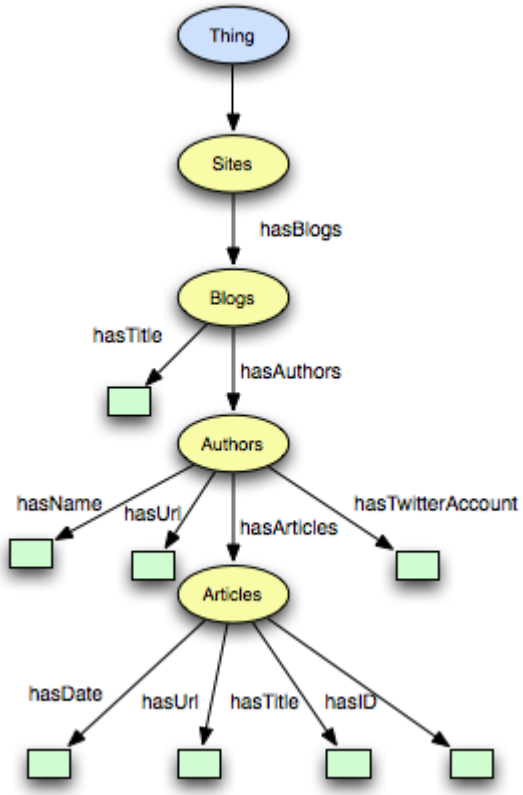
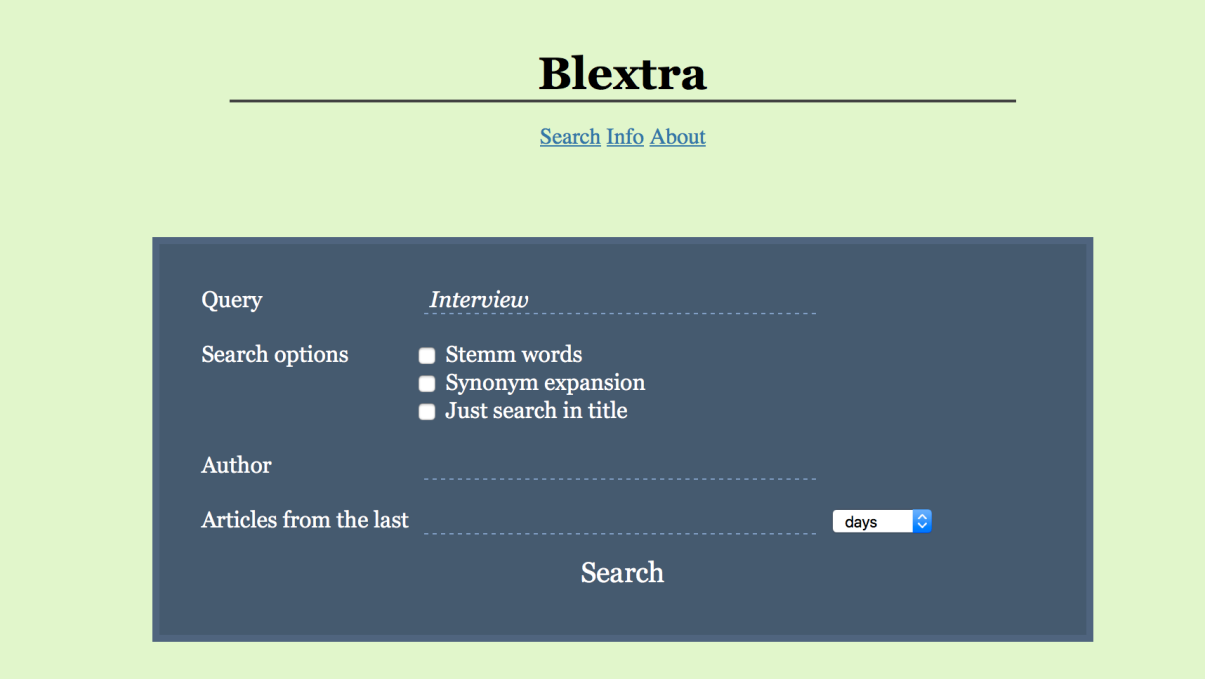


Figure 1: Blog ontology schema

In the above figure the representation of our ontology is displayed. It is shown that a site has many blogs, each blog has an author and each author has many articles. For each class, we have defined some attributes for instance, for class Blogs the *hasTitle* attribute contains the title of the Blog.

Using the *RDFLib* (a Python package work working with RDF) we parsed the ontology schema and then we started creating the individuals (instances) of our blog ontology with the extracted data from json files. Finally we save the ontology of each site (Tagesanzeiger, NZZ, Tribune de Genève) to three different RDF files.

5.3. Search

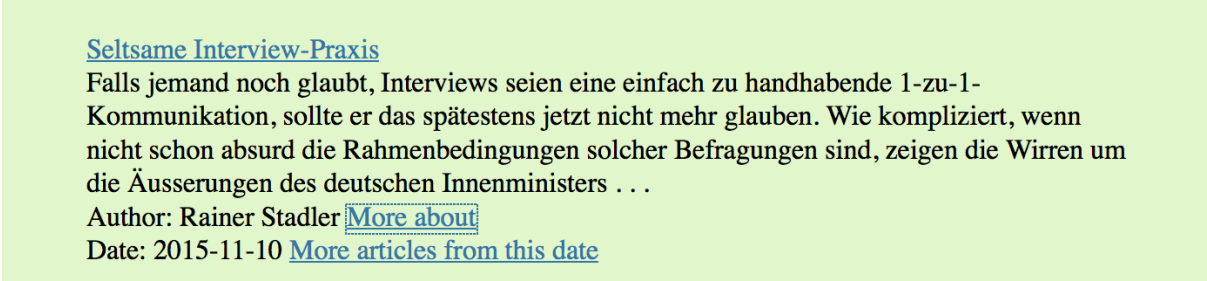


The picture shows how the search forms looks.
At the moment the following searches can be done:

- Search for articles, which are containing specific terms.
- Search for articles from an author.
- Search for articles written in the last X days/weeks/months/years.
- Search for articles, which are containing specific terms written by a specific author.
- Search for articles which are containing specific terms and were written in the last X days/weeks/months.
- Search for articles from an author written in the last X days/weeks/months/years.
- Search for articles containing specific terms, written by an author and in the last X days/weeks/months/years.

In the following it is explained how the program works for the different search queries.

Search for articles which are containing specific terms:
If the user just searches for articles which are containing specific terms, then the text index is searched with the Whoosh library. As a result, the program gets an array with the relevant article-ids sorted by the score. For every article we have a text file which has the article-id as a name. Like this the program can load the corresponding text and presented it to the user.
The program also gets with SPARQL the meta-information for an article from the ontology. The following picture shows how an example of how an article is presented:


Seltsame Interview-Praxis
Falls jemand noch glaubt, Interviews seien eine einfach zu handhabende 1-zu-1-Kommunikation, sollte er das spätestens jetzt nicht mehr glauben. Wie kompliziert, wenn nicht schon absurd die Rahmenbedingungen solcher Befragungen sind, zeigen die Wirren um die Äusserungen des deutschen Innenministers . . .
Author: Rainer Stadler More about
Date: 2015-11-10 More articles from this date

Since the search in the ontology takes still a while at the moment information are just loaded for 5 articles. The user is than presented with a next button to see the next 5 articles. As most of the user are anyway just interested in the top results the user does not have to wait too long.

Search for articles from an author
The program gets from the ontology all the articles-ids for an author. It than loads the corresponding texts and presents the results to the user.

Search for articles written in the last X days/weeks/months/years
The program gets from the ontology all the articles-ids for all articles which were written in the given range. It than loads the corresponding texts and presents the results to the user.

Search for articles which are containing specific terms written by a specific author
The program does a normal text search and gets all article-ids, which are containing the terms. It also makes a search in the ontology to get all the article-ids for one author. Both sets of ids are intersected to get all the ids, which have the specific term inside and are written by this author.

Search for articles which are containing specific terms and were written in the last X days/weeks/months
The program does a normal text search and gets all article-ids, which are containing the terms. It also makes a search in the ontology to get all the article-ids in the date range. Both sets of ids are intersected to get all the ids, which have the specific term inside and are written by this author.

Search for articles from an author written in the last X days/weeks/months/years
Similar to the already explained queries to articles and date, and author and date. Search for articles containing specific terms, written by an author and in the last X days/weeks/months/years.
Article-ids for every part are requested and then intersected.

6. Databases

6.1. Ontology

The ontology is being stored in triple in the N-triples format using the RDFLib. This format is the simplest and slightly faster than the others such as xml, n3, trix.

6.2. Text search

Whoosh stores the terms in an index.

7. Administrator guide

7.1. Installation guide

7.1.1. Requirements

The following packages have to be installed on your machine:

Python 2.7: https://www.python.org/downloads

BeautifulSoup4:
http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-beautiful-soup

Scrapy 1.0.3: http://doc.scrapy.org/en/latest/intro/install.html

Whoosh 2.7.0: https://pypi.python.org/pypi/Whoosh

RdfLib 4.2.2: http://rdflib.readthedocs.org/en/latest/gettingstarted.html

virtualenv: https://virtualenv.readthedocs.org/en/latest/installation.html

7.1.2. Install Flask in a virtual environment

In this project we installed Flask in a virtual environment. The following section explains this installation. It assumes that our Project is put on your machine. Lets call the path to the project "pathToProject".

Open a terminal on your machine.
Type the following commands:

```
cd pathToProject/Project/webApp
virtualenv flask
```

With this a virtual Python environment is created. In this environment we can install all the libraries we need for the Web Application.

Type the following commands in the console:

```
flask/bin/pip install flask
flask/bin/pip install flask-wtf
flask/bin/pip install whoosh
flask/bin/pip install rdflib
flask/bin/pip install flask-whooshalchemy
```

7.1.3. Complete Installation instruction for Ubuntu

Open a terminal and type the following commands.

Commands to get all necessary libraries for the Scraping program:

```
sudo apt-get install python-pip
sudo apt-get install python-twisted
sudo pip install scrapy
sudo pip install beautifulsoup4
sudo pip install whoosh
```

```
sudo pip install rdflib
sudo pip install virtualenv
```

Commands to install the virtual python environment for the Web App.

```
cd pathToProject/Project/webApp
virtualenv flask
flask/bin/pip install flask
flask/bin/pip install flask-wtf
flask/bin/pip install whoosh
flask/bin/pip install rdflib
flask/bin/pip install flask-whooshalchemy
```

7.2. Crawling instruction

The crawling is not automated at the moment. To start a new crawling one has to start the process from the command line. The instructions are listed below.
Every run of a crawler can be done with a option to set the limit auf extracted article.
Type: *scrapy crawl spidername –a limit=number of articles you want to extract (Exmaple: scrapy crawl nzz –a limit=500)*

Open a terminal console. Change directory to the scraping application:

```
cd pathToProject/Project/scraping
```

Crawl for the NZZ blogs:

```
scrapy crawl nzz_blogs -o blogs.csv -t csv
```

Crawl the NZZ blogs

```
scrapy crawl nzz
```

Crawl the Tagesaneziger

```
scrapy crawl tga
```

Crawl Tribune de Geneve

```
scrapy crawl tdg
```

7.3. Building the index

After the crawling is done the index and ontology have to be updated.
If the index was never before initialized the script
create_schema.py under Project/Scraping/scraping/indexing has to be called.

For every other one just has to run the script
main.py
in the root folder. This scripts updates the index and ontology and copies all the text files and index files to the Web application.

7.4. Running webserver

Open a console and type the following commands to start the webserver locally:

```
cd pathToProject/Project/WebApp
./run.py
```

The website can than be opened in a browser with : localhost:5000

8. User guide

8.1. Term search

With Blextra you can do a normal search for terms.
But in the query field the terms you want to search for.
You can conctenate different terms with & (and), | (or) and ! (not).
You can also use parathesis to group your queries.

Example:
(Sommer & Eis) | (Sommer & Bern)

This query would search for all articles which either have the words „Sommer and Eis“ or which have the words „Sommer & Bern“ inside.

8.2. Phrase search

If you put your query in quotation mark you can do a phrase search.

8.3. Searching by author

With Blextra you can search for all the articles of an author. For this put the name oftthe author in the „Author“ field.

8.4. Searching by date

With Blextra you can search for all articles which were written in the last X days/weeks/months/years. Just put the number in the field and set the time-period.

8.5. Combine searching

All this searches can be combined. This means you can search for all articles which have a specific term inside and were written by an specific author or in a specific time-period. Or everything together.

9. Conclusion

9.1. State of the application

To conclude, for this project we created a web crawler using Scrapy and we crawled three blogs (“Tagesanzeiger”, “Neue Züricher Zeitung” and “Tribune de Genève”.). First, we extracted all the relevant data of these blogs. Using these data, we built our blog ontology using Protégé for creating the RDF schema and RDFLib for instances.

Moreover, for the metadata search we used the ontology and for the text search (terms in title and content of the article) we built an index. We also created a synonym expansion only for German language. Finally, a user has the ability to search in the data by using our web application in Flask framework.

9.2. Further development

9.2.1. Expand Ontology

It would be interesting to extend our ontology to use some social networking information like shares of Facebook, Twitter and G+, as well as the use of comments, tags and categories of an article.

9.2.2. Expand Searching

It would be very useful to extend our searching to provide synonyms for French words, translation of queries and finally, searching by using a specific date.

9.2.3. Using of faster databases

Using n-triples format is not the best and the fastest solution and especially if the ontology started getting bigger over the time. Thus, a better solution is to use a scalable RDF database.

9.2.4. Service to crawl regularly

At the moment crawling has to be started manually. To get information regularly (like every day) it is “better” to use a service. When using a service also the synchronization between the Scraping App and the Web App have to be handled different.

9.2.5. Deploying

The whole project is not deployed online yet. After implementing a service and handling the data better (faster) the Webapp can be deployed. Deploying two possible solutions:

- The Webapp and the data files are updated regularly by crawling on the our machine.

- Deploying Webapp, Scraping app and do the web crawling in a Service Cloud like Amazon, which is going to be faster than a local machine. This service can handle multiple threads to do the web crawling which is good and more efficient. Having multiple threads, do the crawling parallel so we have the ability to crawl more blogs and get more information. For instance, one thread per blog could be suitable.

10. Appendix A –Content of USB-Key

10.1. Content of the USB-Key

```
USB-Key
    Project_Karavolia_Schmidt_XXX
        Project_500
            main.py
            Scraping
            WebApp
        Project_Empty
            main.py
            Scraping
            WebApp
```

Scraping: Folder which contains the whole Scraping program.
WebApp: Folder which contains the Web application.

Project_500: A version of our project where already 500 articles per site are crawled. You can start the Web application from here if you don't want to crawl the websites before.

Project_Empty: A version of our project where no article was crawler yet.