Image Search IR System

WS2020-21 Information Retrieval Project

Claudio Maggioni

December 8, 2020

Contents

1 Introduction

This report is a summary of the work I have done to create the "Image Search IR system", a proof-of-concept IR system implementation implementing the "Image Search Engine" project (project #13).

The project is built on a simple *Scrapy-Solr-HTML5+CSS+JS* stack. Installation instructions, an in-depth look to the project components for scraping, indexing, and displaying the results, and finally the user evaluation report, can all be found in the following sections.

2 Installation instructions

2.1 Project repository

The project Git repository is located here: `https://git.maggioni.xyz/maggicl/IRProject`.

2.2 Solr installation

The installation of the project and population of the test collection with the scraped documents is automated by a single script. The script requires you have downloaded *Solr* version 8.6.2. as a ZIP file, i.e. the same *Solr* ZIP we had to download during lab lectures. Should you need to download a copy of the ZIP file, you can find it here: `https://maggioni.xyz/solr-8.6.2.zip`.

Clone the project's git repository and position yourself with a shell on the project's root directory. Then execute this command:

```
./solr_install.sh {ZIP path}
```

where {`ZIP path`} is the path of the ZIP file mentioned earlier. This will install, start, and update *Solr* with the test collection.

2.3 UI installation

In order to start the UI, open with your browser of choice the file `ui/index.html`. In order to use the UI, it is necessary to bypass `Cross Origin Resource Sharing` security checks by downloading and enabling a "CORS everywhere" extension. I suggest this one for Mozilla Firefox and derivatives.

2.4 Run the website scrapers

A prerequisite to run the Flickr crawler is to have a working Scrapy Splash instance listening on port `localhost:8050`. This can be achieved by executing this Docker command, should a Docker installation be available:

```
docker run -p 8050:8050 scrapinghub/scrapy
```

In order to all the website scrapers, run the script `./scrape.sh` with no arguments.

3 Scraping

The chosen three website to be scraped were `flickr.com`, a user-centric image sharing service service aimed at photography amatures and professionals, `123rf.com`, a stock image website, and `shutterstock.com`, another stock image website.

The stock photo websites were scraped with standard scraping technology using plain `scrapy`, while *Flickr* was scraped using browser emulation technology using `scrapy-splash` in order to execute Javascript code and scrape infinite-scroll paginated data.

I would like to point out that in order to save space I scraped only image links, and not the images themselves. Should any content that I scraped be deleted from the services listed above, some results might not be correct as they could have been deleted.

As a final note, since some websites are not so kind in their `robots.txt` file to bots (*Flickr* in particular blocks all bots except Google), "robots.txt compliance" has been turned off for all scrapers and the user agent has been changed to mimick a normal browser.

All scraper implementations and related files are located in the directory `photo_scraper/spiders`.

3.1 Flickr

3.1.1 Simulated browser technology Splash

As mentioned before, the implementation of the *Flickr* scraper uses *Splash*, a browser emulation that supports Javascript execution and simulated user interaction. This component is essential to allow for the website to load correctly and to load as many photos as possible in the photo list pagest scraped through emulation of the user performing an "infinite" scroll down.

Here is the Lua script used by splash to emulate infinite scrolling. These exact contents can be found in file `infinite_scroll.lua`.

```lua
1  function main(splash)
2    local num_scrolls = 20
3    local scroll_delay = 0.8
4
5    local scroll_to = splash:jsfunc("window.scrollTo")
6    local get_body_height = splash:jsfunc(
7        "function() {return document.body.scrollHeight;}"
8    )
9    assert(splash:go(splash.args.url))
10   splash:wait(splash.args.wait)
11
12   for _ = 1, num_scrolls do
13       scroll_to(0, get_body_height())
14       splash:wait(scroll_delay)
15   end
16   return splash:html()
17 end
```

Line 13 contains the instruction that scrolls down one page height. This instruction runs in the loop of lines 12-15, which runs the scroll instruction `num_scrolls` by also waiting `scroll_delay` seconds after every execution.

After this operation is done, the resulting HTML markup is returned and normal crawling tecniques can work on this intermediate result.

3.1.2 Scraper implementation

The Python implementation of the *Flickr* scraper can be found under `flickr.py`.

Sadly *Flickr*, other than a recently posted gallery of images, offers no curated list of image content or categorization that can allow for finding images other than querying for them.

I therefore had to use the *Flickr* search engine to query for some common words (including the list of the 100 most common english verbs). Then, each search result page is fed through *Splash* and the resulting markup is searched for image links. Each link is opened to scrape the image link and its metadata.

3.2 Implementation for 123rf and Shutterstock

The *123rf* and *Shutterstock* website do not require the use of *Splash* to be scraped and, as stock image websites, offer several precompiled catalogs of images that can be easily scraped. The crawler implementations, that can respectively be found in `stock123rf.py` and `shutterstock.py` are pretty straightfoward, and navigate from the list of categories, to each category's photo list, and then to the individual photo page to scrape the image link and metadata.

4 Indexing and Solr configuration

Solr configuration was probably the trickiest part of this project. I am not an expert of Solr XML configuration quirks, and I am certainly have not become one by implementng this project. However, I managed to assemble a configuration that has both a tailored collection schema defined as XML and a custom Solr controller to handle result clustering.

Configuration files for Solr can be found under the directory `solr_config` this directory is symlinked by the `solr_install.sh` installation script to appear as a folder named `server/solr/photo` in the `solr` folder containing the Solr installation. Therefore, the entire directory corresponds to the configuration and data storage for the collection `photo`, the only collection present in this project.

Please note that the `solr_config/data` folder is ignored by Git and thus not present in a freshly cloned repository: this is done to preserve only the configuration files, and not the somewhat temporary collection data. The collection data is uploaded every time `solr_install.sh` is used from CSV files located in the `scraped` folder and produced by Scrapy.

The configuration was derived from the `techproducts` Solr example by changing the collection schema and removing any non-needed controller.

4.1 Solr schema

As some minor edits were made using Solr's web interface, the relevant XML schema to analyse is the file `solr_config/conf/managed-schema`. This files also stores the edits done through the UI. An extract of the relevant lines is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Solr managed schema - automatically generated - DO NOT EDIT -->
```

```xml
<schema name="example" version="1.6">
  <uniqueKey>id</uniqueKey>

  <!-- Omitted field type definitions and default fields added by Solr -->

  <field name="id" type="string" multiValued="false" indexed="true"
         required="true" stored="true"/>
  <field name="date" type="text_general" indexed="true" stored="true"/>
  <field name="img_url" type="text_general" indexed="true" stored="true"/>
  <field name="t_author" type="text_general" indexed="true" stored="true"/>
  <field name="t_description" type="text_general" indexed="true"
         stored="true"/>
  <field name="t_title" type="text_general" indexed="true" stored="true"/>
  <field name="tags" type="text_general" indexed="true" stored="true"/>

  <field name="text" type="text_general" uninvertible="true" multiValued="true"
         indexed="true" stored="true"/>

  <!-- Omitted unused default dynamicField fields added by Solr -->

  <copyField source="t_*" dest="text"/>
</schema>
```

All fields have type `text-general`. Fields with a name starting by "`t_`" are included in the `text` copy field, which is used as the default field for document similarity when searching an clustering.

The `id` field is of type `string`, but in actuality it is always a positive integer. This field's values do not come from data scraped from the website, but it is computed as a auto-incremented progressive identified when uploading the collection on solr using `solr_install.sh`. Shown below is the `awk`-based piped command included in the installation script that performs this task and uploads the collection.

```bash
1   # at this point in the script, `pwd` is the repository root directory
2
3   cd scraped
4
5   # POST scraped data
6   tail -q -n +2 photos.csv 123rf.csv shutterstock.csv | \
7           awk "{print NR-1 ',' \$0}" | \
8           awk 'BEGIN {print "id,t_author,t_title,t_description,date,img_url,tags"}
9                {print}' | \
10          ../solr/bin/post -c photo -type text/csv -out yes -d
```

Line 6 strips the heading line of the listed CSV files and concatenates them; Line 7 adds "{id}," at the beginning of each line, where {id} corresponds to the line number. Line 8 and 9 finally add the correct CSV heading, including the "id" field. Line 10 reads the processed data and posts it to Solr.

4.2 Clustering configuration

Clustering configuration was performed by using the `solrconfig.xml` file from the `techproducts` Solr example and adapting it to the "photo" collection schema.

Here is the XML configuration relevant to the clustering controller. It can be found at approximately line 900 of the `solrconfig.xml` file:

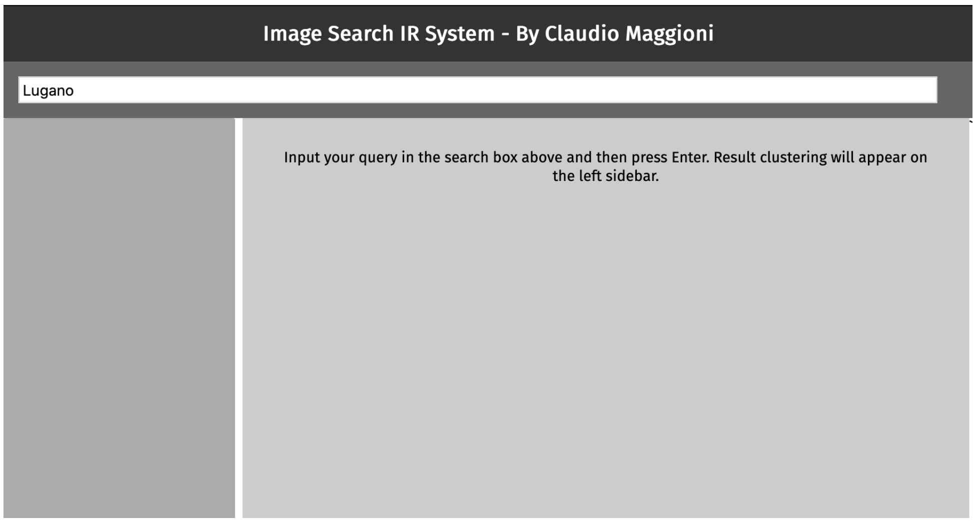
```
 1 <requestHandler name="/clustering"
 2                 startup="lazy"
 3                 enable="true"
 4                 class="solr.SearchHandler">
 5   <lst name="defaults">
 6     <bool name="clustering">true</bool>
 7     <bool name="clustering.results">true</bool>
 8     <!-- Field name with the logical "title" of a each document (optional) -->
 9     <str name="carrot.title">t_title</str>
10     <!-- Field name with the logical "URL" of a each document (optional) -->
11     <str name="carrot.url">img_url</str>
12     <!-- Field name with the logical "content" of a each document (optional) -->
13     <str name="carrot.snippet">t_description</str>
14     <!-- Apply highlighter to the title/ content and use this for clustering. -->
15     <bool name="carrot.produceSummary">true</bool>
16     <!-- the maximum number of labels per cluster -->
17     <!--<int name="carrot.numDescriptions">5</int>-->
18     <!-- produce sub clusters -->
19     <bool name="carrot.outputSubClusters">false</bool>
20
21     <!-- Configure the remaining request handler parameters. -->
22     <str name="defType">edismax</str>
23     <str name="df">text</str>
24     <str name="q.alt">*:*</str>
25     <str name="rows">100</str>
26     <str name="fl">*,score</str>
27   </lst>
28   <arr name="last-components">
29     <str>clustering</str>
30   </arr>
31 </requestHandler>
```

This clustering controller uses Carrot2 technology to perform "shallow" one level clustering (Line 19 disables sub-clusters). `t_title` is used as the "title" field for each document, `img_url` as the "document location" field and `t_description` the "description" field (See respectively lines 9, 11, and 13 of the configuration).

This controller replaces the normal `/select` controller, and thus one single request will generate search results and clustering data. Defaults for search are a 100 results limit and the use of `t_*` fields to match documents (lines 25 and 23 – remember the definition of the `text` field).

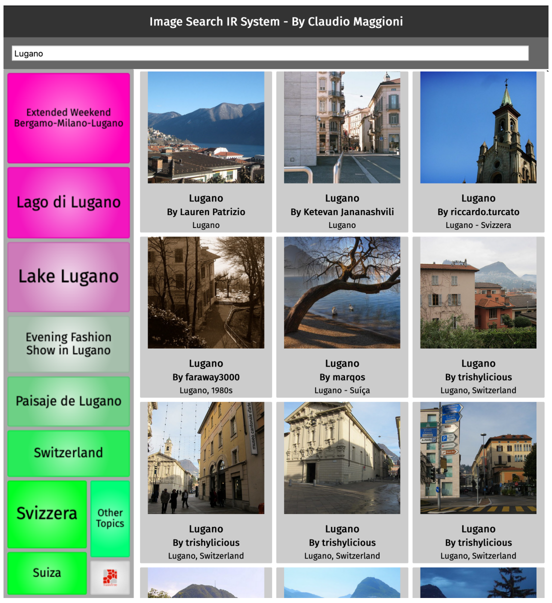
5 User interface

Figure 1 illustrates the IR system's UI showing its features.



Input your query in the search box above and then press Enter. Result clustering will appear on the left sidebar.

(a) The UI, when opened, prompts to insert a query in the input field and press Enter. Here the user typed "Lugano".



(b) After the user inputs a query and presses Enter, resulting images are shown on the right. Found clusters are shown on the left using FoamTree.

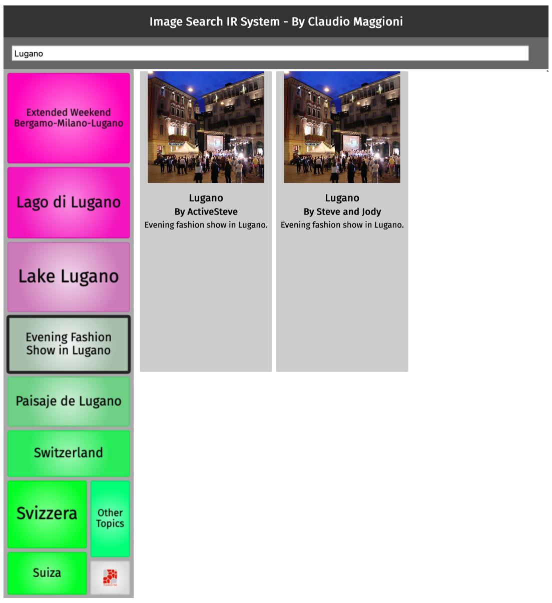


Image Search IR System - By Claudio Maggioni
Lugano
Extended Weekend Bergamo-Milano-Lugano
Lago di Lugano
Lake Lugano
Evening Fashion Show in Lugano
Paisaje de Lugano
Switzerland
Svizzera
Other Topics
Suiza
Lugano
By ActiveSteve
Evening fashion show in Lugano.
Lugano
By Steve and Jody
Evening fashion show in Lugano.

(c) When a user clicks a cluster, results are filtered depending on the cluster clicked. If the user clicks again on the cluster, filtering is removed.

Figure 1: The UI and its various states.

The UI has been implemented using HTML5, vanilla CSS and vanilla JS, with the exception of the *FoamTree* library from the Carrot2 project to handle displaying the clustering bar to the left of search results.

Event handlers offered by *FoamTree* allowed for the implementation of the results filtering feature when a clustering in filtered.

This is a single page application, i.e. all updates to the UI happen without making the page refresh. This was achieved by using AJAX requests to interact with Solr.

All UI files can be found under the `ui` directory in the repository root directory. In order to run the UI, a “CORS Everywhere” extension must be installed on the viewing browser. See the installation instructions for details.

6 User evaluation