Web Information Retrieval
Project 2

Analyzer for Google Scholar – Continued

Manuel Hotz, Philipp Meschenmoser

February 2016

1 Project Motivation

Google Scholar is a large database of scientific documents, however it has no (public) API for the programmatical access of its information. In the age of open data and access initiatives the platform feels rather restricted and is only a specialized search engine, rather than a scientific platform.

The overall goal of our second project is twofold. First, we want to analyze geographical relations between authors and their institutes. Second, we want to directly compare a list of authors that are not necessarily in our database yet. This functionality is especially useful in "Berufungskommissionen", where a quick look onto fair and helpful metrics can be used to decide whether to consider a researcher for an open position or not.

We felt like our first project provided the needed basis for further development in this direction.

2 Conceptual Approach

The approach of our second project is still the same as in the first project, although the feature set was expanded quite a bit.

Scraping of data can still happend on multiple machines ("dockerization" is also possible) and is now, through a service daemon, also accessible through HTTP.

So our architecture is now comprised of four parts: the spiders, the spider daemon, the web application and the database server.

Spiders. The spiders define how a particular page is crawled and what is scraped. The whole scrapy project can be deployed to the spider daemon.

Spider daemon. The spider daemon lists available projects and spiders and is able to schedule spider runs. Its access method is HTTP, so it is especially suited for a web-based access.

Web application. The web application now has two new functionalities: geographic analysis of researchers' organizations and multi-search to scrape information of multiple researchers at once and to compare them side-by-side.

Data gathering. We were able to expand our set of authors significantly. This became possible by explicitly crawling all author ID's belonging to organisation ID's, which we gathered from other author profiles. Meaningful information with respect to organisations became available to us, and so, we thought about ways of analysing Google Scholar on a more aggregated level. Our idea was to develop and apply a methodology to locate our 3000+ organisations on a world map, to visually encode aggregated measures and to show relations between organisations in a geospatial context.

Geo coding. So, at the very beginning we did the geocoding: organisation ID's were given, and values for longitude, latitude and even the address for the corresponding location were desired. We created a small pipeline for solving this task:

1. Preprocessing: Gather 'official' organisation names. It seems that one and the same organisation can be referenced by different anchor texts on corresponding author profiles, i.e.: authors can use different descriptions, writings, languages to name their organisations, typos can also be included. To solve this issue, we programmed a spider, which does the common organisation search (`/citations?view_op=view_org&hl=de&org=16008520586621520646`) and saves the content of the search results' heading (→ official name).

2. First Geocoding : implicit usage of Google Places API. By following this method, we were able to find locations for about 90% of our organisations. Our strategy is based on a Scrapy Spider which crawls the source code of the common Google Maps web interface . The spider's input was the url `www.google.de/maps/place/` followed by an official organisation name we scraped in the first step. In the returned HTML header, we found the tag
   `<meta content="http://...." itemprop="image" property="og:image"/>`.

   The content attribute refers to a link accessing the Google Maps Static Maps API. There, the geographical center gets specified, so we received our longitude and latitude values. The meta tag with the property 'og:description' gave us corresponding addresses. This method worked efficiently and led to very good results, however a few locations were not found: those are identified by a placeholder value in the og:description field.

3. We applied a second geocoding step on organisations we could not find with the strategy, which was developed in step 2. The second strategy refers to an explicit use of the Google Places API. We wrote a node.js script, which reads relevant organisation names from our database and sends those to the Google Places API

via a node module named 'google-locations'. Then, we receive data (lon,lat,addr-much more is possible) with respect to the first suggestion (c.f. autocomplete search on Google Maps). This strategy provides good results, but it is less efficient and the API'S explicit usage is quite restrictive: as a maximum in 24 hours, there are 2100 API calls allowed, combined with another limitation on second level. With this approach, we were able to geocode another 5% of organisations.

Geoanalytical applications. As soon as we had finished geocoding, we developed two geoanalytical applications, which are web-based. Both tools are attached as stand-alone, javascript-only projects, which can be launched via node.js. Simply install node.js and modules in 'backend/node_modules' via 'npm install', adapt database settings in 'backend/config.js' and run 'node backend/main.js' in your command line, followed by opening frontend/index.html in your browser. However, we also integrated the applications into our Flask[1] app and rewrote the tools' backends in Python.
Our first tool 'Geoanalytics - Measures' aggregates measures on organisation level: #counts, avg + sum of h5- and i5-indexes, next to citations during the last 5 years. Those values get mapped onto a Mercator projection via color encoding of transparent circles. The user can vary the circles' radii. This possibility solves overlap issues and provides an interesting combination of geometric and semantic zooming. With respect to the mentioned measures, it is obvious that we only make use of 'current' measurements. Long-term measurements would not reflect the researchers' current location (i.e. circle position) properly ("it is unlikely that an author stays at the same university for 30 years...").

We implemented our first tool using the Google Maps Javascript API, d3.js is used as an overlay to display the circles on SVG elements. There is an details-on-demand approach implemented, which can be also seen as a part of information search and retrieval. When the user performs a right click, we send the corresponding longitude, latitude, zoom-level value and the selected measure to our backend. We determine the organisations which lie within a zoom-dependent distance to the click coordinates and return the top five organisations (name and measurement value).
Our second tool 'Geoanalytics - Coauthors' connects organisations via geodesic polylines, deduced of coauthor information given in single author profiles. Before any polylines are drawn to the map, the user needs to enter the name of the relevant organisation into an autocomplete field. We implemented this field by using jQuery Autocomplete, where AJAX calls build the basis for the communication with our database. In the autocomplete's functionality, there is also a cache implemented via a simple Javascript Object: `{term1: [{name:xyz, id:123}, ...,], term2: [...]}` . In general, our second tool became very efficient by preloading and caching data. Before showing the map at the very beginning, we request and transform data in a way that an object like the following results: `{orgid1: {name:..., lat:..., lon...},..., orgid2800:...}`. When the user wants to add connections for a new organisation, we simply need to return ID's for 'target organisations' from our database, lookup coordinates in the object above, build and display a google.maps.polyline. In addition, we implemented another cache

where organisation ID's are referencing arrays of google.maps.polylines.

Click handlers on very thin polylines are almost never fired. Because of this, we programmed an extra details-on-demand approach: We use precalculated geodesic geometries in our database to identify the target organisation, whose geodesic path is the closest to the click coordinates. We return the ID and the count of coauthor connections.

The request flow for the second application is shown in Figure 1.

3 Implementation

As in the first project, we still rely on Python 2.7 as its library support fit our need better and we did not see any benefits of using Python 3.

Web crawler. For the web crawler we still use the *scrapy* framework[2]. Some spiders were modified and we added a useful spider:

author_complete Receives an ID of an author (`start_authors`) whose profile page is scraped, including co-authors with link depth 2 (configurable by `settings.py`).

org_name Gathers all official organisation names, as described in our conceptual approach.

org_detail Completes organisation items created by the spider above, i.e. does the initial geocoding step and adds values for longitude, latitude and address.

The typical scraping workflow from the first project still exists. However via the spider service each spider can be schedule from an HTTP endpoint.

Web application. Extensive readme's on how to install the tools and the dependencies, and how to run them are provided in the source root directory and in the scrapers subdirectory. Additionally, the project is hosted in a public GitHub repository[1].

The applications new features includes:

Geographic Scraping and Analysis Exploration of authors' organizations and their measures as already described in the above section.

Refresh Author Profile Refresh the data of an author from the profile page.

Scrape Multiple Authors Issue a multi-search and scrape multiple authors at once.

[1] `https://github.com/enplotz/webir2015`

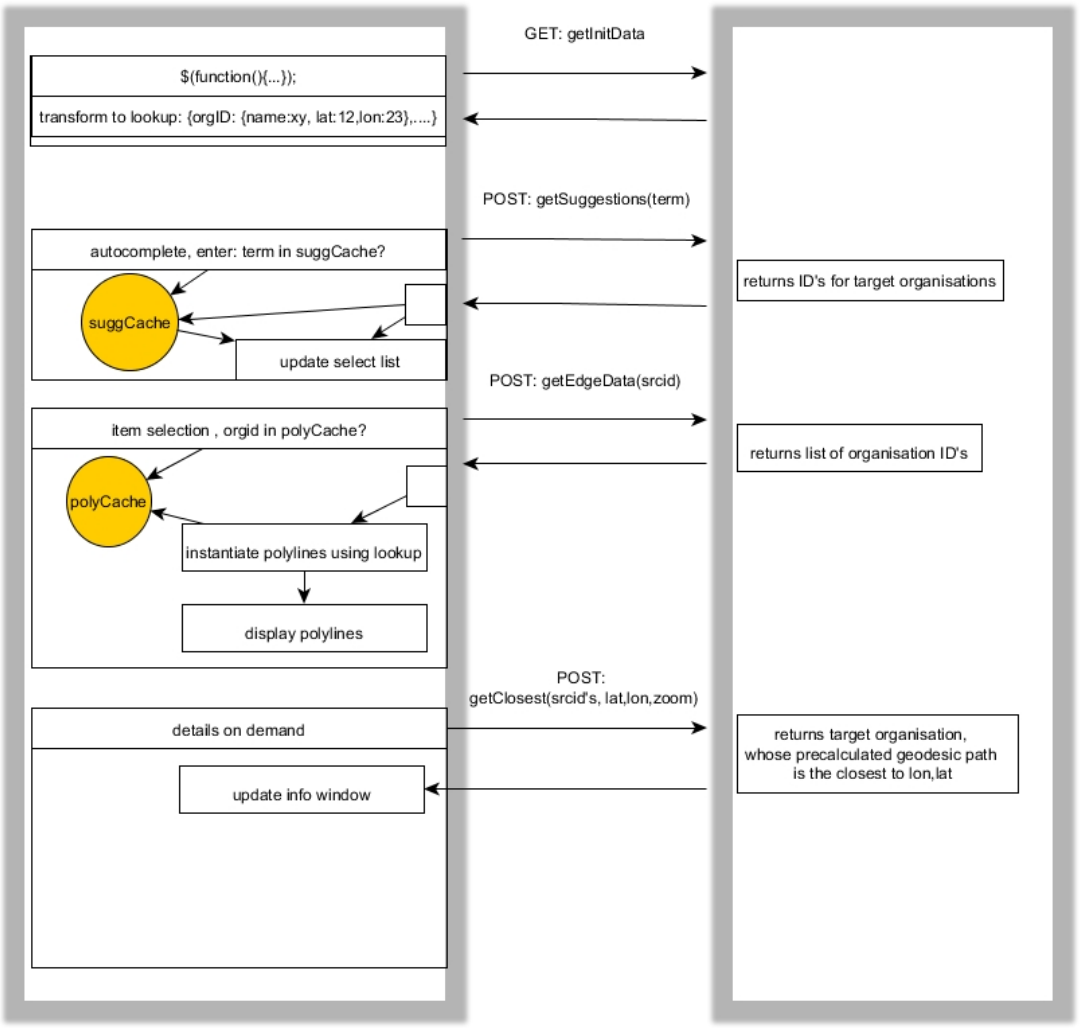

GET: getInitData
$(function(){...});
transform to lookup: {orgID: {name:xy, lat:12,lon:23},....}
POST: getSuggestions(term)
autocomplete, enter: term in suggCache?
suggCache
update select list
returns ID's for target organisations
POST: getEdgeData(srcid)
item selection , orgid in polyCache?
polyCache
instantiate polylines using lookup
display polylines
returns list of organisation ID's
POST:
getClosest(srcid's, lat,lon,zoom)
details on demand
update info window
returns target organisation,
whose precalculated geodesic path
is the closest to lon,lat

Figure 1: Front-end request flow with caching.

4 Biggest Challenges & Difficulties

4.1 Interactive Scraping

One big challenge was how to make the scraping process interactive. The conceptual approach was quite clear at the start: provide a list of authors (this is an easy to understand “interface”) that should be scraped and display scraping results to the user that compare the given authors.

To divide the task into more manageable pieces, we first implemented a simple author profile refresh button: the user visits an author’s profile page, sees that it is outdated (via the displayed timestamp) and can issue a refresh. In the background a scraping task for the authors profile (and co-authors) should automatically be issued.

Because scrapy is based on Twisted[2] we could not simply integrate it into our Flask[1] web application. We found out, that scrapy has a more or less actively maintained service daemon: *scrapyd*[3]. This daemon lets us schedule spiders via its HTTP API. The integration into our web application provided a challenge to us, because one project member had no experience using JavaScript which we wanted to use for the refresh button functionality. A JavaScript-based approach would communicate the scraping process to the user nicely.

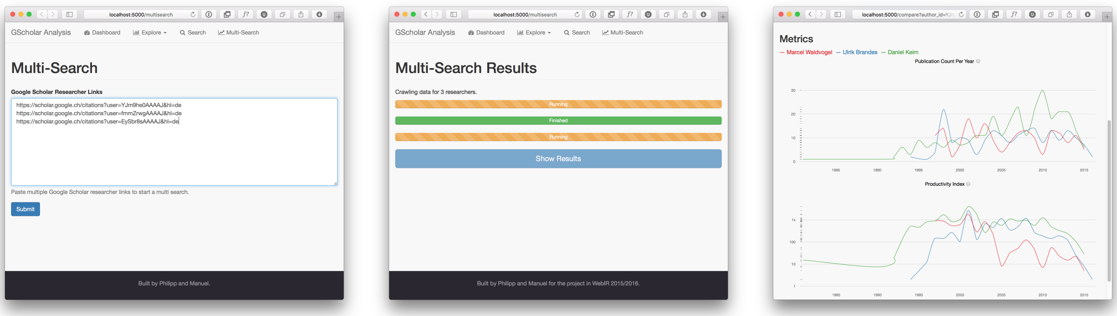
Once we had implemented the refresh button, we proceeded to build the multi-author search. In that we had to schedule the scraping via the Flask app, not by the front-end JavaScript. However, it displays the scraping process to the user by using JavaScript (it actually polls the service daemon and updates the UI).

The next challenge was to display scraped information. For this we took two measures about the authors and combined them in two small timeseries visualizations. This is examplary as a full-featured analytics application would have been out of the scope of the project. The process of comparing multiple researchers is shown in Figure 2.

4.2 Geoanalytical Scraping and Analysis

Probably the most difficult part of our geoanalytical approach was to develop methods, which yield good geocoding results for a lot of organisations. As the disambiguation of organisation names was done, we did a small brainstorming regarding possible approaches to find appropiate coordinates. Initial ideas referred to pattern matching, wikipedia as data source, locating mail servers, etc. . Finally, the Google Places API came into our mind. However, we observed that the explicit usage of the Google Places API via javascript is too restrictive and would not scale very well. We had to find another way and achieved our goal, as we explored the Html content of the common Google Maps web interface. Even though it was not trivial to find and develop our geocoding strategy, we are very happy with our results’ quality and the pipeline’s efficiency.
Difficulties which followed might refer to visual analytics. We had to explicitly decide which visualization methods are the most appropiate ones for our data. For the geospatial

[2]`https://twistedmatrix.com/trac/`
[3]`https://github.com/scrapy/scrapyd`



Multi-Search

Google Scholar Researcher Links

```
https://scholar.google.ch/citations?user=YJm9he0AAAAJ&hl=de
https://scholar.google.ch/citations?user=fmmZrwgAAAAJ&hl=de
https://scholar.google.ch/citations?user=EySbr8sAAAAJ&hl=de
```

Paste multiple Google Scholar researcher links to start a multi search.

Built by Philipp and Manuel.

Multi-Search Results

Crawling data for 3 researchers.

Built by Philipp and Manuel for the project in WebIR 2015/2016.

(a) Search form (b) Scraping progress (c) Visual comparison

Figure 2: Multi-Search. The web application can scrape and compare multiple researchers at once. The progress is continually shown to the user.

visualisation encoding scientific measurements, we tried to apply a heatmap but we recognized that the data points' distribution was not appropiate for this technique, as compared to our current mixture of a dot and symbol map. However, mapping around 3000 circles on a world map, we had to handle issues regarding overlap, which were mostly solved by radius variation and transparency. With respect to our second application, our details-on-demand approach was not trivial to implement, as we had to handle zoom-dependent distances and precalculations of geodesic paths.

5 Results

With regards to the goals we set us for the second project, we implemented two of them (due to constrained time).

Web-based scraping We successfully implemented scraping from our web app and comparison of multiple authors.

Geoanalytics We implemented geolocation and geoanalitcal comparison of author's organizations.

6 `from __future__ import ?`

Due to constrained time, the second project could not solve some tasks we wanted to adress. Specifically and improved temporal and correlation analysis, where e.g. an authors dissertation is automatically located and marked. This way one could compare the scientific work of authors and quickly gauge whether a reasearcher is currently active or not. Also, the indexing of authors without a profile on Google Scholar would be interesting and tie probably into another project based around author deduplication.

Another interesting point to continue would be to expand the two geoanalytical applications by providing the possibility to apply a fields-of-study filter. In addition to this, the coauthor application could be improved by edge bundling, for instance on a state or hierarchical level.

7 Individual Contributions

Philipp Spider Implementation, Page Analysis, (Geo-)Visualizations, JS, JS-API, Basic Scraping Strategy

Manuel Spider Implementation, Porting Attempts of Py3 libraries, Web App, Database Setup, Spider Daemon, Multi-Search/Scraping

References

[1] Armin Ronacher. Flask: Python microframework. `http://flask.pocoo.org`. Accessed: 2015-12-14.

[2] Scrapy Contributors. Scrapy: Open source scraping framework. `http://scrapy.org`. Accessed: 2015-12-14.