Web Information Retrieval
Project 1

Analyzer for Google Scholar

Manuel Hotz, Philipp Meschenmoser

December 2015

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 project is to be able to evaluate the current scientific performance of a researcher using a handfull of meaningfull metrics. A researcher should be compared with similar researchers in the same field or similar fields and over time, considering past publications. This information 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 the project description sounded like a good proportion of information gathering, cleaning and visualization and fit our research interests well.

2 Conceptual Approach

We implemented a scraper and a web application. Both tools are based on the Python programming language and use existing libraries where possible. The overall process is to scrape information using the provided spiders of the scraper, store it in a database, and then display and analyze the data using the web application.

Scraping can happen on different machines than where the web application is hosted, which enables distributed crawls and helps when requests are blocked, as the traffic coming from a single IP can be reduced.

As interesting the process of scraping the data is, the scraped information is not an end in itself. It should enable the user to compare researchers to their fields or examine temporal trends in publications and citations.

For this, the web application currently can compare a researcher to the "average" researcher in any given field. For reasons of simplicity and to keep the project manageable,

we resorted to a normal average of the values of the six metrics we collected (citation count, h-Index, i10-Index, and the three values since 2010).

The dashboard shows what can be achieved using temporal data – an aspect that we want to delve more into and apply to specific author profiles and fields of research, as the temporal activity of a researcher can be a key factor in deciding about the qualification of a candidate.

The node-link visualization on researcher profile pages displays, how the data can be explored in the case of available co-authorship information. Using this interface, a more broad exploration of the data is facilitated.

3 Implementation

As noted earlier, we used Python, more specifically version 2.7 due to the difference in available libraries compared to Python 3.5. We spent two days porting over a couple of libraries to Python 3.5 but decided against further engagement, because the return of our investment was uncertain and we were not sure about the benefits of a Python 3.5 implementation anyway.

Web Crawler For the web crawler we used the *scrapy* framework[1], which is an open source scraping framework. It provides scraping capabilities via its *spider* class[1]. Scrapy schedules requests and passes responses to the running spiders. The spiders then extract *Items*[2] or more requests from the page. Items are the container that scrapy uses to process and pass information. They can be filled directly or via *ItemLoaders*[3], that take *XPath*- or *CSS*-path specifications and extract relevant data. The extracted items are then routed through the item pipeline[4], where further data cleaning can happen. Lastly, the items can be exported to various storage backends. However, as there was no SQL-storage backend readily available, we used a pipeline step to store the items into the PostgreSQL database. Our project includes a couple of spiders. The following listing explains what the most important ones do.

author_labels Searches for the names in `SEED_NAME_LIST` (see settings.py) and scrapes the labels from the list of search results

author_general Searches for all labels in the database and scrapes general author information

author_detail Complements existing author information by requesting the profile page of specific authors

author_co Scrapes co-authorship information of specified authors

[1] `http://doc.scrapy.org/en/1.0/topics/spiders.html`
[2] `http://doc.scrapy.org/en/1.0/topics/items.html`
[3] `http://doc.scrapy.org/en/1.0/topics/loaders.html`
[4] `http://doc.scrapy.org/en/1.0/topics/item-pipeline.html`

A typical scraping workflow using the above spiders would be, to first scrape label information using the popular names, then getting authors for these labels and finally augmenting general author information by detail information regarding scientific measurements or co-authorship.

Web Application The web application is implemented using the *Flask* web framework[2]. It gave us the neccessary freedom to start right away with the prototype, without drowning us in the "convention over configuration" sea. This way, we were able to add additional functionality and libraries along the way, right when they became neccessary. The web application provides a couple of endpoints both to the user and to the Javascript code we wrote. Through the magic of *SQLAlchemy*, it automagically maps the schemas of our database tables to Python objects in the webapp. This way, we are completely decoupled from the crawler models (in the sense of implementation, not schema). We could even specify which tables we want to map and which we do not. Additionally, the application uses a common CSS framework for a visually pleasing presentation and good UI.

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[5].

4 Biggest Challenges & Difficulties

The first big challenge was how to map the unstructured page sources to our structured schemas. We want to analyze performance metrics and therefore need structured data. So we had to figure out, which parts of the Google Scholar website identify a resource in the system – the most valuable being unique identifiers. By analyzing *URL parameters* and the *HTML source code* we could identify unique identifiers as string values, sometimes concatenated by a colon. Relatively late in the project, when we had a notable number of documents crawled, we found out, that what we thought was the unique identifier, was in fact not.

We decided, for ease of simplicity, to scrape only authors that have a profile set up in Google Scholar. This way, we get relatively easy access to their fields of study, published documents and affiliations.

The next challenge hit us early on in the development process. Eight minutes in in the first run of scraping data, our IP address got blocked. We then switched to using Tor as a proxy. We implemented an automatic block detection (basically look for specific HTTP status codes and response headers). However even after getting a new IP, disabling cookie support and waiting 30 seconds we still got blocked. A restart of the crawler was then the only resolve to the problem. Later in the project the blocks got less frequent and were no big problem any more. Yet we would still like to find out, why we got blocked, being the only similarity of our new request to the blocked one, a repeated request to the same URL. To solve the captcha that we got presented with was outside the scope of our project.

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

5 Results

As the focus of the project was laid equally well on the scraping and the analysis side, we did not produce a high amount of visualizations or groundbreaking insights. We rather produced a strong basis for further analysis, both in data gathering and in data visualization. The tool can now scrape with a steady amount of about 700 items per minute (in a couple of overseen runs) without being blocked. To date, we have indexed over 170,000 documents, nearly 80,000 authors with approximately 5,000 of them having detailed information available, as well as nearly 100,000 labels.

6 `from __future__ import ?`

In the second project we would like to work on improving our existing applications. Specifically, we would like to implement the following features.

Full REST API For better access to the data, we would like to implement a fully rest-ful API. This way, the information is not locked in again and can easily be reused in other projects.

Web-based scraping While scraping Google Scholar and implementing the web application simultanteously, we noticed that we had many more authors with missing data, than with crawled details. Often we wanted to test something with the currently open author. With a web app-based crawling request, the data could not only be requested for the first time, but also for repeated indexing, so to refresh the list of publications. Then a user would see when the last refresh of the data happened and issue a refresh. We briefly looked into the current state of web-based activated scraping and found two promising projects. However one was not working anymore and the second one was too complicated to implement quickly.

Improved temporal and correlation analysis We also want to improve the analytical capabilities of our tool. We think we have a good technical basis for an analytical tool, that can deliver on the promise of "comparable research" to quickly gauge the impact of a researcher in comparison to the research field.

Co-Authorship Analysis As there exists a prototype of a co-authorship visualization, we would like to further develop it. This is strongly related to the next point, as we would infer the co-authorship based on the list of publications.

Indexing of authors without profiles As an additional analysis feature, we would like to introduce indexing of authors, that are not registered in Google Scholar. This way, also new and unknown researchers can be looked up. The task provides a certain amount of challenges, as person disambiguation has to be done. As there can be more than one

researcher publishing under a given name, we need to figure out, how to identify them, based on their co-authorship information or relatedness of fields.

7 Individual Contributions

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

Manuel Spider Implementation, Porting Attempts of Py3 libraries, Web App, Database Setup

References

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

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