DATS Program Director

Building An Intelligent Web-Scraping Model For Individual-Level Scholarly Information

Ruchika Singh, M.S.E. Data Science, University of Pennsylvania
Abhinav Bandaru, M.S.E. Data Science, University of Pennsylvania
Mikaela Spaventa, M.S.E. Data Science, University of Pennsylvania
Rut Vyas, M.S.E. Data Science, University of Pennsylvania
Samuel Thudium, M.S.E. Data Science, University of Pennsylvania
Advised by Dr. Sharath Guntuku, University of Pennsylvania

The process of reviewing literature can be painstaking and time consuming. Any researcher, whether a trainee, professor, or industry professional, may have to spend significant time sifting through and reading relevant papers in order to stay up to date with the latest developments in their field. This is compounded by the myriad of databases which hold scholarly texts; discovery of additional information about an author of interest can be difficult to find. In collaboration with the Institute of Electrical and Electronics Engineers (IEEE) we aim to improve upon this process by consolidating individual-level professional information about authors of interest.

I. INTRODUCTION

Under the guidance of a data science team at IEEE, we set out this semester to build an intelligent web scraping model capable of identifying “scholarly entities” of interest using seed information related to an author. For our purposes, scholarly entities refers to active web pages online that contain data or information directly related to an author’s professional work; examples include professional profiles like Google Scholar and LinkedIn, news articles about a component of the author’s work, and GitHub repositories that store code for or build off of an author’s research, among others.

Seed information is defined as data gathered from the IEEE data lake that provide context about a given author; examples include the author’s name, their most recent affiliation, and the names of their publications.

To this end, we have developed a python package, `scholarscraper`. This package is an API allowing access to an end-to-end pipeline that efficiently combs the web for links relevant to authors of interest. We designed this innovative tool from scratch to utilize machine learning (ML) and advanced natural language processing (NLP) techniques for scraping, sorting, and classifying the collected data into predefined categories. We developed two parallel approaches to solve this problem; these strategies are: Top-Down and Bottom-Up. The Top-Down approach leverages Common Crawl, which is a vast repository of web pages scraped since 2008. The Bottom-Up approach harnesses the power of Google to perform a targeted search based on author seed data. These strategies will be described in detail throughout this report.

The package is tailored to integrate seamlessly with the IEEE Data Lake and it necessitates preliminary setup from the user, including Redshift database credentials, a Google API key, and a search engine ID. Overall, this sophisticated tool is capable of tapping into IEEE's proprietary data resources, conducting targeted searches on potentially millions of published authors, and organizing the results according to IEEE's specific categorization standards.

II. DATA REQUIREMENTS

Common Crawl is an open corpus of web pages scraped from across the internet. New crawls consist of several billion web pages and new dumps, called indexes, are added several times per year. At the time of writing, the latest crawl ID is “CC-MAIN-2023-40”, which consists of nearly 3.5 billion records. Additionally, while the crawls do not visit a mutually exclusive set of sites, the overlap tends to be quite small[1].

When testing this resource as a potential repository in which to search for scholarly links relevant to authors of interest, we limited ourselves to the latest crawl. Additionally, we filtered for web domain names known to contain important scholarly information or interest, namely: github.com, wired.com, dblp.org, twitter.com, and .edu. After filtering solely on domains (and not on seed information), we were left with approximately 45,000 matching web pages.

For both pipelines, a crucial step was to gather seed information for authors of interest. There are millions of authors in the IEEE data lake, but we limited ourselves to a high-ranking subset defined as the top 1% of authors by citation count on papers published within the last 5 years. This set alone consisted of over 700,000 unique author IDs. The most informative seed data was determined to be the author name, their last known affiliation, and a selection of their top cited papers; these data were gathered from the appropriate tables in the data lake. The latter step sourced data exclusively from the IEEE's Redshift database. Due to the proprietary nature of this information, we are bound by confidentiality and therefore unable to disclose specifics regarding the database structure.

III. TOP-DOWN STRATEGY - COMMON CRAWL

Overview

Using Common Crawl as our data source, we developed a top-down approach to discovery of scholarly entities; in other words, we began with a vast collection of web pages and filtered them down to a pool of candidates. This was accomplished by querying the Common Crawl Index for domain patterns of interest (as described in the Data section). By passing these domains into the Index searcher with wildcard regex operators, we could extract all pages in a given Index that matched these domains. Our approach was then to continue this top-down methodology by extracting the matching web pages and performing another filtering step in which we look for author seed information within the page. Together these stages should find the most relevant links in the crawl pages should they exist given author seed data. Seed data was generated by querying the IEEE data lake and extracting author names, affiliations, and paper titles.

Workflow

The workflow of the `fetchCC` pipeline, as orchestrated by `cc_runner.py`, consists of three main components: an index, Extractors, and Requesters. Together these classes facilitate the

[1] Common Crawl statistics: https://commoncrawl.github.io/cc-crawl-statistics/

top-down approach described above. The first step is to define the Common Crawl indexes that the user wishes to search; this is accomplished by the CCIndex class (Figure 1 - Step 1). This pipeline then sets up a Requester and Extractor designed to query the Common Crawl Columnar Index for defined web domains of interest (Figure 1 - Step 2). Upon retrieving matching Index records, another Extractor and Requester pair is created, this time capable of processing Web Archive (WARC) files[2]. These WARC processors extract the web page data from links returned in Step 2 and filter them based on the presence of author seed information (Figure 1 - Step 3).

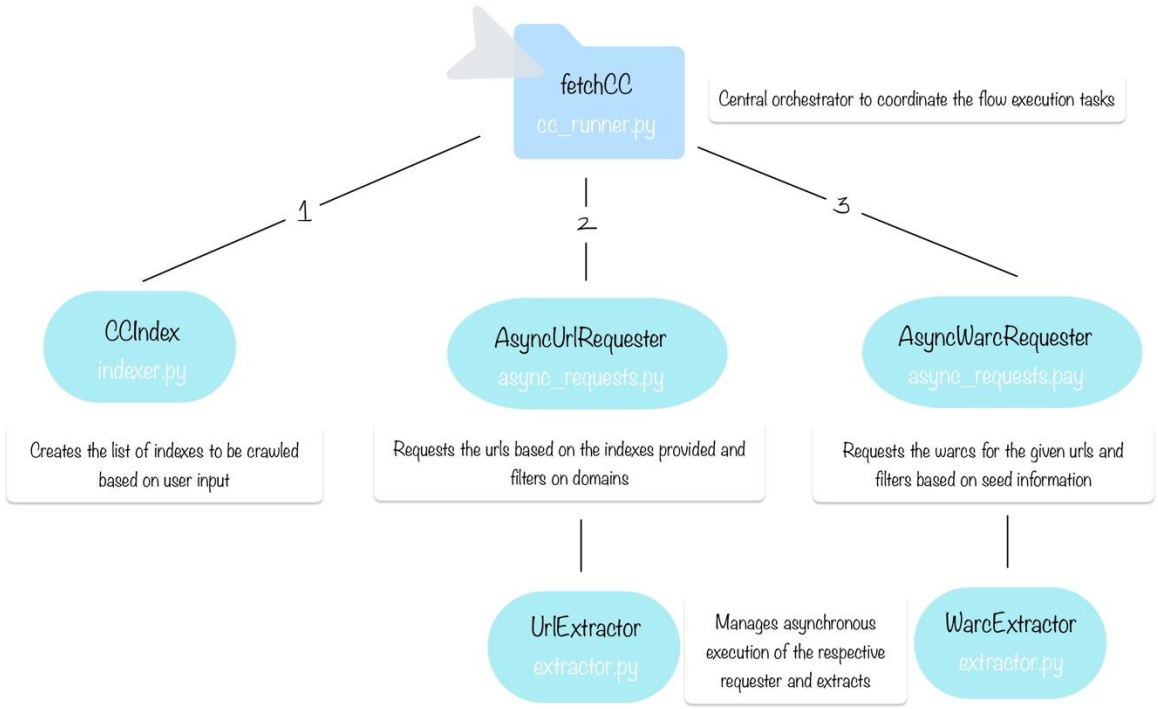

Figure 1: Common Crawl Workflow

Here we elaborate on the functionality of each component of the `scholarscraper.commoncrawl` module. The `CCIndex` class within `indexer.py` allows the user to define in which specific set of Common Crawl indexes they would like to search. This class will collect a list of crawl IDs (strings of the form: 'YEAR-CRAWL', e.g. '2023-40') depending on the user's instantiation of `CCIndex`. Additional crawls means more processing time, but the flexibility of `CCIndex` allows the user to set up their author search on many billions of web pages if they choose to select multiple crawls.

Next, the Requester steps described in Figure 1 are carried out by three classes in `async_requests.py`, which defines asynchronous HTTP request handlers, using the aiohttp package, for the Common Crawl Index and WARC files. The base class, `AsyncRequester`, defines a custom GET request method wrapper around aiohttp.ClientSession. `AsyncUrlRequester` extends `AsyncRequester` and implements a method to process each domain of interest the user wishes to search on the Common Crawl index. `AsyncWarcRequester` also extends `AsyncRequester` and implements the same processing function, but configured to work on JSON metadata returned by `AsyncUrlRequester`. The

[2] WARC file type described: https://commoncrawl.org/get-started

purpose of `AsyncWarcRequester` is to download WARC files using metadata obtained from the Common Crawl Index; these files are only retained if the page content matches information contained in the author seed which was used to instantiate the instance of the class. Additional helper methods are described in the Capstone Github repository.

Another crucial file is `extractor.py`, which contains the `AsyncExtractor` class and its specialized subclasses: `UrlExtractor` and `WarcExtractor`. These classes are designed to orchestrate asynchronous requests and processing of URLs or WARC records, respectively. They implement a producer-consumer workflow for the efficient asynchronous processing of items, allowing many coroutines to run at the same time. Since requests to the Common Crawl server can take a long time, this design allows processing of the successful requests to run immediately. The user has the flexibility to define the relative sizes of the `asyncio.Queue` and the number of workers; while in theory the `asyncio` library supports thousands of concurrent tasks, we have been limited by Common Crawl's ability to handle requests, so have limited the number of concurrent tasks to 25. In the case that Common Crawl is able to handle more requests in the future, the limit could be increased and the number of tasks will be scaled according to the number of items that must be processed. The most important attributes of the Extractor classes are the iterable, from which items (urls or domain records) will be enqueued, and the Requester class, which implements a `process_url()` method.

In summary, the pipeline implemented in `cc_runner` integrates several components to automate and streamline the process of retrieving and filtering web content from the Common Crawl dataset, ensuring efficient data collection in a top-down information retrieval model.

Issues and Limitations

Common Crawl is a promising, free resource that in theory should allow IEEE to access large numbers of web pages containing scholarly entities without having to implement a scraper of the entire web. Unfortunately, in developing this module, we have run into many challenges in using Common Crawl. The most limiting feature about Common Crawl is that it is a free, open resource that is hosted by Amazon under the AWS Open Data Sponsorship Program. This is both a benefit and a major downside to the resource; it means Common Crawl is free to access, but also that any number of people may do so without limit. Moreover, the servers have a low capacity to handle incoming requests. As a result, over the course of the semester, we have received almost exclusively 503 Gateway Timeout Errors in our request responses (Figure 2).

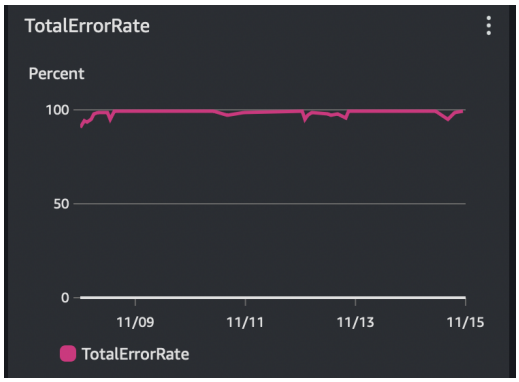

Figure 2: mid-November error rates

Through correspondence with the maintainers of Common Crawl, we learned that this is because the popularity of Common Crawl has grown substantially and there are several entities

that are sending requests at rates well above the manageable limit of the Common Crawl servers. Greg Lindahl, CTO at Common Crawl, informed us that under the AWS Open Data Program, they are able to handle only a thousand requests per second. As an example of the overload on servers, at the time of this conversation, a particular entity was sending 70,000 requests per second for 36 hours straight. Unfortunately, they have no way of identifying who this is as they are open source. The team at Common Crawl has set up a status site that will track the current request metrics to the resource[3] and there is an active Google Group in which the maintainers of Common Crawl post regular updates[4]. In the last several weeks (December 2023), the error rate has improved dramatically and requests have been more successful (Figure 3).

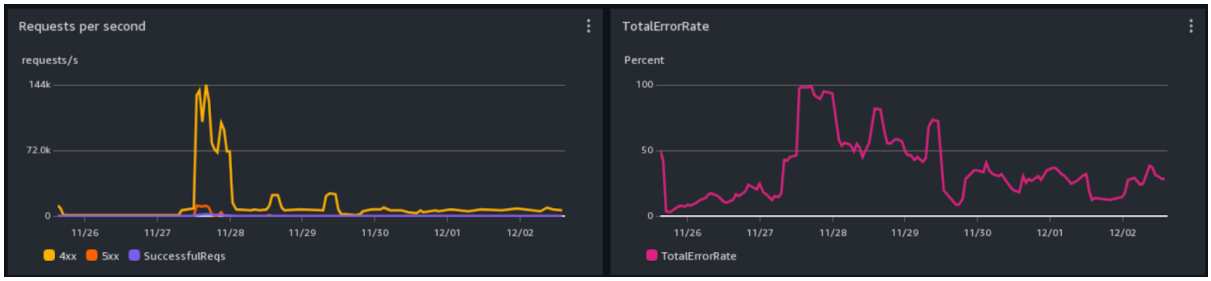

Figure 3: Improvement of error-rate and request load in early December

Furthermore, there are notable issues concerning the quality of link extraction. It's essential to highlight that we depend on the content within webpages of these URLs for filtering using seed information. In the case of **Google Scholar** and **LinkedIn**, recent crawls reveal a minimal presence of extractable links. As for **Twitter**, while some links are captured in the crawl, CommonCrawl appears unable to access the content within, likely due to privacy restrictions on Twitter pages that prevent data scraping. This may be considered a breach of confidentiality, leading Twitter to block crawlers like CommonCrawl.

Regarding **GitHub**, despite the large number of links identified, the private nature of some repositories and potential crawling restrictions imposed by GitHub hinder content extraction. Similarly, for **Wired**, although URLs are present in the CommonCrawl database, the content within these URLs remains inaccessible, impeding our ability to filter based on seed information.

Potential Solutions

Unfortunately, we cannot improve the server load problem. Since the data is open for public use, it is incumbent on users to act ethically. This means that there may always be periods in which the Common Crawl data is essentially inaccessible; the status tracker is a helpful tool for scheduling requests. The graphs on the site are not currently implemented as a live dashboard, but are updated as screenshots periodically by the team at Common Crawl. Thus, it is impossible to know the current server load in real time. Further, since this is another public resource, there is always a possibility that malicious third parties will overload the bucket using the CloudWatch data.

That being said, we do have control over how we initiate and process requests. Scheduling the `scholarscraper.commoncrawl` pipeline to run at times expected to be outside of peak hours would likely improve the overall error rate. If results are not needed quickly, then

[3] Common Crawl status watch: https://status.commoncrawl.org/
[4] Common Crawl Google Group: https://groups.google.com/g/common-crawl

respectful boundaries on the number of concurrent workers in the `AsyncExtractor` class and adding **back-offs between retries** will help lower the server load as well.

Additionally, since the existing Common Crawl Indexes are static - new web pages are not added after an index dump has been published - the search over these crawls for pages matching URLs of interest only needs to happen once rather than for every new run of the pipeline. Storing the records of matching web pages would reduce the number of requests that need to be sent from this pipeline. Our assessment is that a similar solution for the matching WARC files is likely to be untenable given the space requirements to store data from potentially millions of web pages. However, we had limited space allocated to us during this project and IEEE may be able to scale this much further. In the case that space is not a constraint, then it may be worth exploring the possibility of downloading all WARC files that match the scholarly domains of interest and searching them statically in the future.

Lastly, according to the Common Crawl status tracker, the error rate for requests to the Common Crawl S3 bucket is lower than that to the CloudFront bucket (HTTPS requests). The current `commoncrawl` module we have developed sends requests to the CloudFront bucket. However, it is also worth noting that large files retrieved from the S3 bucket are subject to batch requests, making them more susceptible to 503 errors and the return of incomplete data.

IV. BOTTOM-UP STRATEGY - GOOGLE API

Overview

This package is designed with a streamlined, sequential pipeline, where each function is intricately interlinked, depending on the input from its predecessor. A comprehensive breakdown of each component within this pipeline is provided in the subsequent section for a clearer understanding of its inner workings.

Workflow

In the first stage of the pipeline, **querying**, our focus is on establishing a connection with the Redshift database and performing queries. This process begins with initiating a connection to the Redshift database. Following a successful connection, we employ sophisticated, parameterized SQL queries to extract a specific subset of data. This extracted data is then meticulously formatted to suit our requirements before being passed on to the subsequent searching phase.

Building upon the seed information obtained from step #1, we form specific queries. By interfacing with Google's Custom Search API, these queries guide us to retrieve related **search** results from the web. As we are targeting high-quality domains for maximum relevancy, we structure our searches accordingly. An asynchronous queue, implemented in SearchQ, allows requests for many authors to be sent concurrently. The results are partitioned by author and returned as a list for downstream processing in the Scrape stage of the pipeline.

Finally, we apply a **categorization** process to the URLs obtained in step #2. Through a defined set of rules, links are categorized to specific sites such as LinkedIn, Github, Google Scholar, and so forth. Post categorization, the links are processed to extract text content, after which they are further aggregated and stored for subsequent analysis. The Categorizer class is pivotal in efficiently executing these tasks.

In this phase, we **filter** out undesirable links using regex to identify certain phrases in the URL. We do this for the link categories that we are unable to scrape from with the resources we were allotted, such as Google Scholar and LinkedIn, and for the links with obvious extensions, such as Github and edu links. Some examples of this include filtering out LinkedIn links with the extension '/posts' to avoid a webpage of a LinkedIn post since we only want profile links and removing Github links using a variety of extensions to avoid those that are not repositories. Filtering these links out early on allows us to only focus on links that are actually usable. This significantly reduces the time spent in the next phase, sorting using TF-IDF.

In this part of the pipeline, the primary purpose is to sort links in order to relevance so that the most important links appear first. For each author, we have a list of links in each category type, for instance Github, Linkedin, Twitter, News, and Edu, which is based on its url. For some of these link types, specifically edu, github, and news, we use the text scraped from the websites to determine how relevant they are to the query. This is achieved through TF-IDF or term frequency-inverse document frequency, which uses the term frequency in a document and the inverse document frequency of the word across the entire set of documents for an author to assign a score to each word for each document. See the formula below.

$$\text{TF-IDF} = \text{TF}(t, d) * \text{IDF}(t) \quad \text{where t is term and d is document}$$

$$\text{TF}(t,d) = \frac{number\ of\ times\ t\ occurs\ in\ d}{total\ number\ of\ terms\ in\ d}$$

$$\text{IDF}(t) = \log\left(\frac{total\ number\ of\ documents\ for\ a\ set\ of\ authors}{number\ of\ documents\ containing\ term}\right)$$

We chose TF-IDF, because it is a very popular approach to quantify the importance of terms in a collection of documents.

In this process of sorting, we sort one category's list of links for one author at a time. We first preprocess each article's text by removing unnecessary characters, such as punctuation, numbers, and extra space, lemmatizing so that each word is in its root form, tokenizing, and removing stop words, such as 'a', 'the', and so on. Then, each list of tokens is passed to a corpus array, which consists the tokens list for all articles.

From here, we call TfidfVectorizer(), a sklearn module, and fit our corpus data. This will result in an array of size (total number of articles x total number of words) where each row corresponds to a different author and each column represents a word. Therefore, each cell consists of the TF-IDF score of each word in each article.

Next, we tokenize all of the queries used for a given author and remove punctuation and stop words. Then, using this list of query tokens, we find the score of each word in each article and sum the scores row-wise so that we have a vector in which each cell corresponds to the score of an article. Then, we sort each article link according to its score in descending order. As a result, the articles for a specific category are returned in sorted order.

V. RESULTS

Due to the limitations of the Common Crawl resource discussed previously, we limit this section to describing those obtained using the Google API pipeline. Within the constraints of the free-tier API, we were able to perform insightful data analysis on the outputs obtained for a

portion of the most cited authors. We collected this data using two settings of our API: unfiltered links and links with filtering and sorting activated.

Link Distribution Analysis

We observed the distribution of the total number of links per author in the unfiltered (Fig. 4A) and filtered (Fig. 4B) data. The unfiltered data represents the original results and in the filtered data, we apply our tf-idf model and our other filtering methods. We used a sample of 887 authors, which is the most we were able to retrieve. Both graphs have a right skew, and in the unfiltered data, there are many outliers on the right end that are pushed to the left end. As a result, after filtering, none of our authors have more than 25 total links and our mean number of links per author has decreased significantly from 10.28 to 4.89. This indicates the filters' effectiveness in streamlining the results towards more relevant links.

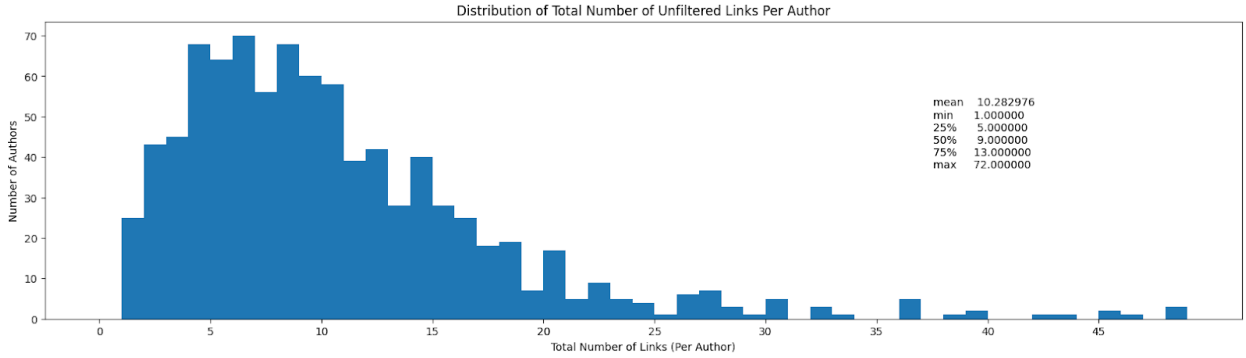

Figure 4A: Distribution of links-per-author for unfiltered data

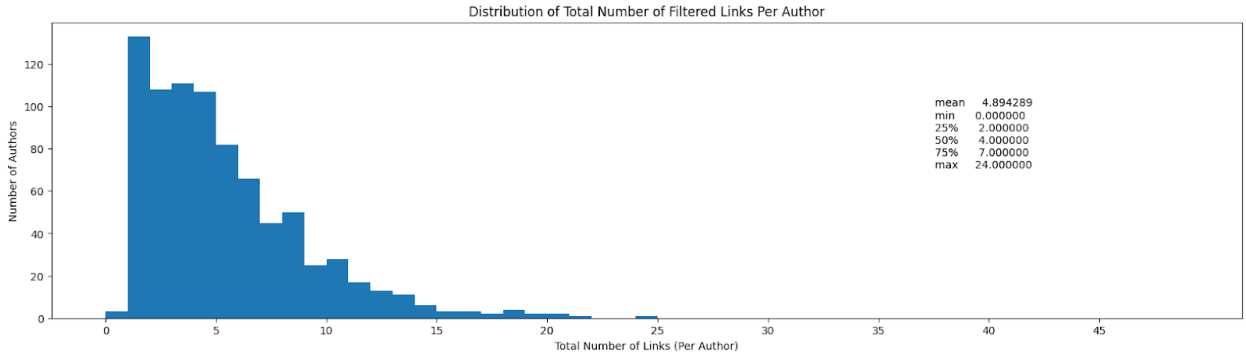

Figure 4B: Distribution of links-per-author for filtered data

Subject Area Composition

We assigned each of the authors in our sample to their academic field, ultimately aggregating related fields into three: biomedical sciences, computer science and math, and other engineering disciplines (Fig. 5). Interestingly, the majority of our sampled authors were in the biomedical sciences domain. This is likely due to the fact that authors in this field are generally more cited. Other areas like computer science and engineering are less represented among highly cited authors.

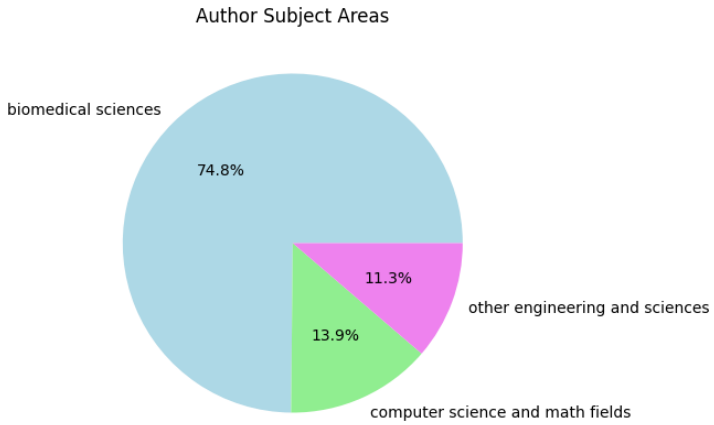

Figure 5: Composition of subject areas for sampled authors

Average Link Count

Next, we found the mean number of links for all authors grouped by category (Fig. 6). There is a clear decrease in the average number of links when we add filtering in the bottom table compared to using no filtering in the top slide. This also shows that on average we get almost exactly the number of links we expect to get after filtering. For DBLP, LinkedIn, Google Scholar, and Twitter we expect to get around one link for every author, and our results reflect that. What's notable is that the Edu category has the largest decrease across all categories with a decrease of over 50%. This reflects the filter's ability to reduce irrelevant academic links effectively and leave us with the links we're expecting to see.

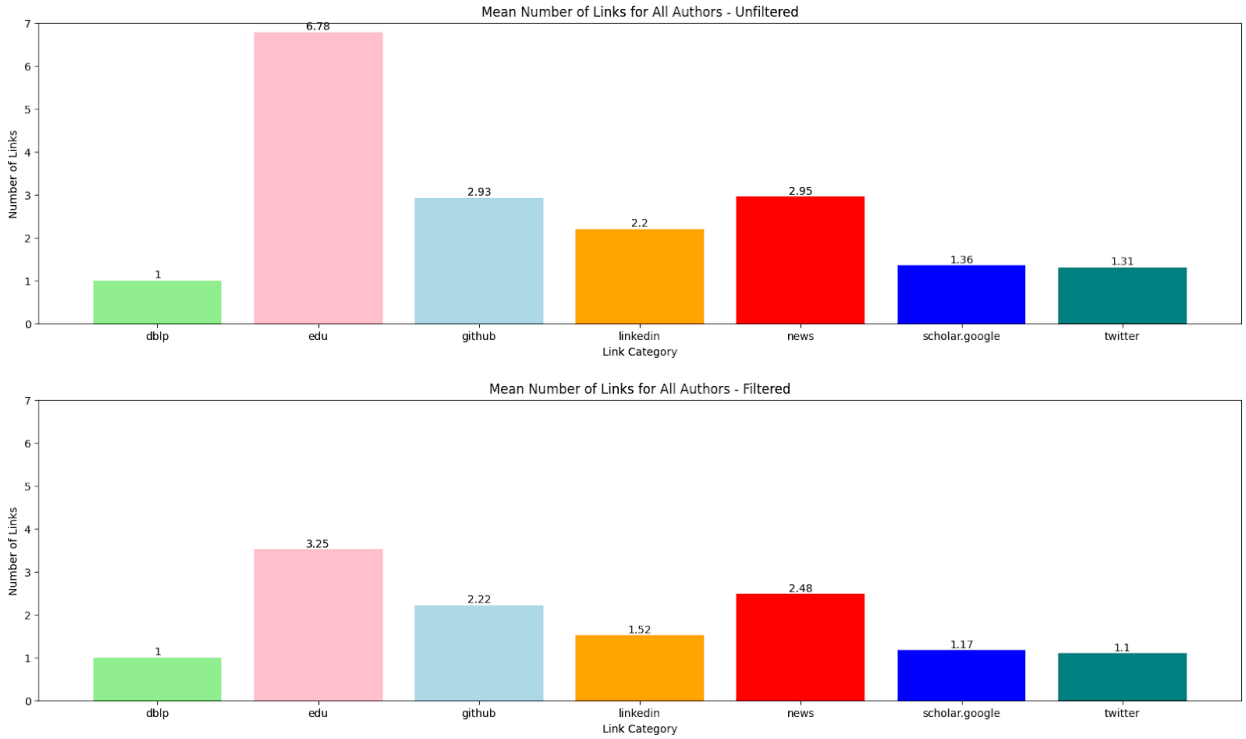

Figure 6: Composition of subject areas for sampled authors

Completeness Metric

Finally, we conducted an analysis of the completeness of our results (Fig. 7). This was computed for each category by taking the sum of authors that have a link of a given category divided by the total number of authors. Note that many authors do not have a strong internet presence. Across all subject areas, our results are fairly consistent (Appendix Fig. 1). LinkedIn, Edu, and Google Scholar links have the highest completeness. Note that authors from the Computer Science and Math-Related Subjects has more complete Github results compared to the other categories, which makes sense given Github is a more popular platform in these fields. Each data point contributes to understanding the web presence of academic authors and the efficacy of the scraping and filtering methods used.

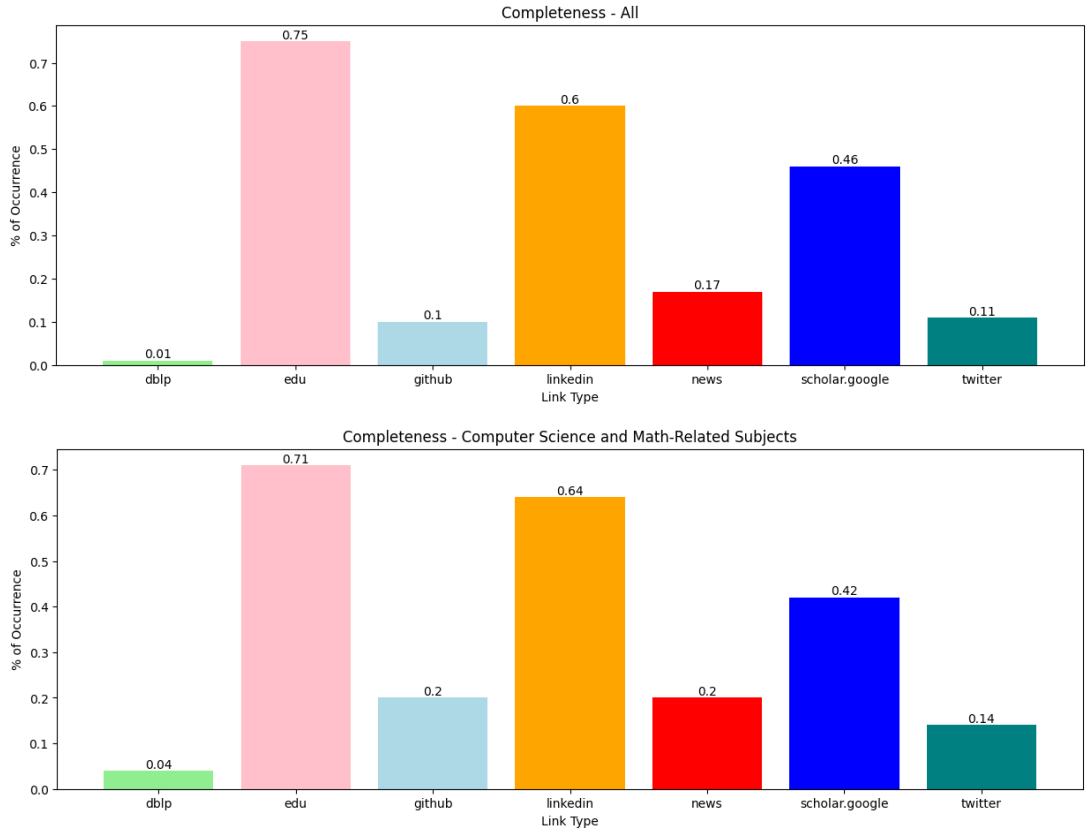

Figure 6: Completeness metrics by link category

Additional Features: Package Testing

This section focuses on the reproducibility assessment of the package, conducted utilizing the pytest framework in Python. Customized functions were developed to examine each phase of the pipeline, encompassing checks for database connectivity, validation of the database schema, authentication of database and API credentials, and ensuring that search outcomes align with expected results.

For newcomers using the package, clear instructions are given on how to run the test files. By executing the 'make tests' command at the end of the setup process, the functions within the test files are executed. This step ensures that all critical aspects of the package are functioning correctly and as expected, essential for its effective usage.

Issues and Limitations

To summarize the capabilities of the Google API for data management and extraction within academic settings, particularly concerning the IEEE data repository, we discovered several strengths and weaknesses. The API provided robust functionality for creating specific search queries and conducting curated searches. It also allowed for meticulous control over the sorting and filtering of search results. These functionalities enhanced our project's efficiency by enabling rapid, scalable, and asynchronous searches.

However, our project also faced several limitations, primarily the **limited query capacity** of the free tier of the Google API, which imposed a search cap of approximately 100 queries per day. This severely limited our ability to conduct broad searches on large portions of the IEEE author set. Additionally, we encountered significant **scraping restrictions** on key academic and social platforms, including: Google Scholar, LinkedIn, Twitter. Scraping of these types of sites is generally forbidden by the companies that own them. Thus, any ranking or filtering of these sites had to be done using the information contained in the URL, not the content of the page itself.

VI. FUTURE DIRECTIONS

The `scholarscraper` package, which currently defines two web-scraping models — Google and Common Crawl — has the potential for significant advancements. Future development can be envisioned in the following areas:

- **Advanced NLP Techniques:** Integration of cosine similarity with word embeddings, FAISS, or a fine-tuned BERT model are other approaches that could result in an improved search relevance and filtering.
- **Data Aggregation:** Combining the Common Crawl and Google pipelines would provide a more comprehensive dataset.
- **Improved API Handling:** A new routine to better manage Google Scholar and LinkedIn data and track API credit limits could optimize resource use.

For the Common Crawl pipeline, additional work could be done to address the points made in ‘Potential Solutions’, namely saving a local repository of matching web pages that match web domains of interest and possibly something similar for the WARC files themselves depending on storage capacity. The scale of the Common Crawl data was a significant hurdle for us given the computational resources available; full Indexes hold terabytes of data. However, it is possible that the space and computational power constraints would no longer pose a problem if this pipeline were reengineered to work on a distributed Spark cluster. In this case, our recommendation would still be to run the first stage of this pipeline - filtering all of Common Crawl down to specific domains of interest - before saving link records or WARC files, ensuring only data from potentially relevant pages is downloaded.

Finally, a limitation of Common Crawl is the types of domains that it scrapes. Common Crawl performs its scrapes in a “respectful” manner, abiding by domains that prohibit their data to be scraped. In these cases, a link in Common Crawl under the prohibited domain will typically direct to a robots.txt file[5]. For this reason, several web domains relevant to this project are lacking, including LinkedIn, Google Scholar, and Twitter. Pages matching Wired.com were also limited, so future directions could include integration of the Common Crawl news dataset[6].

[5] https://developers.google.com/search/docs/crawling-indexing/robots/create-robots-txt

[6] CC-NEWS: https://commoncrawl.org/blog/news-dataset-available