KTH | ROYAL INSTITUTE OF TECHNOLOGY

DATA-INTENSIVE COMPUTING

GitHub Committer Relationship Visualizer

Group "The Stallions"
Mattia Evangelisti - mattiaev@kth.se
Perttu Jääskeläinen - perttuj@kth.se
Gabriele Morello - morello@kth.se

1 Introduction

In this project, we aim to explore the relationships between contributors to selected GitHub repositories. Our goal is to create a graph that visualizes connections among repositories and users, highlighting those users who contribute to common repositories.

We will utilize the GitHub public APIs as our data source. While our project draws inspiration from the Erdős number, our focus is on a specific set of GitHub repositories and we do not intend to recursively explore all repositories.

1.1 Project Goal

The goal of the project can be defined as follows:

- Construct a scraper to gather commit data from selected GitHub repositories
- Implement a streaming implementation that aggregates commits made by users in the selected repositories
- Use the output of the streaming implementation to visualize relationships between users committing in multiple of the selected repositories using a Graph structure

2 Method

Below we describe the methodology used to achieve the project goals presented in section 1.1

2.1 Choice of Dataset

We chose GitHub as the source of our data due to their available public API. First, we selected a subset of repositories to scan for commits. Since the amount of repositories on GitHub is massive, we chose the subset as a set of popular open-source projects. The organizations that we chose for our subsets are the following:

- Apache Foundation (16)
- Facebook (24)
- Google (1)
- Microsoft (21)

The repositories chosen add up to a total of 62, and the actual repositories chosen can be seen in the appendix (figure 5) and in our GitHub (contact authors to request access)[1].

[1] https://github.com/GGmorello/github-graph

2.2 Scraper

The scraper fetches commit data from GitHub repositories using their public API. To facilitate that we use PyGitHub which is a library that provides an interface to GitHub API without having to use HTTP requests. The scraper takes as an argument the name of the repositories (owner/repo) we want to retrieve and saves the results in a JSON file to support future work. The file is a list of commits where for each row we store the repository name, the author of the commit and the date. We also stored a list of authors for each repository in another JSON file. The program has two different modes - the first retrieves all the commits to the date of execution while the other retrieves commits one day at a time. The second mode was developed to have live daily updates instead of having to rerun the full analysis every time we want to update our graphs.

2.3 Streamer

The streamer component reacts to changes in the scraped data and aggregates commits made by users in real-time, keeping track of duration's in which the commits were made and by which user/repository. To do so, we utilize Spark streaming with windowing intervals and grouping functions. The streaming implementation periodically stores data on disk, such that the application can easily be re-run in case of errors in the processing.

As a prerequisite for the streamer, we need to make sure it has the necessary files available from the scraper. We do so by using a separate component that reads the output folder of the scraper, selecting the relevant files for the streamer and moving them to the streamer working directory. To enable visualization of the data, another component reads the output folder of the streamer and exports the relevant files to the visualization component.

2.4 Visualizer

The final step of our project's pipeline is the visualization of the users-repositories graph. The visualizer module receives data streaming of users and repositories information and creates the graph structure. The created data structure has as nodes the repositories, while edges are created between nodes if at least one user has committed to both repositories. For each node, we saved its `node_id` which is the repository name and the node degree. At the same time, each link carries information about the source and target nodes, their weight, and a label, representing the list of users that committed to both repositories connected by the edge.

The visualizer then creates a Flask web application to display the previously explained graph. The graph visualization is dynamically updated if new data arrives from the streamer, this data is sent to the web application through a `POST` request.

3 Results

Below we describe the results for the project - we present the components written, how they are implemented along with descriptions for what data we visualize. To run the code, see the **README.md** file in the project directory. Currently, the steps to run can be seen in the appendix but might be outdated in case the repository is updated.

3.1 Scraper

We retrieved commit histories for 62 repositories and stored the results on disk in JSON files in the **data** directory. It took quite a long to retrieve all the data because the APIs provide batch of commits for each repository and not the full history, therefore, the number of HTTP requests for each repository was quite high and was occasionally limited due to the GitHub rate limit. This is a problem when starting from scratch, but once an initial data set is gathered it is easy to keep up to date, since requests only have to be performed occasionally. The code can be found in the **scraper.py** file.

3.2 Streamer

The streaming implementation consists of three parts to integrate with the scraper (section 3.2.1) and visualizer (section 3.2.3) and to handle streaming data (section 3.2.2).

3.2.1 Importer

The **importer** component reads files from the **data** directory, where each entry in the top level of the directory is another directory, named after the repository to which its underlying files belong. The importer will recursively iterate the **data** directory and any sub-directories, looking for files with the **commit.json** format. When it finds such files, it copies them to the **src/input_files** directory, which the streaming component introduced below reads files from - in order to aggregate commits made by users and dates during which the commits were made. The code can be found in the **importer.py** file.

3.2.2 Streamer

The **streamer** component monitors files in the **input_files** directory, and reads any files with the expected following schema distribution in JSON formatted rows:

- **repo STRING, author STRING, date TIMESTAMP**

The streaming component itself uses 4 week windows to group together events received. This was done to avoid large output files with a single commit per row - instead, we want to perform some aggregation to reduce the rows per file - and since we're only interested in the commits made in individual repositories, the amount of commits was not relevant at the time of implementation.

The aggregation of rows is done through grouping of **repo, author** values, with a count being tracked for each individual author and repository. The streamer stores its output in the **checkpoints** and **output_files** folders. The code can be found in the **streamer.py** file.

3.2.3 Exporter

In order to integrate with the visualizer component, the **exporter** component can be run to transfer the output files of the **streamer** to the visualizer component through a POST request (see section 3.3 for visualizer implementation details). The code can be found in the **exporter.py** file.

3.3 Visualizer

The graph is visualized using `networkx` Python library in a web application presents some features to better highlight the characteristics of our data. Firstly we decided to show only 15 characters out

of the repository name for better clarity in the visualization, while the organization and full name of the repository are shown when the mouse is moved over the node.

The dimension and width of nodes and edges are proportional to their rank and weight respectively. Moving the mouse cursor over a node or a link would result in showing information about it: rank and repository name for a node and weight for an edge. Moreover, we added two sliders to allow the user to filter the nodes and edges to be shown based on their rank and weight.

Finally, when a node is clicked, its neighbors are highlighted, to better show which repositories are directly connected to the selected one. An image of the visualized graph can be seen in the appendix (Figure 1), along with more detailed images. The code can be found in the **visualizer.py** file.

4 Conclusion

We identified some possible improvements that could be done to the project if given more time, or if we wanted to implement a live application that would periodically fetch commits from repositories. Some of these improvements are as follow:

- Automate the fetching process for repositories with most stars
  - Allow the option to give an organization as input, and gather their repositories automatically - ex. **apache** or **google** - rather than explicit names.
- Automate the fetching data to periodically fetch new commits, rather than when explicitly run
- Automate the filtering of fetched data and moving of files to the streaming process
  - Currently fetches input files once when run, doesn't keep track of files that already exist.
- Automate exporting of data from the streamer to the visualizer - similar to the suggestion above
- Add feature to the visualizer to provide additional graphs that might be interesting
  - Visualize commits per repository
  - use different scales for graph nodes/edges through feature flags

We also considered the bottleneck of data size of commits - for example, if we had all the historic data for hundreds of repositories, we'd quickly run into memory problems. This was done as an explicit choice in this project, and is suggested as future work.

The data could also be stored on disk in the visualizer, and only provide possibility to show trends for repositories based on date intervals specified by the user. In the current state, one might run out of memory if the amount of repositories/commits exceeds the available memory of the browser running the application.

The entire pipeline could also be improved with automation - sine the scraper, importer and exporter have to be explicitly run. We made sure the streamer worked dynamically by manually moving files into the working directory, but it would also be interesting to automate the entire pipeline if given time.

Appendix

How to run

Scrape data

1. Navigate to the project directory.
2. Create a `.env` file in the root directory.
3. You can set your API key by running:

```
echo "your_api_key_here" > .env
```

4. Install the requirements:

```
conda install --file requirements.txt
```

5. Navigate to the project directory:

```
cd src
```

6. Run the scraper by running:

```
python src/scraper.py
```

Run the streaming workflow

To run the entire application, make sure to have **Python 3.12.0** installed on the running system. Then, perform the steps below from the project root directory:

- pip install -r requirements.txt
- cd src
- Run the **python visualizer.py** command to start the visualization UI.
- Run the **spark-submit streamer.py** in a new terminal to start reading files from the input files directory
- Run the **python importer.py** command in a new terminal to move files from the data folder (scraper) to the streaming application
  - The moving of the files can also be done dynamically, i.e. in the case of a real streaming environment. The spark app reacts to changes in the directory and updates data as it runs.
- Run the **python exporter.py** command to export files to the visualizer.
- Browse the graph on your configured localhost address. On macOS, it should be **127.0.0.1** (see command line output for actual address when starting the visualizer)

Figure 1: Resulting graph visualized on the web application. The darker blue colors reflect repositories that have a higher degree - i.e. more users who commit to the repository also commit to other repositories, while the lighter blues which are smaller in size, reflect repositories that don't have that many committers in other repositories.

Figure 2: Graph filtered to show nodes with degree greater or equal than 55

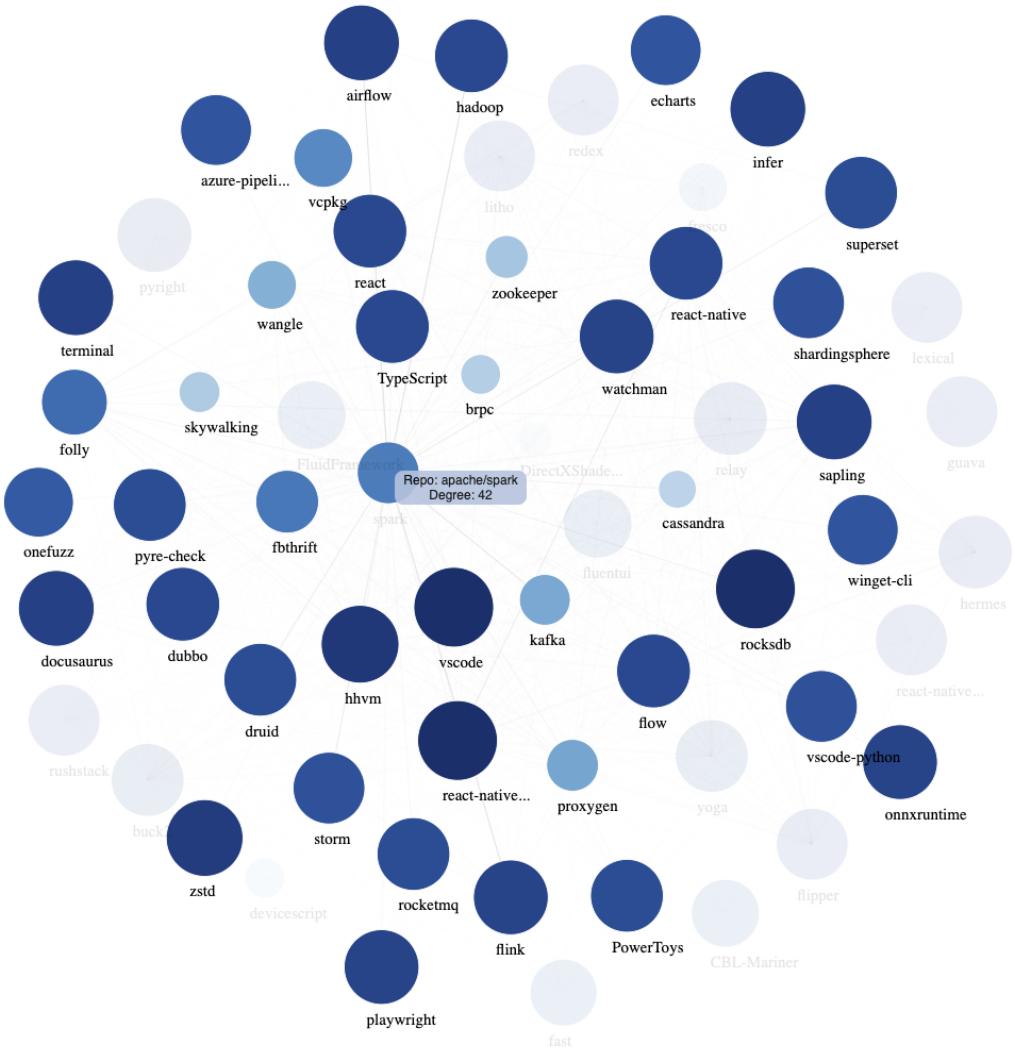

Figure 3: Resulting graph after clicking on the node representing the Spark repository, only the repositories directly connected to the clicked node are highlighted

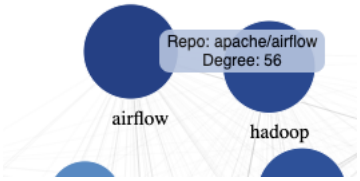

Figure 4: Detail of the graph. When the pointer is moved over a node the node name and degree are displayed

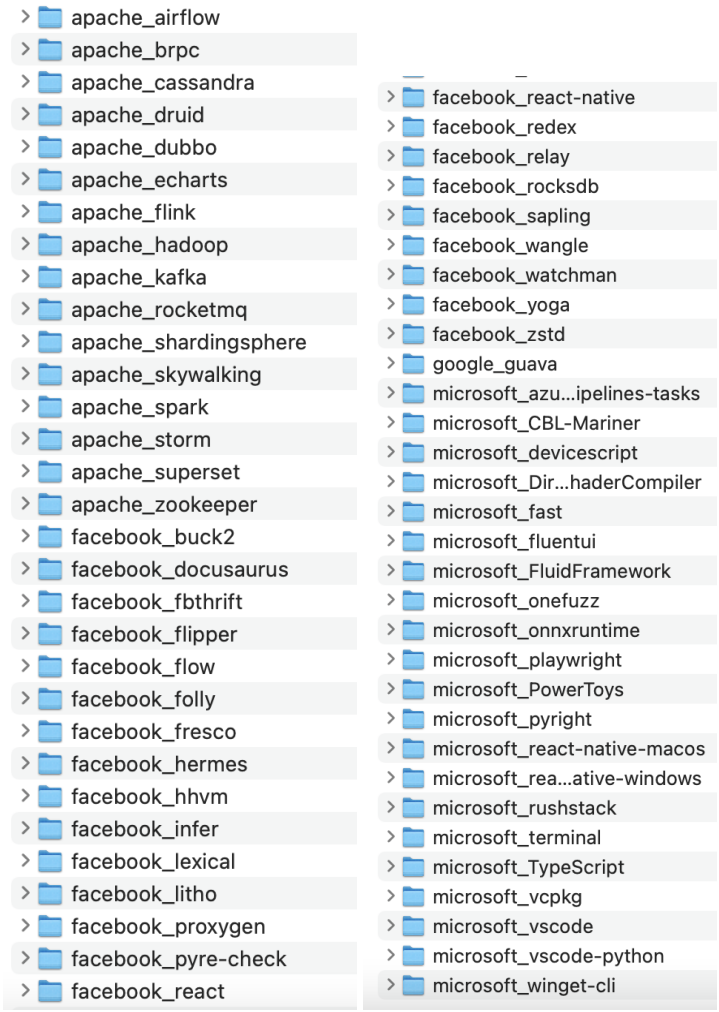

Figure 5: List of the selected repositories for each organization. These can also be found in the GitHub repostiory, in the Data folder. They were populated using the scraper component to fetch commits from GitHub.