School of Electrical Engineering and Computer Science

Project Report

Advisor: Dr. William Braynen

Assistannt: Dr. Vladiszlav Pauk

Graduation and Placement Trends in Philosophy Graduate Programs Using Automatic Web Scraping

Abstract

This project investigates the enrollment and placement trends in Philosophy graduate programs throughout the United States and Canada. By leveraging web archive service, we construct detailed enrollment timelines for numerous graduate programs. The data collection process is facilitated through an advanced web scraping pipeline, which utilizes generative AI scripting to streamline and scale the extraction process. We provide an interactive interface for exploring and analyzing the collected data via web-application Grad Stats. The application allows users to search for universities, view detailed statistics on student data, and gain insights into various graduate programs.

1 Overview and Motivation

Enrollment, graduation and placement data is a valuable resource for academic research, program evaluation, student recruitment, and ranking. Across various disciplines, data reporting varies in terms of consistency and accessibility. Centralized databases akin to the National Science Foundation's Survey of Earned Doctorates[1] and grant-funded initiatives like the Council of Graduate Schools' PhD Completion Project[2] provide aggregate statistics, but lack information on individual programs. Accessing such data is labor-intensive and can be challenging due to legacy systems, privacy concerns, and inconsistent reporting practices.

University academic departments often publish lists of current graduate students on their websites, providing insights into the student body of individual programs. Philosophy programs, for example, often provide detailed information on current graduate students, including their names, graduation status, and placement outcomes. By collecting and analyzing this data over time, we can gain a comprehensive understanding of enrollment trends and assess the outcomes of individual programs. The Wayback Machine web archive service captures snapshots of web pages over time, which enables tracking changes in student listings, and enables longitudinal analysis of enrollment. By monitoring student names in the webpage over time, we can estimate start and end dates of each student's enrollment, providing access to time-to-degree metrics. To assess the outcomes for each student, the placement and graduation information can be retrieved by matching student database against program placement, alumni and other open sources.

Collecting web archive data on graduate students is a challenging task due to the variability in webpage structures, the dynamic nature of web content, and internationalization of names. Furthermore, webpages

[1] https://ncses.nsf.gov/surveys/earned-doctorates

[2] https://cgsnet.org/data-insights/diversity-equity-inclusiveness/degree-completion/ph-d-completion-project

often list supervisors, alumni, and other non-student names, further complicating the extraction process. Using standard web scraping techniques and libraries yields limited results, while manual programming of pattern matching is time-consuming, not scalable, and requires manual maintenance and updates.

Recent advancements in Large Language Models (LLMs) have opened new avenues for automating and streamlining code scripting. Models like OpenAI's GPT exhibit exceptional capabilities in information retrieval and generation tasks, and became a widespread tool in natural language processing. However, the non-deterministic nature of LLMs presents challenges in producing consistently reliable results for direct information retrieval. Moreover, performing inference with LLMs on large collections of web pages is computationally intensive, often resulting in significant overhead. On the contrary, with targeted prompting, it is possible to generate high-quality code snippets that can be validated and integrated into manually coded pipelines. In this way, the strengths of LLMs can be harnessed to automate the generation of web scraping scripts, while maintaining the robustness and reliability of programmatic extraction.

2 Methodology

We determine student enrollment in a program over time based on the student's first and last appearances in the web page source. The actual enrollment date is not available and is uncorrelated with the webpage backup frequency. It is assumed to occur between the time of the first appearance and the preceding snapshot. In case the first appearance is the first snapshot, the start date is not available. The end date is estimated to be sometime after the last appearance but before the next snapshot. The duration of the program for each student is calculated as the time elapsed between the estimated start and end dates, given both are available. If a student is currently listed on the program webpage it is assumed to be active and the end date is not available. We don't consider graduation rate in this module, however our approach can be extended to include this metric.

We estimate the average time-to-degree for the program by calculating the mean duration of all past students. Note, that this definition assumes that all students graduated, which may not be the case. By extending our model with the graduation information, this can be straightforwardly addressed. We don't include the current students in calculating the average time-to-degree, as recent students may introduce bias towards shorter durations.

We make several assumptions to estimate the enrollment duration and program completion:

- The student's name is listed on the program webpage at least once during their enrollment.
- The student's name is listed on the program webpage only while they are enrolled.
- The extracted current students did not graduate yet.
- A student's name is a unique identifier.
- Program and placement pages are up-to-date and accurate.
- The start (end) date are estimated as the date of the first (last) snapshot where the student is listed.

Since our aim is estimating cumulative enrollment and duration of the program, rather than individual student data, the assumptions are reasonable and do not significantly affect the overall accuracy of the analysis. Philosophy programs are typically accurate in listing current students and placement records, and the data is often updated regularly. Name collisions are rare.

Besides currently enrolled students, the web page may list faculty, alumni, or students who have already graduated. To address this, we design the search functions so that they target the currently active students specifically, filtering out other entities. With a minimal intervention, we can increase accuracy of the name search by manually validating the extracted names as a part of the generation pipeline. The frequency of snapshot captures varies per website. Websites in the "Worldwide Web Crawls" are included in a "crawl list", with the site archived once per crawl. A crawl can take months or even years to complete, depending on size. See Wikipedia[3] and Wayback Machine Help[4] for more information.

We propose a hybrid approach that combines the strengths of LLMs with programmatic pattern matching, and facilitates automating the process of extracting information from non-standardized web pages. By leveraging OpenAI's GPT API, we streamline generation of Python functions that extract names from web page sources, providing a scalable and adaptable solution for data collection, while maintaining robustness and reliability of programmatic extraction. The core of the module's methodology

[3] https://en.wikipedia.org/wiki/Wayback_Machine

[4] https://help.archive.org

is the use of OpenAI's GPT API to generate Python functions for name extraction. Using GPT-generated functions offers a non-deterministic solution providing means for automatic adaption to changes in the web page structure. This approach is agnostic to the specific structure of web pages, making it adaptable to various non-standardized sources. The generated functions are deterministic, providing robustness and reproducibility in the extraction process.

Using GPT for code generation, rather than relying on direct inference, mitigates the overheads associated with traditional NLP model approaches, which require iterative execution of resource-intensive models on extensive datasets. Additionally, pattern matching is computationally efficient. Generated code undergoes programmatic validation to ensure it meets the required functionality and correctness. To enhance robustness and reliability, users are prompted to verify the correctness of the generated code. If the generated function returns incorrect results, users can provide feedback to the model, facilitating continuous improvement. For each generated function, the user only needs to inspect the extracted names once to verify the correctness of the code. The pipeline can automatically adjust to changes in web page structures by re-running the generation process, eliminating the need for manual updates and reducing maintenance demands. By modifying the prompt, this approach can be easily adapted to a specific task. We can target specifically graduate students, or generally, students with specific traits. This is difficult to achieve using NLP models or generic pattern matching without hand-crafted rules.

2.1 Architecture

The module architecture is based on three main components: extracting webpage source of snapshots from the web archive, generating Python functions to extract names from the source, and building a database of student information based on extracted data. The list of Wayback Machine snapshot URLs is obtained with `snapshot_url.py` module. The `program_page.py` module handles fetching and processing data from snapshots of web pages. These two modules interact with the Wayback Machine API to retrieve the archived web pages.

The student information is retrieved using `searn_names.py` module which calls generated search modules to extract names from the source. The placement information is retrieved using the `placement_page.py` module which fetches and processes data from the placement pages. The extracted data is processed and stored in a structured format in the database by `database.py` module. The search modules are generated with `module_manager.py`. The module interacts with the OpenAI API to generate Python functions via `gpt_api.py` module. The validation of the generated functions is performed by verifying extracted names. This is implemented in `student_mame.py` module, which handles both programmatic validation and interaction with the user via command line interface.

The challenges arising related to non-determinism of GPT-based coding are handled via a feedback loop between validation and generation processes. The pipeline for code generation involves the following steps:

1. **Validation**: The `search_module` is validated using the `validate_function`.
   - If validation passes, data scraping from archive snapshots proceeds.
   - If validation fails, the code is generated using `generate_code`.
2. **Code Generation**: A function is generated to extract names from the web page source using `generate_code`.
   - The generated code is validated with `validate_function`.
   - If validation passes, the code is saved as a Python file.
   - If validation fails, the code is updated using `update_code` and revalidated.
3. **Update Code**: The code is updated based on the chat response and saved to a Python file.
   - The updated function is validated and saved to a Python file.
   - The process is repeated until the function is validated.

The extracted data is stored in a structured format in `public/data/student_data_v<version>.json` with the following schema:

```
dataset (name, university, url, start_date, end_date, active, years, snapshots, placement)
```

where:

- `name` (str): Full name, primary key of the dataset

- `university` (str): Host university
- `url` (str): Current graduate students page
- `placement_url` (str): Placements page
- `start_date` (str): Start date of the program
- `end_date` (str): End date of the program
- `active` (bool): Whether the student is currently enrolled
- `years` (int): Duration of the program in years
- `snapshots` (list): URLs of snapshots
- `placement` (bool): Whether the student has a placement

The primary program identifier is the domain name of the university.

After extracting the data, it is matched against the most recent database version to identify new entries. The new entries are then added to the database, and the database version is updated.

2.2 Implementation

The implementation of the module is built around efficient error handling, which plays a crucial role in the code generation and validation process. The program involves invoking machine-generated code, which may not always be correct or exhibit the desired behavior or structure. By utilizing exceptions, the architecture efficiently handles the non-deterministic nature of the system and facilitates seamless interaction between the program and the chat API. If a function fails validation, it is treated as an exception, and the error is propagated to the GPT API for context-aware function generation. This approach allows for unified handling of validation, module, and server errors within a single feedback interface. In the `exceptions.py` module, we define custom error classes: `ModuleError`, `ValidationError`, `OpenAIError`, and `WaybackMachineError` to categorize errors from different modules. `ModuleError` and `ValidationError` are redirected to the chat API to update the context for function generation. `OpenAIError` and `WaybackMachineError` are logged and displayed to the user.

The module integrates with the Wayback Machine to retrieve archived snapshots of web pages. The `snapshot_url.py` module first checks for the availability of snapshots for the given URL and then fetches and parses all available snapshots. A retry mechanism is implemented to handle temporary connection and server errors. In Wayback Machine web archive, snapshots of documents and resources are stored with time stamp URLs such as `20240615142741`. The date of the snapshot is encoded in the URL, allowing for easy retrieval of the snapshot date.

The interaction with the OpenAI API is managed by the `gpt_api.py` module via predefined prompts. The prompts for the GPT model in `prompts.yaml` are designed to generate a function that extracts all unique graduate student names from a web page source. It includes the function signature and a high-level description of the required functionality.

```
setup_prompt: |

  You are an expert in Python and web scraping using
    BeautifulSoup, HTML, regex and other libraries.

  You have to follow instructions and respond to prompts as precisely as possible.

generate_function_prompt: |

  Your task is to create a function that extracts
    a list of current graduate student names
    from an HTML source.
   The function should be named ‘extract_phd_student_names‘ and should take
    a BeautifulSoup object as input, returning a list of strings where each string
    is a current PhD student’s name.
   Make sure it only selects all current students.

  Analyze the following HTML source chunks to understand the structure of the web page:
```

```
  {html_chunks}

  Based on the structure and patterns identified in the provided HTML source chunks,
    write the ‘extract_phd_student_names‘ function.
    Make sure to include the necessary import statements.

  ### Expected Response

  def extract_phd_student_names(source: BeautifulSoup) -> list[str]:
      # Your code here

  This function accurately extracts the names of current PhD students
    based on the status specified and all information provided in the HTML source.

update_function_prompt: |

  Your task is to update the function ‘extract_phd_student_names‘
    to fix incorrect behavior.
  You must use matching patterns and HTML structures in the source.

  The function signature should be
    ‘extract_phd_student_names(source: BeautifulSoup) -> list[str]‘.
  In your response, provide only the code with the function
    and necessary import statements included.

  Names extracted by the previous implementation:
  {names}

  Error message:
  {error_message}

  I expect your response to include the following structure:

  from bs4 import BeautifulSoup

  def extract_phd_student_names(source: BeautifulSoup) -> list[str]:
      <your code here>
      return students

  This update ensures that the function accurately extracts
    the names of current PhD students.
```

The response from the GPT model is processed to extract the relevant code within code block markers. The code is then trimmed after the return statement to ensure functionality.

Some pages include pagination. The `program_page.py` module manages it by sequentially extracting names from all pages, using incremental page numbers in the pagination URL query strings (e.g., `?pg=1`). This process continues until no new pages are detected or the page content is empty. The raw HTML content is parsed using the BeautifulSoup library and then broken down into chunks. We randomly sample and present a subset of these chunks to the GPT model for analysis and pattern generation. Sampling has two benefits: it provides a representative view of the page structure and reduces the computational load on the model. In fact, the entire page source won't fit into the GPT model's token limit, so sampling is necessary.

The module processes the programs from `public/programs.csv` file and updates the database after each program scan with new records. In this way, the new data is saved incrementally, allowing for easy tracking of changes and updates. In case of errors, the module logs the error and continues with the next program, ensuring the scraping process is robust and fault-tolerant. Users can monitor the progress and status of the scraping process through the logs, validation output, and the database viewer. When needed, the module can be run for a specific program or a set of programs, allowing for targeted data collection and correction.

3 Data Viewer Application

For accessing and viewing the extracted data, a data viewer application is developed. This application allows users to query and visualize the dataset, providing insights into enrollment durations and placement rates. Grad Stats is a web application designed to provide an interactive interface for exploring and analyzing graduate program data. Users can search for universities, view detailed statistics on student data, and gain insights into various programs.

3.1 Features

The application includes a search functionality, allowing users to type the name of a university or program in the search bar. As users type, a dropdown list of matching universities appears, and they can use the arrow keys to navigate through this list, pressing enter to select a university. Clicking on a university name in the dropdown displays the relevant statistics. Additionally, if the search query is empty and the search bar is focused, a list of all available programs is displayed, offering automatic suggestions.

Grad Stats provides various data views through multiple tabs. The Overview tab offers a summary of graduate programs, including total entries, currently active students, and placement rates. Clicking on a program name in this tab reveals detailed statistics for that program. The Statistics tab displays placement rates and time-to-degree data through histograms, while the Program Summary tab gives a comprehensive overview of the selected program, detailing the number of currently enrolled students, total students recorded, placement rate, average time-to-degree, and more. The Students tab presents detailed student data, including names, enrollment dates, completion dates, time to degree, and placement status. Finally, the Snapshots tab provides links to snapshots of program data, showing the date of the snapshot and the number of students listed.

The data tables in Grad Stats support sorting functionality, enabling users to sort the data by clicking on column headers. This sorting can be done in ascending or descending order, with the current sort direction indicated by an arrow next to the column header. To help users understand the data better, informative tooltips appear when hovering over column headers, offering additional context about the data represented in each column.

Cross-view navigation is seamlessly integrated into the application. Clicking on a program name in the Programs tab or a university name in the search dropdown displays detailed statistics for that program or university, ensuring users can easily move between different views of the data.

3.2 Installation and Usage

To install Grad Stats, clone the repository and navigate to the project directory. Install the dependencies using `npm install` and start the development server with `npm start`. The application will be accessible at `http://localhost:3000`.

Using Grad Stats is straightforward. Start by typing the name of a university or program in the search bar or press enter to open the overview page. Navigate the dropdown list of universities using the arrow keys, selecting a university to view its detailed statistics. The application's tabs offer different views: the Graduate Programs tab displays a summary of various programs, the Summary tab provides detailed program statistics, the Student Data tab shows comprehensive student data, and the Snapshots tab links to program data snapshots.

Sorting the data is simple: click on the column headers in any table to sort by that column. Click once to sort in ascending order, and click again to sort in descending order, with an arrow indicating the sort direction. Tooltips on column headers provide additional information to help understand the data.

3.3 Data Flow

Grad Stats fetches data from external sources using the `fetchVersions` and `fetchStudentData` functions. The data is processed to compute program summaries and indexes with functions `computeProgramSummary` and `computeProgramIndex`. State management is handled using React hooks, including `useState`, `useEffect`, `useRef`, and `useCallback`. Based on the state, various components are rendered to display the data to the user, ensuring a smooth and interactive experience.

4 Results

The analysis focuses on two main features: the placement rate of graduates and the duration of their enrollment in the program.

The average duration in the program is computed by tracking the first and last appearance of student names in web page snapshots, thereby estimating start and end dates. This metric excludes currently active students to avoid skewing the results.

A quarter of the analysed programs have only one snapshot and therefore lack sufficient data to calculate the average duration. We exclude these programs from the cumulative statistics. The remaining programs have an average duration of 2 years and 2 months, with a standard deviation of approximately 10 months. This is less than the typical duration of a Philosophy graduate program, and can be attributed to students that have not completed the program. Indeed, examination of the individual student data reveals that many students were enrolled for short durations of less than a year, suggesting that they may have dropped out or transferred to other programs. To address this issue, we can extend the analysis to include graduation information, which would provide a more accurate estimate of the average time-to-degree, as well as the graduation and attrition rates. Furthermore, the average duration is underestimated due to the lock of data on past graduates who graduated before the first snapshot. Given that the archive history is limited to the last 5-10 years, and the typical duration of a Philosophy graduate program is 5-7 years, this is a source of significant bias.

The placement rate is calculated by matching student names against placement records. The average placement rate across all programs is 22%, with a standard deviation of 10%. Low placement rates are explained by inconsistent reporting practices in reporting non-academic placements, as well as the fact that we consider the cohort of all graduate students rather than targeting PhD students specifically. This can be straightforwardly addressed by selecting only PhD programs and tuning the search prompts. Furthermore, the placement rate is likely underestimated due to the lack of data on recent graduates, who may not have had time to secure placements. Given that the typically the archive history is limited to the last 5 years, many recent graduates are not included in the placement webpages, and at the same time many placed graduates are not accessible due to the lack of snapshots.

5 Conclusion

This project demonstrates the potential of combining LLMs with traditional web scraping techniques to automate and enhance the extraction of chronological data from web sources. Our approach provides an alternative way access the time-to-degree metric using information from the institutions hosting the academic programs without requesting this information directly. Furthermore, it provides an insight into the enrollment history for each program, the information that is often only available to the university administrations and partner organizations, based on only public sources. The main limitations of the method are due to a limited observation time and the lack of the completion information.