CS 410 Progress Report – Automatic Crawler of Faculty Pages

Net IDs: ap41, gangyao2, sg54

Group Name: Team Texas

Overview

This tool is designed to crawl a university web structure and find pages where faculty members are listed. It tries to accomplish this task by looking at the URLs and deciding how to classify those URLs.

The tool can be expanded upon for a variety of use cases involving a full repository of faculty across universities.

Some use case examples include:

- Users trying to establish a faculty census at a university.
- Students looking to contact people working on common areas of research.
- Organizations looking for people with the expertise to solve problems faced in the industry.

Design and Solution Approach

(How is the tool solving the problem?) What are the main components of the tool? How are those components implemented? (libraries)

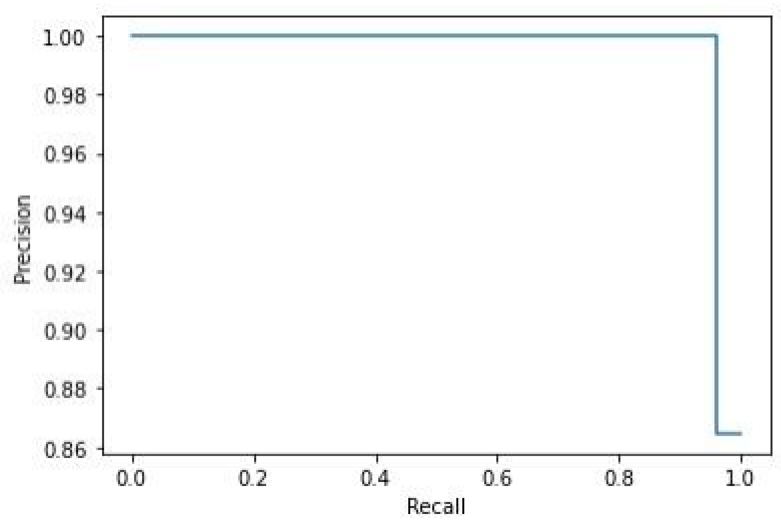
There are 3 main components in the tool:

1. An SVM classifier which makes decisions on URLs

   We implemented a URL based topic binary and URL content-based classification. We denoted 1 for faculty directory and 0 for non-faculty directory.

   In the beginning, we focused content-based classification as using just a URL provides very little information for classification. The basic idea is to scrape the website using the Beautiful Soup library and then extract all the text information from the site. Once the text data is extracted, we then use an Apriori Algorithm and set the Minimum Support to 50 to reduce the vocabulary size because each website has a lot of text and most of the vocabularies present are not useful for our decision. We built 340 training datasets with the correct labels for each URL and use these training datasets to train the SVM classification model. This model is pretty accurate as the input has enough information for classification. However, the problem of content-based classification is scalability and efficiency. First, the training time is pretty long because we need to scrape every URL for the training dataset and extract its text information. Second, the validation time is long as well because we need to scrape the testing dataset URL. Due to the performance problem of content-based classification, we switched to implementing URL topic classification. For this methodology, it doesn't need to scrape every URL and extract the web text information, so the runtime is very fast. The drawback of this method is that there is very little information in the URL itself. In order to resolve this issue, we can use supervised tokens and n-grams derived from the URL to generate the features for classification. For example, given the URL https://www.darden.virginia.edu/faculty-research/directory, after applying the feature algorithm with tokens and n-grams to this URL, we can get the following features: darden, virginia, edu, faculty, research and directory. If we don't use the n-grams technique, we will miss the "faculty" feature and make an incorrect prediction.

We built a SVM classification model based on this concept. We trained the SVM model with the Python sklearn library. Once the SVM model is trained, we use the pickle library to save the model in the local folder. And then we validated the model with another 229 testing datasets by loading the trained model. The average precision of the model is 0.96 and the precision/recall curve is shown here:



2. A web spider or crawler that feeds the URLs to the trained model. This component will produce the positively classified URLs.

   The crawler is using the Python ***scrapy framework***. Below there is a list of the main technical considerations:

   - A *LinkExtractor* that ignores most file extensions like pdf's, images, etc.

- A filter in the *LinkExtractor* is configured to only consider a domain and its subdomains. The crawler will not drift to other places on the internet.
- The Crawler can be switched from *BFO* to *DFO* exploration so we can measure the most effective way to get potential *URL* candidates.
- A configurable *depth-limit* for the crawling process. We estimate that given the main page of a university, the candidate *URLs* should be available at a relatively shallow level.
- There is a pipeline to process the URLs yielded by the Spider, concretely, it asks the classification model to decide on the URLs and produces csv files as output.

3. A GUI where the user can input a University URL and receive a list of faculty directory URLs for that University.

   The GUI primarily consists of a search bar, a results page, and the Flask client that receives the requests form the UI. The UI uses Node React for all the code as well as various NPM packages. The GUI is meant to be intuitive and very easy to use by any user, taking in a base web URL, and running the scraping/classification process in the background in order to get the faculty directory URLs as desired. The main web UI sends the URL request to the Flask client which then runs the scraping/classification process asynchronously. The UI then periodically polls the Flask client until the process is done. The Flask client sends a JSON object to the UI when the process is done, containing all of the faculty directory URLs which are then displayed to the user on the UI.

   The Flask client is deployed on the world-wide web using a service provided by Heroku, which enables us to host the main Flask code as well as the scraping/classification code on a server which can then be interacted with through the Flask endpoints.

   The main web UI is deployed on the world-wide web using the hosting service provided by GitHub called GitHub pages. This then connects to the Heroku server through the POST requests the web UI makes to the Flask client.

Deliverables

Tool is live on this URL: https://sxxgrc.github.io/faculty-scraper/

Demo video: https://github.com/alex-pi/CourseProject/blob/main/doc/UsageDemo.mp4

```
|  cli.py >> command line utility to use the faculty scraper
|  globals.py >> common python definitions
|  README.md
|
+---data
|  |  finalized_model.sav >> svm model data
|  |  output.zip >> output example
|  |  TrainingDataSetTest.csv
|  |  TrainingTestingDataSet.csv
|  |
|  \---output >> csv files will be generated here, one for positive cases, another for all URLs tested
|          .gitignore
|          README.md
|
+---doc
|      Automatic crawler of faculty pages - Project Proposal.pdf
|      ProgressReport.pdf
|      FinalReport.pdf
|      UsageDemo.mp4 >> Demonstration video
+---spiderbot
|  |  scrapy_spider.py
|  |
+---ui >> web content for GUI such as JS, html and css
|  +---public
|  \---src
|          faculty_scraper.js >> Main search page for UI, handles accessing the flask client and polling
|          results.js >> Main results page which displays the JSON result from the Flask client
+---urlclassification
|  |  url_classification.py >> model training and classification
```

Installation and Usage

Installation steps

In order to run the CLI locally:

- Make sure you have installed Pip locally for Python 3
- Run **pip install –r requirements.txt** to install all the libraries, modules, etc. needed to run the client
- Run something like **python cli.py illinois.edu 50** where 50 is the number of directory URLs to scrap

In order to run the Flask client locally:

- Make sure you have installed Pip locally for Python 3
- Run **pip install –r requirements.txt** to install all the libraries, modules, etc. needed to correctly run the client
- Run **python faculty_scraper_ui.py** which will run the client locally and deploy it to: **127.0.0.1:5000** which is the endpoint for POST requests to send a URL to the scraper/classification modules

To run the web GUI locally:

- Make sure you have **npm** installed locally on your system
- Make sure you are in the **ui** directory of the project
- Run **npm install** to automatically install all the packages required to run the UI locally
- Run **npm start** to start the UI locally, it should be located at **127.0.0.1:3000**

GUI Usage

The web GUI is meant to be used in the same way as a traditional search browser, with the user being able to submit a query in the search box and get results for it by hitting the search button or hitting Enter on the keyboard.

Specifically, the base UI page with the search button is meant to accept only university webpages, such that it will only accept URLs that end with **.edu**. The search box will also accept URLs that either begin with **http/https**, contain **www.**, or only contain the primary component of the URL.

For example, to get faculty directory URLs for UIUC, the user can input either: **https://www.illinois.edu**, **https://illinois.edu**, **www.illinois.edu**, or **illinois.edu**. Once this has been inputted, the user can begin the scraping process by hitting the **Enter** button on the keyboard or clicking on the search button. This will begin the main scraping process which will return results to the user in around 30 seconds to 2 minutes.

Once on the results page, the user can scroll through and see various faculty directory URLs, being able to click on each of them to access those webpages.

Making GUI Changes

Making GUI changes with respect to the scraping process can be done by modifying one of the following files: **faculty_scraper.js** in the ui/src directory, or **faculty_scraper_ui.py** in the base directory. Making GUI changes with respect to styling can be done by modifying one of the following files **faculty_scraper.css**, **results.js**, or **results.css**. We dive into each of these processes below.

The **faculty_scraper.js** file consists of the primary code for the search box in the webpage, handling what happens after the user submits a URL query. This manages the communication between the UI and the Flask client which connects to the primary scraping and classification modules. Changes to this file can be made in order to change the way in which the UI communicates with the scraping/classification modules, as well as what it does with the results.

The **faculty_scraper_ui.py** file contains the main Flask client code, servicing external POST requests by taking in the given URL and sending it to the primary scraping/classification modules. Modifying this file can be done in order to change the maximum amount of faculty directory URLs to scrape for, the way in which the URL given from the UI is treated, and the way in which the client interacts with the UI and the main modules.

The various css files and **results.js** deal primarily with how the GUI looks and feels, as well as how results from the user’s query are displayed. These files contain very granular control over how each part of the GUI works and feels, so modification of these files allows the user to have a lot of flexibility with the changes they want to make.

Future improvements

The tool is designed with the possibility of getting future improvements or extensions. Some ideas below:

- Add a second classifier based on content. This will decide on pages coming from the first URL classifier making.
- Leverage scrapy capabilities to save the crawling status for a university and resume later.
- Used the output URLs to implement entity extraction from the Biography pages.
- Apply the backing scraper/classifier in a more specific and useful way, such as being able to scrape multiple universities for faculty URLs related to some topic, or returning only the URLs to faculty websites for a given university (instead of directory URLs).
- An unexpected use from the crawler is to identify domains and links broken across the university pages. For instance, while testing, we discovered illinois.edu has much more death links than other universities.

Team participation

Gangyao2 – SVM Model training and data collection. Model evaluation. Participated in all documentation activities and technical discussions.

Sg54 – GUI design and creation. Integration with the crawler. Participated in all documentation activities and technical discussions.

Ap41 – Crawler creation and performance adjustments. Integrated classification model with the crawler component. Participated in all documentation activities and technical discussions.