selenium_navigation.py

1.0 Introduction

scraper.py is the scraper of the backend. While selenium_navigation.py finds the URLs to scrape data from, scraper.py actually retrieves the text from the job posts, and handles things like the proxy IPs and various error handling features.

2.0 Architectural and component-level design

1. Scraper.__init__(is_headless):
    a. First, set up the proxy server. The proxy server works by accessing the api of our proxy provider, ProxyRack.
        i. There are caveats to this, however. The system that the script is run on has to have its public IP manually whitelisted on the service, as selenium makes authentication for proxies incredibly difficult in its current form.
        ii. A proxy is necessary to avoid detection from LinkedIn’s bot detection. While web scraping LinkedIn is legal, (A judge ruled that since all information is publicly available, there is no violation of user privacy with the use of web scraping) LinkedIn makes its money from collecting and selling user data. As such, they do not like web scrapers collecting this data for free.
    b. We add the proxy to the options of the webdriver, along with whether or not the driver is headless.
    c. We determine the OS using platform.system(), and use the correct driver accordingly.
    d. We create the webdriver and return.
2. Scraper.get_page(url):
    a. The first thing we do is sleep for 1 to 5 seconds, to simulate user activity and avoid bot detection.

   b. As error catching for a rare error, (detailed in known bugs), we use a try and except block along with a loop to retry up to 5 times if the failure occurs.
   c. We try the Selenium get() function to set the driver to the given page.
   d. We update the current URL, sleep again, and return.
3. scraper.get_title()
   a. Returns the current URL
4. scraper.end_scraping()
   a. Closes the web driver.
5. scraper.scrape_linkedin()
   a. First, we check if the description exists in the HTML. If it doesn't, there are a few possible causes:
      i. There is an error with how the page was loaded, because a proxy IP was blocked, the user IP is not properly whitelisted, or the page loaded incorrectly for a myriad of other possible reasons. Alternatively, the user's internet could be out.
   b. If the description does not exist, we retry up to 5 times with a 1-59 second wait. This allows time for the IP in the proxy API to rotate in case the IP was a problem.
   c. After the retry, we check the description again. If it works, we continue. On the 6th attempt, we let Selenium throw the exception and abort the script.
      i. I do this because in general, if such a failure were to happen 5 consecutive times, it is indicative of a user or code related issue. It is possible but rare for it to happen otherwise. Over the course of a 12 hour stress test, it happened once in the middle of the night, likely due to an internet outage. If we were to simply skip to the next job entry, errors would potentially continue to crop up and waste time.
   d. After the description is confirmed to be readable, we look for the "show more" button in the HTML.
   e. First, we check if it exists, then we check if it can be clicked.

        i. There is a rare case where the button exists in the HTML, but can not be interacted with. This seems to be caused by short job descriptions, where the show more button does not exist

   f. We then click the show more button if it is interactable, retrieve the job description data, and return it.

        i. The reason we need to click the show more button is because if we don’t, only part of the job text is loaded.

6. scraper.check_element(element):
   a. A try and except block that tests if an element exists by attempting to use the find_element functionality of Selenium.
7. scraper.try_click(element):
   a. Same as the check_element() function, except that it tries to click instead of just finding the element.

3.0 Testing

1. Primary Testing Method: Due to the nature of the web scraper, conventional unit testing was not viable or useful. The web scraper gathers data from websites through a browser, and as such is expected to handle a non-finite number of possible inputs, in this case job descriptions. Scraping the same job description, or the same page of job descriptions, as a unit test would not prove or disprove functionality. As such, I decided to use full system tests with parameters that could be controlled from web_scraper.py.

4.0 Known Bugs

- Scraper.scrape_linkedin():
    - Very rarely, a page will have an error when the scraper tries to interact with the “show more” button on the job description. This does not cause the operation of the program to stop. The

user will be prompted to check liggs in the command line, this refers to ‘formatted.txt’ if the -o option is selected when the script is run. This is caused by the button existing in the HTML, but

    - There are rare circumstances under which the error handling for scraping a job description in Scraper.scrape_job() will fail, ie. exhaust it’s number of retries. This could happen for a few reasons:
        - The public IP of the system the program is run on is not whitelisted on the proxy service’s API.
        - The webdriver is failing (ie. not loading the proper page, or loading an error page)for an unknown reason.
        - Multiple proxies IPs that are banned are rotated into each other in sequence, leading to the number of retries not properly avoiding said bans.
        - The user’s internet is failing.

      Note that this error is fairly uncommon.
- Scraper.get_page(url):
    - ERR_TUNNEL_CONNECTION_FAILED is a webdriver failure that happens rarely when a URL is accessed with Selenium’s get_page() function. There is code in place to handle this error, however we can not cause it intentionally, and it does not occur at a frequency that allows us to test the error handling code. If this error does occur, and handling does not catch it, Selenium’s WebDriverException will be thrown.