گزارش پروژه ساخت ابر جست‌وجوگر

پژوهنده : رعنا آیرنگ

استاد راهنما: دکتر کیهانی پور

پس از طرح روش پیشنهادی و بیان مسائل تئوری و ساختاری در ساخت فراجویشگر در این فضل به بیان نحوه‌ی پیاده سازی این روش پرداخته شده است. این فصل با تشریح زبان برنامه نویسی مورد استفاده، کتابخانه‌های وارد شده و توابع ایجاد شده و به کار رفته، کلیات پیاده‌سازی روش پیشنهای را رائه می‌دهد.

در قسمت اول این پروژه ما نیاز به ساخت یک خزنده وب برای گردآوری داده‌های هر موتور جست‌وجو نیاز داریم. برای تولید خزنده‌ی وب (**Web Crawler**) به کمک پایتون باید از چند کتابخانه برای این کار استفاده کرد و قسمت های مختلف این کتابخانه‌ها را با هم ادغام کنیم تا نتیجه نهایی را به الگوریتم ترکیب نتایج ارجاع دهیم.

برای این پروژه ما از چهار موتور جست‌وجوی بومی زیر استفاده می‌کنیم:

ریسمون(**rismoon.ir**)

پارسی‌جو (**parsijoo.it**)

قطره(**www.ghatreh.com**)

پارسیک(**www.parseek.com**)

در اولین مرحله ما باید از یک کتابخانه به نام "Selenium" استفاده کنیم که این کتابخانه بر روی زبان‌های

پایتون و جاوا قابلیت استفاده دارد، کتابخانه‌ی selenium برای خودکارسازی تعامل‌های مرورگر وب استفاده می شود و برای ارسال و دریافت اطلاعات از یک سرور به یک کلاینت (و بالعکس) استفاده میشود.

این کتابخانه با باز کردن صفحه ای شبیه به مرورگر، به انجام فعالیت های تعیین شده در وب می پردازد در واقع کتابخانه مذکور یک کتابخانه برای شبیه‌سازی مرورگر وب و انجام فعالیت‌های اتوماتیک روی یک مرورگر ساخته شده است. این کتابخانه از نرم‌افزار هایی به نام webdriver برای شبیه‌سازی فضای مرورگر استفاده می‌کند.

این نرم‌افزارها (webdriver) در اصل موتور اصلی رندرینگ مرورگر هستند که توان پردازش و نمایش فایل های مرتبط با وب (html, css, javascript) را دارا هستند.

در سلنیوم هم صحنه معلومه هم پشت صحنه. یعنی یک صفحه مرورگر باز میشود(کروم، فایرفاکس و...)، آدرسی که شما دادید رو سرچ میکنه و کاری که بهش گفتید رو براتون انجام میده. شما همه این کار ها رو به چشم میبینید تا اگر اشتباهی پیش اومده بود، سریع متوجه شوید. البته که سلنیوم از requests کندتره. ولی قابلیت هایی سلنیوم به برنامه نویس میده خیلی کامل و کاربردی ست.

در سلنیوم این مرورگر ها که به ما اجازه سرچ میدن، driver نام دارند. وقتی میخواهیم driver هارو از کتابخانه سلنیوم فراخوانی کنیم، به این صورت مینویسیم:

```
from selenium import webdriver
```

در مرحله بعدی driver مون رو انتخاب میکنیم. یعنی انتخاب میکنیم که driver مان فایرفاکس باشد یا کروم یا ... که ما کروم رو انتخاب میکنیم:

```
url = "path/chromedriver.exe"
driver = webdriver.Chrome(url)
```

منظور ازurl ، آدرس فایل driver کروم هست.

کتابخانه سلنیوم قادر است یک مرورگر وب را برای تعامل با وب سایت مورد نظر ایجاد کند. برای کار با سلنیوم، باید به درایور مرورگر دسترسی داشته باشیم. بطور پیش فرض درایورهای Chrome، Firefox، Edgeو Safari موجود است.

کتابخانه سلنیوم مانند کتابخانه **Request** عمل میکند اما با این تفاوت که کتابخانه **Requser** با ارسال درخواست ها به صورت پنهان (یعنی ما اتفاقات را نمی بینیم)، فعالیت مورد نظر را انجام می دهد .

نمی شود به طور کلی گفت کدام یک از این دو کتابخانه بهتر هستند. چون هر یک در جایی کاربرد بیشتری دارند. ولی سعی می کنم، به طور کلی، ویژگی های مشترکشان را بررسی کنم.

سرعت :کتابخانه سلنیوم، پنجره ای مثل مرورگر اجرا می کند و همچنین همه چیز را نمایش می دهد (رندر می کند)، پس در نتیجه، سرعت آن کاهش میابد. پس اگر نیاز به برنامه ای پرسرعت دارید، کتابخانه Requests مناسب شما (و برنامه) است.

راحتی استفاده :درباره راحتی استفاده دو نکته مطرح است: راحتی نصب و راحتی کد نویسی. از لحاظ راحتی نصب و اجرای کتابخانه، Requests سیار مناسب است. زیرا فقط با یک دستور، می توانید آن را نصب و سپس به کدنویسی با آن بپردازید. ولی پس از نصب خود کتابخانه سلنیوم، باید وب درایور مرورگر مورد نظرتان را هم دانلود کنید. پس از این کار، باید در هر برنامه ای که با این کتابخانه می نویسید، آدرس محل قرار گیری این وب درایور را وارد کد ها کنید. نصب درایور ها کمی دردسر دارد. چون باید مناسب نسخه دقیق مروگرتان باشد. در نتیجه، در آپدیت کردن هم مشکلاتی ایجاد می شود. از لحاظ راحتی کد نویسی، تفاوت خاصی وجود ندارد.

کنترل آسان درخواست ها :در این ویژگی، کتابخانه سلنیوم، به دلیل نمایش اتفاقات به کاربر، پیروز است.

برای برقراری ارتباط selenium با مرورگر نیاز به یک هدایت کننده یا driver داریم که به آن web driver می گویند و نام این web driver برای مرورگر ما در اینجا geckodriver است.

برای استفاده از هر کدام می توانیم درایور مورد نظر را را دانلود و سپس در برنامه خود آدرس دهی کنیم.

برای درک بهتر تخصصی بودن این موتور های رندرینگ میتوان به این نکته اشاره کرد که موتور رندرینگ مرورگر کروم یک موتور تفسیر جاوااسکریپت جدا را در اختیار دارد و آن را با توجه به استاندارد های روز توسعه میدهد.

یکی از متداول‌ترین و پراستفاده‌ترین موتور های رندرینگ **gecko** نام دارد. این موتور جست‌وجو که توسط شرکت موزیلا توسعه پیدا می‌کند در بسیاری از مرورگر ها من جمله کروم و فایرفاکس مورد استفاده قرار می‌گیرد.

برای شروع به کار با **Selenium** از این موتور استفاده می‌کنیم. آدرس دیگر موتورهای قابل استفاده در این کتابخانه در مستندات این کتابخانه در این آدرس قرار دارد.

در اولین قدم برای گرفتن نتایج مرتبط با هر موضوع درون یک موتور جست‌وجو تگ <**input**> مربوط به سرچ را در صفحه پیدا می‌کنیم.

با استفاده از متد **input** ورودی مورد نظر را به واسطه **WebDriver** وارد تگ مورد نظر می‌کنیم. در مرحله بعد المنت مربوط به **Submit** کردن داده ورودی را در موتور جست‌وجو وارد می‌کنیم.

در این مرحله موتور جست‌وجو نتایج را به صورت یک فرم **html** به مرورگر ارسال می‌کند. بالطبع هر موتور جست‌وجویی با روش خود این نتایج را نمایش میدهد؛ این مسئله مقایسه نتایج را کمی دشوار میکند، در این مرحله ما برای دقیق تر کردن مقایسه نتایج جست‌وجو یک سریالازر طراحی کنیم که نتایج موتور جست‌وجوهای مختلف را در قالب داده های **JSON** خروجی بگیریم.

و این نتایج در مراحل آتی میتوان بصورت موقت یا دائمی درون پایگاه داده ذخیره کرد تا برای پردازش در مراحل آتی مورد استفاده قرار گیرد.

طبق گفته های قسمت های قبل تفاوت مرورگر headless با سایر مرورگرها تنها نداشتن رابط کاربری گرافیکی یا GUI است، تمامی کارهای یک مرورگر گرافیکی مثل render کردن HTML به DOM (مهمترین اصل برای اتوماسیون وب) و تفسیر کدهای JavaScript را انجام میدهد. علاوه بر اتوماسیون وب این مرورگرها استفاده های دیگری مثل تست کارایی شبکه یا حمله های DDoS دارند که همگی درواقع برپایه تکرار ارسال درخواست http و render پاسخ هاست، مهمترین و شاید تنها مزیت آنها سرعتشان است.

مرورگر بدون سر یک مرورگر وب بدون رابط کاربری گرافیکی است.

مرورگرهای بدون سر امکان کنترل خودکار یک صفحه وب را در محیطی مشابه مرورگرهای محبوب وب فراهم می کنند ، اما از طریق رابط خط فرمان یا با استفاده از ارتباطات شبکه اجرا می شوند.

آنها به ویژه برای آزمایش صفحات وب بسیار مفید هستند زیرا قادرند HTML را به همان روشی که مرورگر ارائه می دهد ، درک کنند ، Headless به معنای یک مرورگر وب بدون رابط کاربری است. برای توضیح بیشتر ، مرورگرهای Headless مرورگرهایی هستند هستند که در واقع به صفحه وب دسترسی پیدا می کنند ، اما رابط کاربری گرافیکی از نظر کاربر پنهان است.

برای ساخت یک مروگر chromium بصورت headless در کتابخانه سلنیوم از روش زیر داخل کد پایتون استفاده می‌کنیم. یکی از اهداف عمده کرومیوم، علاوه بر ساخت مرورگری با مدیریت پنجره به صورتtab ، ایجاد بستری برای برنامه‌های تحت وب است که آن را با یک مرورگر اینترنتی مرسوم متفاوت کرده‌است.

```
from selenium.webdriver.chrome.service import Service
chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument('--headless')
```

تنظیمات فوق فایل باینری chromedriver را که در مسیر جاری قرار دارد را بصورت headless اجرا میکند.

متد Service در نسخه های جدید سلنیوم برای اجرای وب‌درایور بدون ورود مسیر خود chromedriver بصورت مستقیم در webdriver مورد استفاده قرار می‌گیرد.

ساخت درایور کرومیوم بصورت زیر صورت می‌گیرد:

```
driver = webdriver.Chrome(service=Service("./chromedriver"),
 options=chrome_options)
```

در ادامه هر کدام از موتورجست‌وجو های ایرانی داده های مورد نظر را دریافت میکنیم.

در اولین مرحله ساختاری که موتور های جست‌وجو به واسطه آن پرسش را پردازش میکنند را پیدا می‌کنیم و تابعی برای ساخت پرسش دلخواه مینویسیم، ساختار تابع برای موتور جست‌وجو های مذکور به شکل زیر است :

پارسی‌جو

```
def url_generator(query: str, page: int):
 query = urllib.parse.quote_plus(query)
url=f"http://parsijoo.ir/web?q={query}&period=all&filetype=any&site=&c
o={(page-1)*10}"
return url
```

ریسمون

```
def url_generator(query: str, page: int):
query = urllib.parse.quote_plus(query)
```

```
url=f"http://www.rismoon.com/search-
fa.html?tab=web&SelectedCategory=00&searchLang=1065&?GroupBySite=yes&n
p={page- 1}&q={query}&t="
return url
```

قطره

```
 def url_generator(query: str, page: int):
 query = urllib.parse.quote_plus(query)
 url = f“https://www.ghatreh.com/news/?q={query}&page={page-1}”
 return url
```

پارسیک

```
def url_generator(query: str, page=0):
 query = urllib.parse.quote_plus(query)
url=““https://www.parseek.com/Search/?q={query}&type=Latest”  ”
 return url
```

ساختار آدرس های بالا با بررسی موتورهای جست‌وجوی داخلی و ساختار صفحه های سرچ آنها بوجود آمده است.

در بخش بعدی source صفحات مورد نظر را از وب‌درایور گرفته و با استفاده از کتابخانه BeautifulSoup این سورس این صفحات که به زبان html نوشته شده را تجزیه می‌کنیم و داده های مورد نظر را با تجزیه ساختار html پیدا می‌کنیم و آنها را در قالب داده های پایتون بسته‌بندی کرده و به بقیه اجزای برنامه برمی گردانیم.

این کتابخانه بمنظور تجزیه کردن یا پارس کردن صفحات وب با پایتون(فایل‌های HTML) و همچنین جهت تجزیه کردن فایل‌های XML می‌توان استفاده کرد.این کتابخانه صفحات موردنظر خود را بصورت یک درخت تجزیه می‌کند که درخت تجزیه این امکان را برای برنامه ایجاد میکند که هرگونه دسترسی به عناصر صفحه html با سرعت بیشتری امکان‌پذیر گردد. با این روش شرایط مناسبی برای جستجوی اطلاعات موردنظر فراهم می‌شود.

حال در ادامه ما به فرآیند بهینه سازی نتایج و روش ترکیب آنها میپردازیم.

در بخش قبلی در مورد Serializer هایی که با آنها نتایج جست و جو گر های مختلف را یکپارچه می‌کنیم صحبت کردیم. در این بخش در مورد ساختار فنی این فرآیند صحبت میکنیم و آنرا بیشتر باز می‌کنیم.

در هر یک از خروجی ها یک داده JSON به شکل زیر وجود دارد.

```
{“id”: uuid.uuid4,"page_title”: “title”, “page_url”: “https:/”...” , “description”: “...“}
```

این نتیجه باید با تعدادی معقول باشد تا بتوان با منابع محدود پروژه مدنظر ما را در محیط تست اجرا شود.

برای این مهم ما میتوانیم تا حدی از ویژگی موتورهای جست‌وجو را در نظر بگیریم. در این روش ما خروجی را صفحه به صفحه از موتورهای جست‌وجو دریافت می‌کنیم. و این نتایج را با هم مقایسه می‌کنیم.

روش دریافت کد منبع هر نتیجه میتواند متفاوت باشد، اگر بخواهیم با استفاده از ساختار کتابخانه requests این کد منبع را دریافت بکنیم ممکن است توسط سرویس های anti-ddos خروجی اشتباهی وارد برنامه کنیم.

پس در این بخش هم مثل بخش های گذشته از selenium برای پیدا کردن کد منبع صفحه مورد نظر استفاده می‌کنیم تا احتمال برخورد با سرویس های این‌چنینی تا حد مقدور کاهش پیدا کند.

ساختار خروجی :

داده هایی که به شکل داده های پایتونی بسته بندی شده اند بر اساس گزارش های قبلی ساختاری مشابه ساختار زیر را دارند:

در مرحله بعدی از داخل این داده ها صفحه موجود در آدرس هر داده را باز می‌کنیم و مشابه ساختار بالا یک آبجکت soup از داده های html داخل صفحات می‌سازیم.

در این بخش متن داخل صفحات که عموما داخل تگ های heading یا تگ p قرار دارد را جدا می‌کنیم و آن را پردازش می‌کنیم.

فرآیند پردازش متن داخل تگ ها بصورت زیر است :

در اولین مرحله تگ های موجود داخل تگ اصلی که ساختار غیر متنی دارند را با یک عبارت باقاعده (regex) پیدا میکنیم.رجکسی که این تگ ها را پیدا می‌کند به شرح زیر است :

```
re.findall(r"(<.+?>)", tag_content)
```

این ساختار همه تگ های احتمالی داخل متن (a, table, tbody, thead, td, tr, b و ...) را داخل متن پیدا می‌کند و به ما اجازه می‌دهد این تگ ها را از داخل محتویاتی که از تگ مذکور گرفته ایم خارج نماییم و به محتوای متنی نزدیک تر شویم.

در مرحله بعدی کاراکترهایی که فرآیند پردازش متن را با دشواری روبه‌رو می‌کنند نیز از داخل محتویات شناسایی کرده و آنها را از داخل متن حذف می‌کنیم.

در مرحله سوم در قسمت هایی که احتمال دارد کارکتر غیر اسکی به کاراکتر اسکی یا عدد به هر کارکتری چسبیده باشد فاصله ایجاد می‌کنیم

متن حاصل را با کمک تابع داخلی ()strip بر اساس فاصله ها جدا می‌کنیم تا کلمات هر صفحه ها بتوانیم جدا کنیم.

در این مرحله ما داخل کد منبع سایت هایی که از موتور جست‌وجو گرفته ایم الگوریتم را اعمال میکنیم.

این الگوریتم چند ساختار مختلف را درون کد سایت جست‌وجو می‌کند که شامل موارد زیر میشود :

تک کلمه (term)

سند (document) ، سند شامل یک مجموعه از چند کلمه می‌شود

تعداد تکرار (N) این بخش مقدار تکرار مجموعه نوشته ها از داخل کد منبع را پیدا میکند.

مجموعه کل نوشته ها (corpus) که شامل کل متن کد منبع میباشد

پس از پردازش نتیجه بالا برای هر یک از نتایج موتور جست‌وجو های داخلی بر اساس کلمات داخل هر صفحه از نتایج الگوریتم tfidf را روی نتایج اعمال میکنیم تا نتایج نزدیک تر به پرسش جست‌وجو یافت شوند.

مقدارِ TF-IDFمخفف دو کلمه است TF :به معنیِ Term Frequencyیعنی تعداد تکرار یک کلمه در یک متن و عبارت IDFبه معنیِ Inverse Document Frequencyکه می‌توان آن را به برعکسِ تعداد تکرار در متون ترجمه کرد.

TF-IDFکه مخفف Term Frequency -Inverse Document Frequency می‌باشد و به معنای فراوانی وزنی کلمه کلیدی است TF-IDF .صرفا میزان تکرار یک کلمه کلیدی یا عبارت را در صفحه نشان نمی‌دهد، بلکه هدف آن نشان دادن اهمیت کلمه کلیدی مورد نظر از طریق مقایسه تعداد تکرار کلمه در متن با تکرار آن کلمه در مجموعه ای بزرگ تر از مستندات (کل صفحات سایت) می‌باشد.

برای اجرای الگوریتم فوق در اولین مرحله لیست کلمات مشترک در بین لیست های کلمات نتایجی که پیش تر پیدا کرده ایم را تشکیل می‌دهیم و با کمک آن فرآیند های TF, IDF, TFIDF را اجرا می‌کنیم.

برای اجرای الگوریتم TF مقدار تکرار هر کلمه را پیدا می‌کنیم و آن را داخل متغیر num_of_words از نوع dictionary قرار می‌دهیم

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

با توجه به رابطه بالا تابع terms frequency را اجرا می‌کنیم.

برای اجرای الگوریتم IDF بعد از پردازش تمامی TF های داده های موجود متغیر IDF را با رابطه زیر می‌سازیم و آنرا مقدار دهی می‌کنیم.

این مقدار برابر است با Inverse Data Frequency و از رابطه زیر به دست می‌آید :

$$idf(w) = log(\frac{N}{df_t})$$

در آخرین مرحله با توجه به مقادیر بالا تابعی که از خروجی دو تابع بالا استفاده می‌کند را صدا می‌کنیم و آن را اجرا می‌کنیم که از رابطه زیر استفاده می‌کند :

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

بعد از اتمام مراحل بالا داده هایی که به دست آورده ایم را بر اساس نزدیکی به ورودی جست‌وجو مرتب کرده و به واسطه اپلیکیشن وبی که با فلسک نوشته شده به کاربر نمایش می‌دهیم.

در مرحله بعد برای پیاده سازی الگوریتم tf-idf کد منبع صفحاتی که از موتور جست‌وجو پیدا کردیم را درون یک ساختار مشخص داخل یک دیکشنری نگه می‌داریم.

```
{"#uuid”: page_source}
```

در این مرحله ما داخل کد منبع سایت هایی که از موتور جست‌وجو گرفته ایم الگوریتم را اعمال میکنیم.

این الگوریتم چند ساختار مختلف را درون کد سایت جست‌وجو می‌کند که شامل موارد زیر میشود :

تک کلمه (term)

سند (document) ، سند شامل یک مجموعه از چند کلمه می‌شود

تعداد تکرار (N) این بخش مقدار تکرار مجموعه نوشته ها از داخل کد منبع را پیدا میکند.

مجموعه کل نوشته ها (corpus) که شامل کل متن کد منبع میباشد