Università degli studi Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di laurea in Ingegneria Informatica

Sviluppo di un crawler per l'estrazione di articoli e citazioni da Google Scholar

Relatore

Luca Gagliardelli

Correlatore

Giovanni Simonini

Candidata

Ferrari Chiara

Anno Accademico

2021/2022

Dedica

Un grazie di cuore a tutti coloro che mi hanno supportato e sopportato in tutti questi anni nonostante tutte le difficoltà

Sommario

1. INTRODUZIONE

Al giorno d'oggi esistono numerosi siti web contenenti enormi quantità di dati. Per questa ragione sono stati sviluppati nel tempo strumenti in grado di accedere a tali siti in modo automatizzato e raccogliere i dati in essi contenuti.

E' diventato perciò importante imparare ad usare questi strumenti per poter riuscire a estrarre dati da un sito web in modo ordinato e eventualmente poi confrontarli con quelli raccolti da altri siti.

Questa tesina ha lo scopo di spiegare come realizzare un programma in grado di estrarre dati dal sito web Google Scholar mostrando il procedimento seguito per ottenere tale risultato, mettendo soprattutto in evidenza i gli ostacoli incontrati e le loro possibili soluzioni.

2. DESCRIZIONE PROGETTO

2.1 Cos'è Google Scholar

Google Scholar è un potente motore di ricerca accessibile liberamente tramite il quale è possibile reperire testi di natura accademica come articoli scientifici e materiali universitari provenienti da differenti autori e case editrici.

2.2 Cos'è il Web Scraping

Per scraping si intende una tecnica con la quale un programma informatico estrae dati dall'output generato da un'altro programma. Per web Scraping si intende dunque quando tale procedimento viene utilizzato per estrarre dati da un sito web.

2.3 Cos'è un crawler

Un crawler detto anche spider o bot è un software in grado di raccogliere dati dal web. Esso è in grado di rilevare il contenuto di un sito e tutti i suoi collegamenti interni ed esterni e memorizzarli all'interno di un database.

2.4 Scopo del programma

Il programma ha lo scopo di reperire dal sito Google Scholar le informazioni relative agli articoli pubblicati da un determinato autore e salvarle in modo ordinato su di un file in formato .csv (Excel) ed in seguito fare lo stesso con tutti gli articoli che citano ciascuna pubblicazione di tale autore.

2.5 Possibile utilizzo del programma

Originariamente il programma è stato realizzato come parte di un più ampio progetto volto ad estrarre dati da due siti Google Scholar e Scopus. I dati estratti dai due siti potranno poi essere inseriti in un database e confrontati con lo scopo di verificare se tali dati coincidono (essendo Scopus un sito più recente ci si aspetterebbe che a questo manchino alcuni dati)

Il programma realizzato si occupa della prima parte del progetto cioè l'estrazione dei dati da Google Scholar e il salvataggio di tali dati su di un file .csv.

2.6 Linguaggio Utilizzato per realizzare il programma

Il programma è scritto in linguaggio Python, e nello specifico si è scelto l'utilizzo di Anaconda, una distribuzione open Source dei linguaggi di programmazione Python e R.

2.7 Programmi Utilizzati

-Python:

è un linguaggio di programmazione dalla sintassi semplice e potente adatto per innumerevoli applicazioni (applicazioni web, applicazioni di calcolo, grafica..)

-Anaconda

è una piattaforma che permette di gestire programmi scritti in Python.

-Scrapy:

è un framework open source scritto in Python utilizzato per il crawling di siti web allo scopo di estrarne dati. E uno strumento utilizzato originariamente per il web scraping, ma può essere utilizzato anche per estrarre dati utilizzando delle API oppure come web-crawler general-purpose

-Selenium:

è un tool open source volto a supportare l'automazione dei browser, utilizzato principalmente per il web testing . Per Selenium si intende una suite di più strumenti composto da Selenium IDE, Selenium Builder, Selenium Webdriver, e Selenium Grid.

In particolare in questo programma verrà utilizzata la componete Selenium Webdriver

-Chromedriver:

è un Webdriver utilizzato per Chrome. Nello specifico il programma utilizzerà Chromedriver mediante Selenium WebDriver uno strumento di Selenium in grado di simulare il comportamento di un utente all'interno di un browser utilizzato solitamente per eseguire localmente o su macchine remote i test all'interno dei browser supportati.

Chrome non è il solo a supportare Webdriver quindi è possibile sostituire Chromedriver con un qualsiasi browser supportato come FirefoxDriver per esempio.

Per il corretto funzionamento del programma è necessaria l'installazione di tutti i programmi sopra elencati ed è fondamentale che il Webdriver che si desidera utilizzare sia presente all'interno della directory in cui è contenuto il programma e che questo sia di una versione compatibile con il proprio sistema operativo.

2.7.1 Cos'è uno User Agent

Uno User Agent è una stringa di testo utilizzata dal web server per identificare il web browser e il sistema operativo utilizzato per connettersi ad un sito.

Quando un browser si connette ad un sito lo User Agent è parte dell'header HTTP inviato al sito web.

2.7.2 Perché utilizzare uno User Agent

Scrapy per il crawlig di un sito utilizza uno User Agent di default identificandosi come Scrapy Bot come conseguenza è possibile che alcuni siti web riconoscendolo lo blocchino impedendo al programma di accedere al sito. Google Scholar uno di questi perciò è stato dunque necessario sostituire lo User Agent di default con altri User Agent che invece sono in grado di accedervi.

2.8 Configurazioni:

Quando viene creato un nuovo progetto in Scrapy questo genererà automaticamente alcuni file di configurazione:

-__init__.py

-items.py :file di definizione degli Items

-middleware.py :file di definizione delle pipeline middleware

-pipelines.py:file di definizione delle le pipeline

-settings.py: file di definizine dei settings

Per il corretto funzionamento del programma è stato necessario aggiungere un ulteriore file denominato:

- utils.py

All'interno di tale file sono stati inseriti:

- una lista di User Agent

- la definizione della funzione get_random_agent() che restituisce in modo casuale uno tra gli User Agent elencati.

Fatto ciò si è aggiunto al file settings.py l'utilizzo della funzione get_random_agent() che va a definire come User Agent uno tra quelli elencati in utils.py .

Esempio:

-file "utils.py"

```
import random
user_agent_list = [
# Chrome
```

```
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)  Chrome/60.0.3112.113 Safari/537.36',
        'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.90 Safari/537.36',
    ]
    def get_random_agent():
        return random.choice(user_agent_list)
```

- file "settings.py"

```
    from tesina.utils import get_random_agent
    USER_AGENT = get_random_agent()
```

2.8.1 Cos'è uno User Agent

Uno User Agent è una stringa di testo utilizzata dal web server per identificare il web browser e il sistema operativo utilizzato per connettersi ad un sito.

Quando un browser si connette ad un sito lo User Agent è parte dell'header HTTP inviato al sito web.

I siti web utilizzano gli User Agent per identificare da dove viene la richiesta di accesso, e adeguare di conseguenza la versione a cui l'utente si interfaccerà, inoltre possono essere utilizzati per identificare e bloccare richieste di accesso da parte di User Agent indesiderati.

2.8.2 Perchè utilizzare uno User Agent

Scrapy per il crawlig di un sito utilizza uno User Agent di default identificandosi come Scrapy Bot che però viene considerato da alcuni siti come indesiderato e dunque viene bloccato. Google Scholar è uno di questi per questa ragione è stato dunque necessario sostituire lo User Agent di default con altri User Agent che invece sono in grado di accedervi.

3. IL PROGRAMMA

3.1 Inizializzazione

In primo luogo è stato necessario dare un nome al progetto, che verrà utilizzato per l'esecuzione dello script in secondo luogo è poi necessario indicare l'URL della pagina di un autore su Google Scholar da cui si desidera cominciare ad estrarre i dati

Esempio:

-inizilizzazione URL e nome script

```
name = "tesi12"
start_urls =[
'https://scholar.google.com/citations?user=qKNoouoAAAAJ&hl=it&oi=ao',]
```

Questo è l'esempio di un url utilizzabile, eventualmente è possibile indicare come url di partenza anche più url contemporaneamente anziché uno solo.

3.2 Estrazione dati

La pagina di un autore contiene una numerosa serie di dati.

Affinché Scrapy possa estrarre i dati richiesti, è necessario che questo conosca dove andarli a cercare.

Per fare ciò è stato necessario ispezionare la pagina web da cui si desidera estrarre i dati selezionando con il tasto destro del mouse sulla pagina la voce "Ispeziona".

Facendo ciò si ha accesso alla pagina scritta in codice Html e a questo punto è sufficiente scoprire sotto quale tag del codice Html della pagina sono situati i dati di interesse ed utilizzare una funzione di Scrapy per estrarli.

Esempio:

-La funzione estrae dalla pagina il titolo di tutte le pubblicazioni dell'autore.

```
for quote in sel.css('tr.gsc_a_tr'):
    titolo = quote.css('td.gsc_a_t a::text').get()
```

3.2.1 Selettori

Scrapy utilizza per l'estrazione dei dati degli elementi specifici. Questi elementi sono chiamati selettori perché selezionano alcune parti del codice HTML mediante l'utilizzo di XPath o CSS.

-XPath è un linguaggio utilizzato per selezionare nodi nei documenti XML, ma che può essere utilizzato con HTML.

-CSS è un linguaggio per applicare stili in un documento HTML. Definisce selettori per associare questi stili con specifici elementi HTML.

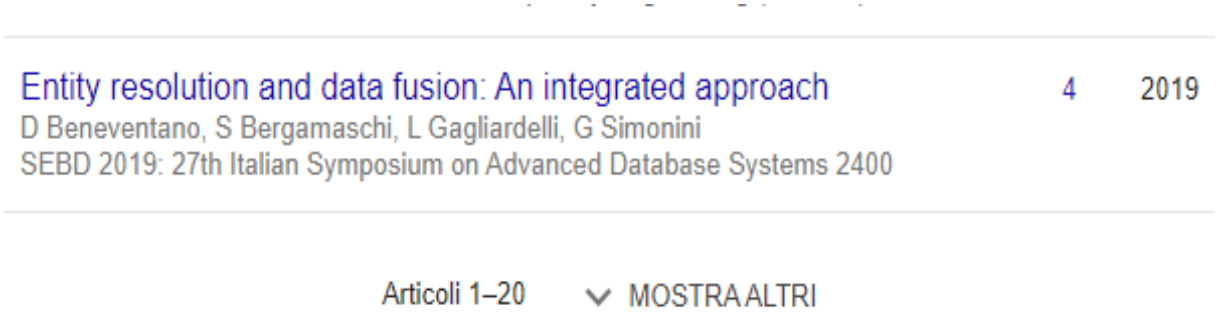
3.3 Load More

la pagina iniziale contiene alcuni dati che richiedono l'interazione dell'utente per essere caricati. (per esempio solo le prime 20 pubblicazioni

di un autore sono caricate inizialmente nella pagina a per caricare le restanti è necessario selezionare il pulsante "MOSTRA ALTRI"

Esempio:

- qui viene mostrato come si presenta la fine della pagina delle pubblicazioni prima della selezione del pulsante "MOSTRA ALTRI"



Poiché Scrapy non è in grado di eseguire questo tipo di azione è stato necessario affiancargli un ulteriore strumento. Per questo programma è stato dunque deciso di utilizzare il Selenium Webdriver il quale è in grado di simulare il click di un utente al fine di caricare tutte le pubblicazioni

Esempio:

-parte del codice html in cui è situato il bottone "MOSTRA ALTRI"

```
▼<div id="gsc_bpf">
  ▼<button type="button" id="gsc_bpf_more" class="gs_btnPD gs_in_ib gs_btn_flat gs_btn_lrge gs_btn_lsu">
    ▼<span class="gs_wr"> == $0
        ::before
        <span class="gs_ico"></span>
        <span class="gs_lbl">Mostra altri</span>
```

-codice Python per simulare il click su "MOSTRA ALTRI":

```
driver = webdriver.Chrome()
driver.get(response.url)
button=driver.find_element_by_xpath("//button[@id='gsc_bpf_more']")
button.click()
```

PROBLEMA:

seppure l'utilizzo di Selenium Webdriver fosse corretto e il driver simulasse il click da parte dell'utente, Scrapy continuava a scaricare solamente i dati delle prime 20 pubblicazioni ignorando le azioni del driver . Ciò è dovuto al fatto che inizialmente veniva utilizzato come riferimento del selettore la pagina caricata ad inizio funzione e che dunque non comprendeva ancora i dati caricati a seguito del click su "MOSTRA ALTRI"

SOLUZIONE:

Per risolvere il problema è stato necessario creare un selettore che facesse riferimento alla pagina caricata dal driver anziché la pagina caricata all'inizio della funzione.

Esempio:

-PRIMA:

```
for quote in response.css('tr.gsc_a_tr'):
    titolo = quote.css('td.gsc_a_t a::text').get()
```

-DOPO:

```
sel = Selector(text=driver.page_source)
```

```
for quote in response.css('tr.gsc_a_tr'):
        titolo = quote.css('td.gsc_a_t a::text').get()
```

3.4 Memorizzazione dati pubblicazioni

In seguito all'estrazione dei dati è necessario salvarli all'interno di un file Excel (formato .csv)

Esistono almeno due metodi per fare ciò:

-il primo metodo consiste nel modificare il file "settings.py" di Scrapy e aggiungere alcune linee di codice che indichino in che formato e dove salvare i dati estratti dal programma e poi indicare quali sono i dati che si vogliono estrarre utilizzando yield, una keyword mediante la quale è possibile salvare e memorizzare i dati estratti da Scrapy .

Esempio:

-settings.py

```
# Desired file format
FEED_FORMAT = "csv"
# Name of the file where data extracted is stored
FEED_URI = "tesina.csv"
```

-script: esempio memorizzazione dati con yield

```
titolo = quote.css('td.gsc_a_t a::text').get()
yield {
```

'titolo' : titolo,

}

L'aspetto negativo di questo metodo è che seppure funzionante non vi è libertà nel decidere come i dati debbano memorizzati. E' inoltre possibile generare solo un unico file .csv contenente tutti i dati senza poterli quindi memorizzare su file separati. In conclusione ciò porta ad un risultato visivamente poco ordinato e poco chiaro da leggere nel caso in cui si volesse visionare il file in cui sono stati salvati i dati.

Esempio:

-file .csv generato utilizzando il primo metodo

1	Authors,N_Citations,Title,Year
2	G Simonini, S Bergamaschi, HV Jagadish,89,BLAST: a loosely schema-aware meta-blocking approach for entity resolution,2016
3	F Benedetti, D Beneventano, S Bergamaschi, G Simonini,65,Computing inter-document similarity with context semantic analysis,2019
4	G Simonini, G Papadakis, T Palpanas, S Bergamaschi,53,Schema-agnostic Progressive Entity Resolution,2018

-Il secondo metodo consiste invece nel generare il file .csv direttamente nel programma così da potersi assicurare che vengano memorizzati in modo più ordinato. E' inoltre possibile generare file separati per ciascuna entità (pubblicazioni e citazioni in questo caso) di cui si vogliono raccogliere i dati, anziché un file unico che le raccolga tutte.

L'organizzazione del file consiste nel creare una riga iniziale disponendo in ogni cella Excel un diverso attributo (titolo, autore, ecc) e facendo poi in modo che i dati di ciascuna pubblicazione/citazione siano poi disposti nella cella coincidente all'attributo corrispondente. Per fare ciò è sufficiente aggiungere dopo ciascun attributo il separatore ";" , un simbolo

che Excel interpreta come segnale di inserire il dato successivo in una nuova cella.

In questo modo il risultato è molto più chiaro e ordinato ed è dunque il metodo consigliato.

Esempio:

-file .csv generato utilizzando il secondo metodo

1	titolo	autori	anno	citazioni
2	BLAST: a loosely schema-aware meta-blocking approach for entity resolution	G Simonini- S Bergamaschi- HV Jagadish	2016	91
3	Computing inter-document similarity with context semantic analysis	F Benedetti- D Beneventano- S Bergamaschi- G Simonini	2019	66
4	Schema-agnostic Progressive Entity Resolution	G Simonini- G Papadakis- T Palpanas- S Bergamaschi	2018	55

PROBLEMA:

Il problema in cui è possibile incorrere durante questa fase è dovuta ad una scorretta interpretazione da parte di Excel del simbolo "," presente talvolta all'interno dei dati che si desidera estrarre (ad esempio quando sono presenti più autori questi sono separati da ","), che come conseguenza può portare ad un'erronea separazione degli attributi di ciascuna entità.

Per questa ragione è stato necessario rimuovere tale simbolo e sostituirlo con un altro ("-" per esempio) utilizzando la funzione replace() di Python per ovviare a tale situazione, prima di salvare i dati nel file.

Esempio:

```
cit=sel2.css('div.gs_r a::text').get()
if cit is None :
```

```
        cit = 'None'
    cit2=cit.replace(',', '-')
```

Osservazioni:

Seppure il secondo metodo sia quello più ordinato e dunque quello più adatto in questo specifico caso, potrebbe non essere però quello consigliato da un punto di vista informatico poiché mentre il primo metodo è facilmente modificabile qualora si desiderasse cambiare il formato o il nome del file in cui si desidera salvare i dati (per esempio si possono salvare i dati anche in formato .json o .txt), modificando semplicemente il file "settings.py", il secondo invece richiede di modificare lo script del programma cosa che ha maggiori probabilità di generare bug .

3.5 Estrazione delle Citazioni

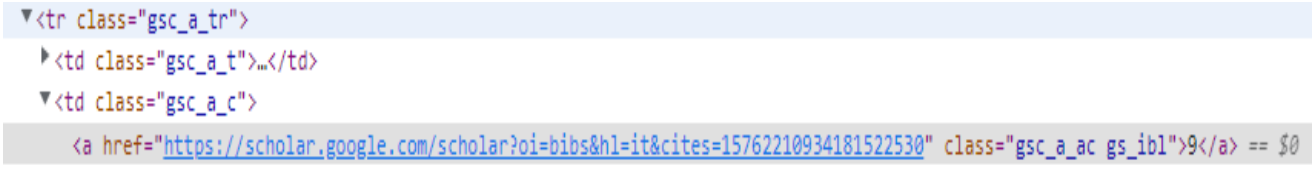
La seconda fase del programma consiste nell'estrazione dei dati che riguardano le citazioni.

Per prima cosa è necessario ordinare a Scrapy di utilizzare i link contenuti nella pagina iniziale per raggiungere le pagine contenenti i dati relativi alle citazioni.

Per fare ciò è sufficiente utilizzare il metodo precedentemente usato per identificare dove nel codice html della pagina sono situati i link alle pagine delle citazioni, estrarre tali dati e ordinare a Scrapy di seguire i link utilizzando la funzione follow().

Esempio:

-codice html da cui è possibile estrarre i link alle pagine delle citazioni:



-codice per seguire i link alle pagine delle citazioni

```
for quote in sel.css('tr.gsc_a_tr'):
    next_page = quote.css('td.gsc_a_c a::attr(href)').get()
    if next_page is not None:
        yield response.follow(next_page, callback=self.parse_citazioni1)
```

Raggiunta la pagina delle citazioni sono possibili due opzioni:

-estrarre i dati direttamente dalla pagina delle citazioni con una tecnica simile a quella utilizzata per le pubblicazioni

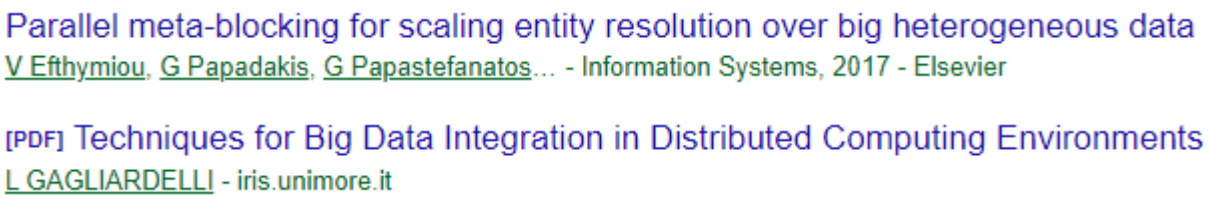
-seguire il link "cita" che dà accesso ad una pagina contenente i dati di una citazione in formato BibTeX, uno strumento utilizzato per la formattazione di testi bibliografici che utilizza un formato di file di tipo testuale, senza informazioni sullo stile di presentazione, contenente un elenco di voci bibliografiche.

Seppure la prima opzione sembri quella più semplice questa non è utilizzabile poiché ad eccezione del titolo e autori i dati relativi alle citazioni sono tutti contenuti in un unico tag html e non possono né essere

estratti separatamente né essere separati in seguito poiché non vi è modo per riconoscere a prescindere dove inizi un attributo e dove finisca l'altro o se questo sia presente o meno.

In conclusione la seconda opzione risulta l'unica utilizzabile.

Esempio:



Come si può vedere le due citazioni qui mostrate contengono attributi differenti tra loro quindi risulta impossibile utilizzare un codice univoco per estrarre tali dati.

3.6 Accesso ai BibTex

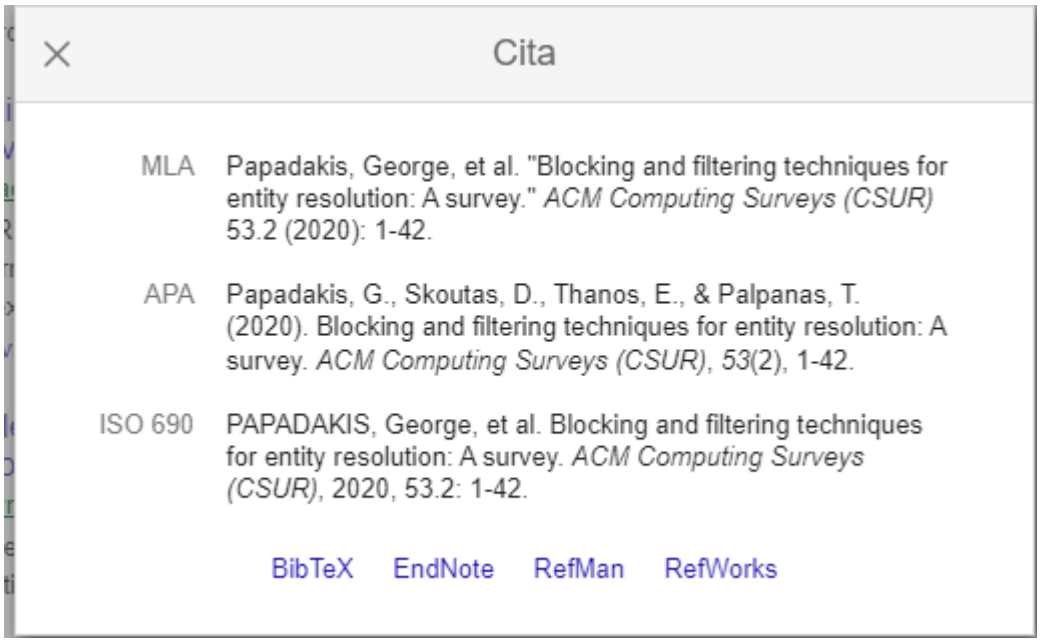
Per accedere ai dati in formato BibTex partendo dalla pagina delle citazioni è stato necessario utilizzare il driver di Selenium e simulare un click sul tasto “cita”.

Questo apre in seguito una nuova finestra che contiene un campo BibTeX che contiene a sua volta il link per accedere alla pagina BibTeX estraibile ispezionando la pagina e poi utilizzando i selettori di Scrapy.

Infine è poi necessario ordinare a Scrapy di seguire i link estratti per accedere a ciascuna pagina BibTeX.

Esempio:

-finestra aperta in seguito al click del pulsante cita



-codice per simulare il click sul pulsante cita

```
targets=driver.find_elements_by_xpath("//a[@class='gs_or_cit gs_or_btn gs_nph']")
sel2 = Selector(text=driver.page_source)
 for target in targets:
target.click()
```

-Finestra aperta in seguito al click sul pulsante BibTeX:

```
@article{papadakis2020blocking,
  title={Blocking and filtering techniques for entity resolution: A survey},
  author={Papadakis, George and Skoutas, Dimitrios and Thanos, Emmanouil and Palpanas,
Themis},
  journal={ACM Computing Surveys (CSUR)},
  volume={53},
  number={2},
  pages={1--42},
  year={2020},
  publisher={ACM New York, NY, USA}
}
```

PROBLEMI RISCONTRATI:

I problemi riscontrati in questa sezione riguardano principalmente l'utilizzo del driver di Selenium.

-Asincronia tra Selenium e Scrapy:

inizialmente lo script prevedeva l'utilizzo di un unico driver ma si è riscontrato che così facendo alcuni dati finivano con il perdersi.

La ragione di ciò è dovuta al fatto che dopo che Scrapy ha raccolto i link di ciascuna pagina contenente le citazioni il driver apre poi tutte le pagine una di seguito all'altra ma senza attendere che sia terminata l'estrazione dei dati prima di procedere con l'apertura di una pagina successiva.

Essendoci un solo driver la nuova pagina sostituirà quella vecchia e, siccome il selettore di Scrapy utilizza il driver come riferimento per estrarre i dati, se una pagina viene chiusa prima del tempo anche i dati raccolti non saranno completi. Poiché l'inserimento di pause è risultato inefficace dato che queste possono rallentare le azioni del driver ma non controllare quando una nuova finestra venga aperta, è stato necessario dunque aprire un nuovo driver ogni volta che si desiderava aprire una nuova finestra.

Esempio:

se si desidera aprire un solo driver è sufficiente definire il driver nella funzione __init__, mentre nel caso in cui se ne voglia aprire più di uno è necessario ridefinire il driver ogni volta.

-codice:

```
def __init__(self):
```

```
options = webdriver.ChromeOptions()
options.add_experimental_option('excludeSwitches', ['enable-logging'])
self.driver = webdriver.Chrome(options=options)
```

3.7 Chiusura finestre

Siccome il selettore di Scrapy utilizza come riferimento il driver per l'estrazione del link ai BibTeX, ripetere il procedimento di apertura della finestra ed estrazione per ciascuna citazione non è sufficiente, poichè una volta effettuato il click su "cita" la finestra che si aprirà sostituirà la pagina delle citazioni come nuovo ferimento dei selettori di scrapy.

E' dunque necessario una volta che si è estratto un link, chiudere la finestra per poter procedere all'estrazione del link BibTeX delle citazioni successive.

Per fare ciò è stato sufficiente ordinare al driver di simulare un click sul pulsante di chiusura della finestra.

Esempio:

-codice per la chiusura della finestra aperta selezionando il pulsante "cita"

```
close=driver.find_element_by_xpath("//a[@id='gs_cit-x']")
time.sleep(1)
close.click()
```

-Next Page:

Poiché le citazioni relative a una determinata pubblicazione possono essere numerose è possibile che queste siano suddivise su più pagine, che devono perciò essere tutte quante esplorate.

E' necessario dunque utilizzare il selettore di Scrapy per estrarre il link alle pagine successive ed utilizzare il comando “Follow_all” per far sì che il programma segua ciascun link.

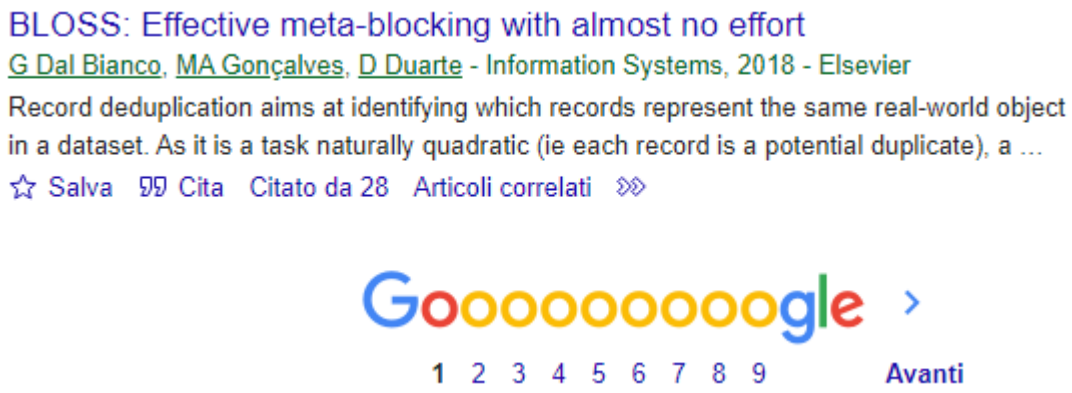
Occorre poi ripetere la procedura di estrazione dei link BibTeX per ciascuna delle pagine.

Esempio:

-codice per seguire tutte le pagine delle citazioni

```
next_page = sel2.css('a.gs_nma')
if next_page is not None:
    yield from response.follow_all(next_page, callback=self.parse_citazioni2)
```

-Esempio di come si presenta la pagina delle citazioni in presenza di molteplici pagine



3.8 Estrazione BibTeX

Una volta raggiunta la pagina BibTex, è sufficiente utilizzare il selettore di Scrapy per estrarre i dati dalla pagina dopo averla ispezionata ed aver verificato dove sono contenuti i dati che si vuole estrarre all'interno dell'html della pagina.

Poiché i dati sono però contenuti in un unico tag html, i dati estratti faranno parte di un unico blocco di testo; è necessario quindi separare ciascun attributo utilizzando la funzione split() di Python.

Esempio:

-codice per l'estrazione dei dati in formato BibTeX

```
sel = Selector(text=driver.page_source)
info =sel.css('pre::text').get()
i=1
start = '{'
end = '}'
#estrazione dati citazioni
if 'title' in info:
        s = info.split(',\n')[i]
        title = (s.split(start))[1].split(end)[0]
        i=i+1
else :
        title = 'None'
```

3.9 Memorizzazione dati Citazioni

Infine si dovrà creare il file .csv che dovrà contenere i dati delle citazioni e poi salvare al suo interno i dati appena estratti, utilizzando le stesse accortezze usate nello scrivere il file delle pubblicazioni cioè sostituire utilizzando il comando replace() il simbolo "," e aggiungere il delimitatore ";" al termine di ciascun attributo così da ottenere un risultato più chiaro e ordinato.

E inoltre importante ricordare di inserire, in tale file Excel, oltre ai dati riportati nei BibTeX anche una colonna contenente il titolo della pubblicazione (già memorizzato nel precedente file .csv) a cui le citazioni si riferiscono

Esempio:

-codice per la scrittura del file .csv contenente le citazioni

```
with open('scholar2.csv', 'a',encoding='UTF8', newline='') as file:
    writer = csv.writer(file)
    delimiter= ';'
    data = [title2 + delimiter + cit2 + delimiter + author2 + delimiter + journal2 + delimiter + volume2 + delimiter + number2 + delimiter + pages2 + delimiter + year2 + delimiter + publisher2]
     writer.writerow(data)
    file.close()
```

-file .csv contenente le citazioni

citedby	title	author	journal	volume	number	pages	year	publisher
BLAST: a loosely schema-aware	Schema-agnostic progressiv	Simonini- Giovanni and I	IEEE Transactions on Knowled	31	6	1208--122	2018	IEEE
BLAST: a loosely schema-aware	Human-in-the-loop data int	Li- Guoliang	Proceedings of the VLDB Endo	10	12	2006--201	2017	VLDB Endowment
BLAST: a loosely schema-aware	The return of jedai: End-to-	Papadakis- George and T	Proceedings of the VLDB Endo	11	12	1950--195	2018	VLDB Endowment

3.10 Limitazioni allo Scraping

Essendo il web un luogo in cui si possono trovare immense quantità di informazioni, esistono individui le cui intenzioni sono quelle di impadronirsi di dati a cui non dovrebbero avere accesso oppure causare disagio sovraccaricando un sito di richieste.

Questi ultimi sono chiamati attacchi DDos e per proteggersi da essi molti siti web utilizzano meccanismi di protezione in grado di individuare e bloccare tentativi di accessi impropri al proprio sito, chiamati “captcha”.

Google Scholar è uno di questi.

Esso al verificarsi di numerose richieste alla medesima pagina da parte dello stesso indirizzo IP, dopo un breve lasso di tempo reagisce, in un primo momento richiedendo una verifica che richiede un'interazione da parte dell'utente al fine di bloccare programmi di estrazione dati automatizzati e in secondo luogo bloccando completamente l'IP per alcune ore.

Tutto ciò risulta dunque un problema poiché il programma appena descritto è di fatto un programma automatizzato che fa numerose richieste a Google Scholar in un breve lasso di tempo, dunque come prevedibile il programma finirà con il bloccarsi dopo poche richieste.

Esempio:

-esempio di Captcha

SOLUZIONE:

- Soluzione manuale: inserimento di opportune pause nel programma per permettere all'utente di risolvere manualmente il "captcha" di google

  Si tratta di una soluzione che rallenta il programma e non compatibile con un programma automatizzato

- Rotazione dei proxy:

l'utilizzo della rotazione dei proxy permette di nascondere l'IP da cui viene realmente la richiesta mascherandola con IP differenti a rotazione. In questo modo il sito web non bloccherà più le richieste poiché le identificherà come provenienti da IP differenti.

Seppure efficacie è però importante ricordare che questa tecnica è difficilmente applicabile con proxy gratuiti poiché la maggior parte di essi hanno una vita molto breve poiché, essendo accessibili a tutti, vengono disattivati in tempi estremamente brevi.

E' dunque necessario per assicurarsi che questa soluzione sia efficace utilizzare Proxy a pagamento

Esempio:

-setting.py

```
ROTATING_PROXY_LIST = ['https://114.55.111.137:22','https://176.57.188.32:443',
'https://157.230.34.219:3128', 'https://140.227.25.56:5678'
,'https://140.227.211.47:8080','https://112.126.70.253:22','https://51.79.207.228:3128']


DOWNLOADER_MIDDLEWARES = {
'rotating_proxies.middlewares.RotatingProxyMiddleware': 610,
'rotating_proxies.middlewares.BanDetectionMiddleware': 620
}
```

3.10.1 Attacchi DDOS

L'acronimo DDos significa "Distributed Denial Of Service" .

Questo tipo di attacco consiste nel tempestare un sito di in elevato numero di richieste al fine di renderlo irraggiungibile utilizzando dei bot in grado di collegarsi contemporaneamente ad un sito e sovraccaricarlo con messaggi e richieste fino a saturare la banda della società attaccata causando così il blocco dei computer alle richieste dei dati in entrata.

3.11 Come eseguire lo script

Terminata la scrittura del programma è possibile eseguire lo script utilizzando il comando

scrapy crawl <nome dello script>

A seguito dell'esecuzione dello script il programma restituirà i due file csv contenenti le pubblicazioni e le citazioni estratte da Google Scholar.

Qualora il programma dovesse interrompersi durante l'esecuzione i dati estratti fino a quel momento verranno comunque salvati all'interno dei due file .csv.

Esempio:

-comando per l'esecuzione dello script

```
(base) C:\Users\Chiara\Desktop\Scrapy\tesina>scrapy crawl tesi12
```

4. CONCLUSIONI

In conclusione abbiamo visto come realizzare un programma in grado di raccogliere dati dal sito web Google Scholar e salvarli all'interno di un file in formato Excel evidenziando le difficoltà in cui è possibile incorrere e tentando di presentare soluzioni efficaci a problemi come la necessità di interagire con la pagina web per il caricamento o l'estrazione di alcuni dati e la memorizzazione degli stessi.

I dati estratti potranno essere utilizzati poi per differenti scopi come per esempio operazioni di confronto con quelli raccolti da altri siti. E' inoltre importante tener conto del fatto che seppure il programma sia destinato al web scraping di uno specifico sito questo può con le dovute modifiche (come per esempio la sostituzione degli url di partenza) essere adattato per estrarre dati da altri siti web simili.

Sono stati inoltre evidenziate alcune criticità relative allo scraping di grandi quantità di dati senza l'utilizzo di strumenti specifici adatti a tale scopo.

4.1 Riflessioni

Grazie a questo progetto ho imparato come utilizzare uno strumenti di scraping quali Scrapy e Selenium e ho potuto verificare quanto sia complesso realizzare tale progetto poiché ogni sito è organizzato in modo differente e può talvolta utilizzare protezioni contro dagli attacchi informatici che possono bloccare ogni tentativo di scraping automatico. Ciò ha reso perciò necessario adattare e modificare ogni azione dello scraper al sito stesso per riuscire ad ottenere un risultato efficace.

5. SITOGRAFIA

1. https://scholar.google.it

2. https://www.selenium.dev

3. https://scrapy.org

4. https://chromedriver.chromium.org

5. https://anaconda.cloud