Project Report: Crawling *ArXiv* Using BeautifulSoup, Scrapy and Selenium

Kunhong Yu(444447), Ludi Feng(437847)
Faculty of Economic Sciences
University of Warsaw
k.yu,l.feng@student.uw.edu.pl

Abstract

In this report, we extend what we have learnt during web scraping with Python lecture to scrape a website: *ArXiv*, which is well-known as academic paper hosting platform, where we can find any recent papers in many subjects(e.g. Computer Science, Physics, Economics, etc.). What we do in this project is try to scrape paper's information, such as, title, authors, abstract, comment, each author's related papers according to subjects users specify and store them into disk using three practical tools: BeautifulSoup, Scrapy and Selenium. Furthermore, we utilize simple dimension reduction(e.g. PCA) and Natural Language Processing(NLP) techniques to understand how papers are related.

1 Introduction

Web scraping[4] is a very practical technique applied broadly in many fields, for example, one uses web scraping to scrape images from different categories and feed them into models to train object recognition tasks in Artificial Intelligence(AI)[1]. There are three great tools are ready to use, BeautifulSoup[8], Scrapy[3], Selenium[5]. BeautifulSoup(BS) is a classical method to get information from static web, BS builds a tree-based model for each web, we can get each component easily; Scrapy utilizes multithread technique to let users run multiple crawling processes at the same time, unlike BS, we can use Scrapy by extending its class and override its functions to accomplish our tasks conveniently; finally, Selenium is more flexible in the situation where we want to do dynamic actions in the web, such as, clicking, filling blank with texts.

In this project, we extend what we have learnt during our lecture, and scrape a famous academic paper hosting platform: *ArXiv* [2], which is developed by Cornell University. Specifically, like shown in Figure 1, when first entering its main page(`https://arxiv.org/`), we split what we need into subject –> subsubject –> subsubsubject(e.g. Physics –> Astrophysics –> Cosmology and Nongalactic Astrophysics), we first get each subsubsubject's link, then skip to the each subsubsubject's page to scrape each paper's link and go to the each paper's page to scrape its title, authors, abstract and comment, finally, we want to know each paper's first author's old papers to see which area this author was dedicated to. In order to analyse what we scrape, we use simple PCA[10] and NLP[9] techniques in machine learning to make 2d and 3d visualization to understand data we obtain.

The rest of report is organized as follows: next section is about mechanism of our project, section 3 gives code organization, section 4 details experiments on three tools we mentioned above and final section draws a conclusion.

2 Related Mechanism

In this section, we go through how *ArXiv* web works and techniques involved in our project.

Main Page | Papers Page for a Specific Subsubsubject

Each Paper's Information | First Author's Papers

Figure 1: The Structure of the *ArXiv* Web Page.

2.1 About *ArXiv*

As mentioned before, *ArXiv* is used to obtain research papers easily for users who work both in industry and academic areas. The organization of the web is already listed in Figure 1. When entering main page, what we need is built into three-level tree structure, we call them subject –> subsubject –> subsubsubject(e.g. Physics –> Astrophysics –> Cosmology and Nongalactic Astrophysics) just like shown in the figure where are highlighted by red underlines. As we can see there are many links related to many areas of research, in our implementation we can scrape all links and skip to each link to get all papers we need, however, in reality, we are often blocked by the web if scraping too much information, therefore, it is safe that we let users choose what categories to scrape, more details will be covered in later sections.

Then, if clicking each link in the subsubsubject, then we skip to its page containing all recent papers in this field, for each paper, we click its link which is marked in the figure to enter the page where includes all information belonging to this paper, what we need in this paper is title, authors, abstract and comment, they are also marked by red lines in the figure. Moreover, in the research field, first author may have contributed quite a lot of work to this paper, therefore, if we want to know how each author's dedication to this field in the past, we can click his/her link to search all papers what he/she has written, just like last subfigure shown in Figure 1. However, it is often that searched results containing some papers that don't belong to this author, they are obtained since the authors' names are similar to the author we search, as a result, we need to filter this information carefully.

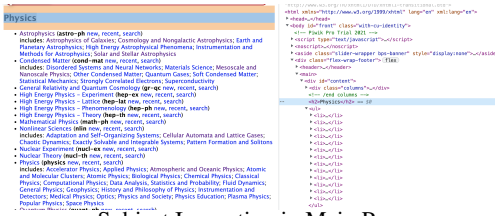
Though pages look neat and simple, the information and structures are valuable, there are many small tricks involved to obtain information we need, it is therefore quite interesting and useful for us to understand how to use three tools to scrape web pages.

Notations: in the report, we use '<x>/<y>' to denote tag <y> is the direct child tag of <x> and '<x>//<y>' to denote tag <y> is the descendant tag of <x>.

2.2 Our Strategy

In this subsection, we go through how we get information and inspect each page. Since three tools scrape the same thing, we follow the same strategy.

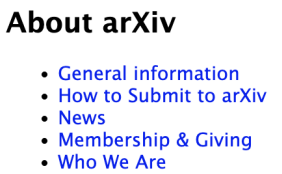
In order to first obtain links of subsubsubject in the main page, we need to follow the procedure, which is first get subject area and then get subsubject area finally reaches subsubsubject area to get links. Specifically, in Figure 2, we can observe that subject can be achieved by finding '<h2>' tag,



Subject Inspection in Main Page

Subsubject Inspection in Main Page

Subsubsubject Inspection in Main Page



About arXiv in Main Page

Figure 2: Inspection of Main Page.

there are multiple '<h2>' tags in this page, so we can iterate them all to go into each subsubject. From Figure 1, subsubject is after a dot sign, and as shown in Figure 2, subsubject is displayed in the first '<a>' tag under '<ul>/<li>' and first '<ul>' is the first sibling of '<h2>'. We expect subsubsubject to be under the '<li>', however, they are all siblings of '<li>', a simple idea is notice that subsubsubject is after text: 'includes:', so we can first find 'includes:', and get all '<a>' after, naturally links of all subsubsubjects are obtained. Note that in the main page, there is a section called 'About arXiv' which is shown in the last subfigure in Figure 2, it is useless for us but it also holds the same structure as above, so we must drop it when scraping; also, there are some blank subsubsubjects, we don't have to deal with this in particular, since Python dictionary is used to store information, getting and checking operations are cheap(e.g. $\theta(1)$). As stated above, in order to store links in tree-like structure, we define a three-level dictionary, where in the first level key is subject name and value is subsubject dictionary, in the second level, key is subsubject name, value is subsubsubject dictionary, in the final level, key is subsubsubject name, value is a link corresponding to this subsubsubject.

Recall after choosing one subsubsubject, we enter the page containing all papers in the related field, note *ArXiv* by default sets 25 papers for each page, they are sorted by the order of released date of papers, which can be spot by '<dl>' tag as shown in Figure 3, and each paper's link can be retrieved by the first '<a>' tag under '<dl>/<dt>//'. However, things are complicated since there are three situations we encounter:

1. There are **usually** more than 25 papers, so we have to skip to next page to find more papers, in our implementation, we also specify a upper limit which can be set by users to scrape how many papers to scrape, if we need to go to other pages, we find next page link in '<small>' tag, and get the '<a>' tag, since there are multiple '<a>' tags in the page, we need to record which one we are currently working on as a pointer, then move pointer to the next one when going to the next page.

2. In some field, there are not many papers, so all papers can be listed in one page, therefore, there are no links in the '<small>' tag descried as above, so when going to the next page, checking if '<small>//<a>' exists is necessary.

3. As shown in Figure 3, there are many dates/'<dl>' tags in the same page, in other words, each day less than 25 pages are released, so multiple '<dl>' tags are in the same page, we need to find them all.

After getting each paper's link, we can get its information by using the link, from Figure 4, we notice that title is easy to get, it lies in '<h1>' with class 'title mathjax', we simple obtain its text and strip its whitespaces. Authors are under the ' ' with class 'authors', which is the first sibling of title, in the figure, we illustrate how we can get first author's link and name, which is useful when we scrape each first author's papers, we store this information into dictionary where key is the name of author, value is the search link. Authors are followed by abstract, abstract is in '<blockquote>' with class 'abstract mathjax', finally, comment is obtained in the second '<td>' under

Figure 3: Inspection of Papers Page.

'<table summary="Additional metadata">/<tbody>/<tr>', however, some paper does not contain comment, so it's not possible to find this tag, our program is also robust to this situation.

Figure 4: Inspection of Paper Page.

After scraping all this information, we store them into the disk with a `csv` file, where each row contains each paper's subject, subsubject, subsubsubject, title, authors, abstract and comment, if there is no comment, we leave it as NA.

Remember we also store each paper's first author's link, in order to add more flexibility, we let users decide whether scrape each author's papers. Just as Figure 1 shows, when we are scraping author's papers, we need to scrape each paper's authors to see if they are matched with author we are searching, in Figure 5, we give a simple illustration on how this works by first retrieving all '<li>' tags with class 'arxiv-result' and go deeper to get title, authors and abstract. Note when inspecting abstract, we are led to the '' tag with class 'abstract-short has-text-dark mathjax', however, in this area, abstract is not shown fully, what we need is text from '' tag with class 'abstract-full has-text-dark mathjax'. Note we also encounter the same situation where going to the next page is possible, from the figure, we can mimic this process by finding next link.

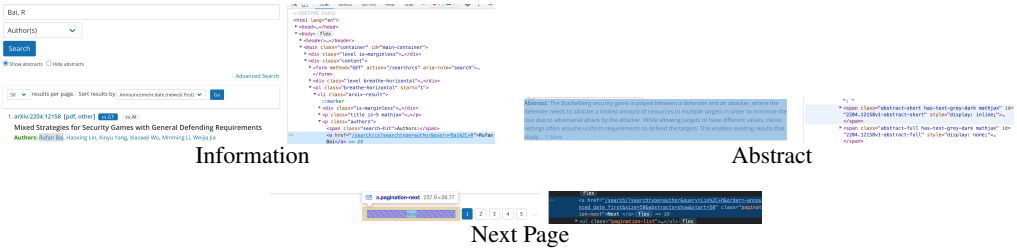


Figure 5: Inspection of Author Papers Page.

Finally, we store all author's all papers into a `csv` file, where each row contains author's name, paper's title and its abstract.

3 Code Organization

In this section, we give our code organization, which is shown as follows in Figure 6.

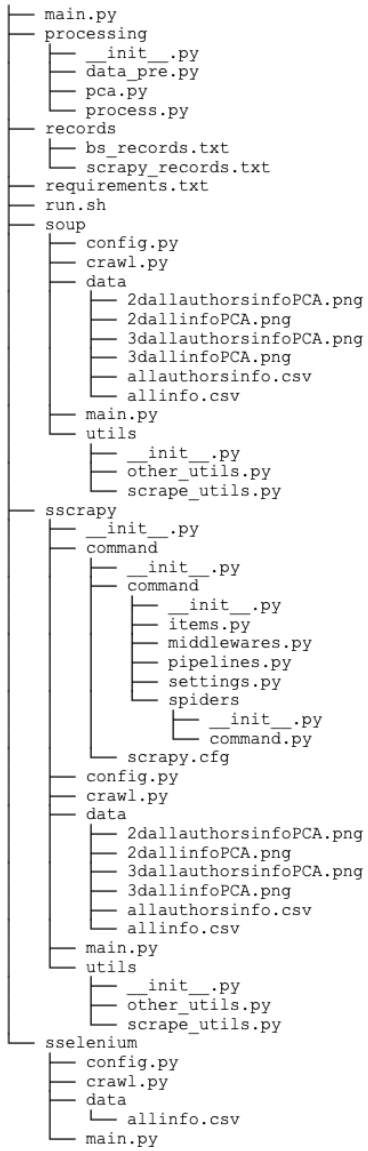


Figure 6: Code Organziation.

We define three main folders: `soup`, `sscrapy`, `sselenium`, which contain all codes corresponding to three tools: BeautifulSoup, Scrapy and Selenium respectively, however, according to project requirements, we have to name them as soup, scrapy and selenium, but it's not possible if we import off-and-shelf packages called scrapy and selenium, so we must rename our folders. In each folder, all files look almost the same, they all have `data` and `utils` folders, in the `data` folder, results are stored and `utils` includes core scraping classes and functions related to the specific tool, moreoever, `config.py` is used to set parameters globally, `crawl.py` is used to call scraping functions in `utils`, finally, `main.py` is the code entrance for the corresponding tool. In `sscrapy`, there is a folder `command`, which is a scrapy project being used to start crawling by typing `scrapy ...` in command line. In `sselenium` folder, we simplify the process with only three common files like in `soup` and `sscrapy` since it's much easier for selenium to work with this page, and we don't scrape each author's related papers with selenium.

`processing` folder contains all operations about data analyses, `records` stores running time records for three tools, moreover, `main.py` in the mainstream folder is the entrance of all the project,

`requirements.txt` describes all needed external packages to run our code, `run.sh` is running script, one can also run code by typing `./run.sh` in command line with default parameters.

More details will be given in the next section.

4 Implementations

In this section, we follow our strategy before to use three tools: BeautifulSoup(BS), Scrapy and Selenium to accomplish scraping tasks. In all folders, there are the same files for each tool, for example, `config.py` is used to store parameters specified by the users. There is a `main.py` function in the mainstream folder, which is the entrance of all the project. In addition, we are asked to add one boolean `page_limit`, when setting it as True, most 100 pages should be scraped, in this project, when scraping papers, we treat 100 pages as 100 papers, and when scraping each author's papers, we recount this number and program stops when 100 papers are scraped. Moreover, we also set an another `limit` parameter which can be controlled by users, when `page_limit` is False, and we will scrape no more than `limit` papers for each subsubsubject, when scraping each author's papers, we set it to be 150 as default for each author.

4.1 BeautifulSoup

In this subsection, BS is utilized. All code is in `soup` folder. BS is simple to use, we won't go deeper into each step, they can be found in the detailed codes. The core code is in `scrape_utils.py`, where defines multiple classes to scrape different pages. In particular, in `LinkScraper` class, we first create `scrape_main_page` method to scrape each subsubsubject's link and store them into a big dictionary as mentioned above. `scrape_specific_page` is used to obtain all papers' links for a specific subsubsubject, and more formally, in case there are many pages for each subsubsubject, we define an inner method `__one_page` to get each page's papers' links, finally, `scrape_paper_links` function combine above functions to get all papers' links.

In the `InfoScraper` class, `scrape_single_info` is used to get information(e.g. title, authors, abstract, comment) of one paper, and `scrape_info` is applied to call `scrape_single_info` in a loop to find all papers' information. Then if we need to get information for each author, there is a `AuthorsScraper` class ready to use, where `scrape` function can achieve this goal efficiently.

In order to use code as simply as possible, we add a `crawl.py` file in the `soup` folder, where `crawl_all` function automatically calls all classes and functions to scrape all information. In the `main.py` file, `main` function calls `crawl_all`. Besides, if one want to run `soup` independently, you can enter `soup` and type `python main.py`.

To avoid being blocked, we add code to change `User-Agent` to overcome this issue, if we still can not handle the blocking problem, we can use the alternative web: `http://xxx.itp.ac.cn`, which is mirror site of *ArXiv*, holding exactly the same structure. To keep what we scrape consistently, we set parameter `limit` as the number of papers of each subsubsubject, and `page_limit` as a boolean variable to denote whether or not to scrape only 100 pages just like as mentioned above, when it is True, `limit` is invalid any more, when scraping each author's papers, we gather all 100 papers belonging to all authors when `page_limit` is set to be True, otherwise, we will scrape each author's papers with upper limit of 150.

4.2 Scrapy

Code is in `sscrapy` folder, since scrapy is already the package name, we can not use it if we want to call function from our implementation, so we rename our folder into `sscrapy`. In scrapy implementation, we have almost the same structure of codes as BS, and logic is almost the identical, however, there are some evident changes needed to point out, which are determined by the mechanism of scrapy. In `scrape_utils.py` file, class `MainPageScraper` also utilizes BeautifulSoup to get links from the main page, we do this since in each script, only one `CrawlerProcess` can be defined, we can not use one `CrawlerProcess` to get return and feed return to the next `CrawlerProcess`, this is because `CrawlerProcess` runs with multithread technique. After getting all links, core class is `PaperSpider`, where `parse` function first gets link for each paper, and uses `scrapy.Request(link, dont_filter=True, callback=self.parse_paper)`,

to use the paper's link to scrape its information(e.g. title, authors, abstract and comment). If there are more papers to be scraped, we also go to the next page by requesting the same function again and again. When scraping each paper, we scrape deeper for the first author to find his/her papers in `parse_author` function. In `ScrapySpider` class, we combine all above classes and functions to accomplish all tasks at once. Remember there are many operations should be operated in parallel, for example, when we scrape all papers, we can do it at the same time, we accomplish this by defining `spider` function under the decorator `@defer.inlineCallbacks`. Instead of designing code which can be executed by typing `scrapy ...` in the command line, we use `CrawlerProcess` class to let scrapy be run and called among scripts, so finally, all scripts can be run like the same.

We notice that since all crawlers are running in parallel when using `CrawlerProcess`, therefore, when limiting the pages within 100, it usually scrapes more than 100 pages because when **global** limit is met, other crawlers are still working, we use one stupid method to truncate what we scraped to limit results within 100, due to the fact that scrapy are crawling all papers randomly(not sequentially like BeautifulSoup), the results we obtain will differ a little with BS. Similar to BS, we set parameter `limit` as the number of papers of each subsubsubject, and `page_limit` as a boolean variable to denote whether or not to scrape only 100 pages, when it is True, `limit` is invalid any more.

In order to run scrapy code using `scrapy ...` command, we add one scrapy project `command` within `sscrapy` folder, where one can go into the `command` subfolder, in `spiders` folder, we create a `command.py` file, which has a scrapy named 'commandspider', one can use this function by typing `scrapy crawl 'commandspider'` in the command line to start scrapy project.

4.3 Selenium

As the article mentioned before, what we need is to crawl the subject, ssubject and sssubject. In order to make it come true, we make some prerequisites, one can find them in the detailed code.

First, we creat a function named `crawl_subject_url`, we use the `driver.get()` to open the main page. And then we inspect the whole page, in order to find the content, we use the `driver.find_elment()` to catch the ID–content, then use the `ul` of the `TAG.name`, every `ul` represents the subject content. Next, it is clearly to observe that every `li` label represents the ssubject, sssubject and the other unimportant stuff, what we want is to catch the ssubject and the sssubject. So we simulate clicking each label `li`, there are a lot of a labels, they represent all the links.

For the ssubject, it appears in the `li` label, every third a label should be the ssubject, then we got the attribute of the `href`. For the sssubject, what we want is the content after the text `includes`. Observe that between the ssubject and the sssubject, there is a common attribute, `aria-labelledby=main-astro-ph`. So in the code part, we first got the a label, for example, if it starts with `aria-labelledby=main-astro-ph`, it should be a sssubject, then we use a for loop to store all links of the sssubject. Finally, we crawl all the links of the ssubject and the sssubject.

In the second part, we have to crawl all the author, title, abstract and the comment, but it is clearly to observe that not all papers have comment, not all of them have the ssubject or sssubject, we start traversing the links, so in `crawl_all`, we filter missing information. For crawling the ssuject, we use the method `request_url` by clicking all collected links, to show all pages, and inspect the page, it is not difficult to find that `dd` represents the link, and then we obtain the title, author, comments and abstract. In the end, we print them all, every time we crawl the new information, it will be appended into the pandas DataFrame instance `df`. Finally, store all the information into `.csv` file like above.

4.4 Analyses

In this final subsection, we make a simple analysis of our scraped data, since the data is almost the same by three tools, so we do it once. **Note that involved analyses techniques's details in this report will not be discussed, one can refer to references to learn more.** Since in `selenium`, author did not implement function of scraping each author's papers, we measure running time with only scraping all papers regarding each subsubsubject.

Before analysing data, we record and compare running time of three scraping tools in `records` folder, where one can find records of all running logs of three tools. Under the same situations in which `limit` and other parameters are set the same, comparison is listed in Table 1.

Table 1: Running Time Comparison. Note we have run each tool 5 times under the same situation and compute their mean and standard deviation.

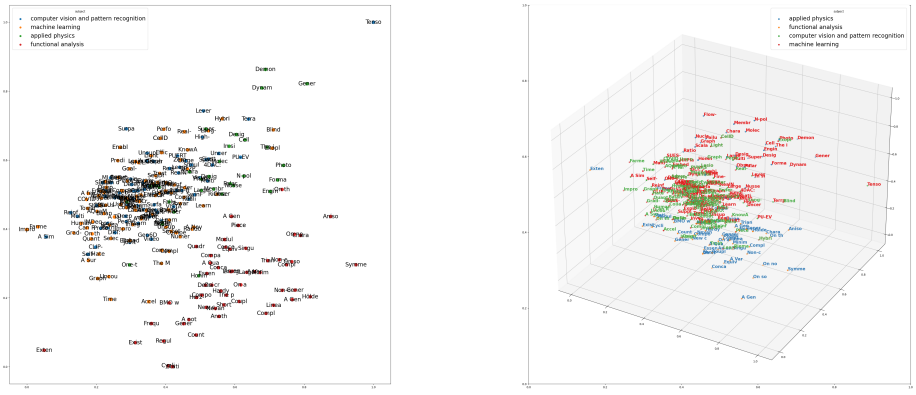
Tool	Running time(s)
BeautifulSoup(BS)	140.67±1.26
Scrapy	41.80±2.33
Selenium	77.21±2.41

We can see that BS runs most slowly since BS runs code sequentially, and scrapy runs much faster because scrapy utilizes concurrency mechanism when crawling, and selenium is also competitive.

In data analyses, firstly, since our data is text stored in the `csv` file, we need Natural Language Processing(NLP) method to make them more understandable, in particular, we utilize standard text preprocessing procedure to firstly preprocess raw text. Then we concatenate title and abstract together for each paper as a long string, then we apply pre-trained GLOVE [7] embedding obtained from `https://nlp.stanford.edu/projects/glove/` to map raw text into 50-dimensional embedding space for each word. For each paper, we calculate mean of embeddings of all words as its new feature and finally use PCA to do dimension reduction to map it from 50d into 2d or 3d dimensions to make visualization.

In implementations, `processing` folder contains all files about data analyses. `glove.6B.50d.txt` is the pre-trained word embeddings. `data_pre.py` loads data, preprocesses data and maps data into 50-dimensional space, `pca.py` defines PCA class from sklearn[6] package, finally, `process.py` includes all steps to analyse data.

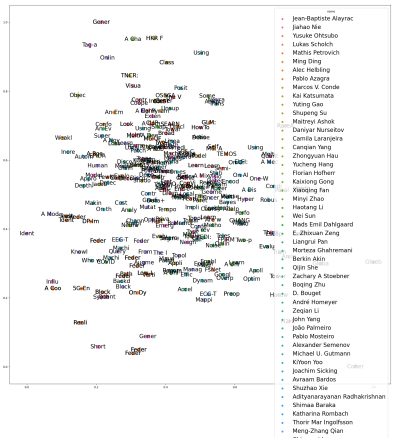
Final functions are `plot_embeddings` and `visualize`, we create these two functions to visualize our analyses results. In the Figure 7 and Figure 8, we visualize our results into 2d and 3d. From Figure 7, all authors' papers results are obtained, for each color, we define it as subsubsubject, and we can see that all papers belonging to the same subsubsubject are clustered together, which proves that our scraped data is correct, if not, we can not see clustered results. Moreover, in Figure 8, we visualize each author's papers, each color defines one author, since there are many authors in the file, we can not fill them all in one plot, we truncate them, from this figure, we also conclude that the data we scrape is valid and reliable.



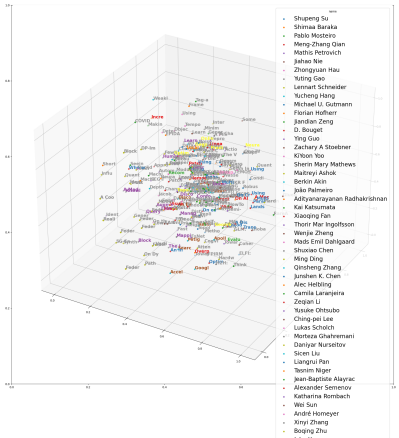
All Papers in 2d

All Papers in 3d

Figure 7: Visualization Using All Scraped *Papers*.



All Papers Corresponding Each Author in 2d



All Papers Corresponding Each Author in 3d

Figure 8: Visualization Using All Scraped *Papers* for Each Author.

5 Conclusion

In this project, we use three tools we grasped during lecture: BeautifulSoup, Scrapy and Selenium to scrape famous academic papers hosting platform: *ArXiv* to obtain each paper's information and each author's papers, finally we use scraped data to make visualization via simple NLP and PCA techniques and understand what's going on in the data. In the implementations, we've done quite a lot work to make every scrape correct. Finally distribution of works is in Table 2.

Table 2: Distribution of Works.

Works	Distribution
Original Idea of Project	Kunhong Yu(100%)
Strategy Designs	Kunhong Yu(100%)
Code Designs	Kunhong Yu(70%)[BS/Scrapy]/Ludi Feng(30%)[Selenium]
Data Analyses Designs	Kunhong Yu(100%)
BeautifulSoup Codes	Kunhong Yu(100%)
Scrapy Codes	Kunhong Yu(100%)
Selenium Codes	Ludi Feng(100%)
Description `pdf` file	Kunhong Yu(80%)[BS/Scrapy]/Ludi Feng(20%)[Selenium]
`README.md` file	Kunhong Yu(80%)[BS/Scrapy]/Ludi Feng(20%)[Selenium]

References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[2] Paul Ginsparg. Arxiv at 20. *Nature*, 476(7359):145–147, 2011.

[3] Dimitrios Kouzis-Loukas. *Learning scrapy*. Packt Publishing Ltd, 2016.

[4] Richard Lawson. *Web scraping with Python*. Packt Publishing Ltd, 2015.

[5] Simon Munzert, Christian Rubba, Peter Meißner, and Dominic Nyhuis. *Automated data collection with R: A practical guide to web scraping and text mining*. John Wiley & Sons, 2014.

[6] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[7] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[8] Leonard Richardson. Beautiful soup documentation. *Dosegljivo: https://www. crummy. com/-software/BeautifulSoup/bs4/doc/.[Dostopano: 7. 7. 2018]*, 2007.

[9] William Song and Jim Cai. End-to-end deep neural network for automatic speech recognition. *Standford CS224D Reports*, 2015.

[10] Jian Yang, David Zhang, Alejandro F Frangi, and Jing-yu Yang. Two-dimensional pca: a new approach to appearance-based face representation and recognition. *IEEE transactions on pattern analysis and machine intelligence*, 26(1):131–137, 2004.