Webscraping and Social Media Scraping
GO Game Project

Group members

- Georgii Bykov - 444449
- Paulina Sereikytė - 444470
- Dawid Szyszko-Celiński - 443709

Introduction

The GO game is believed to be one of the oldest board games in the world. It was invented around 2,500 years ago in China and it is still played mainly in Asia, however in western culture it is not as well known as chess. Standard GO board is 19x19 grids, whereas in chess it is 8x8 and that is one of the reasons why GO is considered a much more complex game. For more information about the game please refer to wikipedia page.

We have chosen to scrape https://www.goratings.org/en/, a website dedicated to tracking the players and games of strategy board game Go. The webpage has a simple structure, however it has a lot of information that can be used and analysed. Practical applications of data gathered from similar websites could include predictions for gambling companies, game improvement and personal insights for the players.

Scrapers mechanics

Beautiful Soup:

1) At the first stage, we should get HTML into an IDE (used PyCharm), so that we could analyse it using the "**requests**" library, more specifically **get** method.
2) Once there is a 'request' object, we can use the parser feature in the BS (we chose **lxml** parser).
3) Then we got a list of all the urls with players from the main page using **find_all** method from BS library and **compile** method from **re** library (regular expressions) to exclude other links. We also made a **list** object with the first 100 links of players' pages at this stage using a **for** loop.
4) In the main phase using **for** loop and **BS** methods **find** and **find_all** we parsed information from each page about the player's name, date of birth, rating, number of wins & losses, total number of games. As a part of the loop, we converted tables to **pandas dataframe** to simplify data extraction from them. We also calculated some statistics within the loop: share of wins and loses depending on players' colour (four values) and average difference between current player and her/his opponent on the basis of the last 10 games. All the data was appended to a **list** object.
5) Finally, we **converted** the list with all the data to a **panda dataframe**, which was later exported to a **csv file**.

Scrapy:

1) First, the links used for data scraping and analysis are scraped using the link_lists scrapy. A link scrapy field is created for scrapy item class. A spider is then created, allowing scrapy to scrape webpages only with the domain name 'www.goratings.org'. Starting from 'https://www.goratings.org' , scrapy finds all the elements that conform to the xpath .//tr/td[2]/a/@href. To ensure that only top 100 links are scraped, a boolean statement is created at the start of the code taking either a specified amount or 100 values. Therefore a *for* loop is introduced and tr[str(i)] refers to either the specified or top 100 rows of the table with the necessary links. The links are called (response.xpath(xpath)), cleaned up (removing '..') and completed with https://www.goratings.org. In the terminal, the scrapy is crawled and the output is saved into link_lists.csv with the command scrapy crawl link_lists -o link_lists.csv -t csv.
2) The second step was to scrape the data collected from the list of links. 11 scrapy fields were created and the allowed domains specified as before. The links from link_lists.csv were then opened. The xpaths from existing necessary data fields were specified. Some of the data fields required for the analysis did not exist in the original webpages so they were calculated at the time of scraping using for loops. Empty lists for colour, win or loss, player elo and opponent elo were created and filled out for the last 10 games of the player. 0 values were initiated for the win/loss and colour combinations which were then augmented according to the values of the i^{th} element in the colours and win/loss lists. The difference in player's ratings was calculated in a similar way, by subtracting the player's elo with the opponent's elo and summing it up for each game. The averages were calculated at the last step, when variables were called.

Selenium:

1) Provide a path to geckodriver
2) Open starting page in a browser - https://www.goratings.org/en/ and wait for 2 seconds to make sure it is fully loaded
3) Create a blank dataframe with proper structure
4) Iterate through chosen number of players pages - by default 100 pages that can be set to another number by “no_pages_q” parameter. Then for each iteration:
5) Set variables ww (white win), wl (white loss), bw (black win), bl (black loss), diff (difference between players ratings) and av_diff to 0 that will be needed for further calculations
6) On the starting page find the object that can be clicked to move to the i-th player page (the html part with link redirect) and then click on that to move to that page
7) On the players page get player name from h1 tag and statistics about wins, losses, total games and date of birth from the first table.
8) From the second table (table about last games) scraper gets player`s rating from last game and we store it as a current elo of the player
9) Then we iterate over last 10 rows from games table
10) For every iteration (last j-th game) we scrape result of the game (win/loss) and colour of pieces that player had (black/white)
11) After getting data about result and colour scraper goes to the if conditions part and based on the scraped data from j-th row it increases ww, wl, bw, bl variables by 1
12) Then we divide each variable ww, wl, bw, bl by 10 to get percentage of games that were won/lost playing certain colour of the pieces
13) Then for every j-th iteration we scrape player rating and opponents rating for the j-th game of the player and for each of last 10 games we calculate difference between player rating and opponent rating (diff variable)
14) At last we divide calculated difference by 10 and it gives us information about average difference between player and their opponents (av_diff variable)
15) All scraped and calculated values are stored in the “player” dictionary, which appends data to previously created dataframe
16) After addition to dataframe scraper goes back to the starting page, gets another link and goes to the next player's page until it reaches the stated number of scraped pages
17) After all iterations firefox browser closes and user can input the path to save csv. If the user does not input any path, by default the file “Go_Players_Data_Selenium.csv” will be saved in the current directory

Description of the output

As a result, we got a csv file with 11 comma separated values:

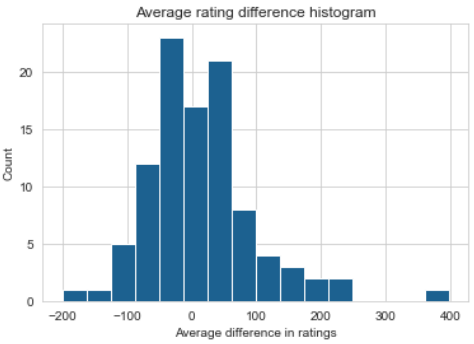
#	name	short description
1	player_name	First and second name of the Go player
2	date_of_birth	Date of her/his birth in YYYY-MM-DD format
3	elo	Player's actual rating
4	wins	Total number of games won
5	losses	Total number of games lost
6	total	Total number of games played
7	ww_perc	Share of games won when playing white from the last 10 games
8	wl_perc	Share of games lost when playing white from the last 10 games
9	bw_perc	Share of games won when playing black from the last 10 games
10	bl_perc	Share of games lost when playing black from the last 10 games
11	avg_diff	Average difference between ones and opponents rating based on the 10 last games

The output csv files for each scraper can be found here.

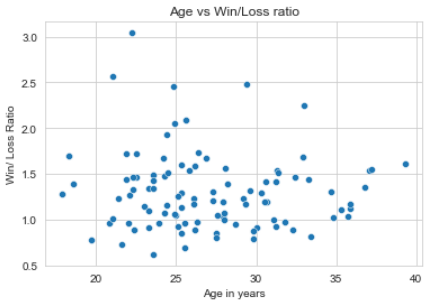
Data analysis

The data gathered was used to perform basic data analysis to showcase the potential use cases of similar web scraping projects.

From the accompanying visualisations it can be observed that the top 100 players have mostly played with opponents that were between 100 rating places lower or higher than their own rating. There is a small dip in values around 0, most likely due to the smaller number of players within that range.

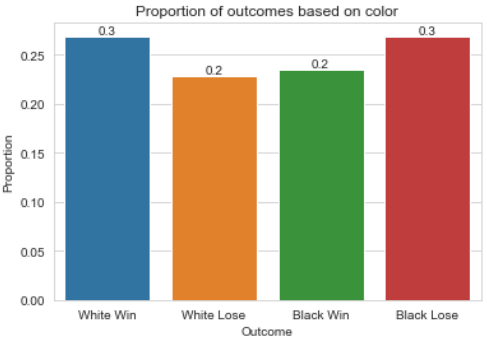


Age does not appear to play a significant part in determining the win/loss ratio, however it can be interesting to note that all top 100 go players were below 40 years of age, averaging at 27 years.



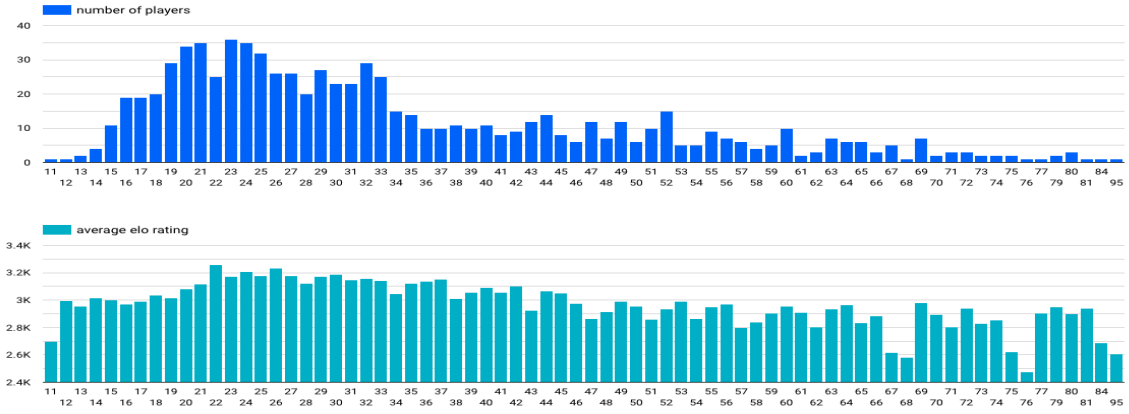
The colour of the piece the player was playing with was also indicated in the data gathered. The proportion of the wins and losses for each colour was calculated. It can be noted that the player playing with white pieces won more times and lost less times than with black ones.

	White	Black
Win	0.27	0.24
Lose	0.23	0.27



It is important to mention the limitations of the analysis. Firstly, it takes into account only the first 100 players' last 10 games. Secondly, some of the games might be duplicated in the analysis if the top 100 players played against each other. Lastly, in case there have been more games played since this analysis has been run, the results could be slightly different.

An analysis was also made for the distribution of age and the relationship between age and the average rating within the full sample. Players (871) with known ages were included in the analysis. The distribution by number of players shows that the majority of players are between 19 and 33 years old. The average rating tends to increase up to 22 years and then gradually decreases starting from 37 years.



Time efficiency

The table below shows 5 measurements of execution time and their average for each of the scrapers in seconds. All measurements were performed on the same machine and with similar conditions, that is why we can assume the hardware/software impact can be excluded. Of course the result will be affected by those factors and the Internet connection as well, but 5 independent tests should give on average the results that can be compared. Measurements of BS and Selenium times were included in code as comments and for Scrapy we used the execution time report visible in the console after running scraper.

Scraper	(1)	(2)	(3)	(4)	(5)	Average
BS	58.39	58.76	59.76	59.14	63.44	**59.90**
Scrapy - links	0.90	0.88	0.96	0.82	0.86	**0.88**
Scrapy - pages	10.05	8.69	9.78	10.26	9.79	**9.67**
Scrapy - sum	10.95	9.57	10.74	11.08	10.65	**10.60**
Selenium	135.85	135.75	133.85	136.38	137.96	**135.96**

At first glance it is visible that there are huge differences between scraping methods. On average Beautiful Soup scraper performed all actions in just a bit less than a minute, which placed this method in second place in terms of time efficiency. The fastest crawling was performed using scrapy. Both spiders (scraping link list and performing pages scraping) were executed in about 10.5 seconds on average, which makes it around 6 times faster than BS. The worst performance was achieved by Selenium scraper, which executed scraping in more than 2 minutes on average.

Of course each of the scrapers should be used for different purposes. For example Selenium can simulate users behaviour and will be used for more complex tasks where some clicking is needed, but in terms of scraping just the html part it is not that great as we don`t need to have visible UI for that. In case of simple and quite fast scraping we would choose BS because of its simplicity. For bigger projects where time is crucial we would go for Scrapy. It's much more complex in terms of coding but as we can see in the table results are outstanding compared to other tools.

Division of work

- Georgii Bykov - Beautiful Soup code / Output files verification and validation and their description/ Data analysis
- Paulina Sereikytė - Scrapy code/ Data analysis and visualisation
- Dawid Szyszko-Celiński - Selenium Code / helping with some issues with BS and Scrapy / adding 100 pages limit / time efficiency part in report / README description