Zachary Jaffe-Notier
CS 406 – Spring 2020

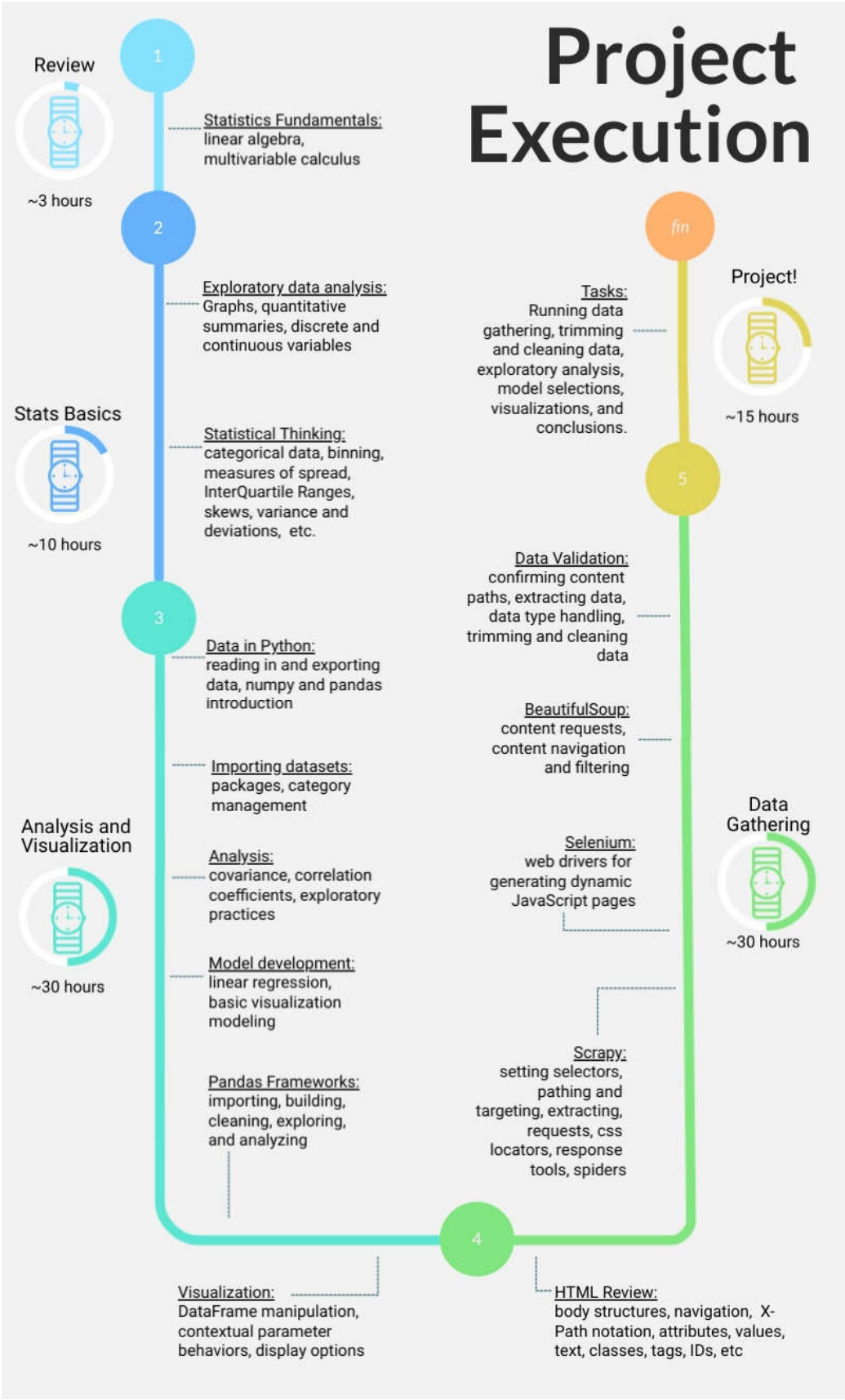
Data Analysis and Visualization in Python

This project was an introduction to the techniques used in Python for data visualization and analysis. To start off on this project, I began with a basic review of fundamental math concepts I knew I would be relying heavily on. This was essentially the only element I was sure of before I began this project, as I had never before done any kind of statistical data analysis. While the project plan I had laid out for covering this material provided decent guidelines for what to cover, I was constantly course-correcting throughout to refocus my efforts on more relevant material. Ultimately, I did not spend nearly as much time covering statistical knowledge as I had hoped to as I spent much more time focusing on the bare minimums of what was required: how to gather data and the tools with which to analyze it. The vast majority of my time was spent familiarizing myself with the ins and outs of the toolsets, navigating webpages to extract data from, and then cleaning the data. On the next page is an approximation of how the project was prepared for, with a following summary of all the steps that were taken throughout the quarter. The project diverted quite significantly from the original plan, and as such so did the expected user experience for the program.

To that end, the program was intended to take a given data set (BoardGameGeek's repository of games, in this case) and provide the user with a series of diagrammatic insights to the data as well as options for exactly what diagrams they would like to see. As it stands, the user must currently run the "bgg_graphs.py" file to generate the analytical graphs and they are hard-coded to generate specific graphs. As this project was only meant to be for half-credit, there is not much of a "user perspective" to detail. Currently, the user will experience some slight annoyance with fact that the files must be opened and altered. Unfortunately, the current state of the program does not have parameters or a UI for customized output so the user must set these variables by opening the file. It was originally planned to be able to take a data set as an argument, then provide a series of options for generating graphs to compare various aspects of the data.

Additionally, it was hoped to allow the user to select the order in which data was gathered and how many games to gather data for. However, after discovering how time intensive the data gathering process is, this will not be in future options. The user can still opt to gather their own data, which will be detailed with the following usage instructions. When it comes to the main experience of generating graphs, after the file has been open and adjusted the user may run it from the console. The console will then describe which graphs it is generating, and if the "display" option was selected the user will get a new open window with the graph they wished to have displayed. After closing or saving the graph, the program will continue displaying messages and graphs in this manner until it displays the "complete" message.

While it would be ideal to still reach the goals of the original plan, after discovering how much data manipulation and cleaning is required for any given analysis, I've realized how much of the tool applications are contextual. In order to draw meaningful conclusions from the data the graphs must typically be more than "X vs Y", which requires extensive data exploration. I may eventually be able to allow parameters for more flexible comparisons, but the user would most likely not be able to draw meaningful conclusions from such graphs.

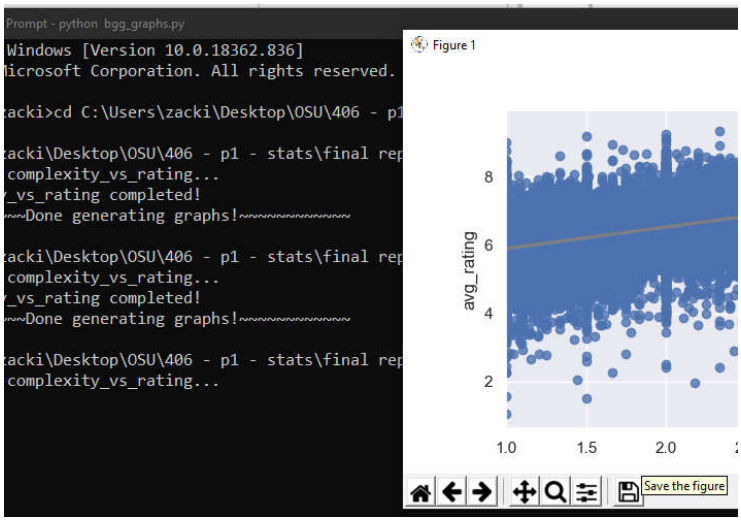

Project Execution
Review
~3 hours
1
Statistics Fundamentals: linear algebra, multivariable calculus
2
Exploratory data analysis: Graphs, quantitative summaries, discrete and continuous variables
Stats Basics
~10 hours
Statistical Thinking: categorical data, binning, measures of spread, InterQuartile Ranges, skews, variance and deviations, etc.
3
Data in Python: reading in and exporting data, numpy and pandas introduction
Importing datasets: packages, category management
Analysis and Visualization
~30 hours
Analysis: covariance, correlation coefficients, exploratory practices
Model development: linear regression, basic visualization modeling
Pandas Frameworks: importing, building, cleaning, exploring, and analyzing
Visualization: DataFrame manipulation, contextual parameter behaviors, display options
4
HTML Review: body structures, navigation, X-Path notation, attributes, values, text, classes, tags, IDs, etc
Scrapy: setting selectors, pathing and targeting, extracting, requests, css locators, response tools, spiders
Selenium: web drivers for generating dynamic JavaScript pages
Data Gathering
~30 hours
BeautifulSoup: content requests, content navigation and filtering
Data Validation: confirming content paths, extracting data, data type handling, trimming and cleaning data
5
fin
Tasks: Running data gathering, trimming and cleaning data, exploratory analysis, model selections, visualizations, and conclusions.
Project!
~15 hours

In order to achieve the limited experience described, prior to being run the user must open the file and adjust the variables within the “Define for program use” section, which will have them select whether they want to display the graphs directly to the screen or save them to a file (or both). If the former is selected, when a graph appears the user must decide what they will do with it (save or simply close out the window) before the program will continue.

```
""" DEFINE FOR PROGRAM USE """
# csv file with BGG data - must have 'mechanics' and 'categories' columns
file_name = 'game_data.csv'

# True is display to screen is wanted - will allow for saving from screen
want_display = False

# True if saved file is wanted, then write the path to where they should be saved
want_file = True
graphs_loc = 'C:\\Users\\zacki\\Desktop\\OSU\\406 - p1 - stats\\testing\\'

# set to True for each graph wanted
pair_plot_graph           = False
average_ratings_vs_years  = False
game_complexity_vs_years  = False
top_mechanics_vs_complexity = False
top_mechanics_vs_years    = False
complexity_vs_gametime    = False
complexity_vs_rating      = False
mechanics_between_1990_2019 = False
min_play_age_vs_complexity  = False

""" END DEFINITIONS """
```

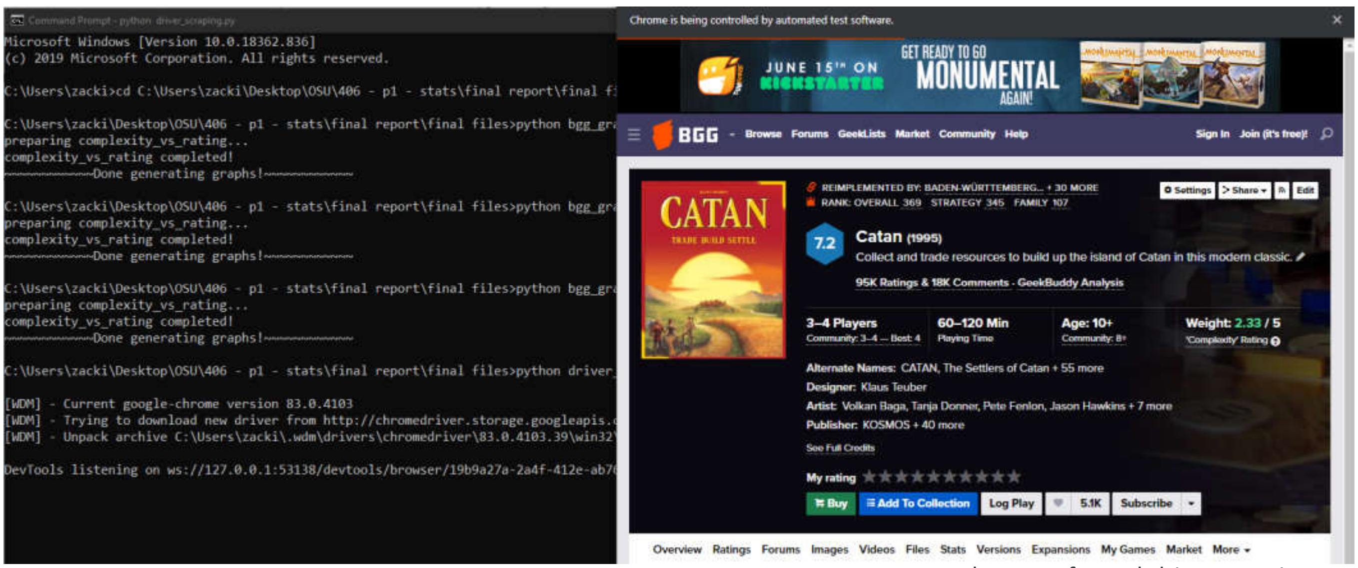
“define for program use” section at the top of bgg_graphs.py

If the latter was selected then the user must also provide the path to where they want to save the graph images, or “graphs_loc” can be set to an empty string (“”) and the graph images will be saved into the same location as bgg_graphs.py. Finally, the user must set the variable for each graph they want to generate to “True”. It has been observed that graph generation is perplexingly inconsistent on rare occasions, so if a graph looks “wrong” the program may simply need to be run again.



popup graph from “display = True”, must be saved or closed

If desired, the user can select from several scraping methods and adjust these options similar to how bgg_graphs.py was prepared. There are three files that were used to approach data gathering from BoardGameGeeks; "driver_scraping.py", "API_scraping.py", and "combined_scraping.py". If more data collection is desired, it is recommended to use "API_scraping" as it is the most reliable as well as the fastest method. However, "driver_scraping" is capable of collecting more information about the games. "Combined_scraping", perhaps predictably, attempts to use both methods to gather data that either one could not alone (eg. Page views or game 'type') but is greatly limited in speed by the webdriver portions. To run any of these files, the must simply enter "python [file_name].py" into their command line. If a gathering method with a driver is selected, a window will pop up that will begin iterating through all of the requested pages – this window must remain open as it is gathering data from the dynamically generated pages.



popup browser for webdriver scraping

Before running data collection, similar to how the "bgg_graphs.py" file was edited the user may edit the variables at the top of the selected file to choose how many games they wish to gather data on and the filename of the csv the data will be saved to. The data will automatically go into a file in the same directory as the chosen scraping file to run. It is intended for the user to be able to select how they want to sort games (by average rating, overall rank, or number of votes) before gathering the top 100X number of games, X being the number of pages the user selects to iterate through, but there are a number of adjustments that need to be made before those become functional. The scrapers currently only go through the games as sorted by number of community votes for the game's rating, as that was the easiest way to filter out games with fewer than 30 votes. This cutoff was decided on because BoardGameGeeks themselves do not consider games below 30 votes to be a meaningful contribution to the data set, as the majority of them are only partial data entries and would skew statistics in an unfavorable manner.

Each scraper gathers game links into a list by making sequential requests to the top X pages of games where the URLs to their gamepages can be scraped, and then if the API is being used the game ID can be taken from the URL for a separate list for the API calls. I believe this may be doing essentially the same work as a Spider, but I never actually implemented a Spider to crawl between pages as it wasn't required for this project. Perhaps for larger data gathering projects this may provide some convenience or inbuilt protections from when a page's HTML is not structured in the expected way.

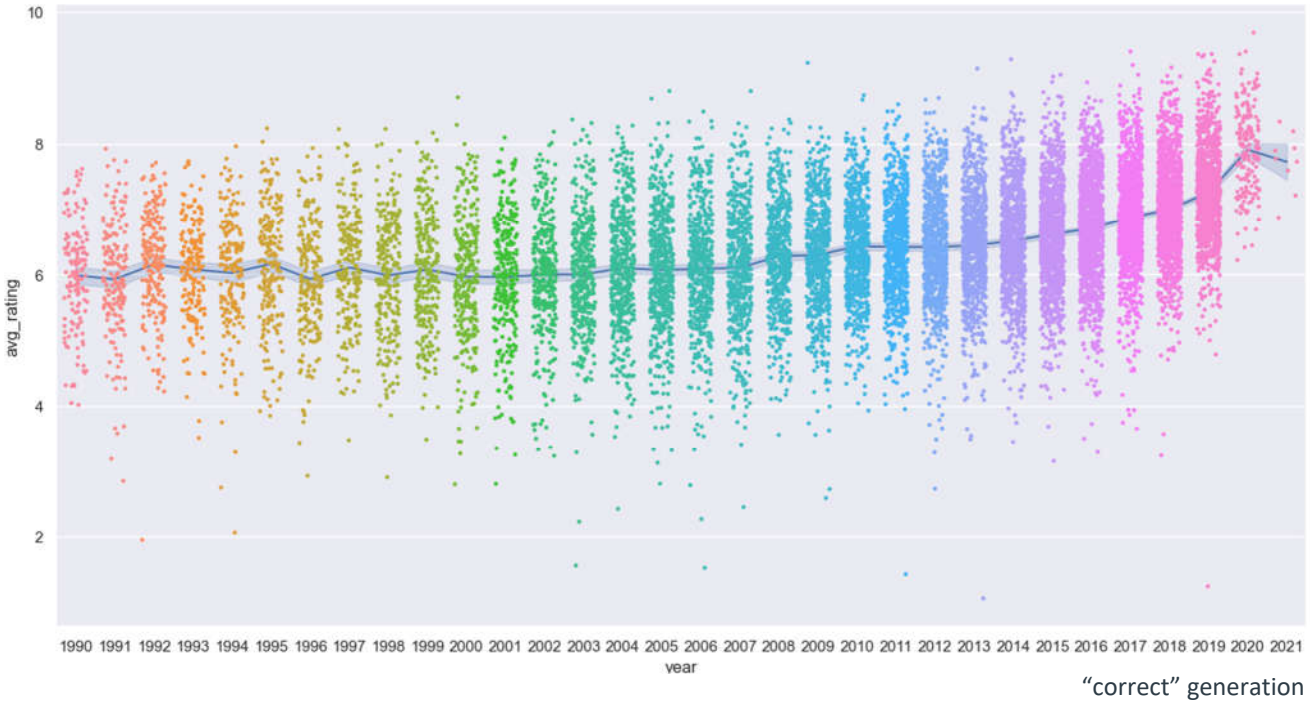
After these lists are generated, the scrapers then use it to begin iterating through all of the board games to start gathering data from either their webpages or API calls. I liked using webpages initially because the "inspect element" feature allowed me to quickly find where in a page's structure an element sat and then generate a path to it. I was also able to gather more data from the loaded webpages, like how many fans follow a game, various pricings for buying it, and how many pageviews it has. However, once familiar with it, using the API was significantly faster and more reliable, as the structure and data were presented in a consistent manner. Regardless of the method used, the scrapers would write rows of data to their respective CSV files on each loop, ensuring that even if the program crashed (which it did, a lot) that dozens of hours of gathering data would not be lost… which I absolutely learned through trial by fire.

An additional advantage of using the API is how much cleaner the data comes out of it. Because of its consistent structure, it's a trivial matter to target the "value" of each desired element, while when extracting data from an element in a dynamic webpage it could be embedded in several layers of unimportant padding and paired with unwanted characters (like '+' in '21+'). When using the web drivers I had to set unique condition checks for each element as the data came out differently for each one, and as such each also required unique cleaning solutions. On the other hand, the API calls simply either had the targeted data or did not, which could be checked for with a simple function to be applied to each sought element.

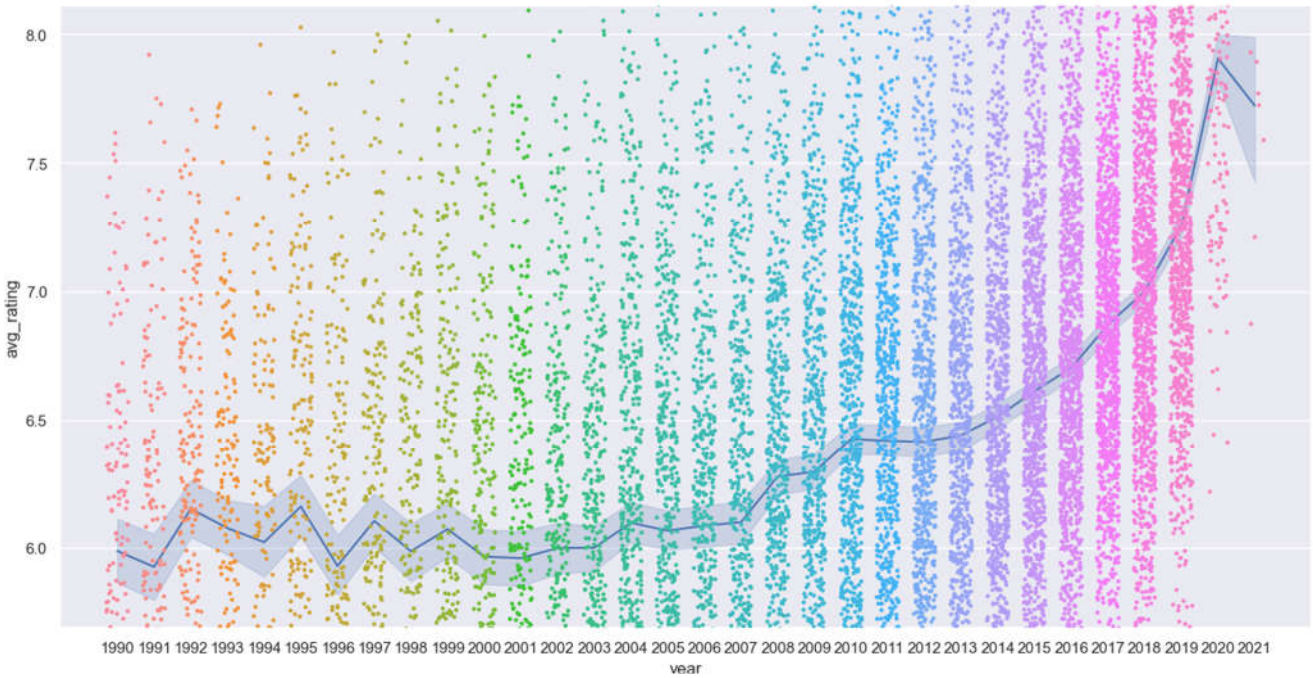
Once the CSV was generated, the majority of the work was exploring, manipulating, and trimming the data. While it was 'clean' for the most part, there were a number of anomalous values that skewed certain comparisons (eg: "9999" for "maximum number of players"). My approach to this work was haphazard and disorganized, which I attribute to not spending as much time on the "statistics" part of "statistical analysis" – I wasn't sure what comparisons to make, what relationships could be considered "significant", or even what a wider variety of comparison methods were outside of the limited toolbox I had prepared for myself. Hopefully as I continue to work with various statistical modeling methods I will better begin to understand which tools are best to use in certain circumstances as well as be able to better interpret their results.

The current bgg_graphs.py file does not have any of this exploration in it, but rather the refined manipulation required for generating specific graphs. While the final code ultimately seems fairly straightforward, it is the result of far more cumbersome methods being paired down to just the essential components and variables being adjusted or generated in ways far more efficient than my initial approaches.

During the data exploration period I encountered a myriad of odd behaviors that produced inexplicable errors, with no real solutions other than “don’t do that”. One of the more surprising results that stuck with me was how the simple swapping of the order for two lines of code, plotting on the same graph, results in the Y-axis of the graph becoming totally mis-represented in one of the generations:

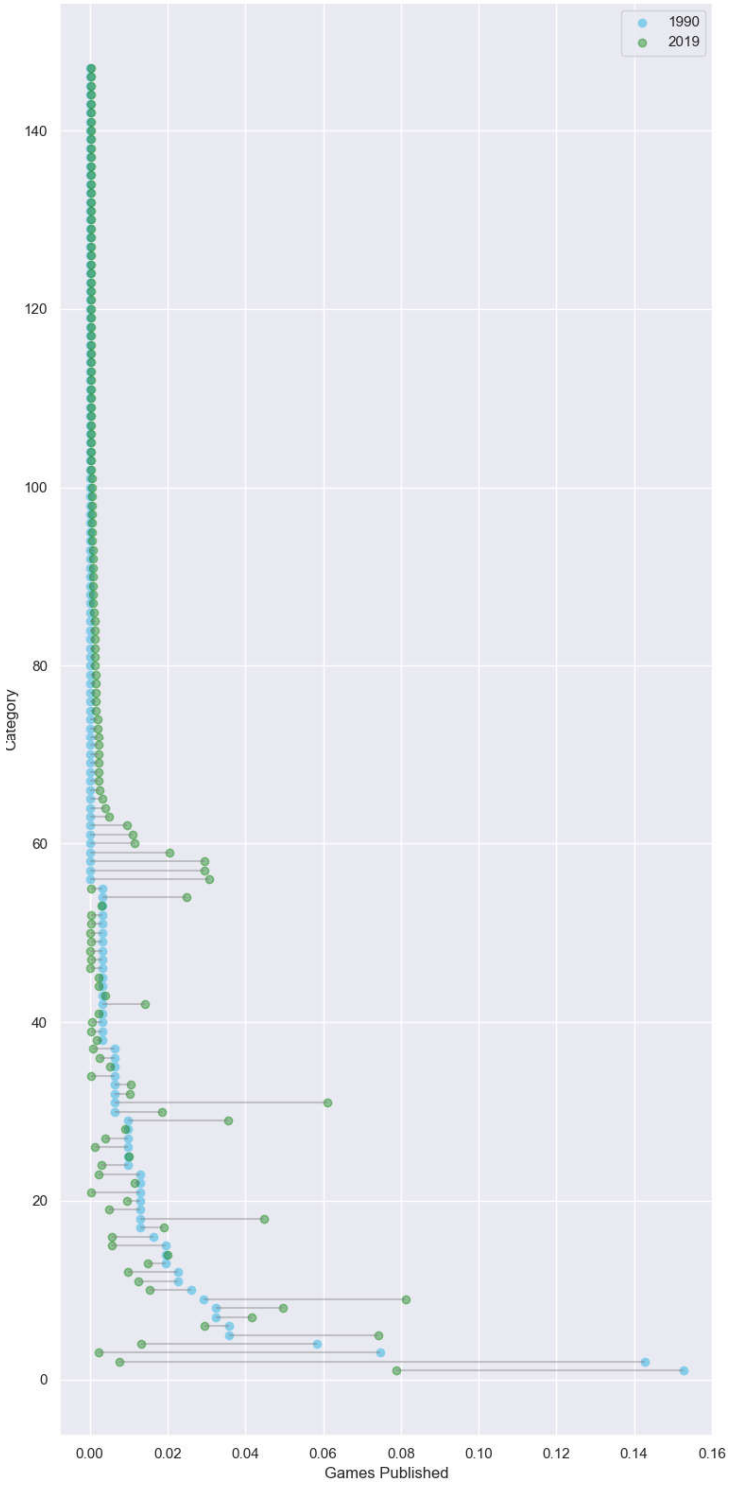


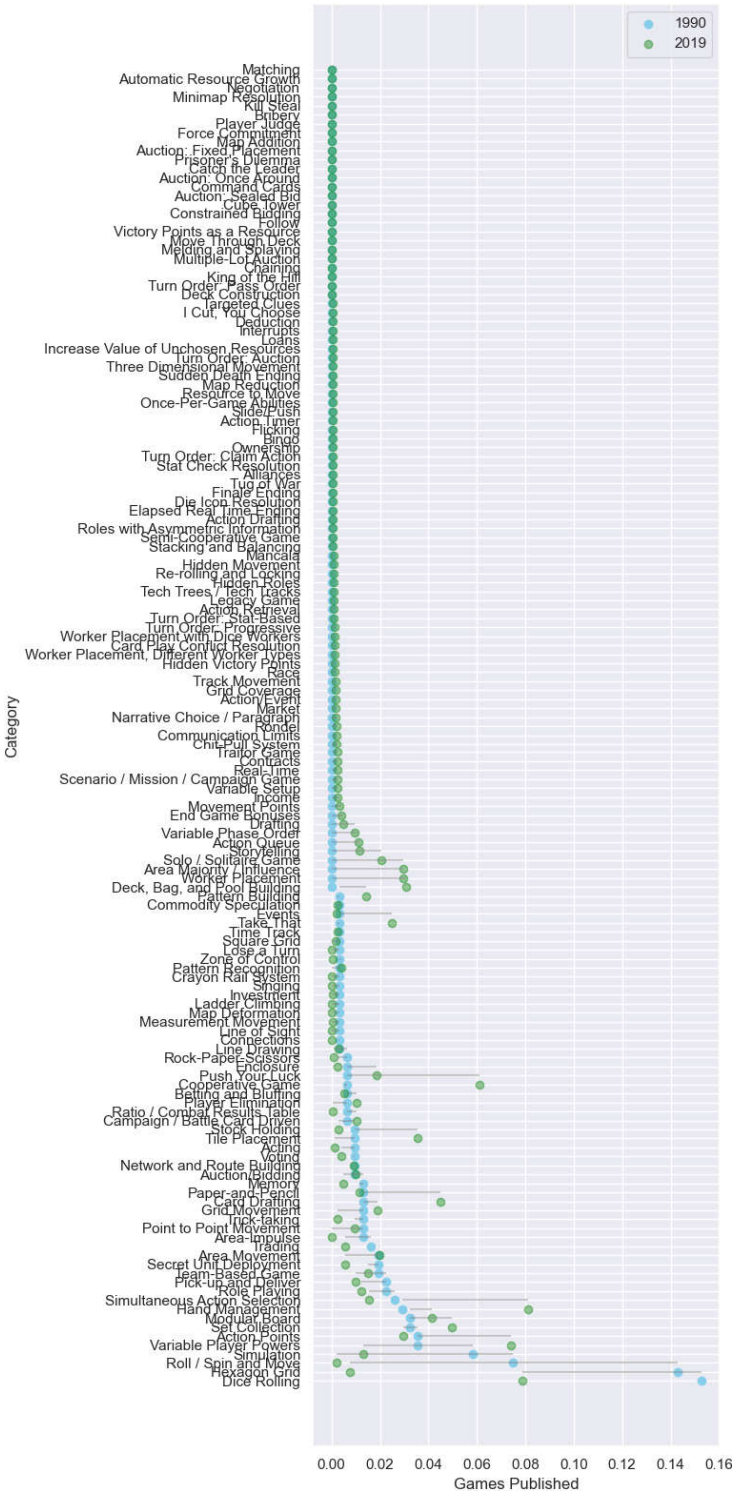
“correct” generation



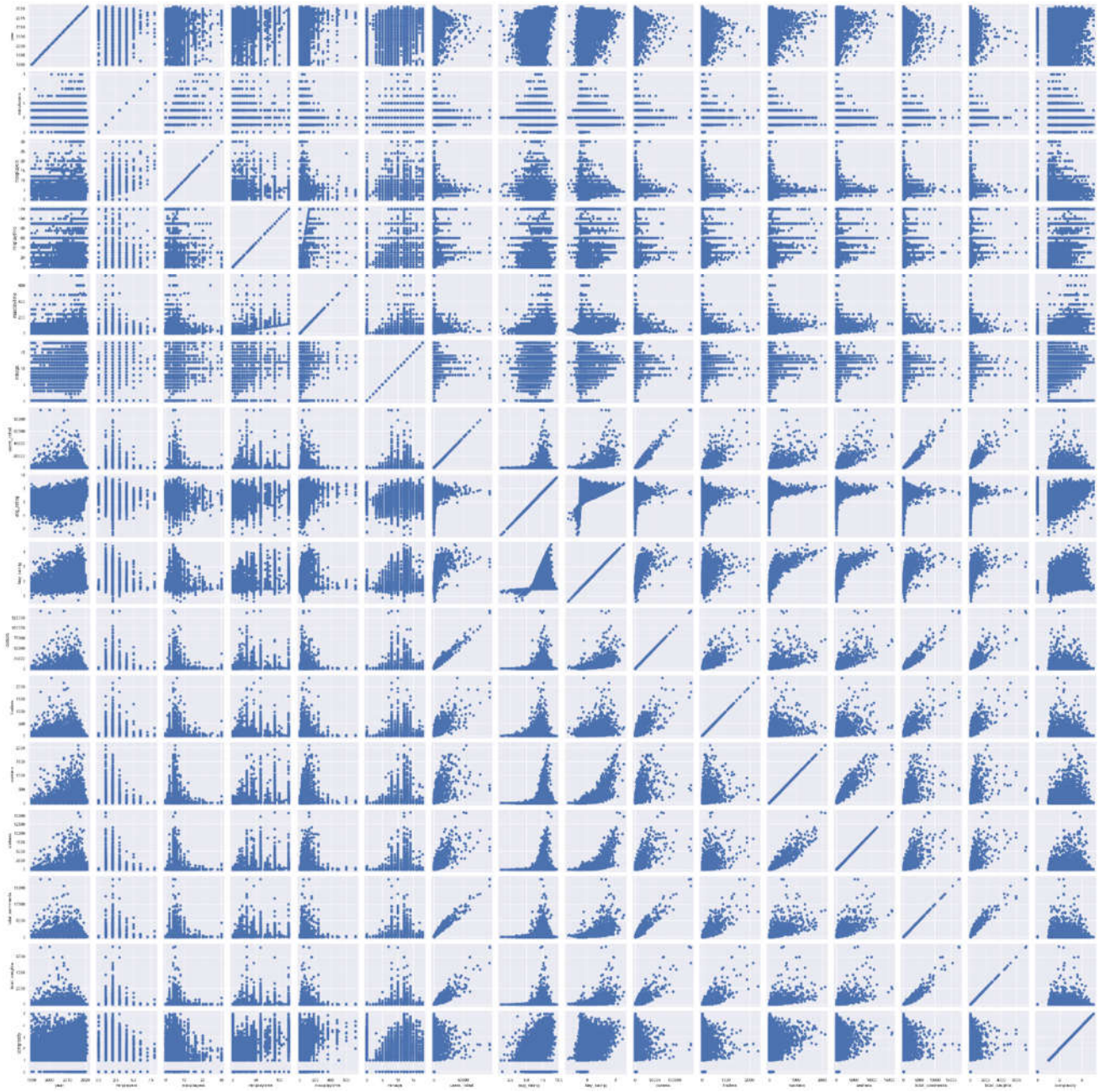
With the order of plotting swapped

Another surprising behavior emerged when I was attempting to measure the most popular board game mechanics in specific years, then measure the difference between them. When I attempt to simply label the y-axis with the appropriate mechanics, the plotting of the data points shifts down for only one year, and the two year no longer match up with their respective lines on the graph:



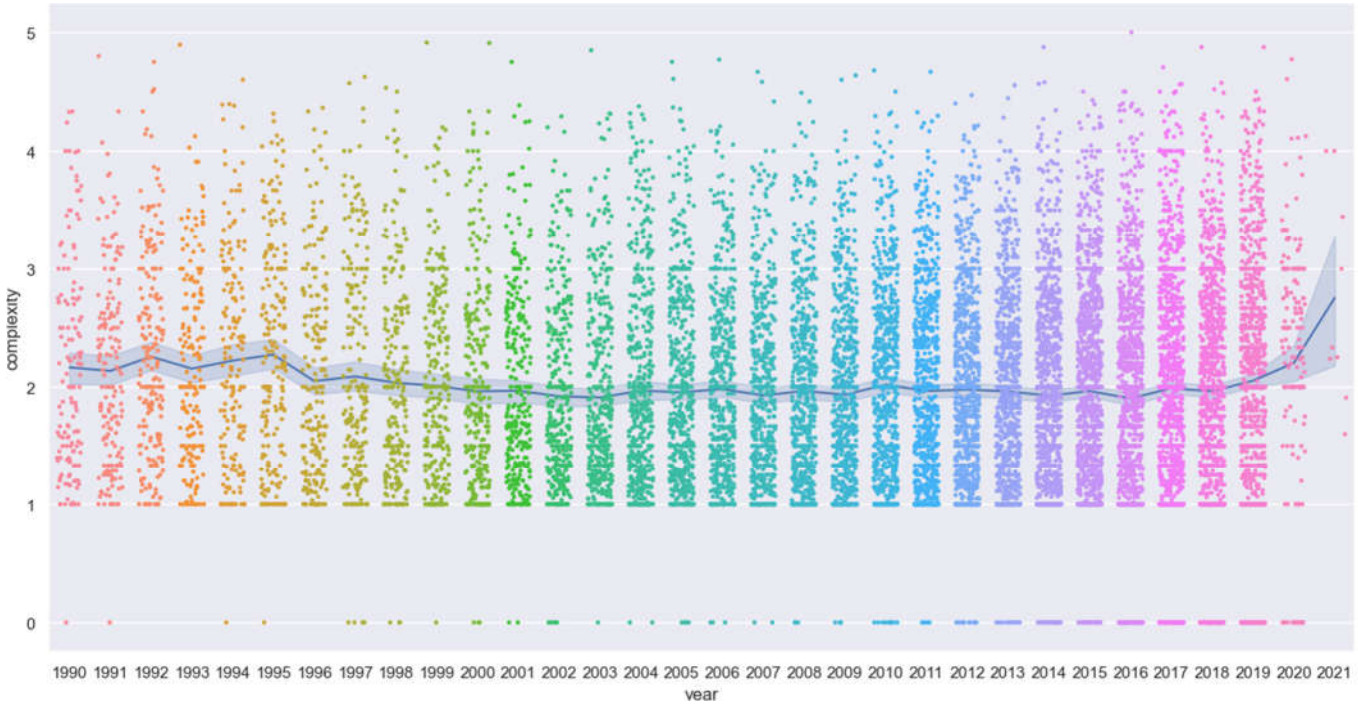


After this discovery process, I began to write bgg_graphs to use the CSV generated from a scraper in the most efficient ways possible. I trimmed out some or the data I did not think would be relevant and wrote pair_plot() to provide a simple scatterplot comparison of values to see if there were any interesting correlations to draw on.



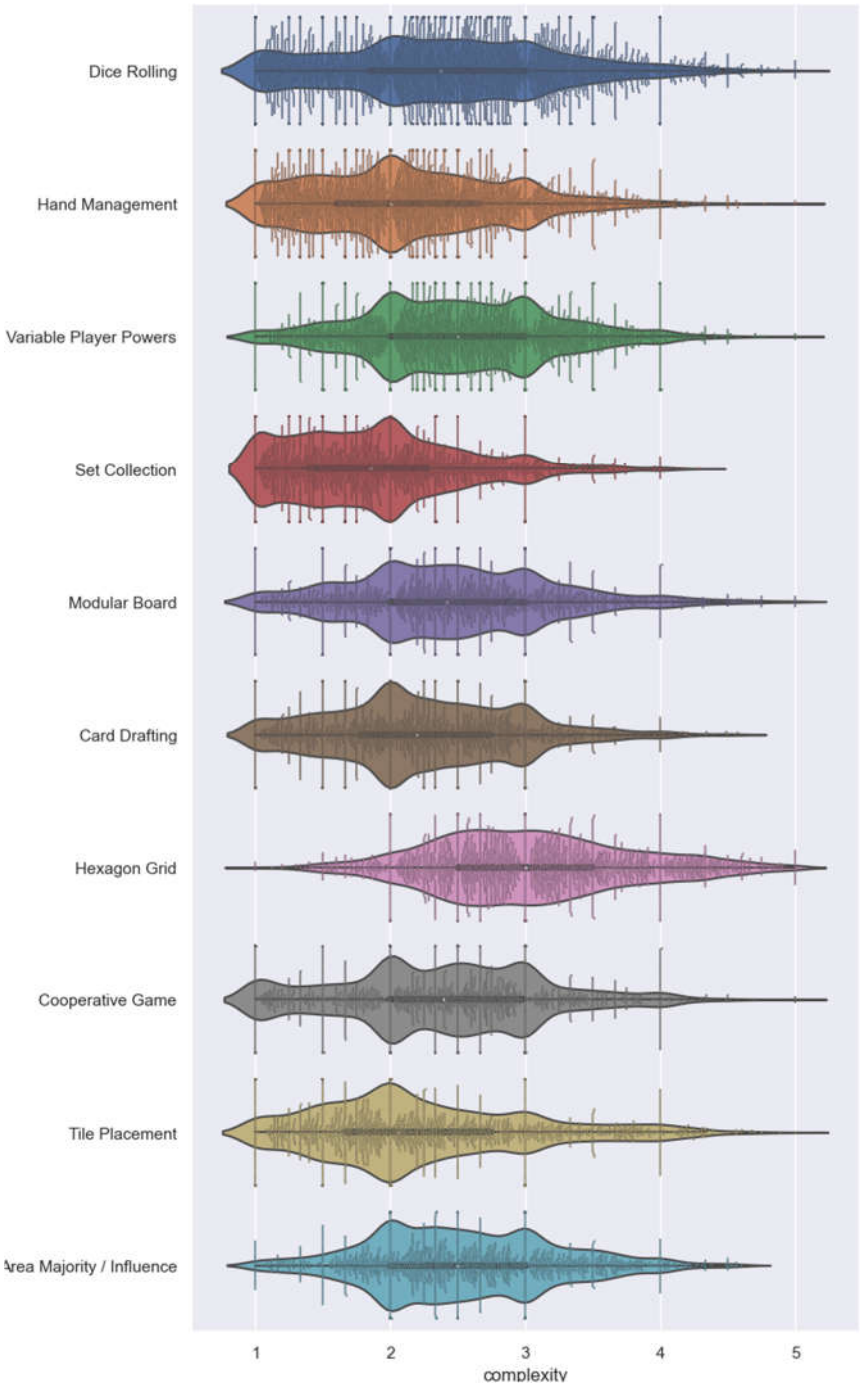
At a glance, I can see data trends that inspired further exploration in the set. That said, I still pursued somewhat predictable relationships (like complexity and playtime) just to see if closer inspection would reveal any new observations. They did not.

One of the functions I wrote to accept an argument is “col_vs_year()” which can take a column in the dataframe and explore its trend through the years, binning the data by year published to display density and trends from within each year as well. The graph presented on page 6 was generated using this method, with that one being the trend of ratings by year. This result could be interpreted as either the quality of games improving as the industry learns and grows, or consumers simply preferring newer games over the old – although it’s hard to make such distinctions between games since there are typically very few qualities which “date” it besides the wear and tear of the product. Another graph generated using this method looks at the perceived complexity of games throughout the years, which is a fairly consistent factor between all games overall.



This observation, along with knowing that games overall have a trend to have more mechanics every year, leads me to wonder if there are specific categories of games that are increasing in complexity through the years and if those would account for the increase in overall average for number of mechanics. Interestingly, another graph showed that there is a positive correlation between complexity and rating of a game – which could either be a bias in need of normalization or a tendency of developers with more ambitious games to also be planned and executed better. This in conjunction with the two col_vs_year() graphs makes me believe it would be the latter, as complexity has no positive trend over the years.

One of the major hurdles I faced was deciding how to handle the dataframe cells that would contain multiple entries, like all of the mechanics for a game. While the scrapers could have potentially parsed and entered values into 40 separate columns for each game’s categories, this would have resulted in an obnoxiously structured data set that was too wide to be wielded efficiently. As such, bgg_graphs needed to be developed to search through the Mechanics and Categories columns and select game entries as appropriate for the context.

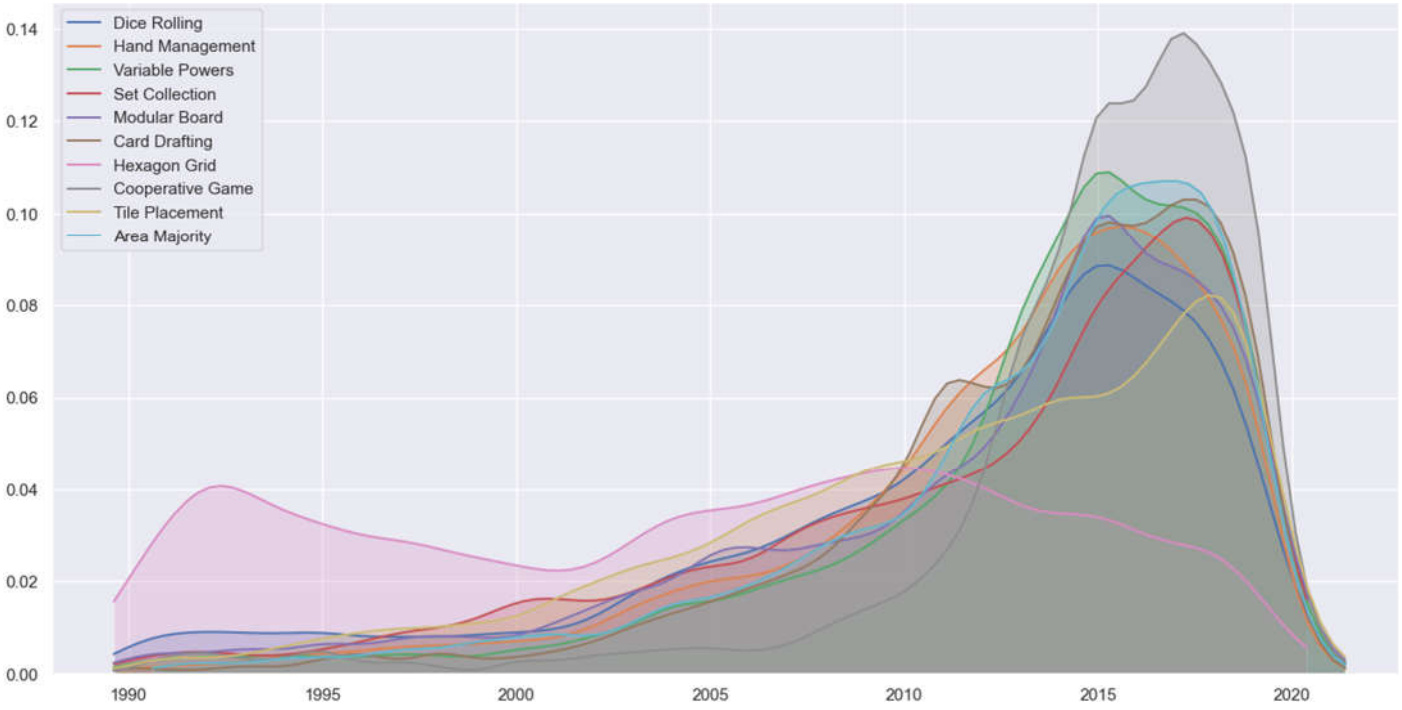


Top mechanics vs complexity

The first of the methods to implement this is top_mechs_vs_complexity(), which is intended to find the top 10 most popular mechanics from all games, and then compare their relative complexities, displaying both their trends and clusters of data values. In addition to being able to notice obvious differences between types if mechanics, like the tendency for games utilizing hexagonal grids to be significantly more complex, we can also see the trends within each set. For instance, hexagonal grid game complexities have a fairly gaussian distribution, while set collection are heavily weighted to the easier end.

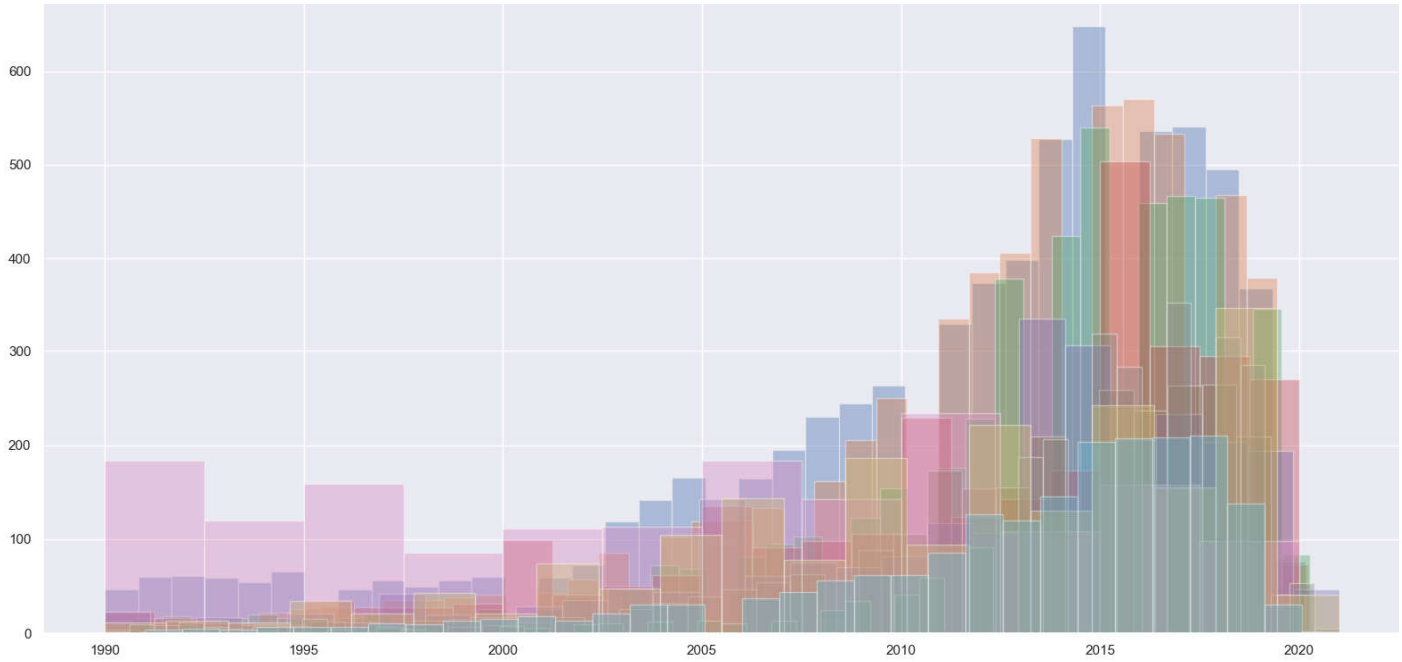
In order to facilitate these sorts of comparisons I have considered targeting the scrapers towards pages with collections of specific game types, but the web driver method of collection would be more tiresome to deal with than simply parsing down the data as it is fed into bgg_graphs.py. If there is a method of doing this through the API, then it would be viable.

The next method was also mechanics-focused, and was also able to use the "adj_mechs()" method I wrote to isolate and simplify rows of games with the mechanics I sought. This method displays the top 10 mechanics as percentages of the total games published through "recent" (last 30) years. This is "recent" because I still remember the 90's far too well for it to have possibly been too long ago.



Top mechanics from all time, from 1990 to 2019

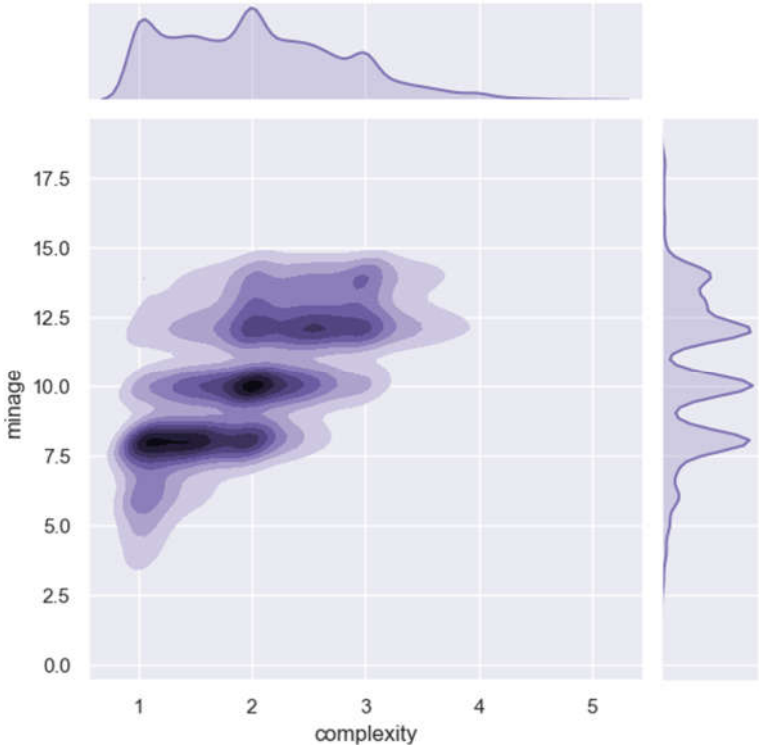
I was surprised to find hexagonal grid games as a standout once again, remaining remarkable consistent, with a trend down in the last 10 years, in its market share of the overall number of games published. This prompted me to look at the actual number of games published for each year, which resulted in a pleasantly artistic graph (if not particularly readable) that was automatically generated with different yearly binning for each category.



"art is hard" - Cursive

This, in turn, led to the last meaningful graph attempted to be generated with the data from the web scrapers – mechanics_years_dumbell() – which can take two years as arguments and graph the difference in mechanics between the two years. This is the graph on page 7 that bumps only some data points down. While writing this paragraph I realized my attempts to reference the string values of the indices could be resulting in odd behavior similar to when I first attempted to plot col_vs_year() and had to convert 'year' to a string value to plot on the same numerical axis as lineplot. I've just adjusted the code and it works as intended, meaning graph will be generated correctly from now on (except now the mechanics text is too large and overlaps…). I've left this in as it is a perfect example of my workflow, and clearly indicative of how much more depth there is to the tools have I only just begun to familiarize myself with.

The final function worth mentioning is "vs_heatmap()" which takes two columns from the provided dataframe as arguments and generates a heatmap of their relationship, as well as having density plots along each axis. While this tool has the potential to reveal unexpected groupings of data points, I simply understand it as another identifier of correlations between sets. It is much simpler than most of the other methods, and as such was easier to parameterize. The chosen graph currently shows the strongest groupings of recommended minimum age to play a game and its complexity, which is a fairly predictable result.



As stated before, hopefully I begin to better understand when specific tools are best applied as I get back into the statistics portion of this project. The original plan had called for at least half of the project timeline to have been focused on Statistical Analysis methods, but in order to get into those it was required to be much more familiar with the tools being used to do the analysis. Fortunately, my development environment in PyCharm allowed me to integrate nearly every aspect of this project into it. Over the course of the quarter I used resources from DataCamp, Udacity, Coursera, and edX as well as numerous forum-style sites (like StackOverflow) to slowly build a toolbox that I might bring to analyze some data. At first I was only using Numpy for basic numerical manipulations, which was paired with Pandas when I began working with dataframes that would store and organize the data I'd be using. I also used matplotlib and Seaborn to generate basic graphical comparisons of certain stats within data sets I gathered from online repositories like Kaggle. When it came to gathering my own data, I

used Python's inbuilt Requests for initial gathering, and Scrapy's Selector to parse static HTML. When it came to scraping webpages dynamically generated with JavaScript, I had to turn to Selenium's webdriver to generate and iterate through various pages. To facilitate this, I used a ChromeDriverManager that would make sure the webdriver was up to date each time it ran. Eventually I became familiar enough with the toolset to use portions of a web scraper developed by Sean Beck to use BoardGameGeek's web API to gather data much faster and more reliably, as the API structure was far more consistent and required fewer requests to be made. I also used BeautifulSoup to parse the returned HTML from BoardGameGeek's API, along with a function from Sean Beck that was able to directly target elements and extract values from them. For all methods of requesting, I used Python's Time module to delay requests as to not overload BoardGameGeek's servers (and so I wouldn't be blacklisted from accessing them).

Near the conclusion of the project I looked back on my original project plan as I began putting together the Project Execution timeline to show what I actually spent my time on. In retrospect, it is easy to see that I didn't allocate nearly enough time to address any more than the very basics of each topic I touched on this quarter. Even after spending significantly more time on the visualization, data gathering and cleaning, and data exploration than I ever intended, I feel like I have not broken the surface. I have no doubt that a better understanding of the tool set will come hand-in-hand with a better understanding of the statistical principles which guided the development of these tools, but to be perfectly honest it's much more difficult to stay motivated to focus on "just math" when there are so many tantalizing coding tools to learn about. After spending more time grinding through resources on statistics, I will return to this project and see if I can make any insightful observations on the data. Ideally I will have a better idea of which tools I would like to implement and how to best use them. Ultimately, I hope to complete the intended minimal user command-line interactions to select graphs. I will also begin preparing it to be integrated with the next half of the project which will be focused on the basics of Machine Learning.