



# Data Analytics

April-June, 2023

“Brocommender”

Robin DEBESSEL

## Table of content

<i>Table of content.....</i>	<i>2</i>
<i>Introduction.....</i>	<i>3</i>
<i>Project Plan, made with Trello.....</i>	<i>4</i>
<i>Data collection &amp; Data sources.....</i>	<i>4</i>
<i>Overall Process.....</i>	<i>5</i>
<i>Data Cleaning, EDA and Visualization.....</i>	<i>9</i>
<i>To SQL or not to SQL.....</i>	<i>18</i>
<i>Entity Relationship Diagram (ERD) .....</i>	<i>18</i>
<i>MySQL Queries .....</i>	<i>21</i>
<i>Machine Learning .....</i>	<i>23</i>
<i>Conclusion.....</i>	<i>24</i>

## Introduction

"**Brocommender**" aims to develop a recommendation algorithm that fits to the individual preferences of a gamer using their Steam ID.

The end goal of this enterprise is to generate a short list of game recommendations that aligns with the genres that the user has spent most of his time playing on the Steam platform.

The recommendations offered by the algorithm are intended to be personalized, hopefully leading the users to a higher likelihood of discovering new games they would enjoy and might have missed or lost track of, enhancing their engagement with the platform while respecting their real motivations.

### Context that explains my choice:

So often we nowadays find ourselves "trapped", if not directly through plain manipulation, at least through not-always-wanted redirection by algorithms tailored for our satisfaction but also accompanied by an ever present incentive : being the most profitable.

My will was to tackle this issue, through a minimalist and unoppressive interface that doesn't contain hundreds of references on one page, most of them being featured and displayed for commercial purpose, with sales tags from top to bottom.

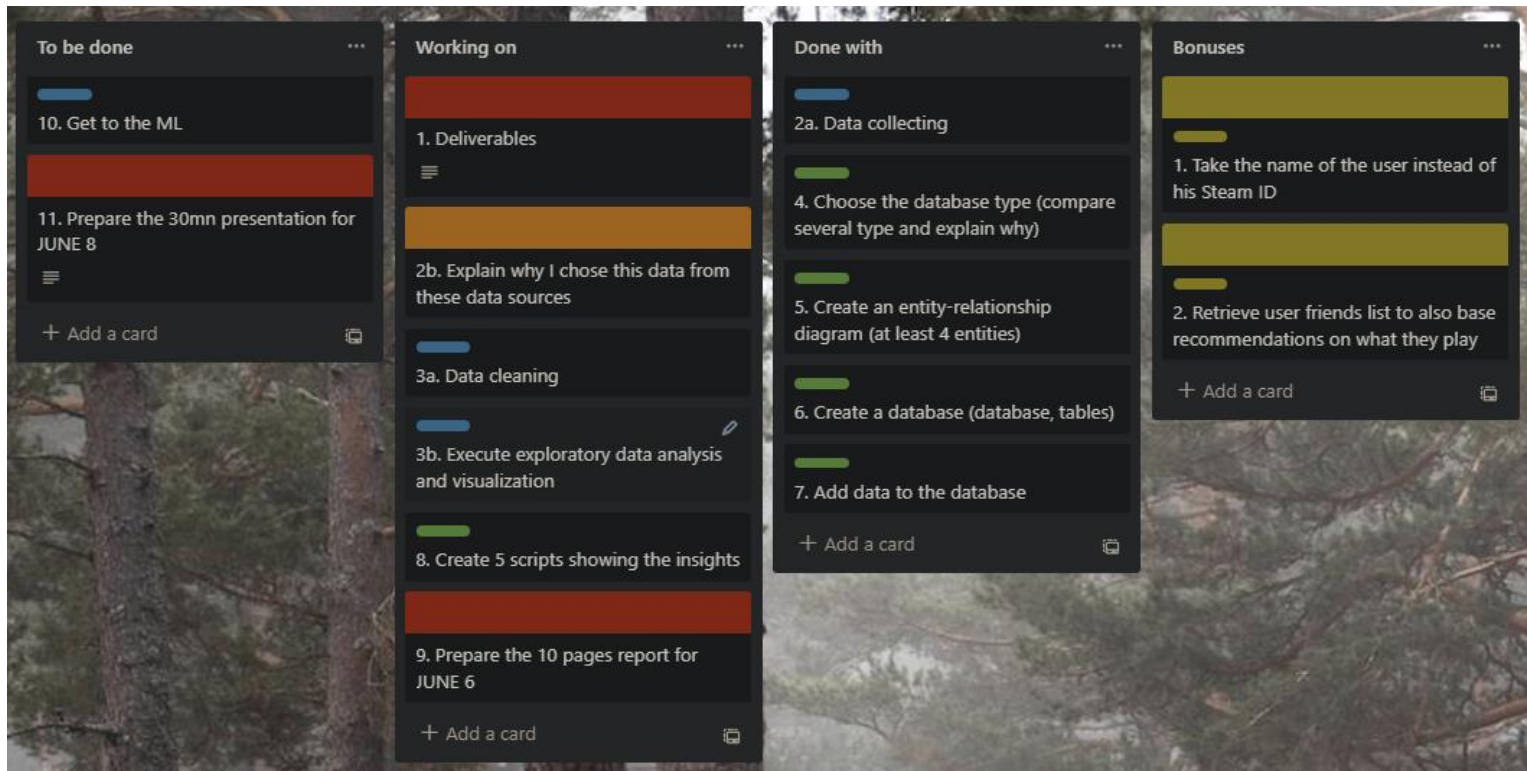
The goal for it was, and will stay, to be interactive enough to keep engagement with the user while getting rid of the things they did not ask to see in order to stay on track and really chase and find what they want and not what "we" want them to find and buy.

### A small disclaimer:

My original ambitions for this project were high and I had to tone them down due to the complexity of tasks.

"Brocommender" asks to be updated (quite a lot) for it to fit my vision but this will be addressed later in development.

## Project Plan, made with Trello



*This snapshot is dated from Tuesday the 6<sup>th</sup> @12:30pm*

## Data collection & Data Sources

### Data Sources:

- The primary data source for the project was a dataset downloaded from [Kaggle](#), a well-known online community of data scientists and machine learning practitioners. The dataset, although four years old, provides a comprehensive snapshot of various game features, which is crucial for the project.

The main attributes considered from the data include the game's genre, the amount of time played, and user ratings.

This dataset is 27000+ rows long and has 18 different columns.

- The second data source we are going to use is the user's Steam games library. This will be taken care of using the Steam API to dynamically retrieve a user's account information in order for us to gather the required data.

### The biggest challenges I faced:

One of the significant challenges during the data collection stage was the use of the Steam API wrapper for Python. As a new concept, I misunderstood it to be the API itself, which led to a few obstacles, during the implementation process.

However, once tinkered with, the Steam API wrapper turned out to be beneficial and helped me out with the process of extracting vital information about the games.

It provided a simple and efficient way to gather essential data elements necessary for our algorithm without having to collect all the data manually from the Steam platform through their basic API, which would have been less practical and probably more time consuming.

The other big difficulty was managing the complex data structure that the genres and other crucial information were embedded in.

Extracting them proved to be a tedious task that took numerous hours.

## Overall Process

The development process for the "Brocommender" project was organized into three separate notebooks, each serving a distinct purpose for safety and clarity measures.

The first notebook was designed as a platform to install and utilize the Steam API and its Python wrapper. This notebook was instrumental in making queries to the Steam API, gathering the required data that is fundamental to the recommendation algorithm. In the upcoming project demonstration, this notebook will serve as the primary showcase, highlighting how we interact with the Steam API to extract game and user data:

### API CONFIG

```
1 from steam import Steam # Steam API Library
2 from decouple import config
3
4 # Retrieves my STEAM_API_KEY in my environment variables
5 KEY = config("STEAM_API_KEY")
6 # Creates a class "Steam" based on the API KEY previously retrieved
7 steam = Steam(KEY)
```

Python

# GET USER'S STEAM\_ID

```
1 # user_steam_id
2
3 # arguments: user_steam_id (input)
4 user_steam_id = input("Please enter your Steam ID : ")
5
6 print("Your Steam ID is :", user_steam_id)
```

Python

Your Steam ID is : 76561197961064971

```
1 # user_details_from_steam_id
2
3 # arguments: steam_id
4 user_details = steam.users.get_user_details(user_steam_id)
5 pprint.pprint(user_details)
```

Python

```
{'player': {'avatar': 'https://avatars.steamstatic.com/1ee3b98c71785486a9b89ae2ef6f3a2ad3ef1919.jpg',
'avatarfull': 'https://avatars.steamstatic.com/1ee3b98c71785486a9b89ae2ef6f3a2ad3ef1919_full.jpg',
'avatarhash': '1ee3b98c71785486a9b89ae2ef6f3a2ad3ef1919',
'avatarmedium': 'https://avatars.steamstatic.com/1ee3b98c71785486a9b89ae2ef6f3a2ad3ef1919_medium.jpg',
'commentpermission': 1,
'communityvisibilitystate': 3,
'lastlogoff': 1685892056,
'personaname': 'Pastek',
'personastate': 0,
'personastateflags': 0,
'primaryclanid': '103582791429521408',
'profilestate': 1,
'profileurl': 'https://steamcommunity.com/id/BoFPastek/',
'steamid': '76561197961064971',
'timecreated': 1063987237}}
```

From the user's data, we can then request to know about his entire games library:

```
1 # user_games_library
2
3 user_games_library = steam.users.get_owned_games(user_steam_id)
4 pprint.pprint(user_games_library)
```

Python

```
{'game_count': 97,
'games': [{'appid': 10,
'content_descriptorids': [2, 5],
'img_icon_url': '6b0312cda02f5f777efa2f3318c307ff9acafbb5',
'name': 'Counter-Strike',
'playtime_forever': 795,
'playtime_linux_forever': 0,
'playtime_mac_forever': 0,
'playtime_windows_forever': 0,
'rtime_last_played': 1403918646},
{'appid': 20,
'content_descriptorids': [2, 5],
'img_icon_url': '38ea7ebe3c1abbbbf4eabdbef174c41a972102b9',
'name': 'Team Fortress Classic',
'playtime_forever': 26,
'playtime_linux_forever': 0,
'playtime_mac_forever': 0,
'playtime_windows_forever': 0,
'rtime_last_played': 86400},
{'appid': 30,
'img_icon_url': 'aad0ce51ff6ba2042d633f8ec033b0de62091d0',
'name': 'Day of Defeat',
'playtime_forever': 0,
'playtime_linux_forever': 0,
'playtime_mac_forever': 0,
'playtime_windows_forever': 0,
'rtime_last_played': 0},
```

```

1 # top_10_app_ids
2 if len(user_games_library) == 0:
3     print("Error : Your games library seem to be empty.\nIt's then impossible to recommend you a new game based on your preferences.\nSowwy.")
4 else:
5     # Orders all the games by "playtime_forever" in descending order
6     sorted_games = sorted(
7         user_games_library['games'], key=lambda x: x['playtime_forever'], reverse=True)
8
9     # Retrieves app_idRécupère les "appid" des 10 premiers jeux avec le plus de "playtime_forever"
10    top_10_apps_ids = [game['appid'] for game in sorted_games[:10]]
11    print("Top 10 AppIDs:", top_10_apps_ids)

```

Python

Top 10 AppIDs: [570, 238960, 730, 393420, 1245620, 526870, 435150, 440, 252950, 1604030]

```

1 # Quick recap
2 for game in sorted_games[:10]:
3     app_id = game['appid']
4     app_name = game['name']
5     playtime_forever = game['playtime_forever']
6     playtime_hours = playtime_forever / 60
7     print("app_id:", app_id, " - ", app_name, " - Total Playtime:",
8         playtime_forever, "or", round(playtime_hours), "hours.")

```

Python

```

app_id: 570 - Dota 2 - Total Playtime: 180723 or 3012 hours.
app_id: 238960 - Path of Exile - Total Playtime: 91380 or 1523 hours.
app_id: 730 - Counter-Strike: Global Offensive - Total Playtime: 41370 or 690 hours.
app_id: 393420 - Hurtworld - Total Playtime: 5771 or 96 hours.
app_id: 1245620 - ELDEN RING - Total Playtime: 3699 or 62 hours.
app_id: 526870 - Satisfactory - Total Playtime: 3080 or 51 hours.
app_id: 435150 - Divinity: Original Sin 2 - Total Playtime: 2300 or 38 hours.
app_id: 440 - Team Fortress 2 - Total Playtime: 2288 or 38 hours.
app_id: 252950 - Rocket League - Total Playtime: 2216 or 37 hours.
app_id: 1604030 - V Rising - Total Playtime: 1411 or 24 hours.

```

After this we will use the games IDs (app\_id here) to get the information relative to each game or app:

## GET THE CORRESPONDING GAME GENRES FOR EACH GAME

```
1 top_10_apps_ids
```

Python

```
[570, 238960, 730, 393420, 1245620, 526870, 435150, 440, 252950, 1604030]
```

We then use a JSON to hold the retrieved information

### Creating the necessary JSON to retrieve the information we want

```

1 # apps_details
2
3 # arguments: app_id
4 app_details = []
5 apps_details = []
6
7 apps_data = {}
8
9 for app in top_10_apps_ids:
10
11     app_details = steam.apps.get_app_details(app)
12     apps_details.append(app_details)
13
14     apps_data[app] = json.loads(app_details)

```

Python

A few fun features we can now work with to offer different links to different websites for the games we want:

```
1 # Steam Store app link
2
3 app_id_store_url = "https://store.steampowered.com/app/" + str(app_id)
4 app_id_store_url

'https://store.steampowered.com/app/1604030'

1 # Official website app link
2
3 apps_data[570]["570"]["data"]["website"]

'http://www.dota2.com/'

1 # Metacritic website app link
2
3 mc_score = apps_data[570]["570"]["data"]["metacritic"]["score"]
4 mc_link = apps_data[570]["570"]["data"]["metacritic"]["url"]
5
6 print("The score this game has been given on metacritic.com is :", mc_score, "/ 100")
7 print("You can find this game's review here :", mc_link)

The score this game has been given on metacritic.com is : 90 / 100
You can find this game's review here : https://www.metacritic.com/game/pc/dota-2?ftag=MCD-06-10aaa1f

1 apps_data[570]["570"]["data"]["detailed_description"]

"<strong>The most-played game on Steam.</strong><br>Every day, millions of players worldwide enter battle as one of over a hundred Dota heroes. And no matter if it's their 10th hour of p

1 apps_data[570]["570"]["data"]["header_image"]

'https://cdn.akamai.steamstatic.com/steam/apps/570/header.jpg?t=1682639497'
```

Afterwards, we can finally gather all our different genres for each games

```
1 top_10_apps_ids_genres = []
2
3 for x in top_10_apps_ids:
4     for item in apps_data[x][str(x)]["data"]["genres"]:
5         genre = item["description"]
6         top_10_apps_ids_genres.append(genre)
7
8     pprint.pprint(genre)
9
10 genre

'Action'
'Free to Play'
'Strategy'
'Action'
'Adventure'
'Free to Play'
'Indie'
'Massively Multiplayer'
'RPG'
'Action'
'Free to Play'
'Action'
'Adventure'
```



We also do this for the game's categories in case we need later to build a better recommender.

Then we have to count our different genres but not before translating them to english as sometimes the info we get from the API is not in english. It's hard to tell why this happens as it really seems to be randomized, especially for the biggest and most played games of the platform.

This is what we end up with!

```
1 # Sort the keys based on their values in descending order
2 top_3_played_genres = sorted(genres_value_counts, key=lambda k: genres_value_counts[k], reverse=True)[:3]
3
4 # Print the top 5 keys and their corresponding values
5 print("The top 3 genres of games you usually play on the Steam platform are:")
6 for key in top_3_played_genres:
7     pprint.pprint(f"{key}: {genres_value_counts[key]}")
```

```
The top 3 genres of games you usually play on the Steam platform are:
'Action: 8'
'Adventure: 5'
'Free to Play: 4'
```

```
1 # Sort the keys based on their values in descending order
2 top_3_played_categories = sorted(categories_value_counts, key=lambda k: categories_value_counts[k], reverse=True)[:3]
3
4 # Print the top 5 keys and their corresponding values
5 print("The top 3 categories of games you usually play on the Steam platform are:")
6 for key in top_3_played_categories:
7     pprint.pprint(f"{key}: {categories_value_counts[key]}")
8
```

```
The top 3 categories of games you usually play on the Steam platform are:
'Multi-player: 10'
'Co-op: 8'
'Steam Trading Cards: 7'
```

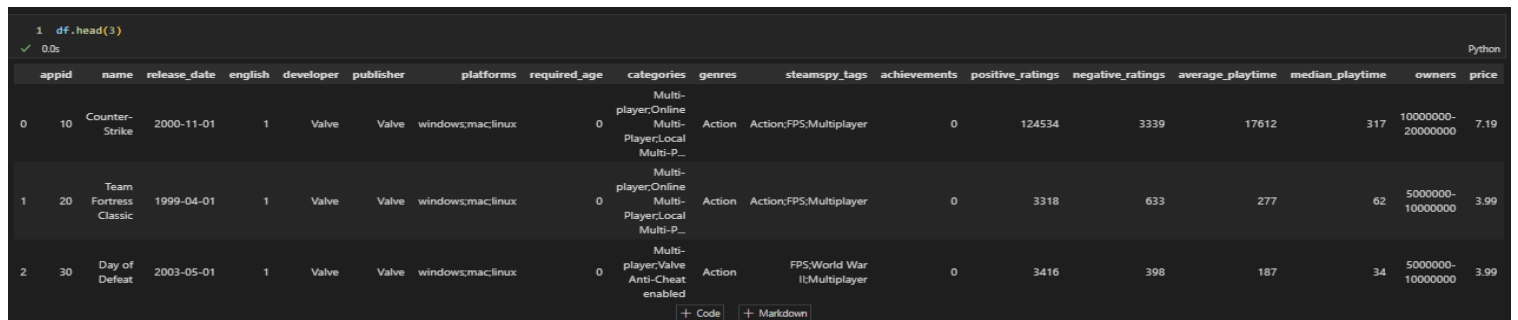
## Data cleaning, Exploratory Data Analysis and Visualization

The second notebook was employed to import, clean, and visualize our primary dataset obtained from Kaggle. It has a vital role in the data preprocessing stage of the project. It includes a few steps to clean the data, ensure its consistency, perform some exploratory data analysis through various visualizations.

We then have to export the cleaned data as SQL tables via appropriate data frames. This process will ensure the data fed into our machine learning model is qualitative as we can make it in order to improve the reliability of our recommendation algorithm.

We will detail this in the following parts of our report.

Our data being very clean from the get go, we are quickly going to go through this part of the report. At least when it comes to coding snippets.



	appid	name	release_date	english	developer	publisher	platforms	required_age	categories	genres	steamspy_tags	achievements	positive_ratings	negative_ratings	average_playtime	median_playtime	owners	price
0	10	Counter-Strike	2000-11-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Online Multi-Player;Local Multi-P...	Action	Action;FPS;Multiplayer	0	124534	3339	17612	317	10000000-20000000	7.19
1	20	Team Fortress Classic	1999-04-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Online Multi-Player;Local Multi-P...	Action	Action;FPS;Multiplayer	0	3318	633	277	62	5000000-10000000	3.99
2	30	Day of Defeat	2003-05-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Valve Anti-Cheat enabled	Action	FPS;World War II;Multiplayer	0	3416	398	187	34	5000000-10000000	3.99

We can however meet and greet our first and biggest frustration for this project:

As you can see in the screenshot, a very interesting column we could use would be the “steamspy\_tags”. They would complement our game genres data greatly.

Unfortunately, after spending a few hours on this matter, I could not find a way to retrieve said tags from any of the game information responses to our request.

Which means that for this version of “**Brocommender**”, we will have to do without...

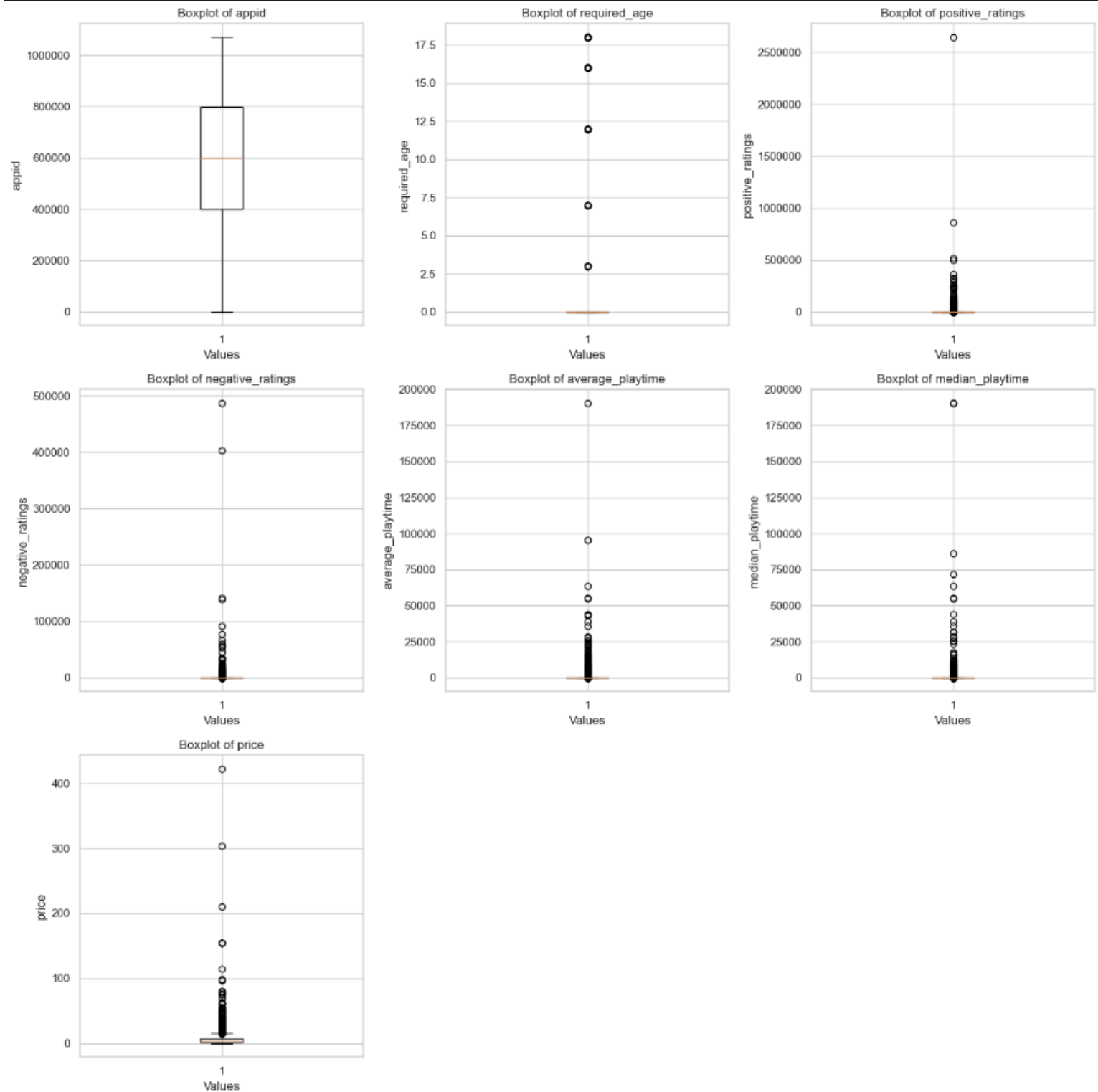
*We can then move onto our next important operation:*

### Categorical and Formatting Operations

Converting Columns Variables to DateTime:

```
1 # Converting columns to datetime
2 df['release_date'] = pd.to_datetime(df['release_date'], format = '%Y-%m-%d')
```

To begin with visualization, a few whiskers boxes to quickly see what's what is always good. In a glance we can gather a lot of useful information through this process:



We will keep our outliers as despite being really big, it seems after closer inspection that they are indeed valid values.

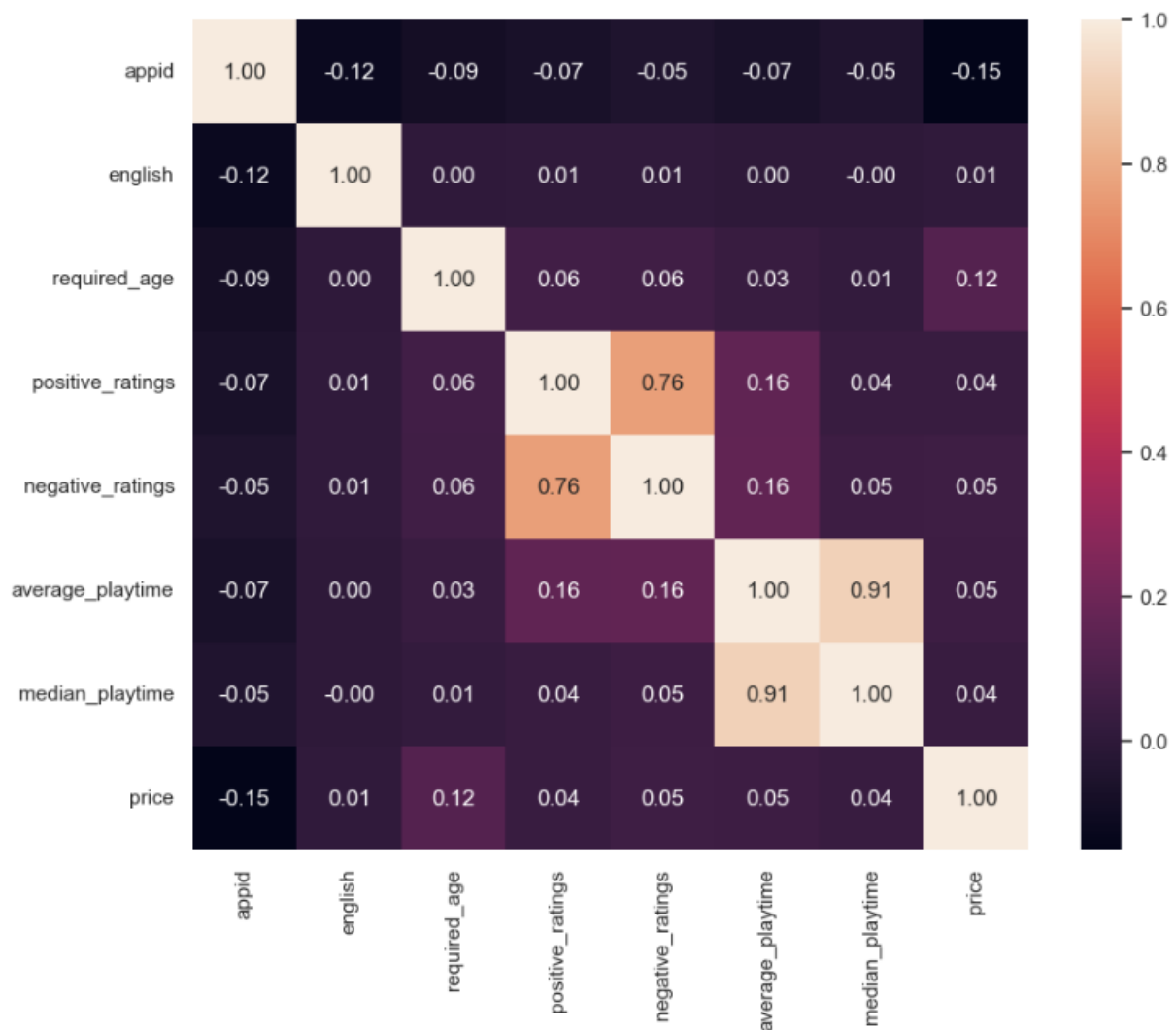
```
1 num_cols = ['positive_ratings', 'negative_ratings', 'average_playtime', 'median_playtime', 'price']
2
3 for col in num_cols:
4     print(f"Top 10 largest values for {col}:")
5     top_10 = df.nlargest(10, col)[['name', col]]
6     print(top_10)
7     print("\n")
8
```

✓ 0.1s

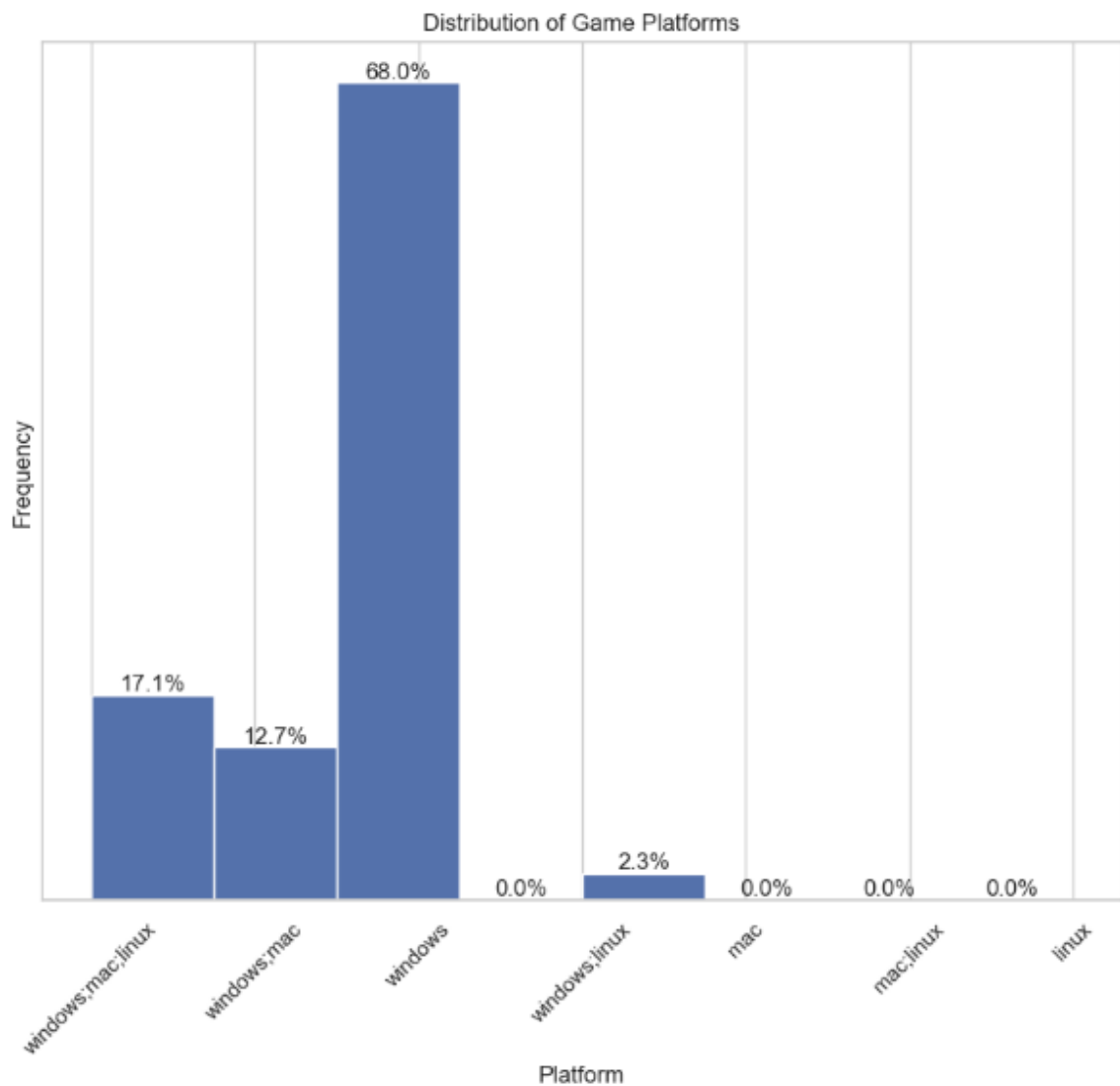
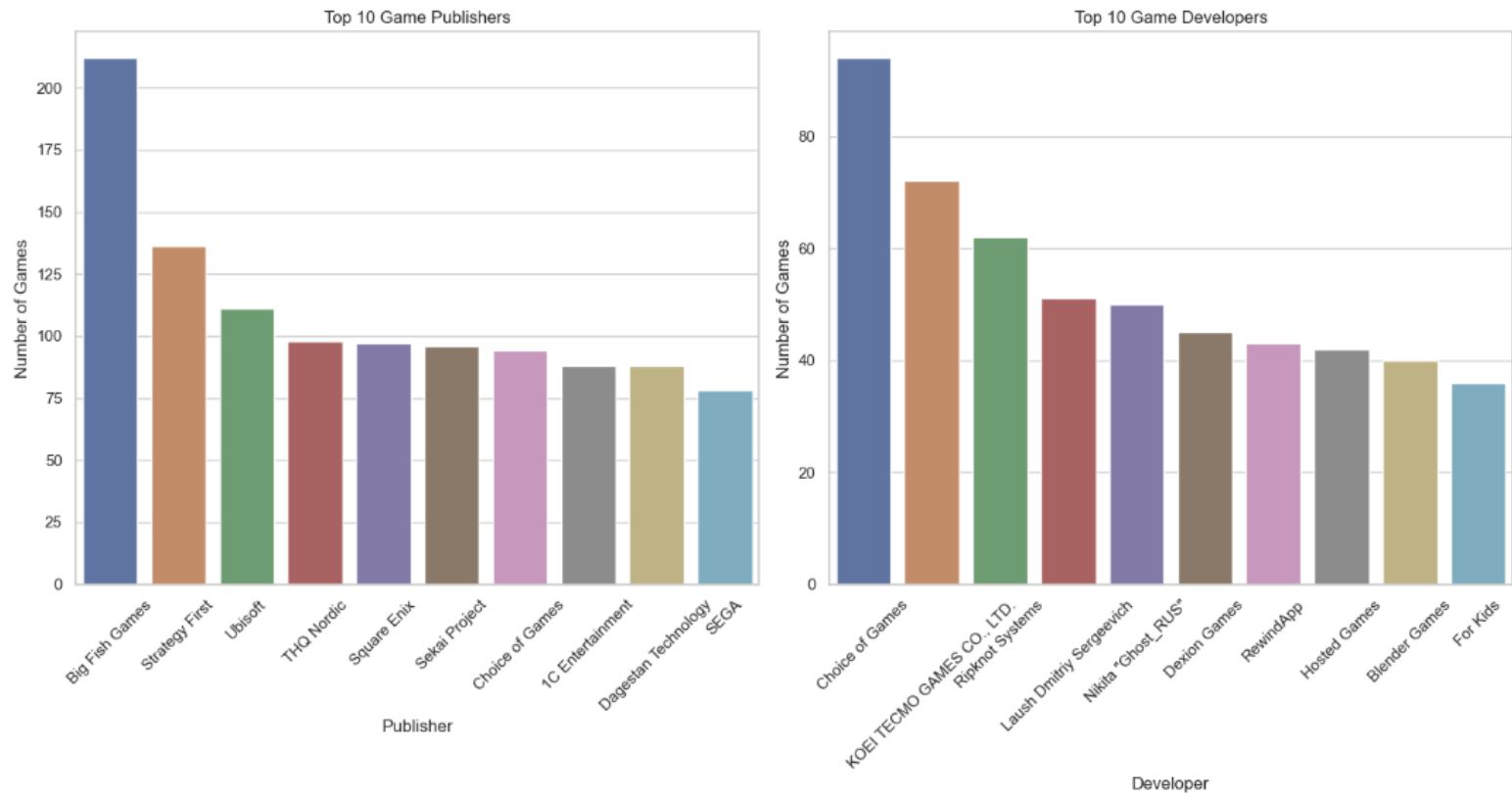
Top 10 largest values for positive\_ratings:

	name	positive_ratings
25	Counter-Strike: Global Offensive	2644404
22	Dota 2	863507
19	Team Fortress 2	515879
12836	PLAYERUNKNOWN'S BATTLEGROUNDS	496184
121	Garry's Mod	363721
2478	Grand Theft Auto V	329061
1467	PAYDAY 2	308657
3362	Unturned	292574
1120	Terraria	255600
21	Left 4 Dead 2	251789

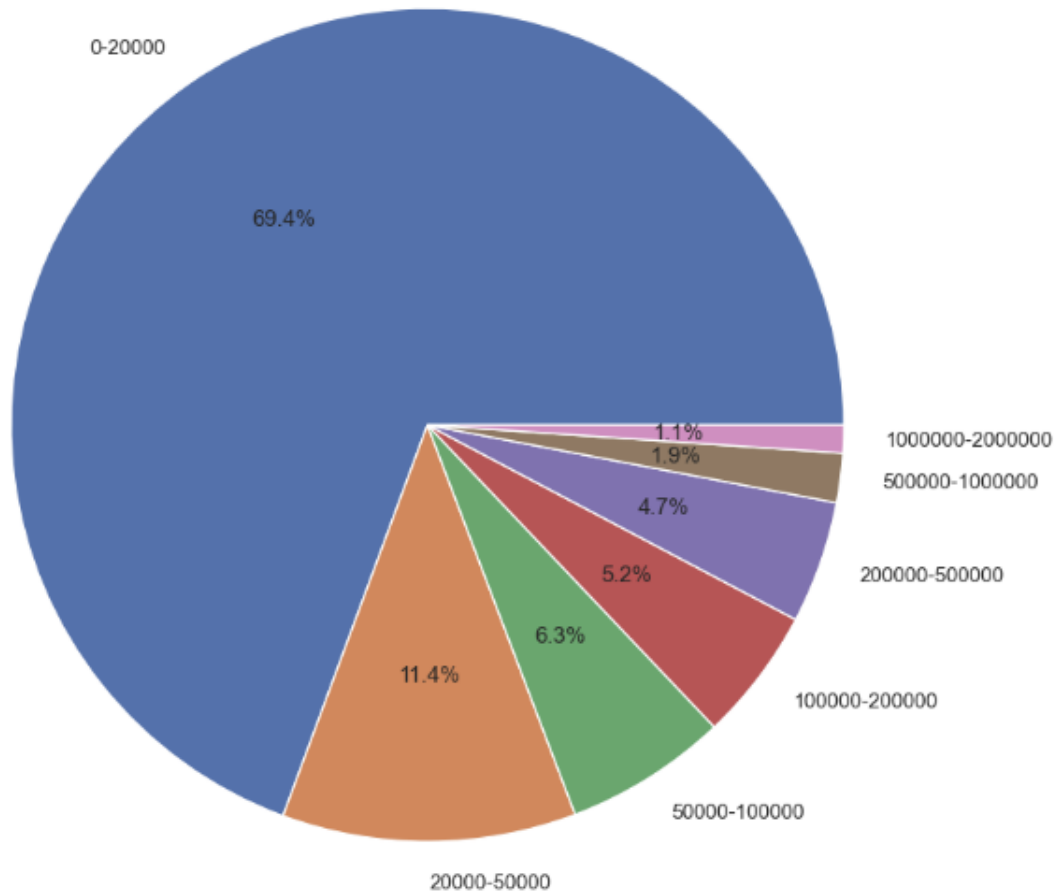
A correlation map is another handy way for us to check our dataset numerical values



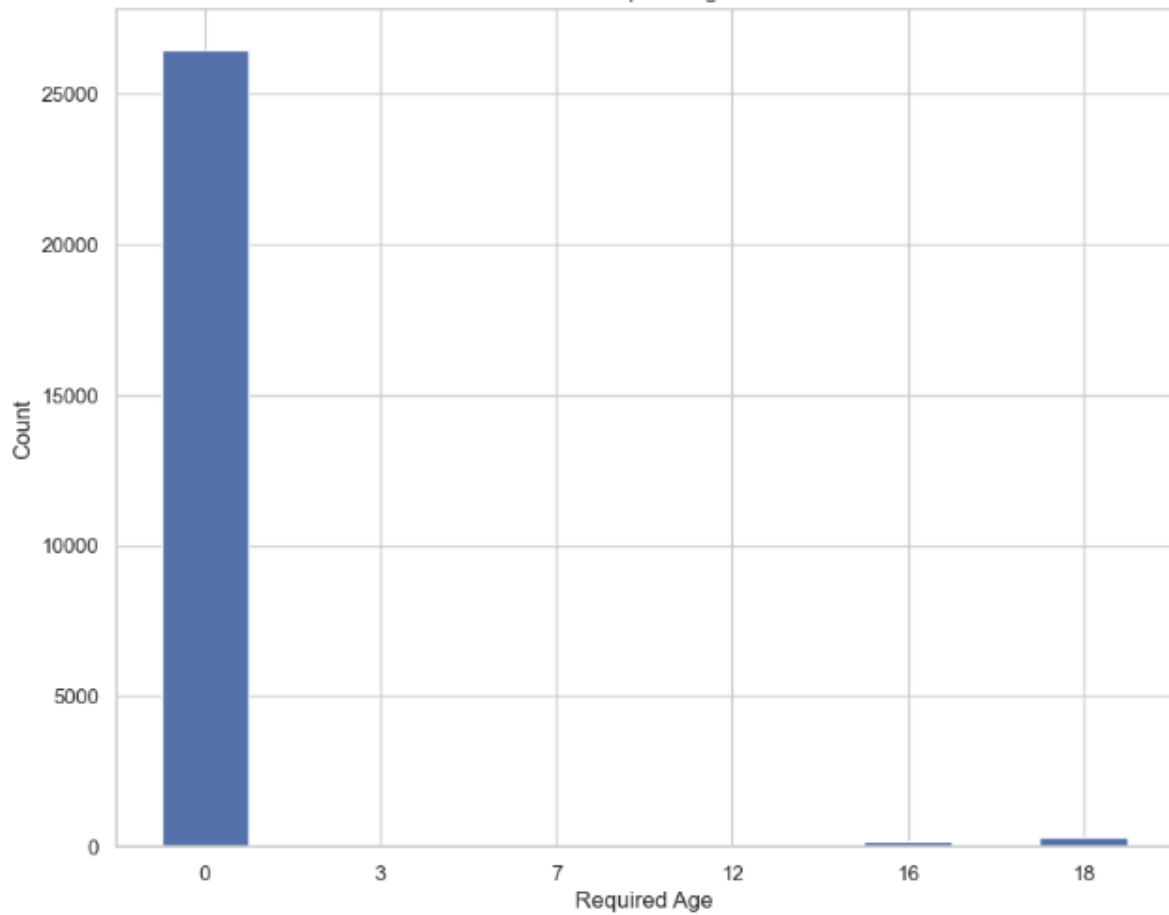
Then we keep on creating a few plots for us to get a better understanding of the values:

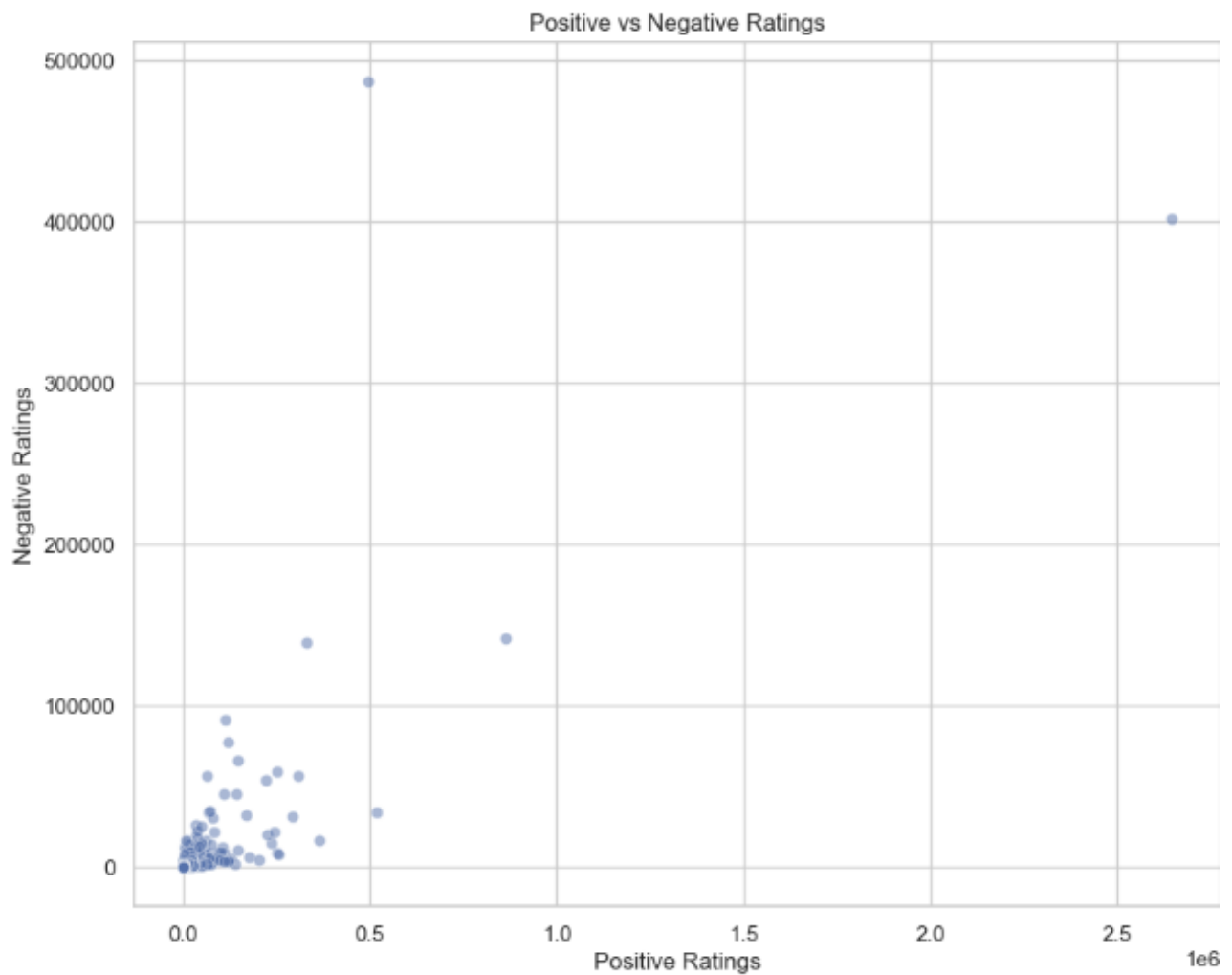


Distribution of Game Owners for Each Game



Distribution of Required Age for Games





We will then tackle an important part of the data tinkering as we have to breakdown our different game genres for the SQL part as follows:

```
1 game_genres_df["genres"] = game_genres_df["genres"].str.split(";")
2 game_genres_df
```

✓ 0.1s

	appid	genres
0	10	[Action]
1	20	[Action]
2	30	[Action]
3	40	[Action]
4	50	[Action]
...	...	...
27070	1065230	[Adventure, Casual, Indie]
27071	1065570	[Action, Adventure, Indie]
27072	1065650	[Action, Casual, Indie]
27073	1066700	[Adventure, Casual, Indie]
27074	1069460	[Adventure, Casual, Indie]

27075 rows × 2 columns

```
1 reindexed_game_genres = game_genres_df.explode("genres")
2 reindexed_game_genres
```

✓ 0.1s

	appid	genres
0	10	Action
1	20	Action
2	30	Action
3	40	Action
4	50	Action
...	...	...
27073	1066700	Casual
27073	1066700	Indie
27074	1069460	Adventure
27074	1069460	Casual
27074	1069460	Indie

76462 rows × 2 columns

```

1 game_genres_list = pd.DataFrame(columns = ['genre_id','genre'])
2 for id, genre in enumerate(set(reindexed_game_genres['genres'])):
3     game_genres_list.loc[id] = {'genre_id':id, 'genre':genre}
4
5 game_genres_list

```

✓ 0.2s

	genre_id	genre
0	0	Nudity
1	1	Violent
2	2	Documentary
3	3	Sports
4	4	Free to Play
5	5	Casual
6	6	Gore
7	7	Action
8	8	RPG
9	9	Sexual Content
10	10	Game Development
11	11	Web Publishing
12	12	Tutorial
13	13	Design & Illustration
14	14	Utilities
15	15	Indie
16	16	Racing
17	17	Photo Editing
18	18	Early Access
19	19	Education
20	20	Video Production
21	21	Massively Multiplayer
22	22	Strategy
23	23	Accounting
24	24	Audio Production
25	25	Software Training
26	26	Animation & Modeling
27	27	Simulation
28	28	Adventure

```

1 reindexed_game_genres_table = reindexed_game_genres.merge(game_genres_list, left_on='genres', right_on='genre')
2
3 reindexed_game_genres_table = reindexed_game_genres_table[['appid','genre_id']]
4 reindexed_game_genres_table

```

✓ 0.1s

	appid	genre_id
0	10	7
1	20	7
2	30	7
3	40	7
4	50	7
...	...	...
76457	849690	23
76458	934710	23
76459	982860	23
76460	982860	2
76461	982860	12

76462 rows x 2 columns

We then just need to create and export our different tables in SQL.

For this we create the corresponding dataframes before the export instruction.

## SQL

### Creating relevant dataframes to export as SQL Tables

```

1 # Table 0: Whole DataFrame
2
3 import getpass
4 from sqlalchemy import create_engine, inspect
5
6 sql_pass = getpass.getpass()
7
8 connection_string = 'mysql+pymysql://root:t3oJbpb38P99T3Jd7cRS@localhost:3306/'
9 engine = create_engine(connection_string)
10
11 # df.to_sql('whole_steam_store', engine, 'steam_store', if_exists='replace', index=False)

```

✓ 3.3s

```

1 # Table 1: Game Info
2 game_info_df = df[['appid', 'name', 'release_date', 'developer', 'publisher']].copy()
3 # # Table 2: Game Genres
4 # game_genres_df = df[['appid', 'genres']].copy()
5 # Table 3: Game Categories
6 game_categories_df = df[['appid', 'categories']].copy()
7 # Table 4: Game Ratings
8 game_ratings_df = df[['appid', 'positive_ratings', 'negative_ratings']].copy()
9 # Table 5: Game Pricing
10 game_pricing_df = df[['appid', 'price']].copy()
11
12 # Table 6: Game Genres ReIndexed
13 game_genres_reindexed_df = reindexed_game_genres_table[['appid', 'genre_id']].copy()
14 # Table 7: Game Genres Listing
15 game_genres_listing_df = game_genres_list[['genre_id', 'genre']].copy()

```

✓ 0.1s



## Exporting relevant dataframes to SQL as Tables

```
1 # Table 1: Game Info
2 game_info_df.to_sql('game_info', engine, 'steam_store', if_exists='replace', index=False)
3 # # Table 2: Game Genres
4 # game_genres_df.to_sql('game_genres', engine, 'steam_store', if_exists='replace', index=False)
5 # Table 3: Game Categories
6 game_categories_df.to_sql('game_categories', engine, 'steam_store', if_exists='replace', index=False)
7 # Table 4: Game Ratings
8 game_ratings_df.to_sql('game_ratings', engine, 'steam_store', if_exists='replace', index=False)
9 # Table 5: Game Pricing
10 game_pricing_df.to_sql('game_pricing', engine, 'steam_store', if_exists='replace', index=False)
11
12 # Table 6: Game Genres ReIndexed
13 game_genres_reindexed_df.to_sql('game_genre', engine, 'steam_store', if_exists='replace', index=False)
14 # Table 7: Game Genres Listing
15 game_genres_list.to_sql('genre', engine, 'steam_store', if_exists='replace', index=False)
16
17 ✓ 6.3s
```

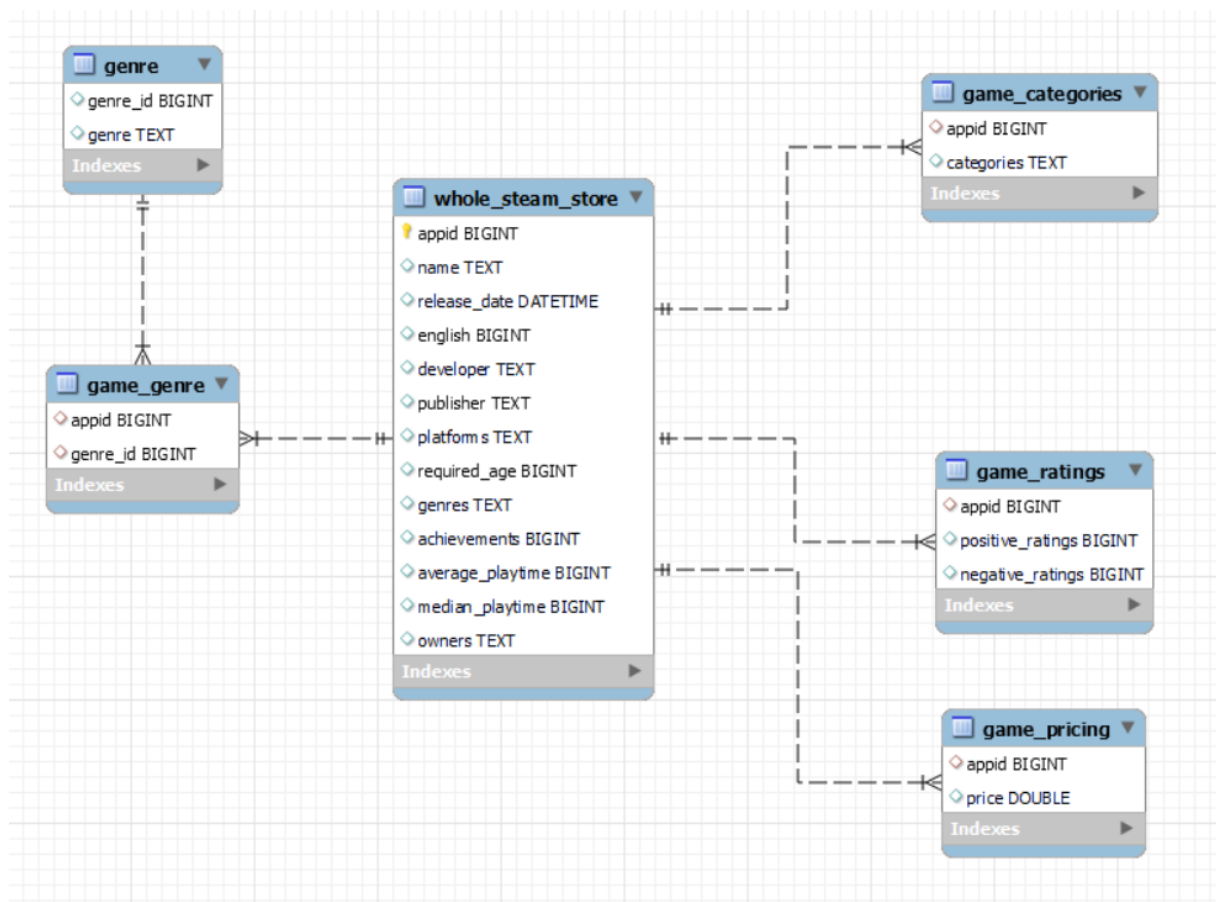
## SQL VS other data base types – the argument

SQL and NoSQL are two different types of database management systems with their own set of characteristics and advantages.

- Relational databases organize data into tables with predefined schemas and use structured query language - SQL - for data manipulation. They provide strong data consistency, integrity, and support for complex queries and transactions. Relational are then suitable for applications that require complex relationships between entities and need ACID - Atomicity, Consistency, Isolation, Durability - properties, such as financial systems, inventory management, and customer relationship management.
- They are characterized by tables with predefined schemas and support for complex relationships, transactions, and queries. Despite my project not being of that size for now, it still seems appropriate to use SQL databases since they are a tool I start to be familiar with.

Another argument I can add is that, as I expect to add more data or other sources later on to take this project to the next stage, SQL will be optimal due to its capacity for vertical scalability and also the ability to use multi-row transactions.

## Entities Relationship Diagram



### *Tables listing:*

- Table: whole\_steam\_store

Columns: «appid», «name», «release\_date», «english», «developer», «publisher», «platforms», «required\_age», «genres», «achievements», «average\_playtime», «median\_playtime», «owners», «price»

Description: This is our main table which contains our primary key: “appid”

- Table: game\_genre

Columns: «appid», «genres»

Description: This table represents the genres associated with each game. It includes the «appid» column as a foreign key referencing the “whole\_steam\_store” table and a column for the genres the game belongs to. Multiple genres can be stored as a comma-separated list or in a separate table if there are many-to-many relationships.

- Table: genre

Columns: «genre\_id», «genre»

Description: This table represents the different game genres we found in our database. It includes the «genre\_id» column as a foreign key referencing the “game\_genre” table and a column for the genres the game belongs to. Multiple genres can be stored as a comma-separated list or in a separate table if there are many-to-many relationships.

- Table: game\_categories

Columns: «appid», «categories»

Description: This table represents the categories associated with each game. It includes the «appid» column as a foreign key referencing the “whole\_steam\_store” table and a column for the genres the game belongs to. Multiple genres can be stored as a comma-separated list or in a separate table if there are many-to-many relationships.

- Table: game\_ratings

Columns: «appid», «positive\_ratings», «negative\_ratings»

Description: This table stores the ratings for each game, including the number of positive ratings and negative ratings it has received. The «appid» column serves as a foreign key referencing the games unique identifier in the “whole\_steam\_store”.

- Table: game\_pricing

Columns: «appid», «price»

Description: This table contains the pricing information for each game, including the «appid» column as a foreign key referencing the “whole\_steam\_store” table and the price of the game.

## MySQL queries

-- QUERY 1 -- Retrieve the top 10 games with the highest positive ratings:

```
SELECT gi.name, gr.positive_ratings
FROM game_info gi
JOIN game_ratings gr ON gi.appid = gr.appid
ORDER BY gr.positive_ratings DESC
LIMIT 10;
```

name	positive_ratings
Counter-Strike: Global Offensive	2644404
Dota 2	863507
Team Fortress 2	515879
PLAYERUNKNOWN'S BATTLEGROUNDS	496184
Garry's Mod	363721
Grand Theft Auto V	329061
PAYDAY 2	308657
Unturned	292574
Terraria	255600
Left 4 Dead 2	251789

-- QUERY 2 -- Retrieve the top 10 games with the highest negative ratings:

```
SELECT gi.name, gr.negative_ratings
FROM game_info gi
JOIN game_ratings gr ON gi.appid = gr.appid
ORDER BY gr.negative_ratings DESC
LIMIT 10;
```

name	negative_ratings
PLAYERUNKNOWN'S BATTLEGROUNDS	487076
Counter-Strike: Global Offensive	402313
Dota 2	142079
Grand Theft Auto V	139308
Z1 Battle Royale	91664
DayZ	77169
ARK: Survival Evolved	66603
Tom Clancy's Rainbow Six® Siege	59620
PAYDAY 2	56523
No Man's Sky	56488

-- QUERY 3 -- Retrieve the top 10 games with the highest positive ratings and display their respective negative ratings:

```
SELECT gi.name, gr.positive_ratings, gr.negative_ratings,
       ROUND((gr.positive_ratings / (gr.positive_ratings + gr.negative_ratings)) * 100, 2) AS positive_ratio
FROM game_info gi
JOIN game_ratings gr ON gi.appid = gr.appid
ORDER BY gr.positive_ratings DESC, gr.negative_ratings DESC
LIMIT 10;
```

name	positive_ratings	negative_ratings	positive_ratio
Counter-Strike: Global Offensive	2644404	402313	86.80
Dota 2	863507	142079	85.87
Team Fortress 2	515879	34036	93.81
PLAYERUNKNOWN'S BATTLEGROUNDS	496184	487076	50.46
Garry's Mod	363721	16433	95.68
Grand Theft Auto V	329061	139308	70.26
PAYDAY 2	308657	56523	84.52
Unturned	292574	31482	90.29
Terraria	255600	7797	97.04
Left 4 Dead 2	251789	8418	96.76

-- QUERY 4 -- Retrieve how many games were released for each year

```
SELECT YEAR(release_date) AS release_year, COUNT(*) AS game_count
FROM game_info
GROUP BY YEAR(release_date)
ORDER BY release_year;
```

release_year	game_count	release_year	game_count
1997	1	2010	238
1998	1	2011	239
1999	2	2012	320
2000	2	2013	418
2001	4	2014	1555
2002	1	2015	2597
2003	3	2016	4361
2004	6	2017	6357
2005	6	2018	8160
2006	48	2019	2213
2007	93		
2008	145		
2009	305		

```
-- QUERY 5 -- Retrieve the names of the most positively rated games and their date of release
SELECT gi.name, DATE_FORMAT(gi.release_date, '%Y-%m-%d') AS release_day, gr.positive_ratings, gr.negative_ratings
FROM game_info gi
JOIN game_ratings gr ON gi.appid = gr.appid
ORDER BY gr.positive_ratings DESC, gi.release_date DESC
LIMIT 10;
```

name	release_day	positive_ratings	negative_ratings
Counter-Strike: Global Offensive	2012-08-21	2644404	402313
Dota 2	2013-07-09	863507	142079
Team Fortress 2	2007-10-10	515879	34036
PLAYERUNKNOWN'S BATTLEGROUNDS	2017-12-21	496184	487076
Garry's Mod	2006-11-29	363721	16433
Grand Theft Auto V	2015-04-13	329061	139308
PAYDAY 2	2013-08-13	308657	56523
Unturned	2017-07-07	292574	31482
Terraria	2011-05-16	255600	7797
Left 4 Dead 2	2009-11-19	251789	8418

```
-- QUERY 6 -- Retrieve the total number of games for each genre and gives us the average corresponding price and rating :
SELECT gg.genre, COUNT(*) AS total_games, ROUND(AVG(gp.price), 2) AS avg_price, ROUND(AVG(gr2.positive_ratings / gr2.negative_ratings), 2) AS avg_rating
FROM game_genre gr
JOIN genre gg ON gr.genre_id = gg.genre_id
JOIN game_pricing gp ON gr.appid = gp.appid
JOIN game_ratings gr2 ON gr2.appid = gp.appid
GROUP BY gg.genre
ORDER BY total_games DESC;
```

genre	total_games	avg_price	avg_rating
Indie	19421	5.05	5.34
Action	11903	6.14	4.85
Casual	10210	4.11	5.15
Adventure	10032	6.12	5.41
Strategy	5247	6.94	4.19
Simulation	5194	7.21	4.03
RPG	4311	6.94	5.25
Early Access	2954	7.04	4.35
Free to Play	1704	0.1	4.66
Sports	1322	7.88	3.74
Racing	1024	7.05	3.46
Violent	843	6.23	3.52
Massively Multiplayer	723	4.32	2.30

genre	total_games	avg_price	avg_rating
Gore	537	5.97	3.30
Nudity	266	7.29	5.58
Sexual Content	245	6.85	6.07
Utilities	146	14.39	4.02
Design & Illustration	87	26.82	4.72
Animation & Modeling	79	27.73	4.66
Education	51	26.79	3.77
Video Production	38	11.32	4.42
Software Training	31	31.8	3.99
Audio Production	29	11.25	5.60
Web Publishing	28	39.7	5.35
Game Development	17	61.68	4.55
Photo Editing	12	14.64	2.46
Accounting	6	2.91	2.52

```
-- QUERY 7 -- Retrieve, for publishers with the highest number of published games, the average price of their games
```

```
SELECT gi.publisher, ROUND(AVG(gp.price), 2) AS avg_price
FROM game_info gi
JOIN game_pricing gp ON gi.appid = gp.appid
JOIN (
    SELECT publisher
    FROM game_info
    GROUP BY publisher
    ORDER BY COUNT(*) DESC
    LIMIT 10
) AS top_publishers ON gi.publisher = top_publishers.publisher
GROUP BY gi.publisher
ORDER BY avg_price DESC;
```

publisher	avg_price
Ubisoft	16.48
SEGA	13.31
THQ Nordic	12.85
Square Enix	12.2
Sekai Project	9.82
Strategy First	7.64
Big Fish Games	6.83
1C Entertainment	6.19
Choice of Games	3.85
Dagestan Technology	2.7

## Machine Learning

Finally, the third notebook addresses the machine learning component of the project. Here, we will engage in model selection, training, evaluation, and prediction. The focus of this notebook will be to train a model that learns the patterns from the cleaned data to eventually recommend games that a user is likely to enjoy based on their gaming history and preferences.

For that we will address the machine learning aspect of the recommendation system.

This process involves training a model on the pre-processed data, enabling it to learn the patterns and characteristics that dictate user preferences.

Once the model will be adequately trained, it will be used to predict potential game recommendations for an existing user based on their games library, playtime history and therefore global preferences.

While this part of the project is not fully fleshed out yet, we can consider, as the data cleaning is now done, that the rest of the general process will follow these steps:

- Feature selection: This step will involve identifying the relevant features that will be used in the machine learning model.
- Normalization and scaling: Once the relevant features have been selected, they can be normalized or scaled to ensure that they are on a similar scale. (This is especially important for models that are sensitive to the scale of the input features, such as linear regression, support vector machines, or k-nearest neighbors).
- Model selection: After preprocessing the data, we will be able to select the appropriate machine learning model.
- Training and evaluation: The selected model will be trained on the preprocessed data, and its performance will be evaluated.
- Final implementation: Once the model's performance is satisfying, it will be implemented for use in making predictions. This could involve deploying the model as part of a larger system or using the model to generate predictions for a new dataset.

I am optimistic that the completion of this stage will result in the elaboration of an efficient recommendation system, capable of enhancing user experience on the Steam platform by offering users a different approach when it comes to them finding their next gem.

## Conclusion

The journey to creating the "Brocommender" algorithm in only a few days' time has already been filled with great learning opportunities, especially in understanding and overcoming new technical concepts on top of the ones I learnt during my formation here at Ironhack.

This is just the beginning.

The magic of coding and computer science residing mostly in automation and constant improvement, version after version.

I really hope and want to finalize the recommender and make it somehow useful, if not only to me, to anybody willing to try it.