# CSC111 Project Proposal: Predictive Steam Game Recommender

Laura Zhan, Olivia Wong, Isabella Enriquez

Monday, April 3, 2023.

## Problem Description and Research Question

The research question we will be studying is: **How can we recommend new Steam video games to gamers based on Steam games they like and Steam community reviews?** In community reviews on Steam, a user can like or dislike a game or provide a descriptive word review (Steam, 2023). So, in this problem, we will be examining the positive and negative reviews from a subset of these Steam reviews to predict what games a person who plays certain Steam games could potentially like or dislike (McAuley). Our project aims at people who play Steam video games (or people who play games on Steam across multiple platforms), which will return results based on said player's favourite games and game genres that they either like or want to try out next! To do this, we will take an input of games liked by the user, and list of genres they like or want to try out, create a graph called a SteamGraph that connects Steam game objects and player objects, and run algorithms on our SteamGraph and the user's input to narrow down the best possible game recommendations. The use of Steam reviews and the user's input will help us filter video games to make predictions in choosing what other games may suit a gamer's enjoyment.

We chose this project question because, based on our personal experience and interest in video games, we find that video game recommendations are not always accurate or suitable based on our interests. Additionally, it is a time-consuming struggle searching for new games to play, as we are unsure whether or not a certain game will be compatible with our interests. So, we want to investigate the different ways we can make more precise and personalised game recommendations to reduce the time spent searching for games. We are interested in the different factors that determine if a game is preferable to a certain player and how we can verify our conclusions.

## Datasets

Dataset 1: australian_user_reviews_v1.json Our first dataset is a collection of Steam users, as well as their corresponding reviews. After opening the json file formatted in a loose json format (using the function steam_game_creater.parse, courtesy of Julian McAuley: the distributor of our datasets 1 and 2), the resulting data is a list of thousands of dictionaries (McAuley). Each dictionary represents a recorded Steam User, and contains the keys 'user_id', 'user_url', and 'reviews'. The keys 'user_id' and 'user_url' are strings of the user's Steam username and the URL to their Steam account webpage, respectively (we do not use 'user_url' in our project). The key 'reviews', which our project is most interested in, is another list of dictionaries, with each dictionary representing a single review for a specific Steam game. There are many keys to these dictionaries, but the only ones we used in our project are 'item_id' and 'recommend', where 'item_id' is an identification number, represented as a string, that corresponds with a Steam game, and where 'recommend' is a boolean value, True meaning a positive review, and False meaning a negative review. In this dataset, there are 25799 player dictionaries, and 59305 reviews. However, 5317 reviews are not tied to a game in our other dataset, so we excluded those reviews from our graph. Below is an example of a dictionary representing a player and their information from the main data list (this player only has graphic):

```
{'reviews': [{'funny': '',
              'helpful': '0 of 1 people (0%) found this review helpful',
              'item_id': '252950',
              'last_edited': '',
              'posted': 'Posted June 16.',
              'recommend': True,
              'review': 'love it'}],
 'user_id': '76561198156664158',
 'user_url': 'http://steamcommunity.com/profiles/76561198156664158'}
```

Dataset 2: steam_games_v2.json Our second dataset is a collection of Steam games and their corresponding information. After opening the json file formatted in a loose json format (using the function steam_game_creater.parse, courtesy of Julian McAuley: the distributor of our datasets 1 and 2), the resulting data is another a list of thousands of dictionaries (McAuley). Each dictionary represents a Steam game, which contains many keys that correspond to many different types of information about the game. In this dataset, there are 31235 Steam games. A quirk of this dataset is that a lot of games have missing keys, and therefore missing information. In order to create our SteamGraph, we had to do some filtering to result in as much of a complete dataset as possible. To do this, we decided that our Game objects, which are a certain type of nodes in our SteamGraph, would need the information from the keys 'app_name' and 'id'. Game objects are only created if the game dictionary has these two keys (There are only 4 games that don't satisfy these conditions, and none of said games have any reviews in our dataset). The key 'app_name' is a string that represents the name of the Steam game application, which is also the title of the game. There is also a key 'title', which also mostly correlates with 'app_name', although this key was missing from more games than 'app_name', so we decided to choose 'app_name' to record the game title (there are only 2 games that have no key 'app_name', compared to 2050 games that have no key 'title'). The key 'id' refers to a unique identifier for a game represented as a string, which is what ties the first dataset to the second! Each player review has a key 'item_id', which creates a link between the game with the same 'id'.

We also wanted to record the genres, or tags, that a game is associated with, so we could create a more extensive recommendation algorithm. On Steam, a game can have both genres and tags (Steam, 2023). Genres are selected by the developer/publisher of a Steam game, while tags are placed by Steam users (Steam, 2023). Generally though, these genres and tags end up being the same for a specific game. Since we had some instances where our game dictionaries had either no 'tags' key or no 'genres' key, or neither, we had to also do some filtering to compensate for this. In the game dictionaries, 'tags' and 'genres' are both represented as a list of strings. We decided to start our Game object 'tags' attribute as an empty list. If the key 'tags' existed in the game's dictionary, Game.tags would be updated to the value corresponding to the key 'tags' (there are 163 instances in our dataset where this is false). If this was false, Game.tags would be updated to the value corresponding to the key 'genres', if it existed (there are 3282 instances where this is false). If both of these keys did not exist, we left Game.tags as an empty list. As mentioned before, genres are not a determinant of the creation of Game objects, they simply just help with specialised algorithms.

The rest of the data goes mostly unused, except for our project output, where some of the returned game recommendation's information is displayed. The remaining keys (if existing for that game) that are used for our output are 'publisher', 'developer', 'release_date', 'specs', 'price', and 'url', which all are corresponding to string values (except for the key 'specs', which corresponds to a list of strings). Here is an example of a game dictionary in its raw, unfiltered form:


```
{'app_name': 'Real Pool 3D - Poolians',
 'developer': 'Poolians.com',
 'early_access': False,
 'genres': ['Casual', 'Free to Play', 'Indie', 'Simulation', 'Sports'],
 'id': '670290',
 'price': 'Free to Play',
 'publisher': 'Poolians.com',
 'release_date': '2017-07-24',
 'reviews_url': 'http://steamcommunity.com/app/670290/reviews/?browsefilter=mostrecent&p=1',
 'sentiment': 'Mostly Positive',
 'specs': ['Single-player', 'Multi-player', 'Online Multi-Player', 'In-App Purchases', 'Stats'],
 'tags': ['Free to Play', 'Simulation', 'Sports', 'Casual', 'Indie', 'Multiplayer'],
 'title': 'Real Pool 3D - Poolians',
 'url': 'http://store.steampowered.com/app/670290/Real_Pool_3D__Poolians/'}
```


## Computational Overview

The kinds of data used to represent our chosen domain are outlined in the previous dataset section in great detail. Using those pieces of data, Game objects and Player objects are created. Game objects represent a unique game in our dataset, and have the instance attributes 'game_name', 'game_id', 'game_tags', and 'reviewed_by'. The attribute

'game_name' is list by the steam_games_v2.json dictionary with key 'app_name', as mentioned in the previous section. This is a string representing the title for the Steam game. The attribute 'game_id' is set by the steam_games_v2.json dictionary with key 'id', also as mentioned in the previous section. This is also a string, representing a unique identifier for the Steam game. The attribute 'game_tags' is a list of strings, where each string is a genre in steam_genres, which is set through some filtering, which is explained in the previous section as well. Finally, the attribute 'reviewed_by' is a mapping of player ids and the corresponding Player object, in which the mapping is of players that have reviewed the Steam game. This 'reviewed_by' attribute creates a link between Game objects (our first type of nodes), and Player objects (our second type of nodes). When Game objects are created, this mapping is empty. When creating a SteamGraph, which is done in steam_game_creator.steamgraph_creator, all Game objects are created first, and are the input to initialise a SteamGraph object.

After the SteamGraph is created, Player objects are created, and their corresponding reviews, which create links between Game and Player nodes. Player objects represent a unique Steam user in our dataset, and have the instance attributes 'player_id' and 'games_reviewed'. The attribute 'player_id' is set by the australian_user_reviews_v1.json dictionary with key 'user_id', as mentioned in the previous section. This is a string representing a unique username/identifier for the Steam user. The other attribute 'games_reviewed', is similar to a Game object's 'reviewed_by', as it is a mapping of game ids and the corresponding Game object and review, in which the mapping is of the games the players have reviewed. The difference in this attribute is that the value stored under the game id key is a tuple, consisting of the Game object and a boolean value, which represents if the review is positive or negative.

The SteamGraph object is a graph that represents our data. Each vertex is either a Player or a Game object, and each Player can only be adjacent to a Game object and vice versa. This means that every path in the graph alternates between Player and Game vertices. Also, each edge represents a review of the game, either with a positive weight or negative weight. Since we use a bool type to represent the review (as a game is either liked or disliked by a player on Steam), positive weight is a True value and negative weight is a False value.

We have three main algorithms that can compute game recommendations, which are selected by our user. The user can either choose to get recommendations based on the three games they input, three genres they input, or based on both. Therefore, the specific algorithm used is dependent on what algorithm the user chooses. This is done in our GUI, which takes our input and sends our output (see the end of this section and the next section for details). For recommendations based on liked games only, the algorithm goes through all the players connected to each liked game and accumulates the scores of the other games the players play. Points are added to the score of a game only if the player likes both the user's liked game and the other game they play. Similarly, for recommendations based on liked genres, the algorithm runs through a list of games and gives one point to its score for each genre it matches with. If the recommendations are based on both liked games and genres, then it first traverses through both algorithms, accumulating the two differing scores: the other games the players play and each genre tag that is matched. Then, the common games that are in both mappings are stored, with a combined score that is the previous two corresponding values added together. In any algorithm, the games with the top three scores are returned. If there's less than three games returned by our algorithms, which may happen if all of the user's liked games have no reviews (and the liked games algorithm is chosen by the user), then the empty spots are filled by randomly selecting games. However, we decided to only randomly select from games with reviews to narrow down our options of random games to games that were played enough to at least have a review. For the both players and genres algorithm, we have two possibilities in filling in the empty spots in the required length three list of games (in the case that the user selected games had no reviews or no genres, or a mix of both). We first examine the two results from our other players and genre algorithms, looking for potential common games that are recommended by both functions; these games and their corresponding scores are accumulated. If there are no games in common, then we randomly choose between the two algorithms to select a game with the highest score to return. If that is not possible, and we do not have enough games to choose from to return three recommended games, then we would randomly generate games from our games with reviews, just like the previous algorithms.

Our project both takes input from the user and reports the results of our computations in an interactive way, through a graphical user interface (GUI) created by Tkinter: an unfamiliar Python library to our course. The Tkinter Python module lets us create a new window containing a wide range of interactive and display widgets created using a grid management system, which overall performs event handling and provides a coherent, interactive measure of interacting with our project. All of the code using the Tkinter module can be found in the tkinter_classes.py file. We created a data class SteamRecRunner to contain all of the code and functions, and the instance attributes are simply variables initialised inside the class that are saved as attributes for easy access.

To start off, a top level widget, Tk(), needs to be created. Widgets are Tkinter objects, which have varying functionalities (Roseman, 2022). We call this Tk() object 'root', as it was called this in many of the TkDocs, and is a "top level widget" (Roseman, 2022). This is what creates our new window, and it can be run by calling root.mainloop(). This opens the window until it is closed by the cross at the top right of the window, like any other computer application. After the root is created, a Frame widget is created, which is a widget that holds all other widgets. Now, all other widgets are created!

Our main objective of using this GUI as our user input was to allow users to input custom inputs from their keyboard, and select an algorithm type, while handling any events (such as invalid inputs, duplicates, missing inputs, etc.). Any text on our input window is displayed using Label widgets, which simply display text. The widget used to allow the user to select an algorithm is called a Radiobutton widget, in which the user is only allowed to select one (out of the three). The selected radiobutton stores a value in a variable using the Tkinter function StringVar(), where the string value associated with the specific selected widget can be retrieved by calling <variable name>.get(). This is useful later when we call our algorithm functions. User input boxes are created by Tkinter Entry widgets, which simply allow users to type into said boxes after clicking on them, where the responses can also be stored in a variable (like the radiobuttons). We wanted to make our GUI a bit more interactive and convenient, so under our genre input boxes, we included an updating Listbox widget, which is a text box that holds multiple separate strings (separated in a list). This listbox updates to show relevant inputs based on what the user has inputted so far into the corresponding entry widget! This is done by binding a Tkinter <KeyRelease> event (whenever the user releases a key from their keyboard) to a function that checks (SteamRecRunner.check()) and refreshes (SteamRecRunner.update()) the listbox. The listbox can also be explored using a computer mouse's scroll, which is useful for users to explore all the genres available to choose (the reason for why we were not able to do this for the game entry widgets is explained in the next section). Finally, after some grid management, and some additional event handling, our input GUI is complete.

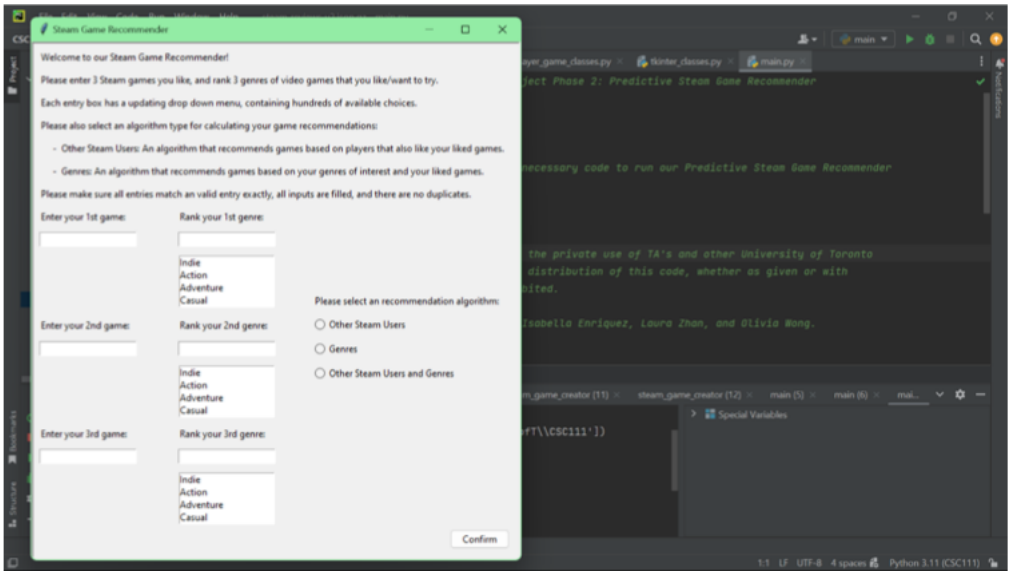
Our output GUI is created similarly. After a successful confirmation of user input is completed, and the corresponding algorithm to our user input is called, the original input window is wiped (the Frame widget is deleted), and a new Frame object is created, alongside new widgets for that new Frame. This window displays a lot of Label widgets, as well as some listbox widgets for game information such as game tags and game specs, as described in the previous section. A new widget is also used in this output frame, which is a Text widget, which allows users to highlight and copy text from inside a box filled with text (This widget also allows for user input, but we have disabled this feature for our purposes). We use this to display a game's 'url', which can be copied and pasted into a web browser to open the recommended game's Steam webpage!

## Instructions for obtaining datasets and running our program

Our requirements.txt file has only PythonTA listed. This is because Tkinter happens to be a built-in library in Python 3.11.

Our three dataset files can be found under the zip file titled "Datasets". Before trying to run our project in main.py, make sure that the Datasets folder (unzipped from the zipped file) is in the same folder as main.py and all of our other modules. It is significant that our datasets are in a folder titled "Datasets", as our function calls use this location to open and format our datasets. (If you want to see our datasets and SteamGraph separately from our GUI, you can open steam_game_creator, and uncomment the lines of code at the bottom of the file in the "if __name__ == '__main__'" block (not including the PythonTA code), run that in the Python console, and access the objects stored in the variables 'data' and 'graph').

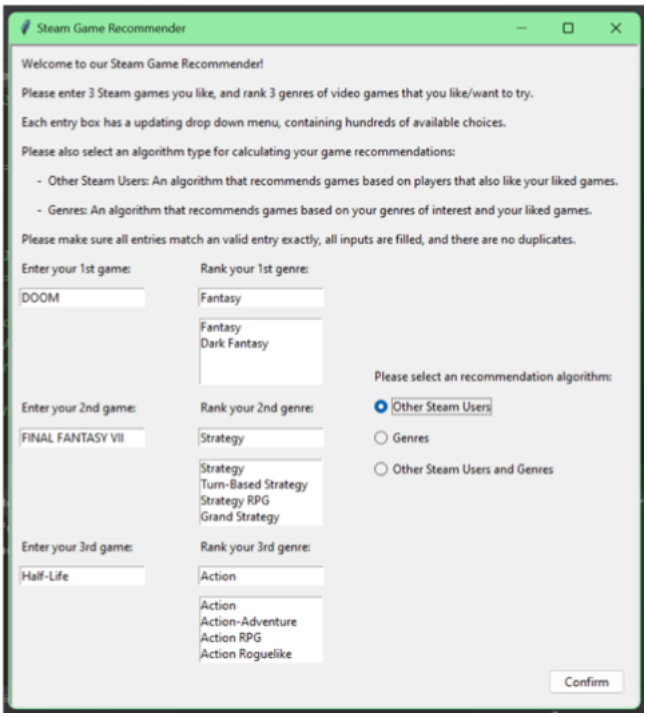
When you run main.py, after a few seconds (our datasets will be formatted and SteamGraph will be created in this time), a Tkinter window titled "Steam Game Recommender" will pop up! It should look something like this: (Ignore the Pycharm window in the background)




This is an interactive GUI, so as a user, you will be able to input Steam game titles, genres, and select an algorithm type. The text at the top of the window is a quick introduction to the interface, as well as provides some explanation to how to use our interface, as well as what our different algorithms do. Note that "Other Steam Users and Genres" is the result of the algorithms "Other Steam Users" and "Genres" combined.

The entry boxes under the labels "Enter your <n> game:" or "Rank your <n> genre:" is where the user can type on their keyboard to input a Steam game title. The listbox under the "Rank your <n> genre:" labels has an adaptive listbox that updates to show relevant inputs as you continue to type! If you have a computer mouse, you can hover over these listboxes and scroll down to see more options. Note however, you cannot click on an item in a listbox and have it fill the entry box automatically, you have to type it in yourself. We weren't able to make this kind of listbox for the "Enter your <n> game:" entry boxes because our datasets have approximately 32000 games, and it affected the running time of the GUI drastically. However, in our submissions, you can find two pdf files titled "SteamGraph_all_game_titles.pdf" and "SteamGraph_reviewed_game_titles.pdf", which contain a list of all the game titles in our steam graph. The first pdf is very lengthy (there are approx. 32000 games after all!), so we created the second pdf, which is a list of all the game titles that have Steam user reviews (which shortens the list of game titles to approx. 3200 games, and will result in more interesting algorithm results, something much more manageable and useful!).

The radio buttons on the side let you select one of our game recommendation algorithms. After you have inputted all of the information, you can click on the "Confirm" button to submit your inputs. If there is anything wrong with your inputs, a corresponding error message will appear at the bottom left of the window, and you can adjust your inputs before pressing "Confirm" again. Here is an example of a valid input, before the Confirm button is pressed:




After pressing Confirm, our algorithms work behind the scenes, and a confirmation message is placed in the bottom left corner of this window (you may not even see this if our algorithms run fast enough!). At the same time, the Confirm button is removed, to prevent another submission while the previous one is running. After our algorithms finish computing, this window is wiped, replaced with new Tkinter widgets displaying the results! Here is an example of an output window, which is the same as the input information from the last screenshot:


This is also a slightly interactive window. As shown, each column of the window represents a recommended game, alongside all of the information present in our datasets about these games. A lot of this information was filtered out in the creation of our SteamGraph Game objects, but this information is retrieved before this window is recreated

to display here! Note, there are many games in our datasets that lack one or multiple of these information types; in those cases, The label objects corresponding to those missing information types are removed. (If you want an example of a recommendation that is missing some of these information types, you can insert "DOOM", "FINAL FANTASY VII", "Half-Life", and "Fantasy", "RPG", "Action", with the algorithm "Other Steam Users and Genres"). In case you're curious, here are the statistics for games with missing information (out of a total of 32131 games):

| Number of games without ___ | # of games |
|---|---|
| Game genres | 139 |
| Publisher | 8052 |
| Developer | 3299 |
| Release Date | 2067 |
| Other Specs | 670 |
| Price | 1377 |

Under the labels "Game genres" and "Other Specs", there are listboxes that contain all of the items in those categories. If you hover over these listboxes, same as the ones on the input window, you can scroll down with a computer mouse to see all of the items. At the very bottom of a game column, all recommended games have a Steam URL, which you can highlight and copy the text to paste it into a web browser, and it will actually bring you to the Steam webpage of that game! After you're done with this window, you can close it with the cross at the top right of the window.

## Changes to Project Plan

In our original project plan, we had two main algorithms for computing the probability of the client user liking a new game. For the first algorithm, we planned on creating a recommendation percentage for the 3 games a user inputs. The algorithm would traverse through the graph to find players who have the most liked games in common and then, we would look through other games that these players like and perform some computations to weigh the recommendation percentage of these games for recommendations. Our second main algorithm was to take in liked genres as well as the user's input of games, and try to find games that best match the tags, alongside the inputted games; this was an extension of the first algorithm, with an added weight of tags used to calculate the recommendation percentages.

However, we found that splitting these two algorithms up into three was a more effective and organised way in computing these scores. We kept our original idea of making the first algorithm compute scores based on other players that have the inputted three games in their likes and traversing those players' games for recommendations. But, for our second algorithm, we focused it on only computing scores based on the three user inputted genres. We added a third algorithm that combines these two and we decided on giving both algorithms equal weights, so that we consider both players and genres scores when we calculate its corresponding combined score. Moreover, for our score count by genre, we made order of inputted genres a factor that determined what a player likes. The first inputted genre would have a higher weight than the second, which has a higher weight than the third. This ensures that our algorithm prioritises games with the genres that matter the most to the user.

## Discussion Section

Our computational exploration resulted in what we hoped for. After running our interactive program, we obtained three appropriate games that matched our required input. Our algorithm generates these recommendations based on what games the user enjoys, which genres they like, or using a combination of both, depending on what the user chooses.

One limitation that we encountered was the size of our datasets, as well as the various levels of completion. In

terms of steam_games_v2.json, a lot of the games had missing information, which created a lot of issues, and had us do a lot of comprehensions to figure out statistics of our datasets (as you have probably noticed throughout our report). The dataset we found is actually quite large, but it does have its limitations. The players and reviews seem to be localised to Australia (based off the dataset file name australian_user_reviews_v1.json), and even if it was not, a subset of all the reviews cannot be as accurate as the entire dataset of all Steam community reviews. Due to our reviews dataset being this size, the majority of the games in our game dataset actually had no reviews in our SteamGraph (only 3195 games out of 32131 games had reviews), which is unfortunate, as our algorithms may have been much more effective and had a much cooler result.

Some limitations we have encountered include the question of how to deal with duplicate game names in the dataset. Since we did not want to complicate user input, we decided on only adding the first instance of a game with the name to our list, which reduced the variety of games. Also, if the inputted games from the user have no reviews, there is no link to other players in our graph, and so we are unable to traverse through the graph to find other games. We decided to deal with this by randomly selecting games with at least insert to fill in any empty spots, but for future exploration, we hope to use another way to generate games to better suit the user's preferences. For example, we could ask the user more questions to narrow down their tastes. Or, we could use their inputted game description and tags to match with other games, even if the inputted game itself is not directly connected to these other games. In a similar manner, we could examine more of the list of recommended games' details, which is especially useful for breaking ties. For instance, we could look at how many have the game installed, how often they play, if they play with friends, and more. Also, we have to keep in mind that not everyone leaves a review of a game they play. Therefore, for our next step, we should consider the number of hours they play and how often they play to account for players who don't leave reviews.

Overall, we are satisfied with how our project turned out. It was a different process from previous assignments because we had more freedom for this one, which meant that we could choose a topic we were interested in. It was also very engaging, as we worked together to figure out how our algorithm should work, and making sure that the methods we write separately will work with each other smoothly. In conclusion, it was a good learning experience and has opened our interests to learning more about how filtering algorithms work.

## References


McAuley, Julian, and Wang-Cheng Kang. "Self-Attentive Sequential Recommendation." ICDM, 2018.

Pathak, Apurva, et al. "Generating and Personalizing Bundle Recommendations on Steam." SIGIR, 2017.

"POPULAR TAGS". Steam, Valve Corporation, 2023. https://store.steampowered.com/tag/browse/#global_492.

McAuley, Julian. "Recommender Systems and Personalization Datasets." UCSD CSE Research. https://cseweb.ucsd.edu// jmcauley/datasets.html#steam_data., https://cseweb.ucsd.edu// jmcauley/datasets.html#steam_data.

McAuley, Julian. "Recommender Systems and Personalization Datasets." UCSD CSE Research. https://cseweb.ucsd.edu// jmcauley/datasets.html#steam_data., https://cseweb.ucsd.edu// jmcauley/datasets.html#steam_data.

Sharma, Dev Prakash. "How to create a combo box with auto-completion in Tkinter?". Tutorialspoint, 2021. https://www.tutorialspoint.com/how-to-create-a-combo-box-with-auto-completion-in-tkinter.

"Steam Reviews". Steam, 2023. https://store.steampowered.com/reviews/.

Roseman, Mark. TkDocs Tutorial, 2022. https://tkdocs.com/tutorial/index.html.

"Tkinter - Python Interface to TCL/TK." *Python Documentation*, Python Software Foundation, https://docs.python.org/3/library/tkinter.html.

Wan, Mengting, and Julian McAuley. "Item Recommendation on Monotonic Behavior Chains." RecSys, 2018.