CSC111 Project Report: Video Game Recommendation System

Yixin Guo, Yifan Li, Richard Soma, Yige Xiong

Friday, April 16, 2021

Problem Description and Research Question

- **Background and Overview:**
  Steam is the first, and one of the world's biggest digital distribution platforms for video games. Launched by the Valve Corporation in 2003, it currently has over 150 million users, gathering a large and diverse gaming community where gamers and developers can buy and sell video games online. In Steam, each user has a Steam Library, a page that stores their collection of games, game statistics, achievements, and friend activities. Steam allows the player to see how many hours they spent playing a game, and this 'Play Time' can be used as an indication of what genres of games the player likes the most, leading to a smarter game recommendation system. In Steam's game library, each game can be classified by their popular tags, genre, languages offered, game mode (i.e. single-player or online multi-player), and a distinct 6 number-digit id. These specific characteristics of Steam games will constitute the core data aspect of this project. (CS Agent)

- **Motivation:**
  During the COVID-19 quarantine, games become highly demanded as many people turned to them to chase away their boredom. One of the most well-known cases is Nintendo's Animal Crossing, a game that caused the Nintendo Switch to be sold out in many different regions (Huddleson). However, the most popular game may not be the right one for everybody, and the difficulty in choosing the 'perfect' game to buy arises, especially for players who have a small budget but want the maximal benefit. This applies to gamers of all types across all levels, including the experienced ones who are growing tired of their current games as well as newcomers who don't know where to start. Although Steam has its own recommendation system, it also has its disadvantages. Steam looks at what games the user has played and what other users have played the same games. Then it recommends additional games that other users have played to the user (Robertson). One imperfection of this algorithm is that labeling a player based on one or two games that they play can be biased. Players with completely different tastes can play the same game! As such, we decided to create an interactive program that helps the user find the games that suit them the most based on their personal preferences and the games that they have played in the past, NOT based on similar users.

- **Project Goal: To recommend new games to people based on their personal preferences and previously played games.**

Datasets

1. steam_games.csv:

   We have found a real-world dataset called 'Steam Games Complete Dataset'. It contains observations for 40833 games on the Steam video game platform. Each observation has 20 variables. The columns that we will use are 'url', 'name', 'all_reviews', 'popular_tags', 'game_details', 'genre', 'game_description', 'mature_content', 'original_price', and 'discount_price'. Here is a sample observation with some of the columns removed:

name	developer	genre	original_price
BATTLETECH	Harebrained Schemes	Action,Adventure,Strategy	$39.99

   This dataset is accessible on Kaggle datasets through this link: https://www.kaggle.com/trolukovich/steam-games-complete-dataset.

2. final_games.csv:

   It has following column headers:

   (a) url: the game's link that directs the user to its Steam page.

   (b) id_num: a unique code (only consisting of numbers) for each game embedded in its url.

   (c) name: the name of the game.

   (d) popular_tags: keywords that describe the game, such as 'Zombies', 'Horror', 'War'.

   (e) game_details: keywords about whether the game supports controller, online multiplayer, split-screen, etc.

   (f) genre: the genre of the game, such as 'Action', 'Simulation', 'Adventure'...

   (g) game_description: a short paragraph that describes the game.

   (h) price: the price of the game.

   (i) popularity_score: a score capturing how popular the game is based on the number of people that rated it and the proportion of positive ratings.

   (j) genre_bools: a list of boolean values that answer 9 genre-related questions; represents a specific path of the decision tree.

   (k) neighbours: all games that have a similarity score larger than 2 with the current game; it's an ordered sequence of game id's separated by semicolons.

   (l) similarity_scores: the similarity scores between the game and its neighbours; it's a sequence of floats (rounded to four decimal places) with the same order as the 'neighbours' column.

   A sample row of this csv file with some columns removed:

id_num	name	price	popularity_score	neighbours	similarity_scores
379720	DOOM	19.99	39146	2320;55230;...	2.1099,2.0693,...

   This is the csv file that our program will use when running the recommendation system.

Computational Overview

- **Data types in weighted_decision.py:**
  - Game: In this project, we defined a custom data type called `Game` to represent each game. It has 12 instance attributes in total, the same as the column headers of 'final_games.csv' described in the previous section (except that it doesn't have 'neighbours' and 'similarity_scores'). We also created a 'recommendation_score' attribute to keep track of how much we recommend each game as the program runs. We stored all the games in a dictionary that maps each game id to their corresponding `Game` object.
  - DecisionTree: To classify the games based on their genres, we created a new data type called `DecisionTree`. Each level of the tree (except the root, which is an empty set) corresponds to a True or False answer to a question related to genre. For example, the first level (right beneath the root) corresponds to the answer to the question 'Is this game an action game?'. If the user chooses 'Don't care' in the questionnaire, the program will randomly choose between True and False and will continue to trace down the path. There are 9 levels in total, corresponding to 9 questions. Each leaf of this tree will be a set of games that have the exact same genres.
  - WeightedGraph: To illustrate the relationship between the games, we created a new data type called `WeightedGraph`. It follows the same structure as the graph ADT we learned in class, with each vertex of the graph representing a game. An edge between two games indicates that they have a similarity score larger than two; this similarity score is the 'weight' of the edge. Some similarities 'weigh' more than others. For example, if two games have similar mature content, we can expect them to have a high similarity score, and their edge will be 'heavier' than others.
  - * Note: The leaves of the tree and the vertices of the graph will only store the id's of the games, not the actual `Game` objects.
- **Computations in data_computations.py:**

- Data wrangling: After obtaining the 'steam_games.csv' dataset, we used the `read_csv` function to filter out the observations with missing data in 'url', 'name', 'all_reviews', 'popular_tags', 'game_details', or 'genre'. For efficiency concerns, the function also directly stores the attributes of the `Game` objects and the edges of the graph (with both the neighbours and the similarity scores) into 'final_games.csv'. In this data wrangling process, we created some new variables (defined in the previous section under 'final_games.csv') and discarded some old variables. For example, we obtained the 'id_num' of each game from the 'url' variable of the original dataset, and we computed the 'popularity_score' of each game using 'all_reviews'.
- Building a tree and a graph: The `load_games` function directly reads 'final_games.csv' and outputs three things: a dictionary mapping game id's to `Game` objects, a decision tree, and a weighted graph. The process of building a decision tree involves the insertion of sequences in `DecisionTree.insert_game`. The process of building a weighted graph involves the computation of similarity scores implemented in `compute_similarity` as we draw edges between vertices (again, the similarity scores were computed inside `read_csv` and stored in 'final_games.csv' to save loading time).
- Algorithms for game recommendation: There are three essential functions that are responsible for computing the results: `tree_computation`, `graph_computation`, and `pop_score_computation`. They all mutate the set of games that we recommend to the user as well as the games themselves by changing the 'recommendation_score' attribute.
  1. The `tree_computation` function uses `DecisionTree.find_games_from_answers` to obtain a set of recommended games based on the user's preferred genres. When there are not enough games that match the user's taste, the function will randomly change the user's answers one at a time to find more games; the more answers the function changes, the lower the 'recommendation_score' of the new games.
  2. The `graph_computation` function finds new games based on the games that the user has already played and the amount of time they have spent playing each game. It locates the given games on the weighted graph and selects a few 'neighbours' of each game using `WeightedGraph.get_neighbours`. The 'recommendation_score' will be updated based on the similarity score and total playtime. The games that the user has already played will not be recommended.
  3. The `pop_score_computation` function first creates a sorted list of the recommended games in ascending order of their 'popularity_score'. It then updates the 'recommendation_score' of those games based on the 'rank' of each game inside the list. The most popular game will receive an increment of 1.0, while the least popular game can receive an increment that's close to 0, depending on how many games there are in the list.

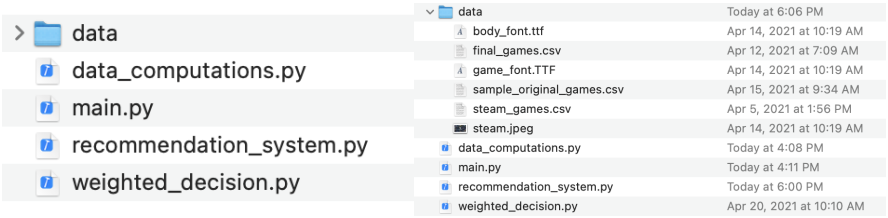
- **Visualization & Pygame in recommendation_system.py**:
  - Overview: Our interactive recommendation system was created using **Pygame**. Some of the most important Pygame features we used in our project include `pygame.Surface`, `pygame.event`, and `pygame.sprite`. We used `pygame.Surface` almost everywhere to represent backgrounds, buttons, tables, as well as text; helper functions like `center_paragraph` and `center_text` were also used to display text nicely on a surface. In order to know the user's preferences, we needed lots of user input; `pygame.event` allowed us to deal with mouse clicks and keyboard entries. Finally, because of the sheer number of buttons that the user can interact with, we used `pygame.sprite` for collision detection and smart update. More specifically, we created an abstract class called `Button`, which inherits from `pygame.sprite.Sprite`. We wrote subclasses like `StartButton` and `BackButton` to represent the various buttons that the user can click on or enter text into. Since our system has multiple 'pages' (or stages), we grouped all buttons on the same page together using `pygame.sprite.Group`, so that only the right buttons will be updated in each iteration of the main loop inside the `main_loop` function. When certain buttons are clicked, the `get_games` method will be called to perform the necessary algorithms as described in the previous section.
  - Stage 0 & Stage 1: The main page (or the menu page) of our recommendation system has a short description of what it does and a `StartButton` that starts the system. The first stage of our system is the 'Q & A' session, which is intended for the user to select their favorite genres. Instead of asking the questions one after another (like in the Python console), we simply layed them out in a table and asked the user to 'tick' their answer to each question by clicking on instances of `SmallButton`. We will obtain a set of games at the end of this stage using `tree_computation`, which is called inside `NextButton.get_games`.
  - Stage 2: The next step is to ask the user for their favorite games and use the weighted graph to find some similar games. We achieved this step by letting the user type in their unique Steam id into instances

of `NumBox` (a sequence of 17 buttons, each storing a digit of the user's id), which was used to create an API URL. The API provided by Steam would return a Steam user object in JSON format. After modifications on the JSON file, we would be able to access various attributes such as owned games of the Steam account and total playtime for each game. We then locate the given games on the weighted graph and do the computation using the `graph_computation` function. Before displaying the final results, we will use `pop_score_computation` to modify 'recommendation_score' one last time. All these will be done inside `OKButton.get_games`.

- Stage 3: The final step is to combine the results of our computations and sort the list of recommended games in terms of the 'recommendation_score' attribute. The top 9 recommended games will be displayed in a table. Each game (row) will have a `ReadButton` which, when clicked, will direct the user to a new page containing the description of the game. On the bottom of the new page there is a hyperlink that directs the user to the game's web page on Steam. The user can also click on the `RestartButton` to return to the main page and restart the whole system.

Instructions for Users

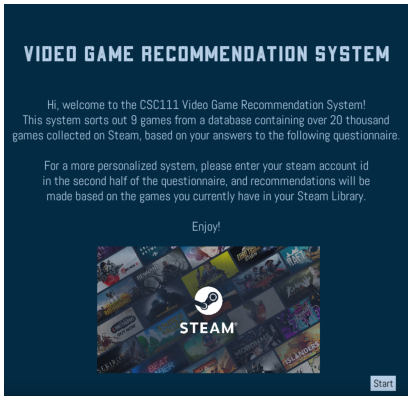
- **Python libraries to install:** See `requirements.txt` file included.
- **Downloading datasets:**
  - Download the folder named 'data'. The folder includes all the csv files as well as some additional `ttf` and `jpeg` files for visualization in Pygame. You may need to unzip some zip files to get the csv files. The 'sample_original_games.csv' file can be used to test the `read_csv` function.
  - Now you should have 4 python files saved in the **same level** with a 'data' folder that contains 6 files, like shown in the images below. **Do NOT modify anything** (i.e., do not create a 'text' folder for the `ttf` files).



(a) What the files should look like (b) 'data' folder expanded

Figure 1

- **Running the program:**
  - Open `main.py` in Pycharm and run it in the python console. A Py-TA check will initiate.
  - Enter `run()` in the python console. An interactive Pygame interface (Figure 2a) will appear.
  - After clicking the 'Start' button in the bottom right corner, a new page will appear where the user can click on the selections that best describe their preferences for video game genres (Figure 2b). Not selecting will have the same effect as selecting 'Don't care'. Click on the 'next' button when you finish.
  - The next page prompts the user to enter their Steam ID (Figure 3a), which is not required but recommended if you do have a Steam account. You can use the 'delete' key if you enter a wrong number. Click on the 'OK' button once you finish. If an invalid or empty Steam ID is provided, the program will continue to recommend games to the user based on Stage 1; a message will appear on the next page in the bottom left corner in yellow, indicating that this Steam ID is invalid (Figure 3b).
  - A results page with top 9 recommended games will appear (Figure 3b). To learn more about each game, click on the 'Read' button to read a description of the game provided by Steam (Figure 3c). The URL of the game will be displayed in the bottom left corner. Click on the link to navigate to the game's official web page on Steam. Click 'Back' to return to the results page.
  - To restart the system, click on the 'Restart' button at the bottom right corner. The main page (Figure 2a) will appear once again. Otherwise, simply close the Pygame interface.



(a) Stage 0: The main page

(b) Stage 1: Choosing genres

Figure 2

(a) Stage 2: Entering Steam ID

(b) Stage 3: Final results

(c) An example of a game description

Figure 3

Discussion

- **Conclusion:** Our program successfully recommended games to people based on their preferences and previously played games. Using a decision tree, we find games based on the user's preferences of genre. Using a weighted graph, we find similar games to the ones that the user has already played. We have also taken into account the user's playtime and how popular the games are. In the end, we displayed the games with the highest 9 recommendation scores to the user, along with their genres, prices, general descriptions, and URL's.

  When we asked one of our teammates to enter his preferred genres and his own Steam ID, five out of the nine recommended games were on his actual Steam wish list. Considering the fact that our teammate is not only interested in those games but has also added them to his wish list, being able to correctly pick five games out of tens of thousands of games on Steam does say something about the accuracy of our program.

  We also feel accomplished regarding the visual aspect of this program, as it includes many interactive features that are easy and intuitive to use. The entire Pygame system follows a cohesive theme with nice colours and fonts that add to the user's overall experience.

  In summary, our program provides an adequate answer to our project question in an interactive and visually-pleasing format. However, there are still limitations as well as room for improvement in terms of algorithms.

- **Limitations:**

  - Dataset deficiencies: There are a few potential issues with our dataset such that we may not be recommending the best games to the user.

    1. Our current dataset is two years old, meaning that all games added to Steam during this time are not included.

2. Almost half of the games from the original dataset lacked some essential information that we needed and were filtered out by the `read_csv` function.

Thus, we may be missing games that could be better recommended to the user but were not included in 'final_games.csv'.

- Under-represented interests: When expanding the decision tree for all of the games, we found that some paths lead to a set containing less than 9 games. Thus, some users' preferences may trace down to a branch in the decision tree with limited options, which prompts the program to randomly change the user's answer to one of the questions until there are at least 9 games in the set. As a result, some of the recommended games may not best reflect all of the user's preferences.

- **Next steps:**
  - Further cleaning of the dataset: Some information in the dataset contains random symbols such as '#@$*'. All of the game descriptions include something like 'About this Game' inserted at the beginning, with issues in formatting. Some games have information written in another language, which goes undetected by our program (for example, we can only process mature content description written in English). Although these small details do not interfere with running the program, cleaning up these areas will help us generate a more user-friendly interface and more accurate recommendations.
  - More preference specifications: In addition to the columns that we used, our dataset includes many variables that can further classify games. These include prices, developers, and more. Although we deemed it unnecessary to include them all in this project, we can of course add new features to our program to provide more personalized recommendations for the user in the future.

References

CS Agent (2019, May 27). *Steam: Everything you need to know about the video game distributor.* Retrieved March 13, 2021, from https://cs-agents.com/blog/steam/

Huddleson, T. (2020, Jun 2nd). *Nintendo Switch: 'Animal Crossing' and coronavirus led to record sales.* CNBC. Retrieved March 13, 2021, from https://www.cnbc.com/2020/06/02/nintendo-switch-animal-crossing-and-coronavirus-led-to-record-sales.html

Robertson, Adi. (2019, July 11) *Steam's New Interactive Recommender Is Built for Finding 'Hidden Gems'.* The Verge. Retrieved April 15, 2021, from https://www.theverge.com/2019/7/11/20690231/valve-steam-labs-interactive-recommender-game-recommendation-machine-learning-tool.

A Steam web API provided by Valve Developer Community is used in order to gain access to a Steam user's game library. The API returns a list of games a player owns along with some playtime information, if the profile is publicly visible. https://developer.valvesoftware.com/wiki/Steam_Web_API