VINCENT BARDUSCO, WILLIAM KHIEU, FLORIAN CELTON, PAULINE ESCAVI

SD201 PROJECT REPORT

INTRODUCTION
STEAM: A VIDEO GAMES PLATFORM

With nearly 100[1] million users, Steam is now a must-see platform when it comes to video games. This platform allows you to purchase video games, update them, purchase additional content, and play multiplayer. This software is downloadable for free on computers as well as on smartphones with a more limited version. An account and internet connection are required to use the software's functionality.

GAME RECOMMENDATION

A Steam account gathers a lot of data, such as a unique ID, a history of purchases and games, etc. So we tried to use some of this information to offer game recommendations to users. These recommendations will be based on games played, time spent on games, similarity of games, basket types etc… The purpose of such a feature would be to:

- enable Steam users to improve their experience of browsing the platform
- introduce users to similar games without them knowing or thinking about them
- limit purchases of games that a user will not play because they do not match their taste

Thus, our project aims to answer the following question: Knowing the games someone bought and how many hours they played them on Steam, what are the other games they would like ?

FOLLOWED STEPS

We have distinguished three major parts in the project:

1. Collect data, clean it up, make preliminary estimates
2. The implementation of the recommendation algorithm
3. Steam data scraping to create our own dataset

For the distribution of work:

- Vincent and Pauline worked on the first part by taking in hand the dataset
- Vincent created a recommendation algorithm based on the k-NN algorithm
- William created a recommendation algorithm based on the a-priori algorithm and frequent itemsets
- Florian worked on the Steam API to scrape and create our own dataset

I - DATASET SELECTION AND PREPARATION

1) THE DATASET

For the dataset, we decided to take the Steam Video Games dataset on kaggle. The dataset has 5 attributes and contains 200,000 lines of data:

[1] https://fr.statista.com/themes/5519/steam/

- ***id***: unique Stem identifier of a user
- ***game***: the name game on which have data
- ***state***: if the user bought or played the game
- ***hours_played***: the number of hours played by the user
- ***0***: a useless column of 0

	151603712	The Elder Scrolls V Skyrim	purchase	1.0	0
0	151603712	The Elder Scrolls V Skyrim	play	273.0	0
1	151603712	Fallout 4	purchase	1.0	0
2	151603712	Fallout 4	play	87.0	0
3	151603712	Spore	purchase	1.0	0
4	151603712	Spore	play	14.9	0

With this dataset we see some issues that we will have to tackle.

First of all, the dataset is a bit strange about the number of hours played. For a user who bought the game, the dataset that corresponds to this, automatically sets the number of hours of play on the line to 1h. That value doesn't really mean much. Some people buy a game but don't play it, so we should have a 0.

We will then choose to use the ***hours*** attribute of the lines where there is ***play*** it is more telling and gives more information.

A second concern is that column ***0*** is useless, and finally, there are no names for the columns. So we will fix all that afterwards.

2) CLEANING THE DATASET

To begin with, we delete column ***0***, we rename the columns, and we check that no 'NaN' values are present.

To get an idea of the scope of the dataset, we calculate the number of different games and users present in it. We get:

- 12394 users with different IDs
- 5155 different games

So we have a wide range of data!

We will also separate the data according to whether the user bought or played the games. We will therefore create two sub-datasets:

- bought_games: ***id***, ***game***;
- `played_games`: id, game, hours_played;

For the `bought_games` dataset, we drop the ***hours_played*** column because it does not fill in as it is a default value that is put in the column.

The data is also standardized so as not to distort the predictions with numbers of hours played too high, and thus prevent outliers from disturbing the predictions. To do so, **we compute the mean and the std for each game**, and we standardize each game time played distribution (otherwise longer games would affect games that are finished faster).

	id	game	hours_played
0	151603712	The Elder Scrolls V Skyrim	0.970886
2	151603712	Fallout 4	0.294820
4	151603712	Spore	-0.193861
6	151603712	Fallout New Vegas	-0.540694
8	151603712	Left 4 Dead 2	-0.223730

Finally, as we want to give recommendations depending on the relations between games, it is important to take care of specific

outliers in the data, in particular players that have played only one game and thus will not be useful for our predictions. We will then **only consider players that have played at least 3 games**. This limit is justified with our tests on the dataset : it is the highest minimum number of games that enables us to keep a significant enough amount of data.

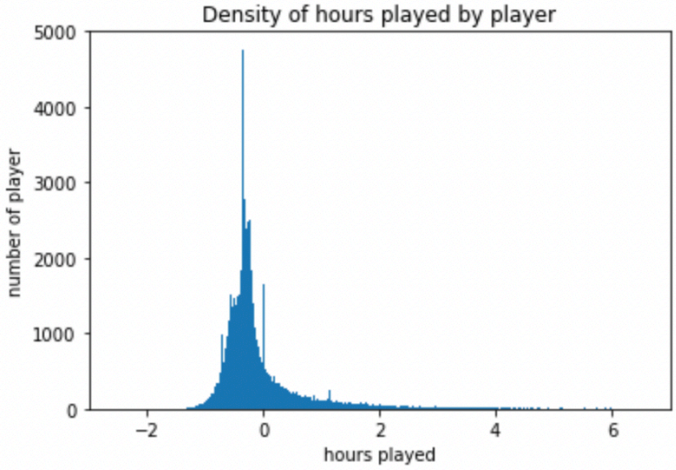
To do so, we apply a `groupby` on ids, and we create a dictionary of games played for each player, and then we only keep the dictionaries containing at least 3 games.

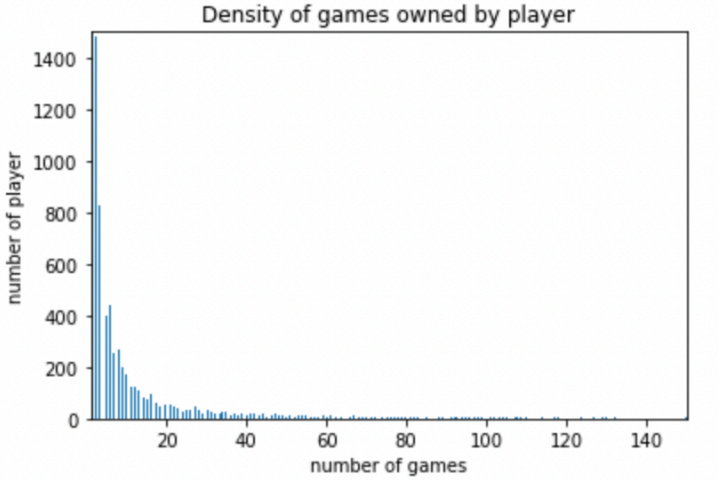
Note on features conversion: as we will use two different approaches, the data handling will be presented inside the further algorithms explanations.

3) FIRST ANALYSIS

In the world of video games, we can distinguish two types of players: those who play little and those who really play a lot. Depending on the profile, we can deduce interesting information such as the number of games that each profile has, etc.

We first wanted to see how the players behave, whether they play a lot or not, we look for the shape of the density of the hours played.





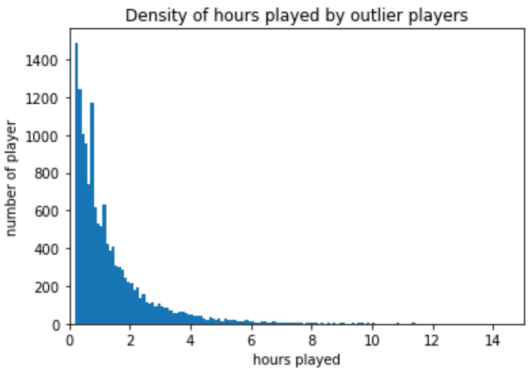
We obtain a density that resembles a density of a Gaussian law centered in about -0.34. And we observe that the majority of players actually play little at their games, at least they play 6 minutes (0.1 hour). The graph shows a few players who play a lot more than the others. We will look at the case of these players afterwards.

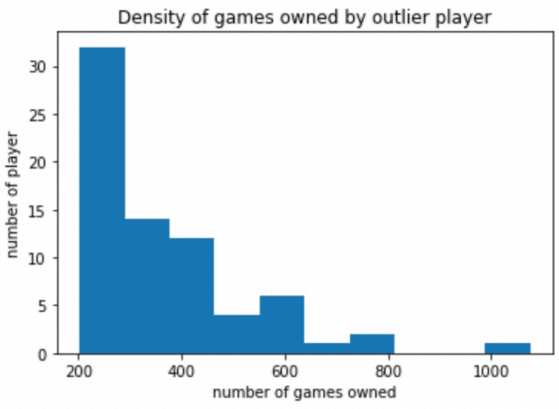
We can also ask ourselves the question: among the players, how many games do they play? What is the distribution of the number of games played?

We observe that the density of the number of games played follows an exponential law. The majority of players have few games, at least 1 only.

So a majority of the players in the dataset play little and have few games.

Let's now focus on players who play a lot. Set a minimum number of hours for this category to 0.2 hours standardized.





Again, exponential distribution seems to be appropriate.

Again, we want to understand the profiles of these extreme players. What is the density of games they have?

Here too we have a distribution that looks like an exponential distribution, modulo the noise at 1000. We deduce from these graphics on extreme players, that there are some players in the dataset who play a lot and have a lot of games.

Knowing more about the dataset and its characteristics, we can know focus more on the recommendation algorithms.

II - RECOMMENDATION ALGORITHMS

1) GENERAL IDEA

In our dataset, we have two different types of data (even though they are linked):

- the games bought
- the time played to each game

With this data, it seemed logical to us to try two different approaches:

- a study of **basket analysis** using apriori algorithm in order to predict what games a user could buy knowing what he previously bought
- a **K-Nearest-Neighbors algorithm** linking a player to several players that like the same type of games, in order to induce what this player may like as well

In both approaches, the schema is the same: we give to the algorithm a steam profile, it computes, and returns a list of **recommendations**.

For these two approaches, we will first present how our algorithms function, and how we progressed until this result.

2) RECOMMENDATION ALGORITHM BASED ON BASKET ANALYSIS

We want to recommend games to a player using a list of games from said player's library. To do this, we will do a basket analysis.

First, we have to build a **list of games for each user id** that we will call that **LIB(id)** (using something similar to a groupby in sql). The way the list is computed depends on

the approach used (games played, games played more than the average,..). We will detail the different approaches below.

Once we have the **LIB(id)**. We can use the Apriori algorithm to compute the **Frequent ItemSets** (**FIS** for short) and then use said FIS to find the **Association Rules** (**AR** for short).

With both the FIS and AR, we can create games recommendations :

We can iterate on the **AR** and for each **rule A→B**, we check if the games in **A** are in the list. If they are, we can add **B** to the list of games recommended.

Once all of this is done, we remove the **LIB(id)** from the list of games to recommend.

3) PROGRESSION OF THE FIS ALGORITHM

Before talking about the different approach we tried, we first need to talk about the different parameters that can influence the recommendation.

The first one is as expected the list of games for a given user id : **LIB(id)**.

If there are a lot of games in **LIB(id),** the list of games that are recommended is also going to be long. If there is only one game in **LIB(id)** then depending on the other parameters, the list of recommendations can be empty. Which is not ideal. The second parameter is the **minimum support** of the **FIS.** In the code, the **support of an itemset** is a number between 0 and 1 that is calculated using the following formula : card(list of games in **LIB** that contains the itemset)/card(LIB).

All itemsets in the **FIS** need to have a support higher than the specified **minimum support** of the **FIS.**

When the support is **closer to 1,** fewer itemsets are proposed by the **Apriori** algorithm. This is to be expected since having a high support means that almost all players have the itemset in their **LIB**. When the support is **closer to 0,** we have more itemsets but the computation time to find the itemsets is higher. For our dataset, having a **lower support is better since it allows us to build more rules** and have more recommendations. **If the number of items (different games here) is high, the support needs to be closer to 0 to be useful.**

Thus a support closer to 0 allows to have more rules to work with thus more games to recommend.

However, having a lot of rules isn't useful if the rules aren't accurate. Which is why we need to filter out the rules with a level of **confidence** that is too low.

'support':

$$\text{support}(A \to C) = \text{support}(A \cup C), \quad \text{range: } [0, 1]$$

'confidence':

$$\text{confidence}(A \to C) = \frac{\text{support}(A \to C)}{\text{support}(A)}, \quad \text{range: } [0, 1]$$

It would be better to have high confidence for our rule but in our dataset it isn't always possible without **filtering too many rules and not having enough games to**

recommend. Indeed, since the number of different items is high in our dataset, the probability that two games appear multiple times in different **LIB(id)** is going to be low. So we need to have low confidence, but if the confidence is too low, the recommendations can be inaccurate. In the contrary, if the support of the antecedent of the rule is too low (which means that this itemset doesn't appear often), having high confidence doesn't mean much since if only two people have this itemset in their **LIB(ib)**, and that they also have the consequent, the confidence is going to be high.

To summarize, large **LIBs** give large recommendations. Low minimum **support** gives more itemsets so more rules so larger recommendations. Low **confidence** filters out less rules so more recommendations.

Now unto the approaches : The first approach that we tried was to compute each **LIB(id)** by grouping the games played by the user of id **id**.

The first criticism that we can make about this approach is that the time the user sinked in each of his games isn't taken into account. For example, the user of id=5250 has in his library : Cities Skylines, Deus Ex Human Revolution, Portal 2, Alien Swarm, Team Fortress 2 and Dota 2. Our algorithm will end up giving recommendations for every single one of these games even if he only really played Cities Skylines and Deus Ex Human Revolution.

Having a lot of recommendations isn't a bad thing but if he didn't like Dota 2 and tried the game for less than an hour then recommending games based on that isn't ideal.

The second and slightly better approach is to compute each LIB(id) by grouping the games played by the user, but only the games where the user played more than the average.

Now the user of id=5250 has in his library : Cities Skylines, Deus Ex Human Revolution.

We can now build recommendations that are slightly more accurate since the rules are only going to be between games that players like to play. For example, people who liked Total War ATTILA are going to have more hours in the game compared to other users. This user will be recommended games that were played by users who like Total War ATTILA. like Total War ROME II or Empire Total War.

However there is a **drawback** to using this approach : Since we have less items in our itemsets, the confidence in our rules is also lower. To compensate, we need a lower **minimum support** and also a **lower confidence**.

If the confidence is too high, some games won't have recommendations : Cities Skylines for example won't have a recommendation unless we lower the minimum

4) RECOMMENDATION BASED ON GAMES LIKED : K-NN

The aim of the algorithm is to predict a list of games someone may like knowing how much he played other games. It means that what we want to predict is **a list of games**, and the features used to do so are, **for each game in the dataset, the amount of hours spent playing this game**.

Neighbors

In this approach, to recommend games for a player, we want to compute k neighbors to this player, who are other players with the same taste as the first one.

To determine the k neighbors, we will use the **euclidean distance** on the vectors of hours played for each game.

With this distance, firstly people that played a lot of the same games will be near each other, but it will also cluster types of players : those who play a lot of different games will be together, and the niche players will be separated.

In our prediction model, we have chosen to compute **20 neighbors** to create the recommendation. (Please check the evaluation part for the justification)

To compute the neighbors, we use sklearn NearestNeighbors model as we are not able to create a faster algorithm to do so. However, the decision process and the recommendations computations have been entirely build by us from scratch, as follows.

The recommendation

Now that the neighbors have been computed, we want to predict what games the initial player may like.

For a prediction, the k-nn algorithm usually chooses the majority class of the neighbors. Thus, we could define a class as the one-hot-encoding of the games played by the user. However, this may lead to 1 class per neighbor, as it is highly possible that every player in the neighborhood, even if they are close to each other, have tried a game the other ones never played.

That's why we choose to give as a result **the games most liked by the neighbors**. This will guarantee that the games resulting will be chosen because they are characteristics of the neighborhood.
More precisely, we define a game as liked when it has been played more than the average time spent on the game by other players. The more it was liked, the more the hours spent were higher than average. With our data standardization (see part I), **the likeness of a game corresponds to the standardized hours_played**.

To compute the most liked games, we will define for each game played by at least one neighbor a **score**, as follows: $S(player, game) = \sum_k \frac{1}{d(player,k)} likeness(k, game)$ with $d(player, k)$the distance between the player and its neighbor computed by the k-nn algorithm, and the likeness as defined earlier. With this score, the more a game has been played by people who like the same games and the more it has been liked by them, the more it will have chances to be predicted. We can then set the maximum number of recommendations we want, and take the games with the highest scores.

The model

If you want to see the entire algorithm compacted in a class, please check the recommendation_algorithm.ipynb notebook provided.

Fitting: (we use a dataframe dictionary like *played_dict_3* shown in data preparation)

- encode the dictionaries in vectors of time standardized time played for each game (replacing NaN values by 0 as a game not in a dictionary has never been played)
- fit a k-nn model on the encoded data
- store a list of games to retrieve the game names after the neighbors computation

- create a dataframe of dictionaries of likeness (as described before), encode it like the times played, and store it to then be able to compute the score function

Predicting n recommendations: (we use a dictionary of hours played indexed by name of games)

- compute the 20 nearest neighbors using the fitted k-nn model
- compute the list of games played by the player
- for each game played by at least one of those neighbors and not by the initial player, compute its score using the likeness stored
- return the list of the n games with the highest score

Example of result

We created a fake player who likes rpgs, and gave him some recommendations:

```
SPM = SteamPredictionModel(20)
SPM.fit(played_dict_3)
```

```
SPM.predict(test_player,20)
```

```
Just Cause 2                    11.437667
Borderlands 2                   11.068734
Far Cry 3                       10.472361
Saints Row The Third            10.142682
Surgeon Simulator                7.690374
Chivalry Medieval Warfare        7.520081
Middle-earth Shadow of Mordor    7.351925
Half-Life 2 Lost Coast           6.965404
Deus Ex Human Revolution         6.564230
Need for Speed Hot Pursuit       6.365282
Grand Theft Auto San Andreas     6.320411
Far Cry 3 Blood Dragon           6.305465
Saints Row IV                    6.213651
The Elder Scrolls IV Oblivion    5.844674
Arma 2                           5.804962
Trine                            5.678657
Worms Reloaded                   5.642036
Wolfenstein The New Order        5.632307
Assassin's Creed II              5.288043
Saints Row 2                     5.279066
dtype: float64
```

```
#Let's create a test rpg player to test the recommendations
test_player = {'The Witcher 3 Wild Hunt':132,'The Elder Scrolls V Skyrim':224, 'Far Cry':90}
encoded_player = pd.Series(test_player,index=games_list).fillna(0)
encoded_player
```

```
007 Legends                                              0.0
0RBITALIS                                                0.0
1... 2... 3... KICK IT! (Drop That Beat Like an Ugly Baby)    0.0
10 Second Ninja                                          0.0
10,000,000                                               0.0
                                                        ...
rymdkapsel                                               0.0
sZone-Online                                             0.0
the static speaks my name                                0.0
theHunter                                                0.0
theHunter Primal                                         0.0
Length: 3564, dtype: float64
```

Model performance

Once the data has been cleaned, the fitting of the model on the data takes a few seconds while the prediction is almost instantaneous.

```
SPM = SteamPredictionModel(20)
%timeit SPM.fit(played_dict_3)
```

```
3.08 s ± 138 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
53 ms ± 4.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The fitting time obviously depends on the size of the dataset, but the predictions are really fast, and thus enables us to scale and recommend a lot of people at the same time without any issue.

5) EVALUATION OF THE RECOMMENDATION ALGORITHM

To properly measure how our model performs, we need to use evaluation metrics. To do so, we began by splitting our dataset in a training set and a testing set, shuffling the data.

The evaluation metrics we have seen in class are fitted to classifiers evaluation. To use them, we will evaluate our algorithm using the testing set like this:

1. remove 5% of the games from the player's played games (among the games liked)
2. use the algorithm to predict the games lacking
3. compare the recommendation to the games that have been removed in step 1

The result of the evaluation is:

- true positive if the game predicted was in the games removed (or if the game removed is in the list of game predicted for fis)
- false positive if the game predicted was not in the games removed(or if the game removed isn't in the list of game predicted for fis)

The number of false negatives (not to predict a game that has been removed) is then exactly equal to the number of false positives, and the number of true negatives is irrelevant here.

That's why we will use the **precision** metrics to evaluate our model, where only true positives and false positives are involved.

FIS algorithm

We will be evaluating our algorithm on the 2nd approach as there's not much time left and it can takes 20min to run everything depending on the parameters.

As we have 2 mains parameters for our algorithm, plotting the precision in function of those parameters is a bit harder. As such we will mostly be using the parameters min_support = 0.0005 and min_threshold = 0.2 (min confidence).

Also, as some players only played one game more than the average, those players will be removed from the testing set.

For those parameters we obtain the following results on our test set composed of 20% of the users (more is possible but too long to compute).

true positives	false positives	precision
88	359	0.197

Which is actually pretty good considering that we sometimes only get one game to use as a basis for our recommendation.

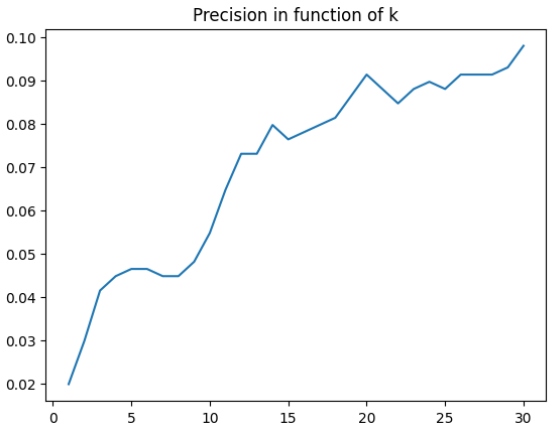
Since a lower confidence mean that we get more recommendations, we can "trick" this measure by having a near 0 minimum confidence (here min_threshold=0.1) :

true positives	false positives	precision
145	302	0.324

true positives	false positives	precision
168	279	0.376

here min_threshold=0.05

The caveat is that there will be so many games recommended that most of them are going to be useless or plain wrong.



K-nn algorithm

Let's try different values of the number of neighbors. We plotted the precision evolution by increasing the

value of k the number of neighbors.

With this result, it seemed fair to choose 20 as the 'ideal' number of neighbors to compute, because the curve tends to stabilize after 20. In fact, with our score function, as we use the inverse of the distance, we could calculate the score with all the players in the dataset without using a knn, but this curve shows that 20 neighbors gives a good result that will be hard to beat with more neighbors, and using only 20 players will ensure us a fast running time and a good scalability.

true positives	false positives	precision
55	547	0.091

In particular, for k=20, we obtain the following results on our test set composed of 20% of the dataset (among players that have at least 3 games).

This precision seems extremely low. Indeed, if we compare it to baselines, a good model should have a precision of at least 75%.

However, here we are trying to predict the exact game we removed from the test player with a recommendation algorithm. To begin with, we have in the dataset a lot of players with only 3 games, meaning that we are in fact trying to predict 33% of the entire data which is extremely hard. Moreover, the aim of the algorithm is trying to induce the type of player it is analyzing, and thus giving him games that correspond to its type. As a lot of similar games exist (we can think of game series for instance, or game genres), it seems really difficult to predict the exact game removed, and thus almost 10% of precision is in fact a good result.

To prove this phenomenon, we will propose a new evaluation method: we will take only players that have played at least 5 games, and for each test player we will remove 1 game, and recommend 10 games, and consider a true positive result if the removed game is in the recommendations. It shows that our model is better performing than what the previous metrics implied, and that it indeed performs well on players that have played a lot of games, which is logical because it gives the model more information on the type of player they are.

III - SCRAPING AND CREATION OF OUR OWN DATASET

In order to test our algorithms and to compare performances between kaggle's dataset and direct steam data, we need to create our own dataset. We scrapped thanks to Steam's API data about users including: their id, games bought and hours played for each of their games.

We used the request python library to exchange with the Steam API. The main problem we faced was that Steam requires us to know a specific steamId to request for game information. However Steam doesn't allow large scraping of steamIds, so we had to find a workaround. Thankfully it happens that requesting for the friendlist of a specific steamId provides also the steamIds of each friend in the list. Hence we started from one of our own steamIds and scraped the steamIds of our friendlists, then of our friends' friendlist… Eventually with enough steamIds, we collected all game data for each steamId.

IV - CONCLUSION

We have seen that our algorithms indeed can recommend games to people depending on what games they played previously and how much they liked them. It was a challenge to do so because we did not have ratings or strict metrics gave by players to define how much they liked a game, and thus we had to deduce it from the dataset. Even if it was difficult to determine a fair way of evaluating our models, we are still satisfied with the results, especially when testing the predictions on our own Steam accounts.

Limitations

As we have seen before, our models are highly dependent on the data given both for fitting and for predicting. Indeed, players that only played a few games cannot be used as relevant neighbors for other players, and it is also almost impossible to make a relevant recommendation for them as there is too little information on their taste.

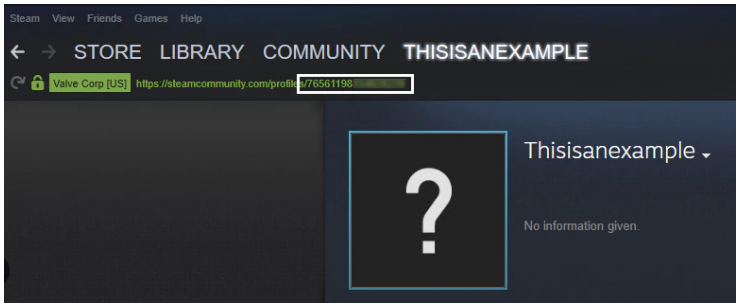
Another limit of the approach is the recommendations for players that like a lot of different types of games. Indeed, the model will then compute really different neighbors, and the recommended games will then probably all have a low and almost equal score, which will come down to a poor quality prediction.

Perspectives

The first improvement to our model that we want to implement is to use the dataset we have built by scraping. Indeed, we need to wait for the scraping which is limited by Steam's API requests per day to gather enough data, but we firmly believe that the results will be even better with this new dataset, as we will have more data and more recent information as well.

V - TEST OUR ALGORITHM

Now that all has been explained, you can test our algorithms with your own Steam profile in order to get recommendations ! To do so, you first need to retrieve your steamID. You can find it directly on Steam application, or on their website going into your account details.



Then, you need to make your games info public so that we can request the API to retrieve your info. To do so, you need to go to your confidentiality settings (https://steamcommunity.com/my/edit/settings?snr=) and set your games info to public.Finally, you need to download the dataset here : https://www.kaggle.com/datasets/tamber/steam-video-games, and put it in the folder with the notebooks.

Once this has been done, you can go to the *recommendation_algorithm.ipynb* notebook, check if everything is well set up by verifying your games using *getGames(steamId)* (where *steamId* is your steamId as a string), and then get your recommendation using the *recommendation* function. Have fun!