Project Report

Steam Games Recommendation

Vincent Bardusco, Florian Celton, Pauline Escavi, William Khieu

SD201

Table des matières

1 Introduction

1.1 Steam : a video game platform

With nearly 100 million users[1], 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.

1.2 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 ?**

1.3 Followed steps

We have distinguished three major parts in the project :

— Collect data, clean it up, make preliminary estimates
— Creating our own recommendation algorithms
— Trying to improve them by scraping a new dataset and using other techniques

1.4 Distribution of work

— Vincent : data cleaning, knn algorithm from scratch with custom recommendation method, evaluation methods, model cleaning, custom dataset implementation, svd researchs, intermediate report, final report
— Florian : dataset scraping
— Pauline : data cleaning, intermediate report, final report
— William : FIS algorithm

1.5 Test our algorithm

After reading the report, please do not hesitate to test our algorithm with your own Steam profile in the *kaggle_dataset/recommendation_algorithm.ipynb* notebook !

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

2 Dataset selection and preparation

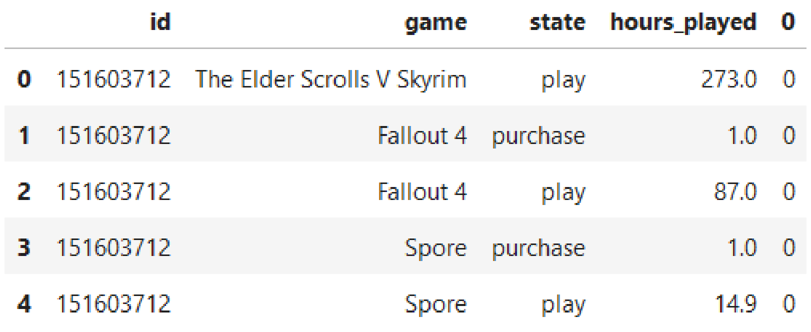
Please check the kaggle_dataset/data_preparation.ipynb notebook if you want to follow the code along the report.

1.1 Dataset presentation

For the dataset, we decided to take the Steam Video Games dataset on kaggle. The dataset contains 200,000 lines of data and has 5 attributes that appears like this after a formal renaming :

— **id** : a unique Stem identifier of the user
— **game** : the name of the game
— **state** : tells if the game was only bought or if it has been played
— **hours_played** : the number of hours the user played the game
— **0** : a useless column of zeros

The challenge given by this dataset is that it does not contain any information on how people like the games, we don't have reviews or rating, and the difficulty in our project will thus be to recommend people only knowing which games they bought or how much they played to certain games.



	id	game	state	hours_played	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

FIGURE 1 – *Kaggle's dataset*

1.2 Data cleaning

To begin with, we delete the entire column of 0, we check that no 'NaN' values are present which is the case, and we reindex the dataset using players ids.
Once this is done, we can analyze the dataset using `groupby` to discover that we have 12394 unique users and 5155 different games, which seems enough to make good predictions.

In our project, we have decided to totally separate the information to set up 2 strategies : one which will be only based on the games bought, and the other one on the time spent playing. It means that we will deliberately loose a part of the information in both strategies, but it will enable us to create models that will perfectly fit our data. That is why we created 2 subsets :

— **bought_games** : id, game
— played_games : id, games, hours_played

In the *played_games* dataset that contains numerical values, we noticed that there is a strong imbalance in the different games popularity, because some games (Dota, CS, TF2, ...) are played by a huge part of the database while some are only niche games. However, we considered that with the method we will use (k-nn based), this imbalance was not a problem, because niche player would gather, while more 'common player' would not be affected.
However, as the average played time for each game are really different, we decided to **standardize** the data to both equilibrate the weights of the different games in the calculations and prevent outliers from disturbing the predictions. We took into account the important fact that the whole dataset should never been standardized (except for visualization purpose) otherwise after the train/validation/test split, the validation and test sets would have been contaminated by the training set. That's why we will only standardize the training set.

Finally, as we want to make predictions depending on the relations between games, we had to take care about specific outliers in the data, in particular players that played only 1 game and thus would not be useful to make predictions. We then had to make a balance between a large number of data and players with a lot of games. We used a boxplot to visualize the distribution of these players :

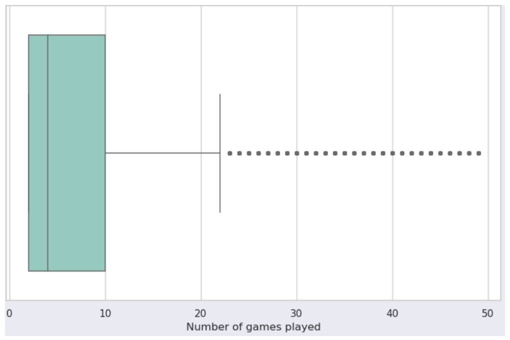


Figure 2 – *Boxplot of number of played games density (¿1, ¡50)*

As we wanted to keep at least $\frac{2}{3}$ of our data, we kept players that played at least 3 distinct games.

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

1.3 Data visualization

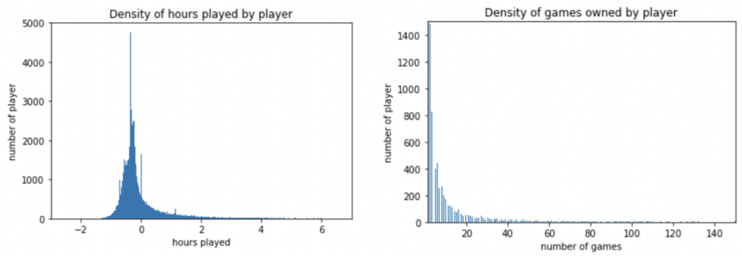


FIGURE 3 – *First analysis*

We observe a density of hours played that is Gaussian, centered around -0.34, which shows that most players play around average (the data is standardized). For the games bought, the density is exponential : most players only buy a few games.

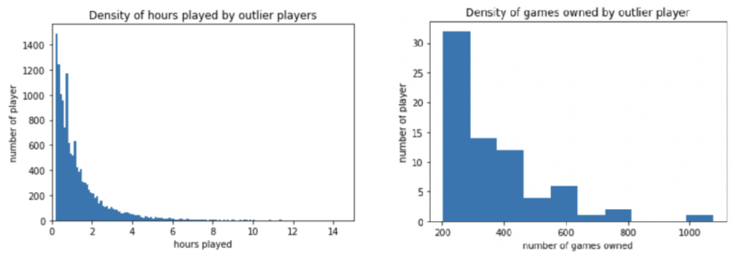


FIGURE 4 – *Outliers analysis*

We observe that the distributions seem similar for outliers : they are also exponential (non stardized hours there).

2 Recommendation algorithms

2.1 General idea

In our dataset, there are two different types of data (even though they are linked) : the games bought and 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 and 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.

2.2 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 $A \to B$, 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.

2.3 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 :

$$\frac{\text{card(list of games in } \textbf{LIB} \text{ that contains the itemset)}}{\text{card(LIB)}}$$

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. 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.

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

Source : http://rasbt.github.io/mlxtend/user_guide/frequent_patterns/association_rules/

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.

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. 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 confidence to 0.1 but in the first approach, a minimum confidence of 0.49 give a fairly good recommendation for the game : Civ V

2.4 Recommendation based on games liked : K-Nearest-Neighbors

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 **the amount of hours spent playing each game of the dataset**.

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.
As we do not have the information on how someone like a game in our data, we had to deduce it by ourselves. To do so, we decided to assert that a game was liked if it was **played more than average**. Obviously, this is not true for people that just bought a game and did not have yet the time to play it, but as we do not have the information on the bought date, we could not take this into account, and admitted that it represented a negligeable part of our data. To be more precise, we will define the likeness as the difference between the time spent of the game with the average (standardized for each game). 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 neighbors computed by the k-nn algorithm. 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.

Progression that led to this algorithm

Please check the kaggle_dataset/knn_progress.ipynb notebook if you want to want to check the detailed process along with examples.

The first approach that we used was based on the entire dataset, without limiting the minimum number of games played, and it resulted in outlier cases where popular games were the only games played by a lot of players, and all the neighbors had the exact same time spent on those games. We then decided to limit the minimum number of games played.
Then, the approach was to give as a recommendation the union of games played by all the neighbors, but this gave too much recommendations and more importantly, it also gave games that neighbors tried but did not like. That is when we implemented the concept of game liked, to limit the recommendation to game liked.
Finally, to be able to recommend only a certain number of games, and to give a probability of likeliness, we implemented the system of score as presented before.

The model

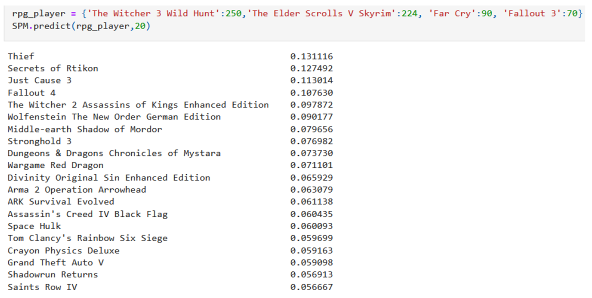
If you want to see the entire algorithm compacted in a class, please check the recommendation_algorithm.ipynb notebook provided.

Fitting : (on a dataframe of dictionnaries containing hours played)
1. store the average played time for each game
2. encode the dictionaries in vectors with a component for each game in the dataset (size of 5155, replacing NaN by 0), and standardize each column of the encoded dataframe
3. fit a k-nn model on the encoded data
4. compute the likeness for each player in the fitting data (using average time played) and standardize it for each game

Predicting n recommendations : (for a dictionnary of hours played)
1. get the list of the games already played
2. encode the vector (as before) and compute the k neighbors using the fitted model
3. for each game liked by at least one of those neighbors and not by the initial player, compute its score using the likeness stored
4. return the list of the n highest score items

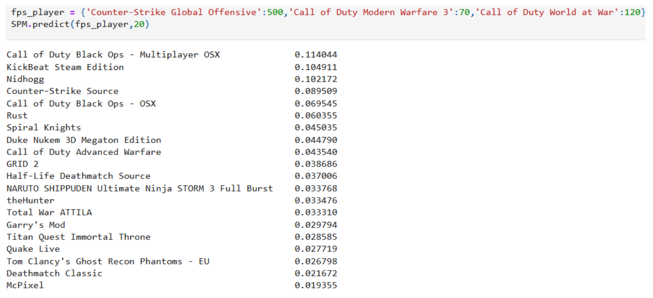
Examples of results

We created dummy stereotyped players to test our algorithm before evaluation and the results were really impressive :



```
rpg_player = {'The Witcher 3 Wild Hunt':250,'The Elder Scrolls V Skyrim':224, 'Far Cry':90, 'Fallout 3':70}
SPM.predict(rpg_player,20)
```

```
Thief                                             0.131116
Secrets of Rtikon                                 0.127492
Just Cause 3                                      0.113014
Fallout 4                                         0.107630
The Witcher 2 Assassins of Kings Enhanced Edition 0.097872
Wolfenstein The New Order German Edition          0.090177
Middle-earth Shadow of Mordor                     0.079656
Stronghold 3                                      0.076982
Dungeons & Dragons Chronicles of Mystara          0.073730
Wargame Red Dragon                                0.071101
Divinity Original Sin Enhanced Edition            0.065929
Arma 2 Operation Arrowhead                        0.063079
ARK Survival Evolved                              0.061138
Assassin's Creed IV Black Flag                    0.060435
Space Hulk                                        0.060093
Tom Clancy's Rainbow Six Siege                    0.059699
Crayon Physics Deluxe                             0.059163
Grand Theft Auto V                                0.059098
Shadowrun Returns                                 0.056913
Saints Row IV                                     0.056667
```

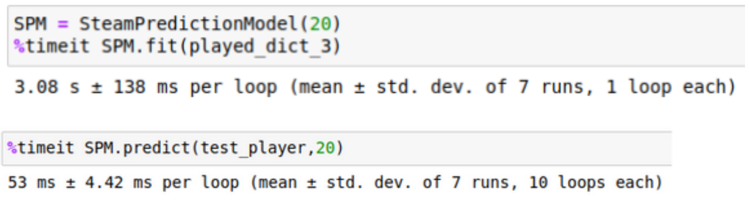


```
fps_player = {'Counter-Strike Global Offensive':500,'Call of Duty Modern Warfare 3':70,'Call of Duty World at War':120}
SPM.predict(fps_player,20)
```

```
Call of Duty Black Ops - Multiplayer OSX           0.114044
KickBeat Steam Edition                             0.104911
Nidhogg                                            0.102172
Counter-Strike Source                              0.089509
Call of Duty Black Ops - OSX                       0.069545
Rust                                               0.060355
Spiral Knights                                     0.045035
Duke Nukem 3D Megaton Edition                      0.044790
Call of Duty Advanced Warfare                      0.043540
GRID 2                                             0.038686
Half-Life Deathmatch Source                        0.037006
NARUTO SHIPPUDEN Ultimate Ninja STORM 3 Full Burst 0.033768
theHunter                                          0.033476
Total War ATTILA                                   0.033310
Garry's Mod                                        0.029794
Titan Quest Immortal Throne                        0.028585
Quake Live                                         0.027719
Tom Clancy's Ghost Recon Phantoms - EU             0.026798
Deathmatch Classic                                 0.021672
McPixel                                            0.019355
```

FIGURE 5 – *Results examples*

Model performance



```
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)
```

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

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

FIGURE 6 – *Performance tests*

SVD

Finally, we wanted to try a last technique to improve our model which was dimension reduction. Indeed, as we compute on really large vectors and matrices (length of the number of games which is huge), we thought that a good way of improving our method was to lower

the number of components we used. To do so, we used Singular Values Decomposition in order to project our matrix in a new space where components would not only be associated with a single game, they would represent real groups in the data, and would be ordered along their importancy in data correlation. For the neighbors computation, we chose to use cosine similarity as a distance (even if not a mathematical distance) because it was the most fitted metrics to such a matrix.
The idea was to use SVD to compute a new input matrix, as a pre-treatment, and to run our k-nn based recommendation model on this new input, with a lower dimension and more sense. When we made the decomposition and studied the meaning of the principal components checking outliers, the results where impressively good, describing the exact real types of people playing on Steam. However, the model suing SVD as a pre-treatment performed really badly compared to the other ones.
We thus think that SVD is a technique with a high potential for this project, but it would need further research, maybe trying to tune both the number of neighbors and of components with a grid-search, or by forgetting the k-nn aspect and only relying on components (giving games prefered by the 'representatives' of the most fitting components).
If you are interested in what have been done on this part, we invite you to check the *kaggle_dataset/svd_knn.ipynb* notebook.

2.5 Evaluation of the K-nn algorithm

Note : no evaluation has been implemented for the fis algorithm because the algorithm implemented did not give a score to the predictions, and optimizing it by tuning the hyper-parameters would have meant training it to predict all the games in the dataset

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, a validation set and a testing set, shuffling the data. Indeed, as we had enough data, we preferend using a hold out method instead of a cross validation. We then optimized our k-nn based algorithm tuning its parameters on the validation set, and finally tested it on the unknown test set. We only standardized the training set not to contaminate the other ones.

Unfortunately, it was difficult getting a visual representation of the results such as drawing decision boundaries because we do not have a clear classification algorithm, and then have a huge number of 'classes'.

There are some metrics to evaluate recommendation algorithms that already exist like NDCG, however with our dataset composed of people that played not much games, it was not possible to use that metric (which requires to predict a lot for the same person) and we thus decided to create our own metrics from scratch, as follows :

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
— false positive if the game predicted was not in the games removed

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.

We used a 'grid search' (with only 1 parameter) method to tune the hyperparameter of the model which is the number of neighbors :

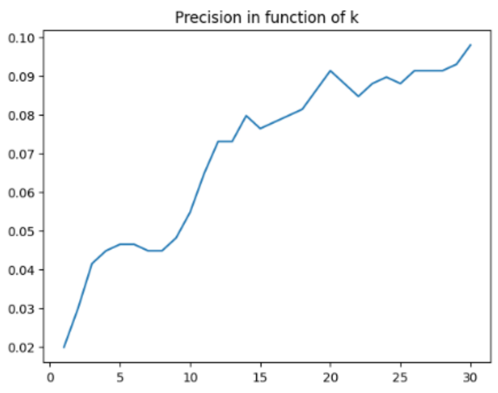


Figure 7 – *Grid-search tuning*

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.

In particular, for $k = 20$, we obtain a **precision of around 5.5%**. If this score seems extremely low, we can in fact consider that our model is performing really well. Indeed, we have to take into account that there are 5155 games in the dataset, which means that the model is performing **200 times better than random**. Moreover, 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. 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.

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. With this metrics, we reached a **precision of 17.6%**, which is around **600 times better than random**. We are thus really proud of our predictions, especially when testing on dummy players and on our own profiles.

3 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 data about users

thanks to Steam's API, including : their id, games bought and hours played for each of their games. The method used was to start with an id, scrap its friends, the friends of his friends, etc... The associated notebook can be checked for more details.
However, when using this dataset with our k-nn based model (in the *custom_dataset* folder), we quickly realized that this dataset was really particular, composed of players with a tremendous amount of games and hours played, which led to huge issues of computation time and harware requiring. The results of the evaluation are pretty good, they even seem higher than those on the kaggle's dataset, but as this dataset has been gathered by scrapping friends of friends, we can think that we gathered the same types of players, and thus that the predictions are easier inside this dataset : the testing set is not really independant from the training set !

Due to this and to the calculation issues, we decided to keep our main presentation on the Kaggle's dataset.

4 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 extremely satisfied with the results, especially when testing the predictions on our own Steam accounts.

4.1 Limitations

As we have seen before and especially with our scrapped dataset, 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.

4.2 Perspectives

The first way of improving our results would obviously to have a wider dataset, with more various types of players, and a more recent one.
To do so, we could use our own dataset but with a very meticulous cleaning process to determine which games are negligible, how to reduce the calculation times, etc...
We could also explore further the SVD technique, or different PCA possibilities (with Word2Vec for instance) in order to use dimension reduction.
Finally, we could add other features in the dataset, using game genres, reviews or grades from players for instance to be more accurate and have other hyperparameter tuning possibilities.