UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as Ingeniería Informática (modalidad bilingüe)

BACHELOR THESIS

Building a videogame recommendation system from scratch based on user and game data

Author: Jorge González Gómez
Advisor: Alejandro Bellogin Kouki

May 2023

Jorge González Gómez
Building a videogame recommendation system from scratch based on user and game data

Jorge González Gómez

PRINTED IN SPAIN

To my family, friends and girlfriend; and those who misjudged me.

“Victory comes from finding opportunities in problems.”

Sun Tzu

Agradecimientos

Thanks to my tutor, Alejandro Bellogin, who has guided me throughout the development of this project, and to my family, friends and girlfriend, who have supported me during this time.

Resumen

Los videojuegos representan casi el 50% de los ingresos de la industria del entretenimiento, y la industria del videojuego no deja de crecer año tras año. Esto ha hecho aumentar el número de videojuegos disponibles para el público general, lo que dificulta que los jugadores encuentren los juegos que les interesan. Los sistemas de recomendación pretenden resolver este problema ofreciendo recomendaciones personalizadas a los jugadores. Sin embargo, la mayoría de las tiendas, ya sean digitales o físicas, carecen de este tipo de sistemas, y las que lo tienen no son muy precisas.

En este trabajo construiremos un dataset con datos obtenidos de Steam, una de las tiendas digitales más grandes del mundo, con más de 30.000 juegos y más de 130 millones de usuarios activos. A partir de una muestra de más de 80.000 usuarios de este dataset, implementamos diferentes enfoques para representar el interés (o puntuación) de los usuarios sobre los videojuegos y diferentes sistemas de recomendación, y compararemos diferentes combinaciones para estos. Nuestros mejores resultados muestran una precisión y recall en los resultados de hasta un 20% de media, un resultado prometedor para un sistema de recomendación implementado desde cero.

Palabras clave

Sistemas de recomendación, videojuegos, crawling, modelado de usuarios

Abstract

Videogames account for almost 50% of the entertainment industry revenue, and the industry keeps growing year on year. This has skyrocketed the number of games available to the public, making it difficult for players to find games they are interested in. Recommendation systems aim to solve this problem by providing personalized recommendations to players. However, most online and physical storefronts are lacking in this regard, and the ones that do have a recommendation system are not very accurate.

In this thesis we will build a dataset with data obtained from Steam, one of the largest online storefronts in the world, with over 30,000 games and over 130 million active users. From a sample of over 80,000 users from this dataset, we implement different approaches to represent the interest (or score) of users on games and different recommendation systems, and compare different combinations for these. Our best results show a precision and recall values of up to 20% on average, a promising result for a recommendation system implemented from scratch.

Keywords

Recommendation systems, videogames, crawling, user modeling

Table of Contents

Lists

List of algorithms

List of codes

List of equations

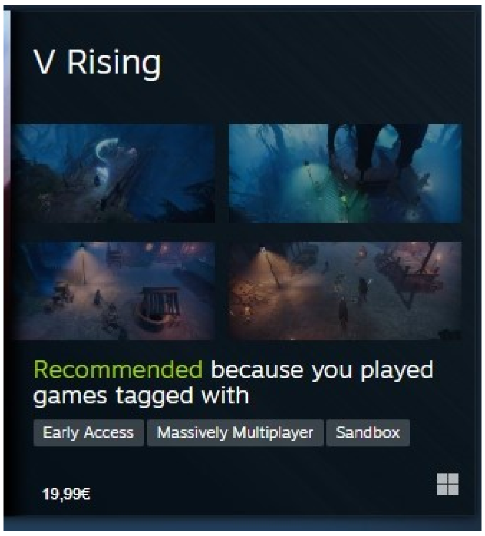
List of figures

List of tables

1 Introduction

The videogame industry accounts for almost 50% of the entertainment industry revenue in the UK, according to the Entertainment Retailers Association (ERA) of the UK [1], and it is likely the same can be said around the globe. The number of games available to the public have skyrocketed and it is difficult for players to find games they are interested in: thus, in this thesis, we will explore different approaches to build a recommendation system for videogames, and discuss the results obtained. In this chapter we will go over the motivation, objectives and structure of this Bachelor thesis. As well, we will briefly go over important terms and concepts that will be used throughout the document.

1.1 Motivation



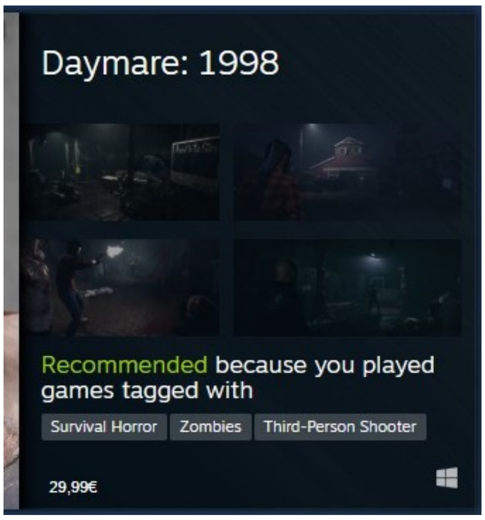
V Rising

Recommended because you played games tagged with

Early Access | Massively Multiplayer | Sandbox

19,99€

(a) Main page example, after playing Tower Unite, a social game unrelated to an action RPG Vampire-themed game.



Daymare: 1998

Recommended because you played games tagged with

Survival Horror | Zombies | Third-Person Shooter

29,99€

(b) Main page example, after playing Resident Evil 4, recommending a Survival Horror, the game's genre.

Figure 1.1: Steam's recommendation examples

A **videogame** is multimedia product that can be interacted with in a personal computer (a PC videogame) or in a console (a console videogame) and provides visual and auditory feedback depending on the player's state, which can go from a simple point-and-click adventure to a first-person-shooter that involves aiming, movement. In this thesis we will narrow down the definition to only take Steam videogames into account.

Steam, created by Valve Corporation in 2023, is a digital game distribution platform, which sells games from any videogame developer, and as of writing over 67,000 games have been published in Steam according to SteamSpy [2].

Steam provides a recommendation system to recommend games to purchase next, called "The Discovery Queue" [3]. However, around the web you can find people complaining about the discovery queue for many reasons: for one, as Figure 1.1 shows, the discovery queue is usually not very personalized, mostly showing popular and new releases, and not taking into account the user's preferences. In fact, the discovery queue might choose to show you popular games without taking into account your personal preferences, and if it does take into account your personal preferences, it will take into account your recently played games. For example, in my personal experience, as I was playing Tower Unite, a purely social game with a bunch of minigames, tagged with "Sandbox", "Massively Multiplayer" and "Early Access", the discovery queue showed me a seemingly 'random' topdown action RPG in the likes of 'Diablo', as shown in figure 1.1(a). However, after playing Resident Evil 4, a survival horror game, the discovery queue would have shown me a lot of survival horror games, as shown in figure 1.1(b).

1.2 Scope

The scope of this Bachelor thesis is to cover basic videogame recommendation systems and different implementations from scratch, exploring which one is more precise and accurate, and which one is more efficient and faster. We will only cover static recommendation systems, with no Artificial Intelligence (AI) or Machine Learning (ML) involved.

We will also only use Steam's data, since every other storefront either requires scrapping, does not have public profiles or does not have a public API [1] [2]; and since Steam is the most popular storefront [3], the data from other storefronts might not be as valuable as the data from Steam.

Ideally, we would like to replicate Steam recommendations. However, we do not know how Steam does its recommendations, as we do not have access to the data they use, and we do not know how they specifically do it. Moreover, we cannot fetch their recommendations for any user unless we have

[1] API stands for Application Programming Interface, which, in the case of storefronts, allows an external app to interact through a set of endpoints.

[2] GOG.com has an unofficial API available in `https://gogapidocs.readthedocs.io/en/latest/`, and the Epic Games Store has another unofficial API available in `https://github.com/SD4RK/epicstore_api`.

[3] Steam had 132M monthly active users as of 2021 [4], and they continued to grow in 2022 [5]. The second biggest PC storefront which reports their user count, the Epic Games Store, claimed they had 68M monthly active users as of December 2022 [6].PlayStation, a console which features the PlayStation Store [7], which is another digital storefront, claimed they had 112M monthly active users users as of 2022 [8]. Xbox, Microsoft's videogame department, claimed they had 120M monthly active users across Xbox consoles, PC, mobile games and cloud [9]. Nintendo, creators of the Switch console, claimed they had 114M 'Million annual users' [10]

access to their account, and we cannot get the data they use to compute their recommendations. Thus, we will not try to replicate their results or systems, but we will nonetheless evaluate our results and see which one is more precise and/or faster.

1.3 Objectives

The goal is to create a recommendation system that is able to recommend games to users based on past preferences, not only taking into account recently played games, through the use of Collaborative-Filtering and Content-Based-Filtering algorithms. We will also explore and compare algorithms between themselves, as well as approaches to see which one is the fastest and if the trade-offs to make it faster or more precise are worth it.

We will compare the tradeoffs between precision and performance, and see if optimizing matters in this case, or if the algorithms are fast enough to not generally need optimization.

1.4 Thesis structure

This document is structured as follows:

Chapter 1. Introduction. This chapter introduces the problem and the motivation behind it, as well as the scope of the thesis and the objectives, and the methodology used to achieve them.

Chapter 2 State of the art. This chapter presents the state of the art of the problem, as well as the different approaches that have been used to solve it.

Chapter 3. Design. This chapter presents the design and implementation of this thesis, including the data model, the data sources, the recommender systems and the evaluation.

Chapter 4. Results. This chapter presents the results of the evaluation, as well as the discussion of the results.

Chapter 5. Conclusion. This chapter presents the conclusions of the project, as well as the future work that could be done to improve upon the work of this thesis.

Annex A. Code. This annex presents parts of the code used in the project. The rest is in the GitHub repository [11].

Annex B. Results. This annex presents the full results obtained.

2 State of the art

In this chapter we will introduce the state of the art in recommender systems, and more specifically, in videogame recommender systems. We will also introduce the metrics that will be used to evaluate the different algorithms, and the data that will be used to test them. Finally, we will introduce the algorithms that will be used in this thesis, and the different approaches that will be used to evaluate them.

Important notes:

• In this thesis, we will use the terms *videogame* and *game* interchangeably. It refers to the items we are trying to predict, and they have some attributes like game tags, genres and categories.

• *'Playtime'* refers to the time a user has spent playing a game. We will use this as the input for our algorithms, sometimes normalized based on the user's top playtime.

• We will also use 'user' and 'player' interchangeably. It refers to the person or client that is using the recommender system, and we will use their 'playtimes' for owned games.

• A *ranking* is a list of games sorted by their score, and a *score* is a number that arbitrarily represents how likely a user is to play a game, in our case, without scale and will be hidden to a user (for example: a recommender system might return a score of 10 for app X and it might be top 1, while another might return a score of 1000 for app Y and it might be top 4, in this case app X might be more relevant to the user).

2.1 Recommender systems

Recommender systems are a type of information filtering systems that seek to predict the rating or preference that a user would give to an item, based on previous ratings, usage, etc. [12]. Recommender systems do not have to use explicit rankings, they can even use read time for articles, as GroupLens would do [13].

These systems are used in many different fields, such as movies, music, videogames, news, books, research articles, search queries, social tags, and many more [12].

We also need to introduce the concept of '*similarity* functions'. These define the similarity between two items or users, and its up to the developer to define them. This concept is useful as more similar items or users to the one we are trying to predict for have a higher impact than less similar ones.

2.1.1 Collaborative filtering

According to the article "Recommender systems" from P. Resnick et al [13], Collaborative Filtering was a term first used by the experimental mail system called Tapestry [14], which was a system that used a collaborative approach to filter out unwanted messages.

In the modern day, Collaborative Filtering (CF) is a technique used by recommender systems to predict a user's rating or preference for an item, based on the ratings or preferences of other users. This is done by finding users with similar tastes to the target user, and then recommending items that those similar users have liked in the past.

Quoting "Introduction to Recommender Systems Handbook" from F. Ricci et al [12]:

"[. . .] the first RSs applied algorithms to leverage recommendations produced by a community of users to deliver recommendations to an active user, i.e., a user looking for suggestions. The recommendations were for items that similar users (those with similar tastes) had liked. This approach is termed collaborative-filtering and its rationale is that if the active user agreed in the past with some users, then the other recommendations coming from these similar users should be relevant as well and of interest to the active user."

kNN or k-Nearest Neighbors is an algorithm that is used in a wide variety of fields, including machine learning, pattern recognition, data mining, and intrusion detection [15].Regarding Collaborative Filtering recommender systems it is a popular method used for classification, used to find the k most similar users to the target user, and then recommend items that those similar users have liked in the past.

2.1.2 Content-based filtering

Content-Based filtering (CBF) is a technique that uses the features of the items to recommend similar items. It is based on the idea that if a user likes an item, then they will also like a similar item. The similarity between items is calculated using a (content-based) similarity function. It can be based on the features of the items, be it its description, its tags, in the context of multimedia like movies or videogames its genre, in the context of articles it may be the author's set keywords, etc. and measures the similarity between two items based on those features.

Usually, but not necesarily, a similarity function can be a function that returns a value between 0 and 1, where 0 means that the items are completely different and 1 means that they are exactly the same, sometimes as simple as a boolean function that returns 1 if the items are equal and 0 otherwise. It can be defined in many different ways, and it is up to the developer to choose the one that best fits their needs.

IDF (Inverse Document Frequency) is a technique used by search engines to weight the importance of a word in a document, by measuring how many documents contain that word, under the assumption that the more documents contain a word, the less information this word would give (e.g. the word ‘the’ appears in every document, but the word ‘goose’ only appears in documents related to geese, so if we were to find 3 documents by searching ‘the anatomy of a goose’ we would ignore the word ‘the’ but return those with ‘anatomy’ and ‘goose’). We will measure if this assumption holds true in the case of CBF recommender systems.

2.1.3 Hybrid recommender systems

Hybrid recommender systems are recommender systems that combine two or more different recommender systems to get a better result.

The most common ways to combine different recommendation systems are [16]:

Method	**Description**
Weighted	Each recommender system is assigned a weight, and the final score is the weighted sum of the scores from each recommender system.
Switching	Each recommender system is assigned a threshold, and the final score is the score from the recommender system that passes the threshold.
Mixed	The final score is a combination of the scores from each recommender system.
Feature Combination	The features from each recommender system are combined to create a new recommender system.
Cascade	The first recommender system is used to create a list of recommendations, and then a second recommender system is used to re-rank the list of recommendations.
Feature Augmentation	The output of a recommender system is used as an input for another recommender system.
Meta-level	The model learned by a recommender system is used as an input for another recommender system.

Table 2.1: Popular approaches to hybrid recommender systems, as described in Table III from [16]

In the results, we will discuss different non-hybrid approaches and try to find which combination of recommender systems would give us the best results, but given the time it takes to process the results (discussed in Section 4.1) it would take too much time to test all possible combinations for every single user.

2.1.4 Minhashing

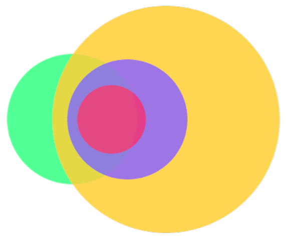
MinHashes, first introduced by Andrei Z. Broder in the year 2000 [17], estimate the Jaccard index, or Jaccard similarity coefficient (also known as resemblance) [18] between two sets. If we have two sets

A and B, then the Jaccard index $J(A,B)$ is equal to the size of the intersection of A and B divided by the size of the union of A and B:

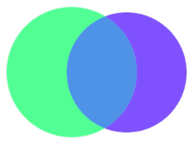
$$Jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{2.1}$$

We can use MinHashes and Locality Sensitive Hashing (or LSH) [19] to quickly filter out items that are not similar to the target item in various ways, saving us time. For this purpose, the library *datasketch* [20] implements various MinHash algorithms and provides various ways to query: Locality Sensitive Hashing (LSH [19,21]), LSH Ensemble [22] (a modified version of LSH which uses containment instead of Jaccard's similarity, explained below, in equation 2.2) and LSH Forest [23] (also explained later). Their purpose is to avoid the need to compare every single item to the target item, which would be O(n).

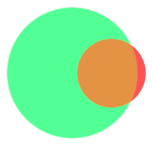
Let us take an hypothetical example to explain how Locality Sensitive Hashing works in a general recommender system: we have a set of items G (represented as green in figure 2.1) and various set of items B (blue), R (red) and O (orange), and we want to find out how similar these are to our target G.



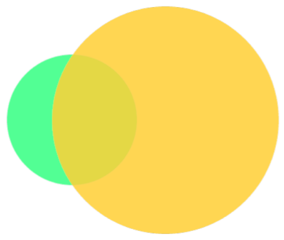
(a) All figures overlapped. We can see the orange O set's intersection is the biggest with G, our target set.



(b) G overlapped with B.



(c) G overlapped with R.



(d) G overlapped with O.

Figure 2.1: Visual examples of intersecting sets to explain our approach

As we can see in the figure 2.1(b), following Jaccard's similarity, the set B would be the most similar to G, since the intersection is the largest one and $|B|$ is the smallest. However, if we look at the figure 2.1(d), we can see that the orange set has the largest intersection with G, but it also has the largest size. If we only took the intersection into account, the orange set would have the highest similarity. For this purpose, we can use the 'containment' function, which is defined as follows:

$$Containment(A, B) = \frac{|A \cap B|}{|A|} \quad (2.2)$$

We want to search similar ones in sub-linear time without resorting to comparing every minhash against each other one, Locality Sensitive Hashing indexes (we will abbreviate them as LSH's) are helpful: LSH and LSH Ensemble both work with a threshold t, and they will return all the items that have an estimated Jaccard similarity or containment (equation 2.2) respectively of t or more with the target item in sub-linear times. LSH Forest [23] , however returns a 'ranking' of items, but it is only useful if we only need to use Jaccard's similarity, even if just to filter out items. LSH Forest is also usually used with 'post-processing', for example: if we wanted K items, we could use LSH Forest to get $2K$ items and then use an accurate Jaccard's similarity to get the K most similar ones.

The regular LSH uses the Jaccard similarity function, which punishes big sets, which often contain more information and would yield better results in our recommender systems. LSH Ensemble uses the aforementioned 'containment' function instead of Jaccard to filter out likely irrelevant items or users without punishing those with bigger MinHashes. This way, it only takes into account the size of the intersection and the size of the target set only, ignoring the size and items of the other set. Containment or Jaccard itself is not also always the best option, but it filters out irrelevant items or users and applying custom similarity functions to these filtered items should be faster than not filtering out items at all: after all, when computing similarity, most of the times the ones with more items in common are the most similar to each other.

2.1.5 Other optimization techniques

When doing recommender systems, it is usually good practice to have some kind of optimization to filter out potentially irrelevant items. This is done to reduce the amount of items that need to be processed, and thus reduce the time it takes to process the recommendations. We will not go over every optimization technique, however, we will briefly go over the ones we used in our implementation besides MinHashing, and some that complement MinHashing.

Related to MinHashing

The caveat with MinHashing and LSH Ensemble is that we would either be forced to use the classic LSH which penalizes big sets (as explained in the previous chapter) and be able to use Weighted MinHashes, implemented in datasketch too [24], or use LSH Ensemble and be forced to use sets of items without any weight.

Weights [1] would provide a very useful feature: we could use them to give more importance to some items than others, as an example: let us have, in Collaborative-Filtering, an hypothetical user u_1 rated

[1] In computer science, weights can be simply explained as multipliers: if we have a list of tuples $val, weight$ one can see between $x, 5$, $y, 3$, and $z, 2$ x would have the biggest weight in the final result.

item i_1 with 5 out of 5, i_2 with a 4/5 and did not rate i_3, i_4 and i_5, but has shown some interest in them. Using LSH Ensemble, we ca not give i_1 more "weight" in any straightforward way, as it does not support Weighted MinHashes.

However, we want to find similar sets of items given a threshold t; we can ignore "not really important items" and only add user's u_1 top items. This also comes with a caveat: In this case, their "top K items" are i_1 and i_2, K = 2; but we can easily find some user $\boldsymbol{u_i}$ with "top $\boldsymbol{K}$ items" being, for example, $\boldsymbol{i_3}$ = 5, $\boldsymbol{i_4}$ = 4.5 and $\boldsymbol{i_5}$ = 4, K = 3. If we were to use MinHashes of size $\boldsymbol{K = 2}$, we would be ignoring $\boldsymbol{i_3}$ for this hypothetical user $\boldsymbol{u_i}$.

Therefore, various observations come to mind:

1. Take a fixed number (as we have explained, we will leave out some important items)
2. Take a fixed percentage of the user's item (for example, the top $\boldsymbol{70\%}$ of the user's items, which would be $\boldsymbol{K = 3}$ for the previous example and would randomly pick a item from $\boldsymbol{i_3}$, $\boldsymbol{i_4}$ and $\boldsymbol{i_5}$ for the hypothetical user $\boldsymbol{u_1}$)
3. Take items that have been a rating above a certain number (for example: items with a rating above 4.0)
4. Take items that have been rated in relation to their top item (for example, items that have been rated for at least 50% of the rating of their top item)

It is not intuitive why the 4th approach would matter unless we introduce the concept of 'implicit ratings'. Let us define interest (or implicit ratings) $\boldsymbol{I}$ as an arbitrary function that returns a number between 0 and 1, where 0 means no interest and 1 means maximum interest. We can then define the interest of a user $\boldsymbol{u}$ for an item $\boldsymbol{i}$ as $\boldsymbol{I(u, i)}$. As an example with items that can be used over indefinite periods of time: if $\boldsymbol{u_1}$ used item $\boldsymbol{i_1}$ for 100 hours and item $\boldsymbol{i_2}$ for 80 hours, and $\boldsymbol{u_2}$ used item $\boldsymbol{i_1}$ for 50 hours and item $\boldsymbol{i_2}$ for 40 hours, they would relatively have the same interest in both items: $\boldsymbol{I(u_1, i_1) = I(u_2, i_1) = 1}$ and $\boldsymbol{I(u_1, i_2) = I(u_2, i_2) = 0.8}$. If we were to use the 3rd approach and take the top 75%, we will be ignoring $\boldsymbol{i_2}$ for both users. Instead, by using the 4th approach, and using also a 75% threshold we would be taking both items for both users. This way, in this case, when querying the LSH Ensemble we would only take into account the most relevant items for each user, which have the biggest impact on the final 'similarity' result of most recommender systems; thus skipping having to process items that will very likely have a very small impact on the final result.

Other optimizations

When using weighted attributes we can simply index items that have a weight below a certain threshold $\boldsymbol{wt}$ using a simple dictionary $\boldsymbol{attribute \rightarrow item\ a \rightarrow i}$ and then simply compute the similarities between our item $\boldsymbol{i_o}$ and every other item returned by this dictionary. We do this since we are searching for relevant items with attribute $\boldsymbol{a}$, and we can assume that items with a weight below $\boldsymbol{wt}$ will not be as relevant for this tag as items with a weight above wt. This skips having to process every item with attribute a.

2.1.6 Metrics for recommender systems

To measure the accuracy of the results, we will first split a random portion of the items into a training set I_{train} and a test set I_{test}, where the training set is bigger and will be used to train the recommender systems, and the test set will be used to evaluate the performance of the recommender systems.

Let Rel be the set of relevant items in the test set I_{test}, and Ret be the set of retrieved items from a recommender system r_i, which uses the training set I_{train}, for an user to be tested u_i, the following metrics are used to evaluate the performance of the system:

Precision is defined as the number of relevant items retrieved (items retrieved which also appear in I_{test}) divided by the total number of items retrieved:

$$Precision(r_i, u_i) = \frac{|\{Rel\} \cap \{Ret\}|}{|\{Ret\}|} \tag{2.3}$$

Recall is defined as the number of relevant items retrieved divided by the total number of relevant items in the test data I_{test}.

$$Recall(r_i, u_i) = \frac{|\{Rel\} \cap \{Ret\}|}{|\{Rel\}|} \tag{2.4}$$

We will use these two methods to see how accurate our results are. Specifically, we use Precision@k and Recall@k, where k is the number at which we are calculating the Precision and Recall (e.g. Precision@10 and Recall@10 are the precision and recall of the first 10 items retrieved by the recommender system, respectively).

2.2 Recommender systems in videogames

Recommendation systems in videogames

Let us first introduce the domain of videogames: 'items' refer to videogames, and 'users' refer to players. We will use the terms 'item','game' and 'videogame', and 'user' and 'player' interchangeably, respectively, throughout the rest of this thesis.

It is worth mentioning that in 2016, a paper called "Condensing Steam: Distilling the Diversity of Gamer Behavior" [25] was published, which claimed crawling all 108.7M users at the time of the crawling. They also provide a dataset available in `https://steam.internet.byu.edu/`. However, this dataset, although massive, is outdated and does not mention how they fetched every user.

More recently, in 2022, a paper called “Large-scale Personalized Video Game Recommendation via Social-aware Contextualized Graph Neural Network” [26] examined different approaches to recommendation systems based on Graph Neural Networks [2], and concluded that SCGRec (their proposed recommendation algorithm) performed better in every metric than every other previous approaches, making it the state of the art in videogame recommender systems. They also do not provide a meaningful explanation on how they fetched the data, but they do provide a dataset of 500.000 users. However, the goal of this thesis is to collect the data for ourselves, and then implement and compare different recommendation algorithms, all from scratch, and, as such, we will not use this paper as a baseline, but it should be considered in the future for completeness.

Recommendation systems in the videogame industry

When it comes to PC or computer videogames, there are various storefronts, which have different approaches to how they sell their games. As mentioned in Section 1.1, Steam has the Discovery Queue [3], a recommendation system that recommends games to purchase next. It is usually not very personalized, mostly showing popular and new releases, and never takes into account user’s old preferences. It also has other systems, mentioned in Figure 2.2, “Games like this” and “More from this developer” sections, also one called “Players like you love” which uses Collaborative-Filtering. [27]

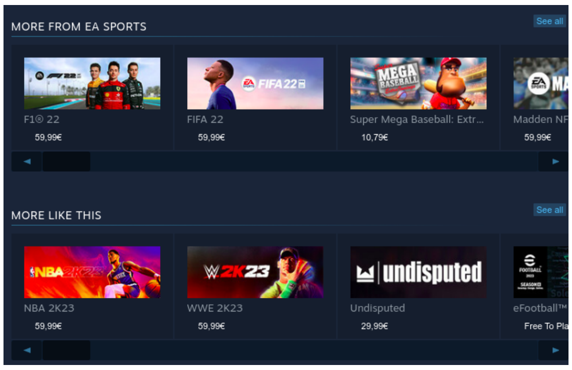
Valve explained how “Players like you” works on Steam [27]: “The Interactive Recommender uses a machine learning model that is trained based on the playtime histories of millions of Steam users. It is not directly affected by tags or reviews—it instead learns about the games on Steam by looking at what users actually play. The basic idea is that if there are other players with similar play habits to you, who also play a game that you have not tried yet, then that game is likely to be one you’ll enjoy too.” They also provide a frontend for this interactive recommendation system, which is shown in figure 2.3.

GOG.com [28], another storefront, uses a DRM-free approach and did not use to track user’s playtime before GOG Galaxy 2.0 (a client that is optional to use), so it mostly only tracks ownership and reviews. They have a "Recommended for you" section, which in principle is based on the user’s library, but we have not been able to find any information about how it works.

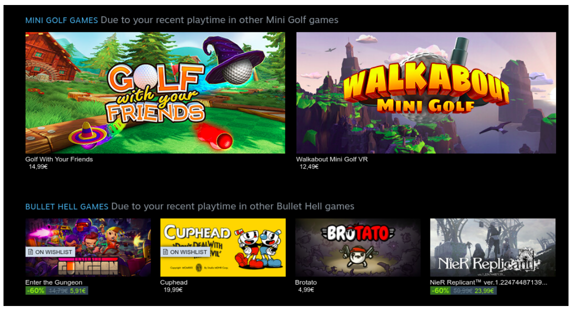
Xbox PC, as well, has "Picks for you", but it is not clear how it works, and they do not explain how it works anywhere, and there is no information about it.

PlayStation, a non-PC platform but the second biggest videogame platform (111M monthly active users in 2021, 108M in 2022 Q4 [8]), had a webpage called “Just for you” that was not very helpful [29], and as far as we could find, someone made a Medium article explaining they only have a “More like this” section (which, as shown, is not very accurate) and “Expand your games”, which is a section that

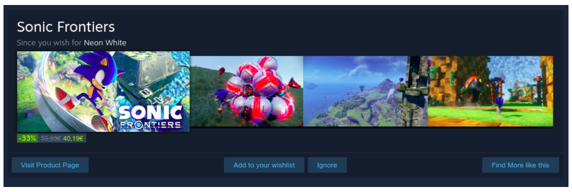
[2] Graph Neural Networks represent the data as a graph, but its details are out of the scope for this thesis.



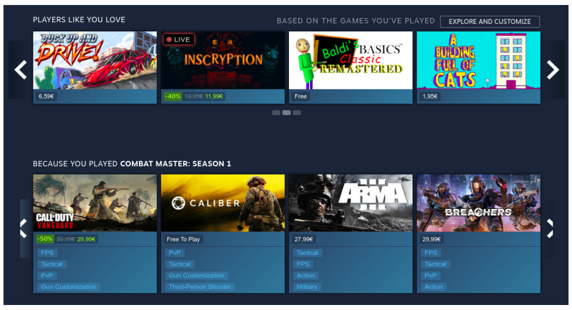
(a) Example for FIFA 23 of 'More from this developer' and 'More like this'


MINI GOLF GAMES Due to your recent playtime in other Mini Golf games
Golf With Your Friends
Walkabout Mini Golf VR
BULLET HELL GAMES Due to your recent playtime in other Bullet Hell games
Enter the Gungeon
Cuphead
Brotato
NieR Replicant™ ver.1.22474487139...

(b) Two examples of 'More like this' for one single tag

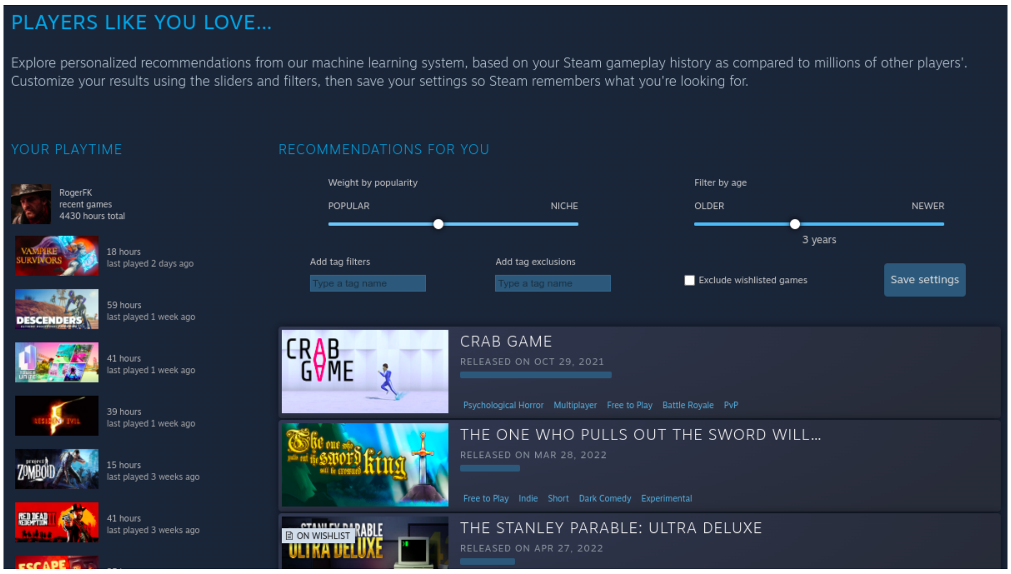


(c) Example of 'More like this' for a single game in your wishlist


PLAYERS LIKE YOU LOVE
BASED ON THE GAMES YOU'VE PLAYED
EXPLORE AND CUSTOMIZE
BECAUSE YOU PLAYED COMBAT MASTER: SEASON 1

(d) Example of 'players like you like' and 'more like Combat Master' (a Call of Duty, first person military shooter, clone)

Figure 2.2: Other Steam recommendations



PLAYERS LIKE YOU LOVE...

Explore personalized recommendations from our machine learning system, based on your Steam gameplay history as compared to millions of other players'. Customize your results using the sliders and filters, then save your settings so Steam remembers what you're looking for.

YOUR PLAYTIME

RogerFK
recent games
4430 hours total

18 hours
last played 2 days ago

59 hours
last played 1 week ago

41 hours
last played 1 week ago

39 hours
last played 1 week ago

15 hours
last played 3 weeks ago

41 hours
last played 3 weeks ago

RECOMMENDATIONS FOR YOU

Weight by popularity
POPULAR NICHE

Filter by age
OLDER NEWER
3 years

Add tag filters
Add tag exclusions
Exclude wishlisted games
Save settings

CRAB GAME
RELEASED ON OCT 29, 2021
Psychological Horror Multiplayer Free to Play Battle Royale PvP

THE ONE WHO PULLS OUT THE SWORD WILL...
RELEASED ON MAR 28, 2022
Free to Play Indie Short Dark Comedy Experimental

THE STANLEY PARABLE: ULTRA DELUXE
RELEASED ON APR 27, 2022

Figure 2.3: Steam interactive recommendations

only sells DLC even if you own them [30].

As for the other main storefront on the PC market [6], the Epic Games Store, it does not have any kind of personalized recommendations. There are also more storefronts, ommited, since they are either not too popular (like itch.io) or specific to one publisher, like EA app, Battle.net or Ubisoft Connect, and would not be very helpful.

To the best of our knowledge there also is no API in any of the aforementioned storefronts except for Steam, which offers a public API to get information about games, users, etc. (except for tags, which need to be scrapped from the web). As a consequence, Steam provides the easiest way to get data about users and games and happens to be the storefront with the most users and games.

Steam's API

As we have mentioned earlier, Steam is the only storefront that officially provides an API. Valve provides a Python wrapper for the API, called *ValvePython/steam*, available in GitHub [31]. It is mantained by Valve, but python-steam-api [32], an unofficial wrapper, is more intuitive and easier to use.

Publicly available previous work on the field does not detail how they gathered new users, and the API does not provide any way to fetch existing users in batch. By reading the official Steamworks Web API documentation [33], this leaves us with some options:

- Generate random valid SteamIDs and check if they exist through the *ISteamUser/GetPlayerSummaries*. This is not an useful option as it would take a lot of time to find existing users, even if the API allows getting users in batches of 100 users per API call.
- Programming a spider to crawl friends of friends through the *ISteamUser/GetFriendList* endpoint.
- Getting all reviews for a game through the *appreviews* endpoint and storing the users that wrote them.

Both the second and third options are viable, but the third one is more reliable, as we can not guarantee every user is connected to each other in the second option, and we might only get a small portion that centers around, for example, a single genre or a saga of games.

By fetching reviews, we can pick popular games that we know are aimed at a general public, thus increasing our chances of getting a more diverse set of users, as it is also more likely that a user who has written a review is intuitively to be more likely to interact with Steam's store page more and purchase more games than someone who only befriends other users or plays with friends, as people who only ever play with friends might not be interested in purchasing new games.

We have mentioned we will use *python-steam-api*, however, to fetch reviews we will use *steam-reviews* [34], a Python library that uses the Steamworks Web API to fetch reviews which features rate-limiting and handles pagination for us.

3

Design and implementation

3.1 Overall design

The basic structure of the project developed during this Bachelor thesis is shown in Figure 3.1

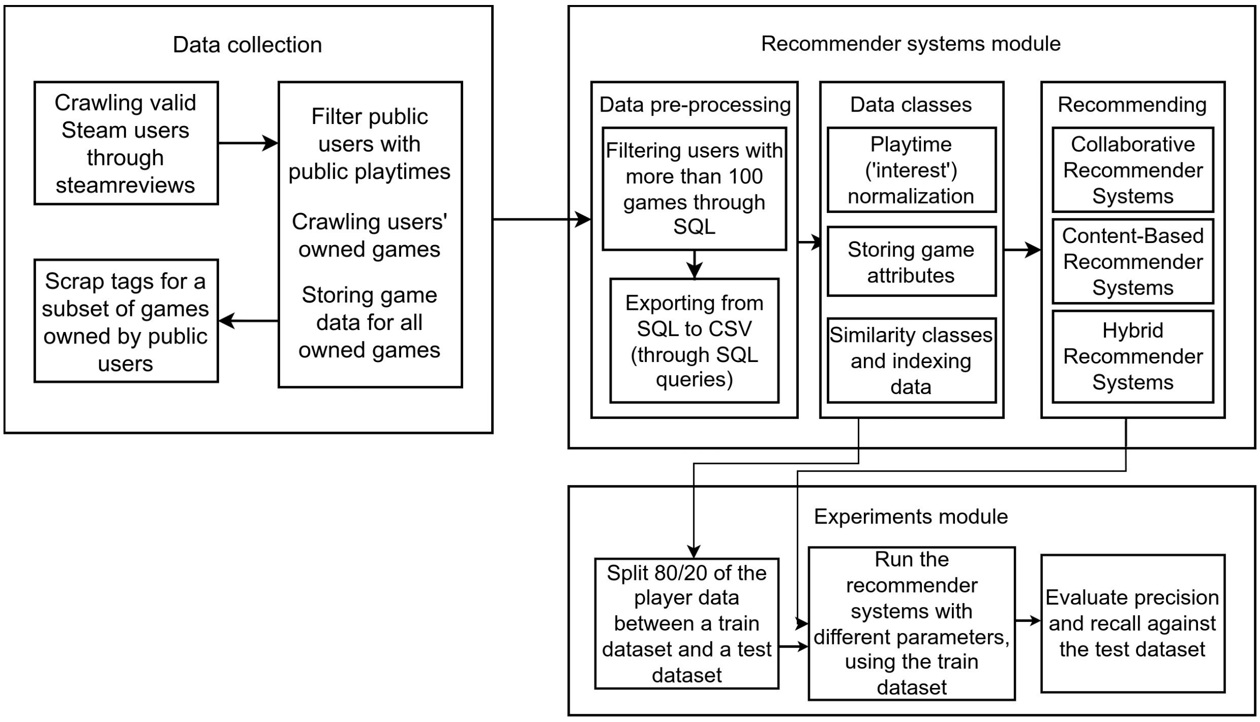

Data collection
Crawling valid Steam users through steamreviews
Filter public users with public playtimes
Crawling users' owned games
Storing game data for all owned games
Scrap tags for a subset of games owned by public users
Recommender systems module
Data pre-processing
Filtering users with more than 100 games through SQL
Exporting from SQL to CSV (through SQL queries)
Data classes
Playtime ('interest') normalization
Storing game attributes
Similarity classes and indexing data
Recommending
Collaborative Recommender Systems
Content-Based Recommender Systems
Hybrid Recommender Systems
Experiments module
Split 80/20 of the player data between a train dataset and a test dataset
Run the recommender systems with different parameters, using the train dataset
Evaluate precision and recall against the test dataset

Figure 3.1: Basic structure of developed project.

Let us now describe each of the modules and submodules included in this structure.

Data Collection module

This module is subdivided in 3 parts, and all read from and write to a SQL database:

'App details' (basic game data) and review data

This submodule gets every review for hand picked videogames, as the Steam API does not offer any way to get any list of their users. It uses the steamreviews [34] library and stores the ‘appdetails’ (basic game data) and user basic info from these reviews.

User owned games

This second submodule goes over the users from the previous submodule and gets all the games they own, only if the player’s profile is public and their owned game list is public. We will mark users as “private” and not use them for our recommendations if their profile or game list is private.

Game tags

This third submodule gets the tags for each game, which better represent the game’s genres, themes and how Steam users perceive the game, as they are sorted by popularity. This helps us better classify the games, since the tags are more specific than the genres and as they are sorted by popularity, they are more accurate describing the game’s features.

Recommendations module

Player playtime normalization and data structures

This submodule normalizes the data, previously exported as CSV from the SQL database through any SQL client, and produces normalized data in various ways in RAM, using Pandas [35]. As well, we have a data structure for each of the separate game attributes: tags, genres, categories, developers, publishers and game details; and one for the users’ playtime for each game. This data is used in the next submodule.

Recommender systems

This submodule takes normalized data and produces recommendations for each user and implements various techniques (explained in Section 3.6) to produce recommendations, and outputs it as a Pandas [35] DataFrame.

Experiments and evaluation module

The last module takes the ‘Recommender systems’ module’s output and evaluates the recommendations using a 80/20 split to predict the 20% of the test data, calculates various metrics (explained in Section 4.1) and produces graphs, tables and average results to better visualize the data. It also exports results as CSVs.

3.2 Requirements

Functional requirements

Data collection module

FR 1 A script must crawl all reviews exposed by Steamworks API [33] for each game from a list of Steam game IDs (list of *appid*), and extract the user info (*user_id*, *num_reviews*, *num_games_owned*) and app details (name, number of reviews, etc.) from the Steamworks API [33] for each review, and insert it into a database.

FR 2 A script must fetch unprocessed players (players with no information about private/public profile, and private/public games owned) from the SQL database and query user data from the Steamworks API [33] for each player, as well as owned games if the player's profile is public, and insert it into the database.

FR 3 A script must take a list of unprocessed games (games with no information about tags) from a list stored in a CSV file, must crawl and scrape the tags for each game from Steam's webpage for each game, and output the extracted tags into a JSON Lines file [1].

FR 4 A script, as part of this system, must convert the extracted tags from **FR 3** as JSON Lines into the SQL database.

Recommender system module

FR 5 A script must take the player playtime data from a CSV file, previously exported from the SQL database through any SQL client (detailed in **NFR 8**), allow the user to pick different approaches for normalization (linear, logarithmic or using nth-roots), normalize the playtime data, and output it as a Pandas [35] DataFrame [2]. It must allow the user to implement their own normalization function easily, by wrapping everything except the normalization function.

FR 6 The system must take all player playtime data and game data from a CSV files, previously exported from the SQL database through any SQL client, and allow the user to pick different approaches for normalization from **FR 5** to normalize the players' playtime data to be used in the recommender systems.

FR 7 The system must allow the user to pick different approaches for recommender systems (content-based, collaborative filtering, hybrid) and output the recommendations for each user into a Pandas DataFrame or a CSV file, only taking the user's Steam ID into account.

FR 8 The recommender system algorithms must be implemented from scratch.

Experiments and evaluation module

FR 9 The system must split the same exported data as **FR 5** (detailed in **NFR 8**) into training and testing sets.

[1] JSON Lines objects represented as JSONs separated into one line per object

[2] A DataFrame represents data in two dimensions, with rows and columns, where each column defines the type of data it stores and each row stores a 'record', just like an Excel spreadsheet or SQL.

FR 10 The system must take the recommender systems from **FR 7** with Pandas [35] DataFrames as input, and output the data as a CSV file, per user.

FR 11 The system must process and calculate the result averages into CSV files using precision (equation 2.3) and recall (equation 2.4) as metrics.

Non-functional requirements

General

NFR 1 The system must be written in Python 3.11 [36], in English.

NFR 2 All algorithms, except explicitly said otherwise, must be implemented from scratch.

NFR 3 The system must use Anaconda [37]'s package manager.

Data collection module

NFR 4 The system must use MySQL [38] as the database management system (DBMS), and install it locally.

NFR 5 The system must use Scrapy [39] for web crawling and scraping.

NFR 6 The data crawling system system must use steamreviews [34] and python-steam-api [32] to crawl data from Steam.

NFR 7 The system will implement every other web crawling algorithms from scratch (except for the tag crawler and fetching steam reviews)

NFR 8 A SQL query will be used to export data from the database as CSV files for the recommender system and experiments and evaluation modules.

Recommender system module

NFR 9 The system must use Pandas [35] for data manipulation.

NFR 10 The Collaborative-Filtering recommender system must use the library 'concurrent' [40] to speed up the process of calculating the similarity between users.

NFR 11 The recommender systems system must use datasketch [20] to index users and fetch them faster.

NFR 12 The system will be able to store results through Pickle [41].

NFR 13 The system will be able to cache data both in RAM and in Pickle files.

Experiments and evaluation module

NFR 14 All of the experiments and evaluation module will be implemented from scratch, except for plotting, which will use Matplotlib [42].

3.3 Data crawling and scrapping

3.3.1 Data model

As our data model, as mentioned previously, we will use MySQL to store all the data, while the recommender systems will however use subsets of the data, stored in CSV files, to have a ‘frozen’ subset of the data to work with, and to be able to work with the data without having to query the database every time. Our data model is shown in figure 3.2.

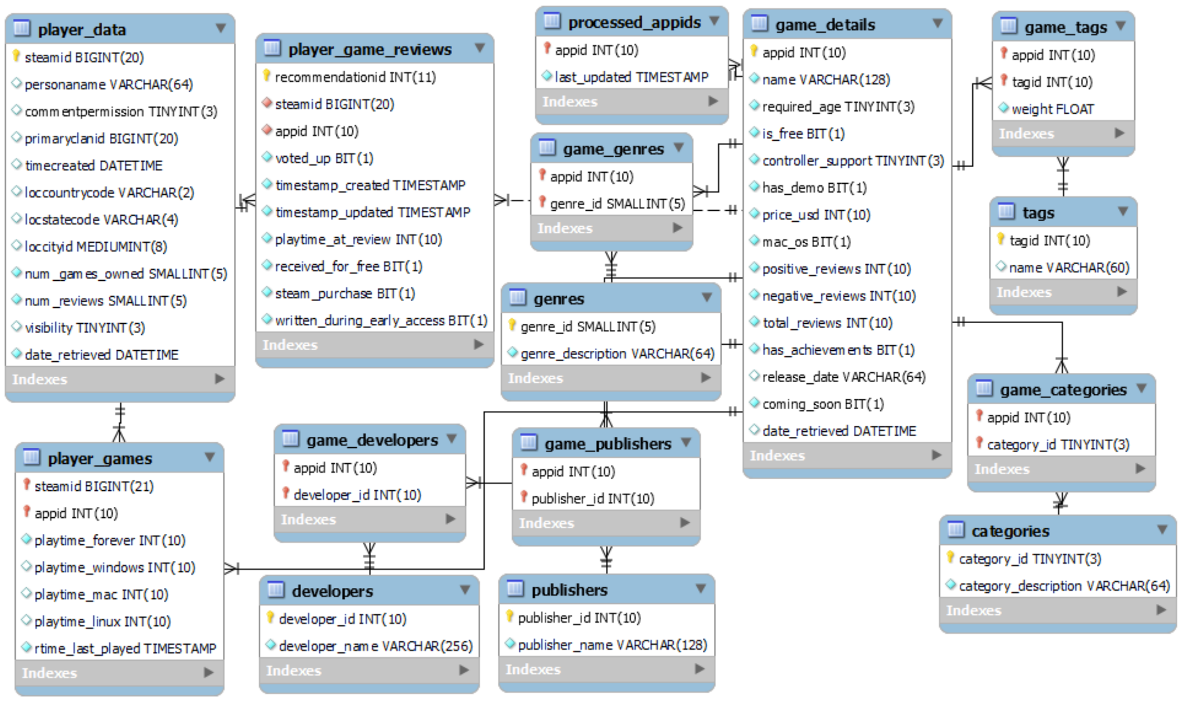


Figure 3.2: Our data model. Will be exported to CSV files for the recommender systems to use.

3.3.2 Data sources

The data sources we will use are the **Steamworks API** [33] endpoints, and we will use the *python-steam-api* [32] as our wrapper. We have mentioned Valve (the creators of Steam) provide their own library [31], however, it does not provide any real advantage over *python-steam-api* in our case, and *python-steam-api* provides their own classes, which are more helpful. The only real advantage would be their “throttler” [3], but we will implement our own rate limiter either way since a simple throttler would mean we would not use the 100,000 API requests per day, the maximum allowed by Valve [43].

We will use the *steamreviews* [34] library to get the reviews, as it is a wrapper for the Steam API and handles rate limiting for us. To get tags, we will use Scrapy [39], a web scraping framework, to scrape

[3] Throttling refers to the practice of purposefully delaying calls to a service to save on bandwidth or to handle rate limiting.

the Steam store page for each game, as the Steam API does not provide any way to get the tags for each game.

3.4 Data crawling implementation

3.4.1 Getting users through reviews

As mentioned previously, the easiest way to get heterogeneous users is to crawl them using *steam-reviews* [34]. We implemented a script that gets the reviews for a list of games, and then gets the users from those reviews. We also store the details for the game we are scrapping (table *game_details* in figure 3.2) by using *python-steam-api* [32], calling the 'appdetails' endpoint. The script is shown in listing A.4. The important extract is shown in listing 3.1.

Code 3.1: Simplified code with no error control to get users from reviews. Full script in Code A.4

```
def get_app_data(app_id):
    response = request("GET", "https://store.steampowered.com/api/appdetails", params={"appids":
        app_id, "cc": "us", "l": "english"})

    appdata = json.loads(response.text)
    if appdata[app_id]["success"]:
        name = appdata[app_id]["data"]["name"]

    # We have to wait for steamreviews to finish crawling all the reviews
    review_dict, query_count = steamreviews.download_reviews_for_app_id(app_id)
```

3.4.2 User games and game details

Once we have the users, we can get their games and the details for those games. We call the Steam-works API [33] to get details for users in batches of 100, using the *GetPlayerSummaries* endpoint [4]. We use *python-steam-api* [32] to get the games for each public user, and if we get a list, we find out if the playtime is public. To store new games we can re-use the previous script from Section 3.4.1, but only getting the 'appdetails'. An extract of this script is shown in Code 3.2.

3.4.3 Scraping game tags

To scrap game tags, we chose to use Scrapy [39], which provides tools to throttle connections, rate limit and avoid getting banned or flagged by Steam.

[4] This endpoint can take up to 100 SteamIDs and return the summaries for users, and most importantly if their profile is public.

Code 3.2: Simplified code with no error control to get users from reviews. Full script in Code A.1

```python
def crawl_player_data(query_count=0, reviews=False, only_games=True):
    players = get_100_unprocessed_players()
    response = steam.users.get_user_details(",".join([str(player[0]) for player in players]), single=False)
    for player in response["players"]:
        steamid = player["steamid"]; name = player["personaname"]
        # check for private profile
        if player["communityvisibilitystate"] < 3: process_steam_user(steamid, name, 0, 0); continue
        ... # part to get user details, unimportant here
        owned_games = steam.users.get_owned_games(steamid)
        # check for private game list
        if "games" not in owned_games: process_steam_user(steamid, name, 1, 0, ...); continue
        for game in owned_games["games"]:
            appid = game["appid"] ; success = game_exists(appid)
            if not success: success, query_count = get_app_data(str(appid), ...)
            if success == SKIPPED: continue
            elif success == ERROR or success == FAULTY:
                add_dead_hidden_game(appid) if success == ERROR else add_faulty_game(appid)
            visible_playtime |= game["playtime_forever"] > 0
            # Here we insert the player-game data
            insert_player_game_data(steamid, appid, game["playtime_forever"], ...)
        process_steam_user(steamid, name, 2 if not visible_playtime else 3, ...)
```

Code 3.3: Script to crawl tags, that yields a dictionary with the appid, its name and the crawled tags

```python
class TagSpider(scrapy.Spider):
    name = 'tags'
    allowed_domains = ["steampowered.com"]
    bypass_age_cookies = { 'mature_content':'1', 'birthtime': '945730801', 'lastagecheckage': '21-0-2000' }
    def start_requests(self):
        appids = []
        url = "https://store.steampowered.com/app/"
        try:
            appids = pd.read_csv('appids.csv', encoding='utf-8-sig')
            appids = appids['appid'].astype(str).tolist()
        except FileNotFoundError:
            print("appids.csv␣not␣found.␣Make␣sure␣it's␣in␣this␣directory:␣" + os.getcwd())
            return
        for appid in appids:
            yield scrapy.Request(url=url + appid, cookies=bypass_age_cookies, callback=self.parse_tags)
```

Code 3.4: Function to parse tags, that uses Scrapy

```
    def parse_tags(self, response):
        if '/agecheck/app' in response.url:
            g_sessionid = response.xpath('//script[contains(text(),
                "g_sessionID")]/text()').extract_first().split('"')[1]
            yield scrapy.FormRequest(
                url='https://store.steampowered.com/agecheckset/' + "app" + "/" +
                    response.url.split('/')[4],
                method='POST', callback=self.parse_tags,
                formdata={ 'sessionid': g_sessionid, 'ageDay': '21', 'ageMonth': '2', 'ageYear': '2000' }
            )
        else:
            yield {
                'appid': response.url.split('/')[4],
                'name': response.css('div.apphub_AppName::text').extract_first(), # useful for debugging
                'tags': [tag.strip() for tag in response.css('a.app_tag::text').extract() if tag != '+']
            }
```

As we can see in Code 3.3, we define the *scrapy.Spider* class and gets all appids from a CSV file, previously exported from SQL. We just extract the 'appids' from the database, and we do a HTTP request using scrapy's library.

The second part of the spider, shown in Code 3.4, parses the tags from the HTML response, which handles the age check and makes a second request if necessary.

3.5 Data preprocessing and classes

3.5.1 Data pre-processing

If we were to use the entirety of the database, we would be looking at almost 300.000 public users with their videogame list set as public, with varying amounts of games, as shown in Table 3.1.

Privacy setting	Number of users
Private profile and private game list	121,993
Private profile and public game list	664,831
Public profile and private game list	26,135
Public profile and public game list	285,260

Table 3.1: Number of users with private profiles, public profiles with private game lists and public profiles with public game lists, also split by people with private playtimes and people with public playtimes.

We need to export a portion of the database to CSVs, as we cannot load the entirety of the database into RAM (which, as-is, is over 7GBs), so we will filter the data as mentioned in Code 3.5. We will be using SQL to export the data to CSVs, as defined in Code 3.6.

Code 3.5: Filtering SQL data according to our preset parameters to get a subset of players.

```sql
CREATE OR REPLACE VIEW player_with_X_games AS (
    SELECT steamid, COUNT(*) AS games
    FROM player_games --NATURAL JOIN appid_owner_count
    WHERE player_games.steamid IN (SELECT steamid FROM player_data WHERE
         player_data.visibility = 3)
    GROUP BY steamid
    HAVING games > 30 --people with X games
);

CREATE OR REPLACE VIEW appid_owner_count AS (
    SELECT appid, COUNT(*) AS visible_owners
    FROM player_games
    WHERE player_games.steamid IN (SELECT steamid FROM player_data WHERE
         player_data.visibility = 3 AND steamid IN (SELECT steamid FROM player_with_X_games))
    GROUP BY appid
    HAVING COUNT(*) > 10 --games with more than X ownerships
```

Code 3.6: Exporting filtered users from SQL to CSV. A similar query, using our temporary table, is used to export tags, genres, categories and game details.

```sql
DROP TABLE IF EXISTS temp_result;
CREATE TEMPORARY TABLE temp_result
SELECT pd.steamid, pg.appid, playtime_forever
FROM player_data pd JOIN player_games pg ON pd.steamid = pg.steamid
WHERE pd.visibility = 3
    AND pd.steamid IN (SELECT steamid FROM player_with_X_games)
    AND pg.appid IN (SELECT appid FROM appid_owner_count);

SELECT *FROM temp_result
  INTO OUTFILE 'player_games.csv'
  FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
  LINES TERMINATED BY '\n';
```

3.5.2 Data classes

recommender.py, the script which encapsulates every data class and every recommender system, is a very complex file of over 3,000 lines long (including comments and abstract classes), so we will not include it in this document nor the Annex, but it is available in the project's recommender systems GitHub repository [11]. We will, however, explain the most important parts of the code.

We will store all of our data as Pandas [35] DataFrames. Before anything else, we have to normalize the playtime data, since the playtime for each game for each user is vastly different. We assume the playtime is the 'interest' the user has in the game, so we will normalize it to a 0-1 scale, where 0 is no interest and 1 is maximum interest. We will use the formulas in equation 3.1 to normalize the playtime:

$$\text{Linear norm.: } \frac{x}{max(user)}, \text{Log norm.: } \frac{\log(x)}{\log(max(user))}, \text{Root norm.: } \frac{\sqrt{x}}{\sqrt{max(user)}} \quad (3.1)$$

The code allows for different approaches to the denominator, in our case in equation 3.1 we use the maximum playtime for each user, but we can also use the maximum playtime for each user or the sum of all playtimes for each user. As well, we will set all owned games with no playtime to 60 minutes, otherwise, games with 0 playtime would be considered the same as games that the user does not own, shown in Code A.9. The full script for normalization is split into different codeblocks, available in the annex (A.9, A.10, A.11, A.12, A.13, A.14).

Storing game attributes and player playtimes

All of our data is stored in various classes, one for each kind of data we want to use to recommend:

1. *PlayerGamesPlaytime*, which stores all data related to users, their games and their playtimes.
2. *GameTags*, which stores all data that relates games to their tags.
3. *GameGenres*, which stores all data that relates games to their genres.
4. *GameCategories*, which stores all data that relates games to their categories.
5. *GameDevelopers*, which stores which games were developed by which developers.
6. *GamePublishers*, which stores which games were published by which publishers.
7. *GameDetails*, which stores all details for each game.

All of these classes inherit [5] from *AbstractRecommenderData*, which defines the methods that all of these classes must implement, as well as some helper methods.

We implemented a Locality Sensitive Hashing index, as explained in section 2.1.4, to speed up the computation of the similarity between games and users, using the LSH Ensemble [22] from the library *datasketch* [20]. We implemented the LSH Ensemble for every type of data except for developers and publishers, as developer and publisher data is smaller than the rest of the data and for a recommender we only need to check if a game was developed or published by the game we are computing the similarity for.

In the case of *PlayerGamesPlaytime*, to enhance performance, and as mentioned in Section 2.1.5, we will only take into account the games the user has played beyond a certain percentage relative to

[5] A class that inherits from another one has all of their attributes/variables, methods/functions and must implement all abstract methods.

their most played game, that is, only those with a playtime above $max(playtimes) * t_{rel}$ for that user will be taken into consideration when MinHashing. In our case, due to time constraints, for the experiments we will only take into account the games the user has played more than 80% of their most played game, but the class allows for any percentage to be used. In the case of LSH Ensemble thresholds (as explained in section 2.1.5), we will use different values for each type of data in our experiments.

Another last notable optimization to mention is that most of our data for our experiments would be shared between different classes, thus, we implemented a Global Cache dictionary that stores the data that is shared between classes, so that we do not have to load the same data multiple times. Before doing this, if we wanted to load the data for different combinations of LSH Ensemble thresholds for all of the classes, as well as different thresholds for *PlayerGamesPlaytime* as mentioned previously, the program would run out of RAM and start filling up the storage through paging files in Windows. Using this technique we managed to use less than 24GBs of RAM for every experiment, whereas previously Windows would not show the exact memory usage but it would run out of storage after filling up 32GBs of RAM.

For *GameTags*, as mentioned in section 2.1.5, we have a different kind of optimization: we can just take tags above a "weight threshold" wt and, for these tags, have a simple dictionary **tag** $\rightarrow$ **item**, and then simply compute the similarities between our item i_o and every other item returned by this dictionary.

3.6 Recommendation algorithms

We implemented various similarity classes which inherit from *AbstractGameSimilarity* and *RawUser-Similarity*, which both inherit from *AbstractSimilarity*. To this avail, we implemented 3 user similarity classes ('Raw' (equation 3.2), 'Cosine' (equation 3.3) and 'Pearson' (equation 3.4)), 3 game tag similarity classes ('Raw', 'Cosine' and 'Pearson'), one similarity class for Game Details, which uses various comparison methods, and one game similarity class for each of the other types of game attributes using Jaccard's similarity (genres, categories, developers and publishers).

Let I_i be an item, and $R_{u,i}$ be the rating of user u for item i. Then, the similarity between two users u and v is defined as follows:

$$\text{Raw:} \sum_{i \in I_u \cap I_v} R_{u,i} \cdot R_{v,i} \tag{3.2}$$

$$\text{Cosine:} \frac{\sum_{i \in I_u \cap I_v} R_{u,i} \cdot R_{v,i}}{\sqrt{\sum_{i \in I_u \cap I_v} R_{u,i}^2} \cdot \sqrt{\sum_{i \in I_u \cap I_v} R_{v,i}^2}} \tag{3.3}$$

$$\text{Pearson: } \frac{\sum_{i\in I_u\cap I_v}(R_{u,i}-\overline{R_u})\cdot(R_{v,i}-\overline{R_v})}{\sqrt{\sum_{i\in I_u\cap I_v}(R_{u,i}-\overline{R_u})^2}\cdot\sqrt{\sum_{i\in I_u\cap I_v}(R_{v,i}-\overline{R_v})^2}} \tag{3.4}$$

The same can be applied to games, but instead of a user rating an item, we have an item being tagged with a tag, which has a weight, which can be interpreted as a rating.

Code 3.7: Code that parallelizes the calculation of user similarities.

```python
def get_similar_users(self, steamid: int, n: int = 10) -> DataFrame:
    """Gets n top similar users to a user. Specially useful for collaborative filtering.
    [...] (comments removed to save space) """
    rough_similar_users = self.pgdata.get_lsh_similar_users(steamid)
    self._own_games_played = self.pgdata.get_user_games(steamid)
    logging.info(f"Finding {n} similar users to {steamid}. Please wait...")
    priority_queue = PriorityQueue(n + 1)
    if self.parallelize:
        futures = []
        max_workers = GLOBAL_THREAD_EXECUTOR._max_workers
        rough_similar_users = list(rough_similar_users)
        count_per_worker = len(rough_similar_users) / max_workers / 4
        for batch in self.chunks(rough_similar_users, math.ceil(count_per_worker)):
            futures.append(GLOBAL_THREAD_EXECUTOR.submit(self.similarity_batch, steamid,
                batch))
        for future in cf.as_completed(futures):
            similar_users = future.result()
            for similarity in similar_users:
                priority_queue.put(similarity)
                if priority_queue.qsize() > n:
                    _ = priority_queue.get()
    else:
        for similar_user in rough_similar_users:
            if similar_user == steamid:
                continue
            similarity = self.similarity(steamid, similar_user)
            priority_queue.put((similarity, similar_user))
            if priority_queue.qsize() > n:
                _ = priority_queue.get()
    logging.info(f"Finished finding relevant similar users.")
    self._own_games_played = None
    return self.player_similarities_from_priority_queue(priority_queue)
```

The methods which calculate similarities for users (but not for games) are parallelized to speed up the computation. This is done by splitting the set of users into n subsets, and then calculating the similarity between each user in a subset and all the users in the other subsets. This is done in parallel for each subset, and the results are then merged. Our *RawUserSimilarity* class handles everything, as shown in Code 3.7.

RawUserSimilarity implements every method required to calculate the similarity between users in such a way *CosineUserSimilarity* and *PearsonUserSimilarity* can inherit from it, and only require the implementation of the *similarity(user, other)* method, which returns the similarity between the user and another one. As well, *AbstractGameSimilarity* implements every method required to calculate the similarity between games in such a way all of our game similarity classes can inherit from it. This is done to avoid code duplication, and every similarity class encapsulates their respective data classes (e.g. *GameTagSimilarity* encapsulates *GameTags*), which will allow us to only require the similarity class to generate recommendations. We also implemented a technique similar to IDF (Inverse Document Frequency) to weight the tags based on how many games have that tag, under the assumption that we would avoid giving tags that are too common (e.g. 'Indie') too much importance.

In the case of the Game Details similarity class, we implemented various methods to compare the game details, as shown in Code A.15. We compare various attributes of the game using different methods, and then return the weighted sum of all the comparisons.

We also implemented four recommender systems, which inherit from *AbstractRecommenderSystem*: *RandomRecommenderSystem*, our baseline, which randomly picks games, *PlaytimeBasedRecommenderSystem*, which recommends games based on the playtime of the user relative to other people's playtimes (our Collaborative Filtering recommender), *RatingBasedRecommenderSystem*, which merely scores games based on their ratings (useful for our *HybridRecommenderSystem*), *ContentBasedRecommenderSystem*, which recommends games based on the similarity of the game to the games the user has played (our Content-Based recommender), and *HybridRecommenderSystem*, which combines the previous recommender systems (our Hybrid recommender).

PlaytimeBasedRecommenderSystem and *ContentBasedRecommenderSystem* use the LSH Ensemble through the similarity classes to speed up the computation of the similarity between games and users, as explained previously. These both are implemented in such a way that they can be used with any similarity class without having to implement any specific code for that similarity class. The *PlaytimeBasedRecommenderSystem* only requires a *PlayerGamesPlaytime* instance and a *UserSimilarity* instance, and the *ContentBasedRecommenderSystem* only requires a *PlayerGamesPlaytime* instance and a *AbstractGameSimilarity* class.

The *PlaytimeBasedRecommenderSystem* uses an implementation of kNN to find the k most similar users to the user we want to recommend games to, and then recommends the games that the k most similar users have played, but the user we want to recommend games to has not played, sorted by score.

ContentBasedRecommenderSystem, however, generates a 'weight map' taking into account the game attributes (tags, genres, developer, etc.) of the games the user has played, adding them to the weight map with a weight equal to the normalized playtime multiplied by the attribute's weight (in the case of genres, categories, developers and publishers, the weight is 1) for every attribute of the game (e.g, if a user has played three games for 1h with tags [Action, RPG], [Action, Adventure] and [Action, Adventure] respectively, assuming the tags all have weight 1.0 and 0.5 respectively, the user map would then be Action $\rightarrow$ 3, RPG $\rightarrow$ 1, Adventure $\rightarrow$ 2.5). Then, the recommender system finds the most similar games to the user map through the LSH Ensemble and recommends the games the user has not played, sorted by score. The code for the item weight map calculations can be seen in Code 3.8.

Code 3.8: Code that calculates the item weight map for an user.

```
def get_item_weights(self, steamid: int, user_games: DataFrame) -> Dict[int, float]:
    """Gets or generates the item weights for a user. [...]"""
    if steamid in self.user_item_weights: return self.user_item_weights[steamid]
    item_weight = {}
    for _, row in user_games.iterrows():
        appid = row["appid"]
        pseudorating = row["playtime_forever"]
        for itemid, weight in self.get_game_items(appid):
            if not itemid in item_weight:
                item_weight[itemid] = 0
            item_weight[itemid] += self.weight_function(weight, pseudorating)

    item_weight = sorted(item_weight.items(), key=lambda x: x[1], reverse=True)
    if self.item_weight_max_length > 1:
        item_weight = item_weight[:self.item_weight_max_length]

    item_weight = dict(item_weight)
    self.user_item_weights[steamid] = item_weight
    return item_weight
```

4 Results

4.1 Approach and caveats

To measure our performance, precision and recall, we first split part of the dataset into a training set and a test set, by picking 1,000 users at random and splitting their game owned list randomly into 80% of the playtime for the training set and 20% for the test set. The training set is used by the recommender to generate the recommendations, and the test set is not used by the recommenders, but is instead used to evaluate all the methods, to compare against the list of ‘relevant items’ we are trying to predict, as mentioned in Section 4.2.

Initially, our experiment script instantiated every similarity and recommender class in parallel, with different parameters, to test out the different combinations. However, we quickly saw RAM go above 32GB quick and had to create a “global cache” to share all the data between classes, as mentioned in Section 3.5.2. With this fix, it was enough to run the experiments sequentially, as the bottleneck was the CPU, not the RAM.

Another caveat was time constraints and, as such, we could not run every experiment we wanted to run with different configurations for our recommender systems for these 1,000 users. Instead, we created a smaller subset of 100 users and ran an exhaustive experiment on it, and then picked the best combinations for our 1,000 users experiments. Otherwise, our estimations showed that we would have had to wait 2-3 months to run all the combinations, which was not feasible. Still, the total runtime for all recommenders was at first over 11 days, 14 hours and 30 minutes, without taking into account some results that we had to delete which took over 15 days to run in total. Thus, we believe our results are a good approximation of what we would have gotten if we have had the time to run all the combinations, as we focused on the most important ones after running an exhaustive experiment over 100 hand-picked users. The experiments were run on a computer with a Ryzen 5 5600X CPU and 32GB of RAM, and the Collaborative-Filtering recommenders had to be run in parallel, which means the CPU cache usage was not optimized, and the runtimes may vary.

4.2 Precision, recall and runtime results

Let us start our experiments with two simple strategies or baselines, which are presented in Table 4.1. The ‘Random’ one simply returns random games, and the ‘Top rated’ one returns the games with the highest average rating. These results are a good starting point to compare our results against, and to see if our recommender systems are actually doing better than a random result or a simple method based on top rated items.

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
Random	0.0046	0.0041	0.0042	0.0014	0.0025	0.0048	4.71
Top rated	0.0016	0.0010	0.0073	0.0005	0.0006	0.0087	705.54

Table 4.1: Baseline results. P@N stands for Precision at N, and R@N stands for Recall at N. Time (s) denotes the time in seconds.

As we can see from the results in this table, our ‘Random’ recommender system is better than the ‘Top rated’ one, which is suprising, since we would expect Steam’s user base to pick games that are good, and thus the ‘Top rated’ recommender system to be better than a random one. However, this is not the case, and we can see that the ‘Top rated’ recommender system is actually worse than a random one. Therefore, our baseline are the results of the ‘Random’ recommender system.

4.2.1 Content-Based Filtering

Our subset of games, obtained through the query in Code 3.5, is composed of of 4,822 games and 81,415 users: thus, we expected the experiments for CBF to take less than the ones for CF, since CBF only compares against other games and not users, so we started with those first, and thus made more experiments with them. The full results for CBF are separated in various tables in Tables B.3 to B.6. We show an extract of the best results in Table 4.2. The best results are highlighted in bold, the best of each recommender system is underlined. t is the threshold for the LSH Ensemble index, w_{idf} is the IDF weight for tags and wt denotes our tag $\rightarrow$ game dictionary approach with weights (no LSH Ensemble, only for Game Tags).

Categories refer to our different categories-based recommender systems, which compare categories to the user’s preferred categories (just like Code 3.8, detailed in Section 3.6, we will refer to “preferred attributes” as their weight map, which scores the user’s interest in each attribute based on his playtime from games with those attributes). Genres refer to the game genres, which are compared to the user’s preferred genres. Tags refer to the game tags, which are compared to the user’s preferred tags, and take a second parameter: the weight of the IDF. This weight is used to multiply the IDF of each tag and add it to the final score of the game, which, as a reminder from Section 2.1.2, penalizes tags that are common among all games. We have three aproaches for tags: ‘Raw’, ‘Cosine’ and

'Pearson', which, as mentioned in Section 3.6, score their similarities by just multiplying weights, by multiplying weights and dividing them by their norms, and by using the Pearson correlation coefficient, respectively. We also have 'Developers', which compares the developers of the games the user owns to the developers of all games, and sorts them by how many games the user has for each developer, and 'Publishers', which is the same as 'Developers', but for publishers instead. Finally, we have a recommendation system that uses the game details (price, release date, name, etc.) to compare the games to the user's preferred attributes, and sorts them by their score.

Categories	**P@5**	**P@10**	**P@20**	**R@5**	**R@10**	**R@20**	**Time (s)**
t=0.42	0.0370	0.0304	0.0251	0.0114	0.0185	0.0302	1869.48
t=0.55	0.0242	0.0150	0.0082	0.0076	0.0094	0.0103	**63.37**
Genres							
t=0.30	0.0030	0.0035	0.0044	0.0007	0.0018	0.0049	4119.41
t=0.80	0.0016	0.0022	0.0022	0.0003	0.0013	0.0027	235.48
Raw Game Tags							
t=0.30, w_{idf}=0.60	0.0472	0.0382	0.0318	0.0137	0.0217	0.0364	9379.58
t=0.42, w_{idf}=0.60	0.0476	0.0382	0.0317	0.0138	0.0217	0.0361	9571.94
Cosine Game Tags							
t=0.42, w_{idf}=0.60	0.0376	0.0316	0.0284	0.0105	0.0178	0.0320	1626.74
Pearson Game Tags							
t=0.55, w_{idf}=0.60	0.0274	0.0240	0.0210	0.0076	0.0134	0.0235	561.59
Others							
Details	0.0440	0.0334	0.0257	0.0134	0.0199	0.0310	21246.19
Developers	**0.0606**	**0.0554**	**0.0467**	**0.0187**	**0.0343**	**0.0571**	1397.43
Publishers	0.0570	0.0502	0.0402	0.0168	0.0301	0.0477	3021.03

Table 4.2: CBF top results, separated by approach. P@N stands for Precision at N, and R@N stands for Recall at N. Time (s) denotes the time in seconds.

Unsurprisingly, we can see that the genres recommender system performs the worst (and has worser results than our baseline), as genres on Steam are very few and broad, and thus not very useful for recommendations compared to tags, which are more specific. In terms of categories, however, we see a big improvement over the genres recommender system, and even outperforms our Pearson and cosine game tag recommender systems. We can observe our Pearson and Cosine game tag implementations are vastly outperformed by every other recommender system except for the genres recommender system: this is because a 'Cosine' approach (mentioned in Equation 3.3) to similarity regarding tags penalizes games with 20 tags (the maximum amount of tags for Steam), and using Pearson's similarity (Equation 3.4) simply penalizes the bottom half of tags. The runtimes for these Cosine and Pearson are superior because, after running the 'Raw Game Tags' approaches first, we optimized the code further to speed up running our experiments, but we have no doubts the 'Raw Game

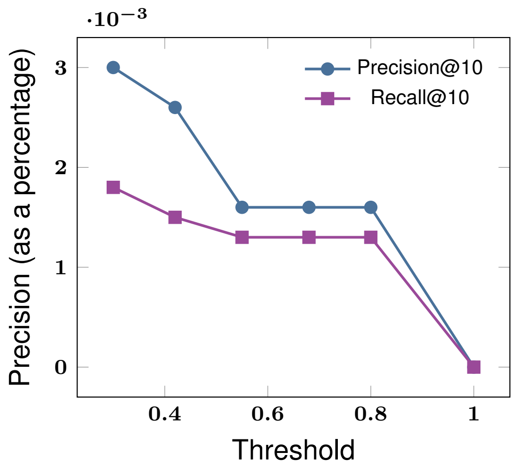
Tags', as of today, run faster than the 'Cosine Game Tags' and 'Pearson Game Tags' approaches. We can also see that the 'Details' recommender system, which compares the game details to the user's preferred attributes, is really close to the 'Raw Game Tags' recommender system but still worse, but it took twice as long to process. We have not shown different IDF configurations in Table 4.2, but as seen in any table from tables B.3 to B.5, the best configuration is w_{idf} = 0.6, and we might have been able to get better results with higher IDF weights, but we leave that analysis as future work. We can see our best performing recommender system is 'Developers', with a Precision@5 of 0.0606 and a Precision@10 of 0.0554, Recall@5 of 0.0187 and Recall@10 of 0.0343. This approach retrieves the games developed by the same developers of the games the user owns, sorted by how many games the user has for that developer. We can also observe minor differences below 0.5 threshold for the LSH Ensemble index, which includes more games to compare our preferred tags to in tables B.3 to B.5

We show graphs that compare the Precision@5 and Recall@10 for genres, categories and game tags approaches in Figures 4.1 to 4.3 when modifying LSH Ensemble indices thresholds, with their respective time graphs. We also show all IDF weight combinations for the game tags approaches. As we can observe, there is a linear correlation between thresholds and Precision and Recall, and the time graphs, when represented logarithmically, are shown to also be linear and closely resembles the pattern seen in the Precision and Recall graphs. We can also see that the variations in Precision and Recall for game tag approaches are not as big as the ones for genres and categories, but we can see the time taken when increasing the threshold for the LSH Ensemble index is also linear when representing it logarithmically. We can see the precision and recall for the 'Raw' and 'Cosine' approaches for game tags is lowered with higher thresholds, but the 'Pearson' approach sees its precision and recall improved when increasing the threshold.

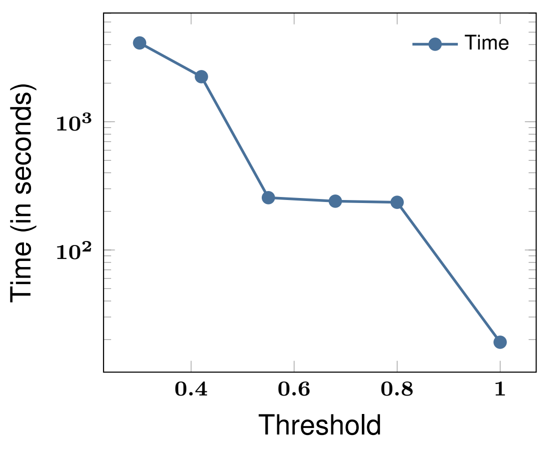
4.2.2 Collaborative Filtering

When it comes to CF, as mentioned earlier, we had to run experiments over less users than originally planned. We found that threshold t = 0.8 for the LSH Ensemble and relevant threshold t_{rel} = 0.6 (mentioned in Section 2.1.5 and Section 3.5), where only games with normalized playtime above $max(playtimes) * t_{rel}$ are taken into consideration when indexing, are both our best and fastest results for a list of 100 players (instead of 1000) to predict for, as shown in Table B.2. We see the results for each user similarity method followed by the normalization approach in parentheses in Table 4.3.

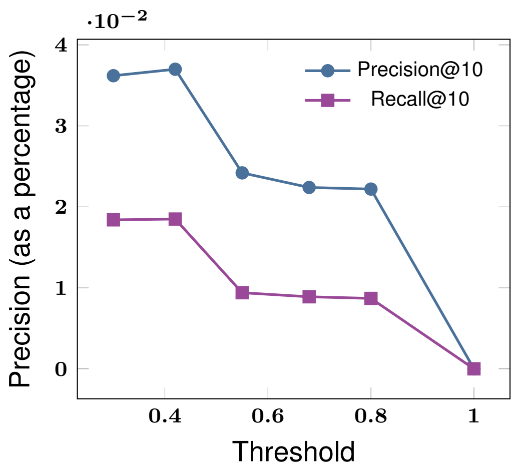
We can see our **Pearson User Similarity** (defined in Equation 3.4) implementation outperforms the second best approach for every normalization method by approximately 3 times at precision and Recall@5, and over 3 times and even 4 times at Recall@20. We can see the best normalization method for Precision@5 and Recall@5 is the logarithm method, which uplifts games with lower playtime: counter-intuitively, however, we found the square root method, which punishes low playtimes, provides the best results at 10 and 20 in both metrics, but only for our Pearson approach. We can also see that the **Pear-**



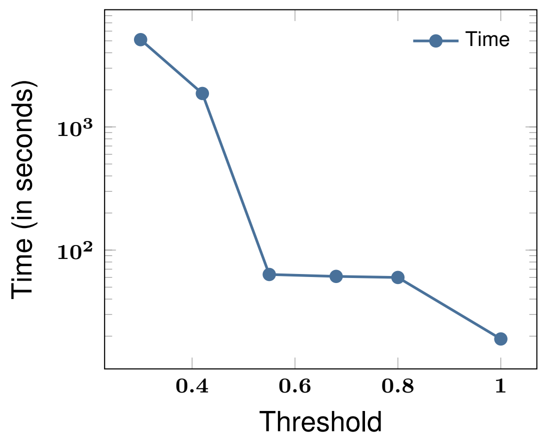
(a) Genres (Precision@5 and Recall@10 vs LSH index threshold)



(b) Genres (Time vs LSH index threshold)

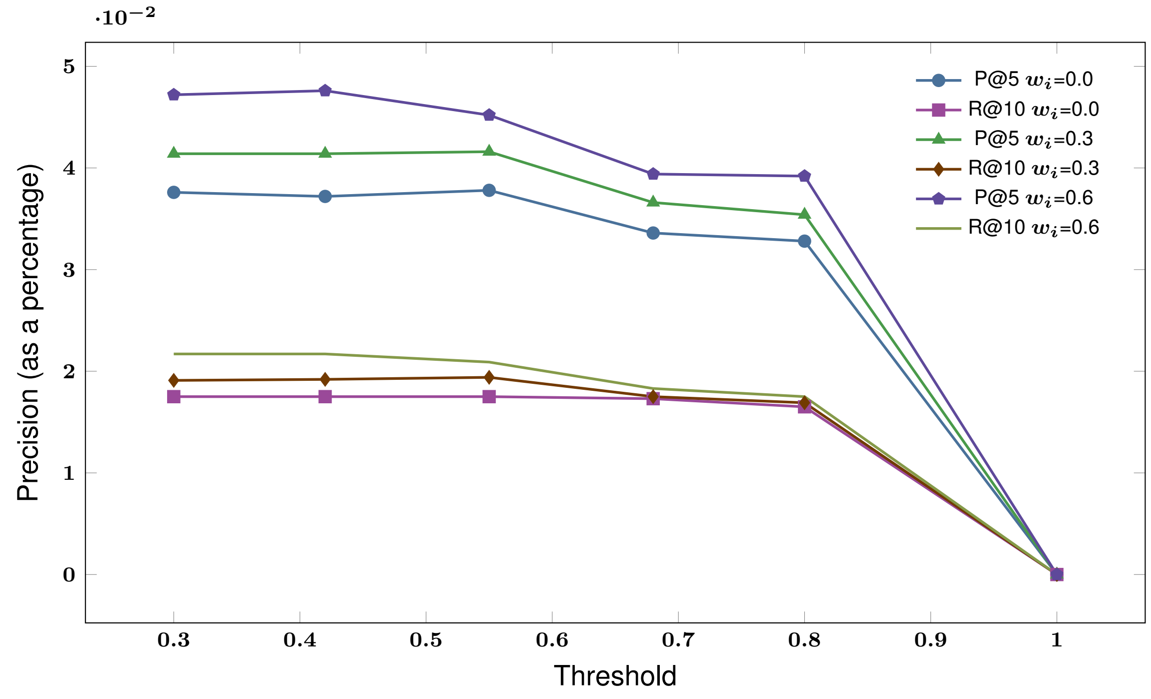


(c) Categories (Precision@5 and Recall@10 vs LSH index threshold)

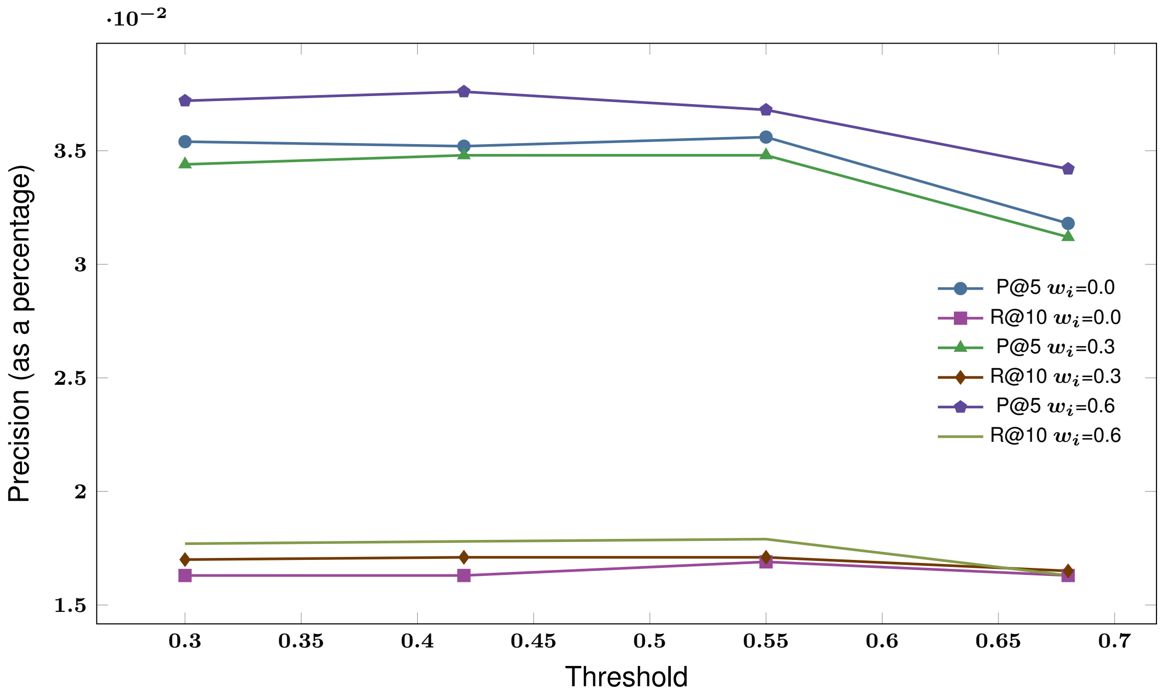


(d) Categories (Time vs LSH index threshold)

Figure 4.1: Comparison of our full results in Table B.6, comparing Precision@5 and Recall@10, and a second graph comparing times against the threshold for the LSH Ensemble index, for genres, categories and game tags approaches.

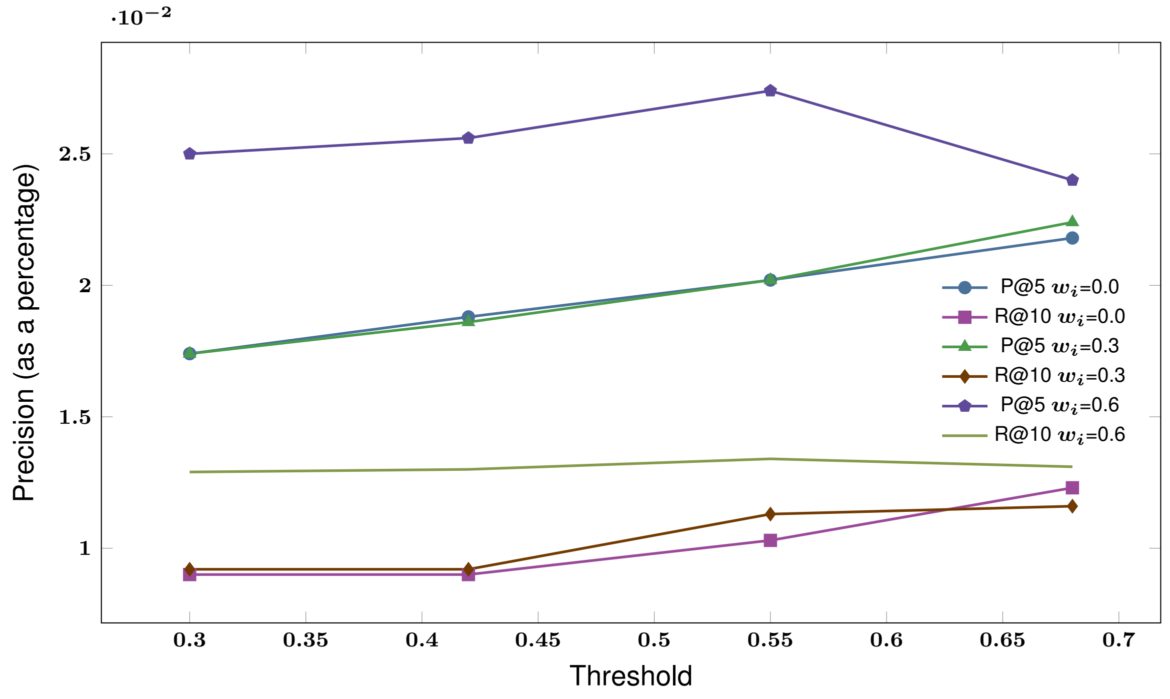


(a) Comparison of results for different parameters for Raw Game Tags (Precision@5 (P@5) and Recall@10 (R@10) using different IDF weights, represented as w_{idf} vs LSH index threshold)

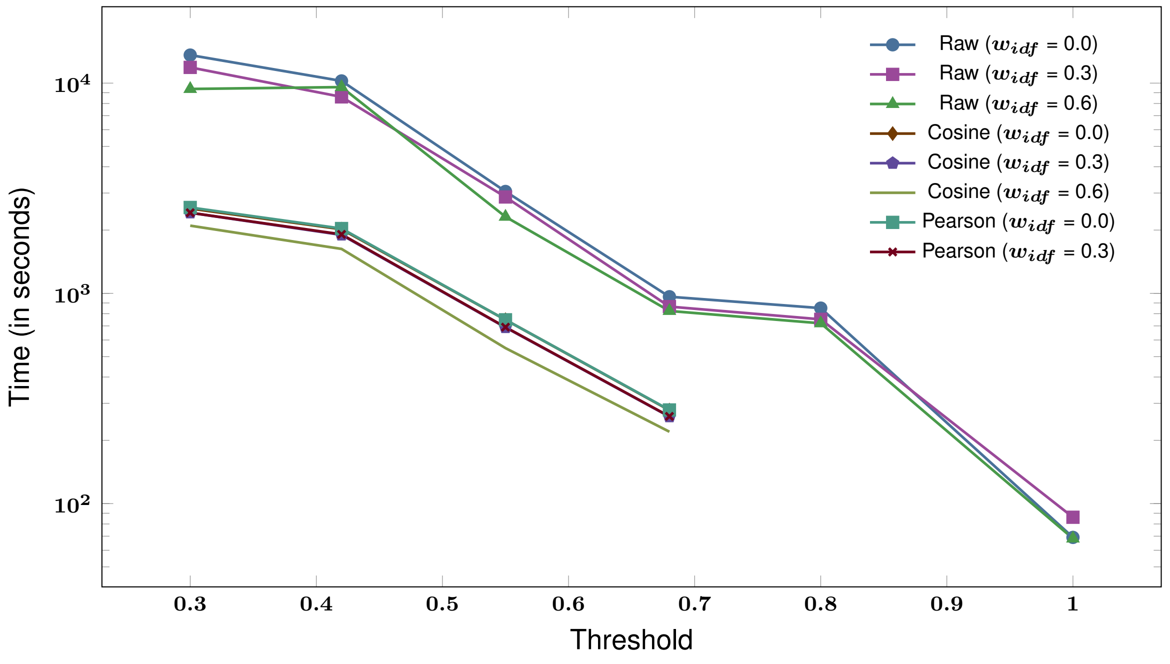


(b) Comparison of results for different parameters for Cosine Game Tags (Precision@5 (P@5) and Recall@10 (R@10) using different IDF weights, represented as w_i vs LSH index threshold)

Figure 4.2: Comparison of the raw game tags and cosine game tags results in Tables B.3 and B.4, comparing Precision@5 and Recall@10 against LSH Ensemble thresholds.



(a) Comparison of results for different parameters for Pearson Game Tags (Precision@5 (P@5) and Recall@10 (R@10) using different IDF weights, represented as w_i vs LSH index threshold)



(b) Time to execute each Game Tags approach vs LSH index threshold. Note: we did not run thresholds beyond 0.68 for Cosine and Pearson approaches.

Figure 4.3: Comparison of the Pearson game tags results in Table B.5, comparing Precision@5 and Recall@10 against LSH Ensemble thresholds, and a second graph comparing times in Tables B.3 to B.5 against the threshold for the LSH Ensemble index, for all game tags approaches.

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
Raw (Linear)	0.0530	0.0401	0.0314	0.0146	0.0220	0.0348	65261.46
Raw (Log)	0.0692	0.0528	0.0401	0.0191	0.0289	0.0438	65443.07
Raw (Square Root)	0.0594	0.0479	0.0364	0.0164	0.0261	0.0399	65674.90
Cosine (Linear)	0.0540	0.0400	0.0312	0.0149	0.0218	0.0346	61063.86
Cosine (Log)	0.0770	0.0575	0.0435	0.0214	0.0312	0.0480	61105.11
Cosine (Square Root)	0.0596	0.0483	0.0369	0.0165	0.0264	0.0407	**61033.86**
Pearson (Linear)	0.1954	0.1672	0.1281	0.0581	0.0983	0.1512	94042.03
Pearson (Log)	**0.2136**	0.1753	0.1356	**0.0632**	0.1030	0.1587	95043.25
Pearson (Square Root)	0.2056	**0.1787**	**0.1389**	0.0605	**0.1048**	**0.1625**	95983.37

Table 4.3: Precision and recall results for CF. Parameters used: t_{rel} = 0.6, t = 0.8. User similarity followed by the normalization approach in parentheses.

son User Similarity is the slowest approach, taking aproximately 1.44 times longer than the second slowest approach, which is the raw similarity approach.

4.2.3 Analysis of the results

Our results show a Collaborative-Filtering approach is more precise and yields better recommendations than a Content-Based approach, but it is also slower, with the overall best approach taking 95,983.37 seconds (approximately 1 day, 2 hours and 40 minutes to process 1000 users) versus the best approach, recommending games from the users' top developers, taking 1,397.43 seconds (approximately 23 minutes) to complete. Nonetheless, our best Collaborative-Filtering approach sees up to 3 times improvements in both precision and recall. Put into perspective, however, the PC market has many bundles [1] and sales from the same developers, and therefore we may have split users who had many bundles in their accounts, thus skewing the results to favour recommendations for people who usually buy games in bundles. This is a limitation of our dataset and using a mean average of the metrics to represent the overall performance of the algorithm may not be the best approach, since we checked and the top precision and recall results for the 'Developer' recommender system were above 70% while half of the results were below 1%. As well, all of our game tag comparison methods use the linear playtime normalizer due to time constraints, but we do not expect these parameters to make much of a difference as shown by the playtime based recommender systems.

[1] In the PC market it is common to see bundles of developers who put their entire franchises for a low price. For example: Valve Complete Pack [44] offers all of their 22 games, as such if we split 10 of their games into the test set, we might have had the other 12 games setting Valve's score too high and recommending the other 10 games, thus having a really high recall.

5 Conclusion and future work

5.1 Conclusion

In this thesis, we have studied how to implement a data crawler from scratch to gather a large dataset from Steam, the most popular videogame store on the market, process this data with different approaches for normalization, and implemented 3 variations of Collaborative-Filtering recommender systems as well as 8 different Content-Based recommender systems.

We also show in our results that our IDF technique, mentioned in 3.6, helps boost uncommon tags to personalize recommendations further by improving the results for the game tags approach up to 27.96% for precision and up to 27.77% for recall in Table B.5. As well, in our full results (Tables B.3 to B.5), we show how using different thresholds for the LSH Ensemble filters out undesired games to compare against, and that the best results are obtained when using a threshold of 0.42.

As for our CF results, we would have loved to test different LSH Ensemble thresholds and relevant game thresholds, but due to time constraints we were unable to do so. Nonetheless, we show various CF approaches and have found that using Pearson's similarity is the best approach to collaborative filtering by a noticeable enough margin to warrant a 44% increase on runtime compared to every other single approach. Thus, a re-implementation of our CF approach using Pearson's similarity in a language faster than Python could be desirable.

5.2 Future work

We would like to propose the following as future work regarding this thesis and recommender systems in general:

- **A more performant language**: Python shows its limitations when it comes to performance, and while we have tried to use libraries that are performant, such as *datasketch* and *pandas*, we have found that the performance is still lacking, and we would suggest trying to use a more performant language, such as C, C++ or Go, to see if the theoretical performance uplifts are worth a re-implementation.

- **More data and comparisons**: We have only used a small subset of the data available due to time constraints, and we only found out about the dataset from the current state of the art recommender system mentioned in Section 2.2 after we had already started working on the project. Plus, an objective of this thesis was to crawl our own information from publicly available APIs. As such, we would like to see how our results compare to the current state of the art, including runtime, power consumption and code readability, and if we can improve upon them. As well, we would have liked using more combinations to our parameters, but due to time constraints we were unable to do so, as it would have taken months.

- **Same optimizations into different domains**: We have used a few optimizations that are not specific to recommender systems, mainly the LSH Ensemble library from datasketch [22], and we would like to see how it performs in other domains.

- **Real-world deployment and viability**: We would like to see how our recommender system (after some tweaks) performs compared to current state-of-the-art recommender systems, and see which approach could be viable for a userbase of over a 100M users, and hope the methods and code presented in our GitHub repo [11] can be used as a starting point for future work.

- **Using our recommender systems for performant comparisons between ways to represent the interest of the users**: In this work, we used playtimes as a way to represent interest of users, but we would like future work to use our recommender systems when using other ways to gather interest of users, such as wishlists, ratings, reviews, cross-storefront playtimes, etc. and using achievements, average playtimes for each game, etc. to get different information (e.g., the user beat the game because they have a certain achievement, or the game is endless and the user has played it for a long time relative to other users, instead of plainly using playtimes).

Bibliography

[1] Entertainment Retailers Association (United Kingdom), “Era yearbook.” `https://eraltd.org/media/72514/2022-era-yearbook_interactive.pdf`, 2022.

[2] SteamSpy, “Monthly summaries.” `https://steamspy.com/year/`, 2023.

[3] Steam, “Discovery Queue.” `https://store.steampowered.com/about/newstore`, 2023.

[4] Valve Software, “Steam - 2021 Year in Review.” `https://store.steampowered.com/news/group/4145017/view/3133946090937137590`, 2021.

[5] Valve Software, “Steam - 2022 Year in Review.” `https://steamcommunity.com/groups/steamworks/announcements/detail/3677786186779762808`, 2022.

[6] Epic Games, “Epic Games Store 2022 Year in Review.” `https://store.epicgames.com/en-US/news/epic-games-store-2022-year-in-review`, 2022.

[7] Sony Interactive Entertainment, “PlayStation Store.” `https://store.playstation.com/`, 2004.

[8] Sony Interactive Entertainment, “FY22 Q4 Suplemental Information.” `https://www.sony.com/en/SonyInfo/IR/library/presen/er/pdf/22q4_supplement.pdf`, 2022.

[9] Microsoft, “Microsoft Fiscal Year 2023 Second Quarter Earnings Conference Call.” `https://www.microsoft.com/en-us/Investor/events/FY-2023/earnings-fy-2023-q2.aspx`, 2023.

[10] Nintendo Co., Ltd., “Financial results explanatory material fiscal year ended march 2023.” `https://www.nintendo.co.jp/ir/pdf/2023/230509_3e.pdf`, 2023.

[11] J. González, “Steam recommender systems in pure Python, using Pandas and datasketch.” `https://github.com/RogerFK/steam-recsys/`, 2023.

[12] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, eds., *Introduction to Recommender Systems Handbook*, pp. 1–35. Boston, MA: Springer US, 2011.

[13] P. Resnick and H. R. Varian, “Recommender systems,” *Commun. ACM*, vol. 40, p. 56–58, mar 1997.

[14] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, “Using collaborative filtering to weave an information tapestry,” *Commun. ACM*, vol. 35, p. 61–70, dec 1992.

[15] S. Raschka, “L02: Intro to supervised learning: Knn.” `https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/02_knn_notes.pdf`, 2018.

[16] R. Burke, “Hybrid recommender systems: Survey and experiments,” *User Modeling and User-Adapted Interaction*, vol. 12, 11 2002.

[17] A. Z. Broder, “Identifying and filtering near-duplicate documents,” in *Combinatorial Pattern Matching* (R. Giancarlo and D. Sankoff, eds.), (Berlin, Heidelberg), pp. 1–10, Springer Berlin Heidelberg,

2000.

[18] P. Jaccard, “The distribution of the flora in the alpine zone.1,” *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[19] J. Leskovec, A. Rajaraman, and J. Ullman, “Chapter 3 - Mining of Massive Datasets.” http://infolab.stanford.edu/~ullman/mmds/ch3n.pdf, 2012.

[20] E. Zhu, “datasketch: Big Data Looks Small.” https://ekzhu.com/datasketch/index.html, 2023.

[21] E. Zhu, “MinHash LSH - datasketch documentation.” https://ekzhu.com/datasketch/lsh.html, 2023.

[22] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller, “Lsh ensemble: Internet-scale domain search,” *Proc. VLDB Endow.*, vol. 9, p. 1185–1196, aug 2016.

[23] M. Bawa, T. Condie, and P. Ganesan, “Lsh forest: Self-tuning indexes for similarity search,” in *Proceedings of the 14th International Conference on World Wide Web*, WWW ’05, (New York, NY, USA), p. 651–660, Association for Computing Machinery, 2005.

[24] E. Zhu, “Weighted MinHash.” https://ekzhu.com/datasketch/weightedminhash.html, 2022.

[25] M. O’Neill, E. Vaziripour, J. Wu, and D. Zappala, “Condensing steam: Distilling the diversity of gamer behavior,” in *Proceedings of the 2016 Internet Measurement Conference*, IMC ’16, (New York, NY, USA), p. 81–95, Association for Computing Machinery, 2016.

[26] L. Yang, Z. Liu, Y. Wang, C. Wang, Z. Fan, and P. S. Yu, “Large-scale personalized video game recommendation via social-aware contextualized graph neural network,” in *WWW ’22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, pp. 3376–3386, ACM, 2022.

[27] Valve Software, “Introducing The Steam Interactive Recommender.” https://steamcommunity.com/games/593110/announcements/detail/1716373422378712841, 2020.

[28] GOG sp. z o.o., “GOG.com.” https://www.gog.com/en/about_gog, 2023.

[29] benny_a, “Step aside Amazon, PlayStation now has a recommendation system.” https://www.neogaf.com/threads/step-aside-amazon-playstation-now-has-a-recommendation-system.925528/, 2014.

[30] R. Howsden, “Building a Playstation Network Recommendation System from Scratch.” https://deceitfuldata.medium.com/building-a-playstation-network-recommendation-system-from-scratch-6e0927e4088a, 2022.

[31] Valve Software, “Intro - steam 1.4.4 - steam 1.4.4 documentation.” https://steam.readthedocs.io/en/latest/, 2023.

[32] D. Salazar, “python-steam-api - PyPI.” https://pypi.org/project/

python-steam-api/, 2023.

[33] Valve Software, “Steamworks Web API Reference.” https://partner.steamgames.com/doc/webapi, 2022.

[34] woctezuma, “steamreviews - PyPI.” https://pypi.org/project/steamreviews/, 2021.

[35] PyData, “pandas - Python Data Analysis Library.” https://pandas.pydata.org/, 2023.

[36] Python Software Foundation, “Python 3.11.0.” https://www.python.org/downloads/release/python-3110/, 2022.

[37] Anaconda Inc., “Anaconda - The World’s Most Popular Data Science Platform.” https://www.anaconda.com/, 2023.

[38] Oracle, “My SQL.” https://www.mysql.com/, 2023.

[39] Z. et al., “Scrapy | A Fast and Powerful Scraping and Web Crawling Framework.” https://scrapy.org/, 2023.

[40] Python Software Foundation, “concurrent.futures — Launching parallel tasks.” https://docs.python.org/3/library/concurrent.futures.html, 2023.

[41] Python Software Foundation, “pickle — Python object serialization.” https://docs.python.org/3/library/pickle.html, 2023.

[42] The Matplotlib development team, “Matplotlib — Visualization with Python.” https://matplotlib.org/, 2023.

[43] Valve Software, “Steam Web API Terms of Use.” https://steamcommunity.com/dev/apiterms, 2023.

[44] Valve, “Valve Complete Pack.” https://store.steampowered.com/bundle/232/Valve_Complete_Pack/, 2012.

Appendices

A

Annex I: Code

Code A.1: Full script that crawls users. It uses certain methods from database.py and config.py. This portion goes from the start up to the end of the loop where we retry in case Steam is currently down, or we are being rate limited and our rate limiter is not properly accounting for rate limiting. Continued in Code A.2

```python
num_processed_players = 0

def crawl_player_data(query_count=0, reviews=False, only_games=True, verbose=False):
    global num_processed_players
    unprocessed_players = get_100_unprocessed_players()
    while len(unprocessed_players) > 0:
        start_time = time.time()
        steam_ids = [str(player[0]) for player in unprocessed_players]
        while True:
            try:
                query_count = check_rate_limit(query_count)
                response = steam.users.get_user_details(",".join(steam_ids), single=False)
                if "players" in response:
                    break
                else:
                    print("Error while requesting user details, no players. Waiting 10 seconds: ")
                    print(response)
                    time.sleep(10)
            except Exception as e:
                print("Exception while requesting user details. Waiting 10 seconds: ")
                print(e)
                time.sleep(10)
```

Code A.2: Part two of the script that crawls users. It uses certain methods from database.py. The second part of the function, where we attempt to gather basic user info, and handles all of the code to fetch users' game lists. Continued in Code A.3

```
        players = response["players"]
        for player in players:
            steamid = player["steamid"]
            cvs = player["communityvisibilitystate"]
            num_processed_players += 1
            if cvs < 3:
                if verbose:
                    print("Private/Friends␣only␣profile,␣setting␣basic␣player␣profile...")
                process_steam_user(steamid, player["personaname"], 0, 0)
                continue

            commentpermission = player["commentpermission"] if "commentpermission" in player else None
            primaryclanid = player["primaryclanid"] if "primaryclanid" in player else None
            timecreated = player["timecreated"] if "timecreated" in player else None
            loccountrycode = player["loccountrycode"] if "loccountrycode" in player else None
            locstatecode = player["locstatecode"] if "locstatecode" in player else None
            loccityid = player["loccityid"] if "loccityid" in player else None
            while True:
                try:
                    query_count = check_rate_limit(query_count)
                    resp = requests.request(
                        "get",
                        "https://api.steampowered.com/IPlayerService/GetOwnedGames/v1/",
                        params={"steamid": steamid,
                                "include_appinfo": False,
                                "include_played_free_games": True,
                                "key": KEY},
                    )
                    if resp.status_code == 200: # I didn't trust the API to return stuff properly when rate limited
                        owned_games = json.loads(resp.text)["response"]
                        break
                    else:
                        print("Error␣while␣requesting␣owned␣games.␣Waiting␣3␣seconds:␣")
                        print(resp.text)
                        time.sleep(3)
                except Exception as e:
                    print("Exception␣while␣requesting␣user␣details.␣Waiting␣10␣seconds:␣")
                    print(e)
                    time.sleep(10)
```

Code A.3: Part three of the script that crawls users. It uses certain methods from database.py. The last part of the function, which handles the calls to the database.

```python
        if "games" not in owned_games:
            if verbose:
                print("Game list is private, skipping...")
            process_steam_user(steamid, player["personaname"], 1, 0,
                               commentpermission=commentpermission, primaryclanid=primaryclanid,
                               timecreated=timecreated, loccountrycode=loccountrycode,
                               locstatecode=locstatecode, loccityid=loccityid)
            continue

        visible_playtime = False
        for game in owned_games["games"]:
            appid = game["appid"]
            success = game_exists(appid)

            if not success:
                if verbose:
                    print(f"{appid} doesn't exist in database, crawling...")
                success, query_count = get_app_data(str(appid), reviews=reviews, query_count=query_count, only_games=only_games, verbose=verbose)
            visible_playtime |= game["playtime_forever"] > 0
            if success == ERROR:
                if verbose:
                    print("Error while crawling game, adding as 'DEAD_HIDDEN_GAME' with no info...")
                # NOTE: this is specially useful to see 'if you used to play appid X and you now play appid Y, others might like appid Y'
                # for example: Project Cars 1 and 2 are dead, but you might like Project Cars 3 without genre context
                # also PUBG Test Server, Realm Royale Public Test, etc. are hidden
                add_dead_hidden_game(appid)
            elif success == SKIPPED:
                if verbose:
                    print("Non-game/mod skipped, skipping...")
                continue
            elif success == FAULTY:
                if verbose:
                    print("Faulty game, inserting as faulty with 'FAULTY_GAME' with no info...")
                add_faulty_game(appid)
            # insert_player_game_data(steamid, appid, playtime_forever, playtime_windows, playtime_mac, playtime_linux, rtime_last_played)
            insert_player_game_data(steamid, appid, game["playtime_forever"], game["playtime_windows_forever"], game["playtime_mac_forever"], game["playtime_linux_forever"], game["rtime_last_played"])

        process_steam_user(steamid, player["personaname"], 2 if not visible_playtime else 3, owned_games["game_count"],
                           commentpermission=commentpermission, primaryclanid=primaryclanid,
                           timecreated=timecreated, loccountrycode=loccountrycode,
                           locstatecode=locstatecode, loccityid=loccityid)
```

Code A.4: Full script that crawls reviews (part one). It uses certain methods from database.py and config.py. This portion goes from the start up to the end of the loop where we retry in case Steam is currently down, or we are being rate limited and our rate limiter is not properly accounting for rate limiting

```python
def get_app_data(app_id: str, reviews=False, query_count=0, only_games=True, verbose=True) -> Tuple[int, int]:
    retry = True
    consecutive_retries = 0
    game_data = get_game_data(app_id)
    exists = game_data is not None
    if (is_processed(app_id)):
        print(f"App {app_id} already processed.")
        return FULLY_PROCESSED, query_count
    elif (reviews == False and exists):
        print(f"App without reviews {app_id} already exists in database.")
        return ALREADY_EXISTS, query_count

    if not exists:
        while retry and not exists:
            try:
                query_count = check_rate_limit(query_count)
                response = request("GET", "https://store.steampowered.com/api/appdetails", params={"appids": app_id, "cc": "us", "l": "english"})
            except Exception as e:
                print("Exception while requesting appdetails: " + str(e) + "\n")
                consecutive_retries += 1
                if consecutive_retries > 3 *5: # maximum waiting time for the connection to be back: 5 minutes
                    # this high number also ensures the user may fix the problem without having to restart the script
                    consecutive_retries = 3 *5

                print(f"Waiting {(consecutive_retries * 20.0 / 60.0)} minute(s) before retrying...")
                time.sleep(consecutive_retries *20.0)
                continue

            retry = response.status_code != 200
            if retry:
                print(f"REST API error ({response.status_code}): " + str(response.text))
                print("Headers: " + str(response.headers) + "\n")
                consecutive_retries += 1
                if consecutive_retries > 3: # maximum waiting time for the API to respond again: 2 minutes
                    # this high number also ensures the user may fix the problem without having to restart the script
                    consecutive_retries = 3

                print(f"Waiting {((30.0 + consecutive_retries * 30.0) / 60.0)} minute(s) before retrying...")
                time.sleep(30.0 + consecutive_retries *30.0)
```

Code A.5: Part two of the script that crawls reviews. This part accounts for getting the basic data from the app and filtering out DLCs. Please note: demos and mods also have playtimes, and are relevant to us.

```
        if len(response.text) == 0:
            # example: https://store.steampowered.com/api/appdetails?appids=2124470
            # it exists in SteamDB and the store, but the API returns an empty response
            print(f"Empty response for faulty app ID {app_id}")
            return FAULTY, query_count
        appdata = json.loads(response.text)

        if appdata[app_id]["success"]:
            # First, check if this is a game
            if only_games and "type" in appdata[app_id]["data"] and\
                not (appdata[app_id]["data"]["type"] == "game"
                    or appdata[app_id]["data"]["type"] == "mod"
                    or appdata[app_id]["data"]["type"] == "demo"):
                if verbose:
                    print("Skipping non-game: " + appdata[app_id]["data"]["name"])
                return SKIPPED, query_count
            else: name = appdata[app_id]["data"]["name"]
        else:
            print(f"Error for app ID {app_id}: " + response.text)
            return ERROR, query_count
    else:
        name = game_data[1]
```

Code A.6: Part three of the script that crawls reviews. It uses certain methods from database.py. The last part of the function, which gets reviews only if asked to. We are also using this method to get basic user info through download_the_full_query_summary if reviews is set to False, specially useful when crawling user games. It also handles the calls to the database.

```
    # Then, check if this game has reviews
    if reviews:
        if verbose:
            print("Getting reviews for " + name)
        request_params['cursor'] = '*'
        review_dict, query_count = steamreviews.download_reviews_for_app_id(int(app_id),
                                                                chosen_request_params=request_params,
                                                                query_count=query_count,
                                                                verbose=True)
        query_summary = review_dict["query_summary"]
    else:
        success_flag = False
        while success_flag == False:
            if verbose:
                print("Getting review summary for " + name)
            try:
                success_flag, query_summary, query_count =
                    steamreviews.download_reviews.download_the_full_query_summary(app_id,
                    query_count, request_params)
                if not success_flag:
                    print("Sleeping 10 seconds")
                    time.sleep(10)
            except exceptions.ConnectionError as e:
                print("Error " + e)
                print("Sleeping 10 seconds")
                time.sleep(10)

    if not exists:
        process_game_data(app_id, appdata[app_id]["data"], query_summary)

    if reviews: # we need to execute this after process_game_data, because of the FK constraint
        process_game_reviews(app_id, review_dict["reviews"])
        mark_as_processed(app_id) # it's only fully processed when the reviews are processed
    if verbose:
        print("App ID: " + app_id + " done.")
    # NOTE: we should probably check the rate limits here, but we're ~10 requests under Steam official
        limits
    return SUCCESS, query_count
```

Code A.7: Script that inserts all popular tags from Steam, to get an ID -> tag name mapping

```python
lang = "english" # future work: language is shared through config.py
popular_tags_url = "https://store.steampowered.com/tagdata/populartags/" + lang
def get_popular_tags():
    response = get(popular_tags_url)
    if response.status_code == 200:
        return json.loads(response.text)
    else:
        print("Error: status code {}".format(response.status_code))
        return None

def insert_popular_tags():
    popular_tags = get_popular_tags()
    if popular_tags:
        for tag in popular_tags:
            insert_tag(tag)

if __name__ == "__main__":
    print("Downloading and inserting popular tags...")
    insert_popular_tags()
    print("Done.")
```

Code A.8: Script that inserts all tags from *output.jl* from 3.4

```
def insert_tags_from_scrapper(filename):
    try:
        tags = pd.read_json(filename, lines=True)
        tags = tags.drop_duplicates(subset='appid', keep='first').set_index('appid')
    except FileNotFoundError:
        print(filename + "␣not␣found.")
        return
    except ValueError:
        print("Error:␣" + filename + "␣is␣not␣a␣valid␣JSON␣Lines␣file.")
        return

    requires_manual_intervention = []
    for appid, row in tags.iterrows():
        if appid == 'app' or row is None: # error
            continue

        if not game_exists(appid):
            # first try to get the appid, since it might come from a redirect, like Total War: Shogun 2
            try:
                print("Game␣with␣appid␣not␣inside␣the␣database␣(redirects,␣betas,␣etc.):␣" + row['name'] + "␣(" + str(appid) + ")")
                gdata = get_game_data_from_name(row['name'])
                if gdata is None:
                    raise RequiresManualIntervention([], row['name'], "No␣appid␣found␣for␣game␣" + row['name'])
                appid = int(gdata[0])
            except RequiresManualIntervention as e:
                print("Unambiguous␣appid␣returned␣for␣game␣with␣appid␣not␣inside␣the␣database␣(redirects,␣betas,␣etc.):␣" + e.name + "␣(" + str(e.appids) + ")")
                requires_manual_intervention += (e.name, e.appids)
                continue

        insert_game_tags(appid, row['tags'], max_len)

    if len(requires_manual_intervention) > 0:
        print("Manual␣intervention␣required␣for␣the␣following␣games:")
        for game in requires_manual_intervention:
            print(game.name + ":␣" + str(game.appids))
```

Code A.9: Script that handles all playtime normalization techniques (part 1)

```python
class AbstractPlaytimeNormalizer(ABC):
    def __init__(self, denominator_function: str = "max", playtime_approach: str = "minutes_always_more_than_60", output_multiplier: float = config.RATING_MULTIPLIER, inplace: bool = False):
        self.inplace = inplace
        self.denominator_function = denominator_function
        self.playtime_approach = playtime_approach
        self.output_multiplier = output_multiplier

    def normalize(self, data: DataFrame) -> DataFrame:
        """The data to normalize, with a structure of:
            "steamid","appid","playtime_forever"

        Arguments:
        ---
            data (DataFrame): Expects a DataFrame with the above structure
        Returns:
        ---
            DataFrame: A DataFrame with the same structure as the input, but with the playtime_forever column normalized
        """
        self.validate_data(data)
        if not self.inplace:
            data = data.copy()

        # we need this to prevent division by zero, wrong sum, max, etc.
        if self.playtime_approach == "ignore":
            pass
        elif self.playtime_approach == "minutes_always_more_than_60":
            data["playtime_forever"] = data["playtime_forever"].apply(lambda x: 60 if x < 60 else x)
        elif self.playtime_approach == "hours_but_add_1":
            data["playtime_forever"] = data["playtime_forever"].apply(lambda x: math.floor(x / 60) + 1)
        else:
            raise ValueError("playtime_approach must be 'minutes_always_more_than_60', 'hours_but_add_1' or 'ignore'")

        if self.denominator_function == "max":
            denominators = data.groupby("steamid")["playtime_forever"].max()
        elif self.denominator_function == "sum":
            denominators = data.groupby("steamid")["playtime_forever"].sum()
        else:
            raise ValueError("divide_by must be either 'max', 'sum'")

        data["playtime_forever"] = data.apply(lambda x: self.output_multiplier * self.normalize_value(x["playtime_forever"], denominators[x["steamid"]]), axis=1)
        return data

    ...
```

Code A.10: Script that handles all playtime normalization techniques (part 2)

```
class AbstractPlaytimeNormalizer(ABC):
    ...

    @abstractmethod
    def normalize_value(self, playtime, denominator):
        pass

    def validate_data(self, data: DataFrame):
        # check if it's a DataFrame
        if not isinstance(data, DataFrame):
            raise TypeError("Argument 'data' must be a DataFrame")

        # check if the data has the correct columns
        if not all([col in data.columns for col in ["steamid", "appid", "playtime_forever"]]):
            raise ValueError("The DataFrame must have the columns 'steamid', 'appid' and 
                'playtime_forever'")

        # check if the data has the correct types
        if not all([data[col].dtype == "int64" for col in ["steamid", "appid"]]):
            raise ValueError("The DataFrame must have the columns 'steamid' and 'appid' as integer 
                values.")

    def __repr__(self) -> str:
        name = self.__class__.__name__
        name = name[0:name.index("PlaytimeNormalizer")] + "PN"
        return name + f"_{self.denominator_function}" + "".join([tok[0] for tok in
            re.findall(r"[a-zA-Z]+|[0-9]", self.playtime_approach)]) + f"_{self.output_multiplier}x"

    def __str__(self) -> str:
        return self.__repr__()
```

Code A.11: Class that does not normalize data

```
class NoNormalization(AbstractPlaytimeNormalizer):
    """Doesn't normalize the player_games data
    """
    def normalize(self, data: DataFrame) -> DataFrame:
        self.validate_data(data)
        return data
    def normalize_value(self, playtime, denominator):
        return playtime
    def __repr__(self) -> str:
        return "NoNormalization"
```

Code A.12: Class that linearly normalizes our data

```
class LinearPlaytimeNormalizer(AbstractPlaytimeNormalizer):
    """Normalizes the player_games data using the playtime_forever
    """
    def normalize_value(self, playtime, denominator):
        return playtime / denominator
```

Code A.13: Class that logarithmically normalizes our data

```
class LogPlaytimeNormalizer(AbstractPlaytimeNormalizer):
    """Normalizes the player_games data using the log of the playtime_forever
    """
    def normalize_value(self, playtime, denominator):
        return np.log(playtime) / np.log(denominator)
```

Code A.14: Class that takes the square root of our data

```
class RootPlaytimeNormalizer(AbstractPlaytimeNormalizer):
    """Normalizes the player_games data using the N root of the playtime_forever
    """
    def __init__(self, denominator_function: str = "sum_max", playtime_approach: str =
         "minutes_always_more_than_60", nroot: int = 2, output_multiplier: int = 5, inplace: bool =
         False):
        super().__init__(denominator_function, playtime_approach, output_multiplier, inplace)
        self.nroot = nroot

    def normalize_value(self, playtime, denominator):
        return np.power(playtime, 1/self.nroot) / np.power(denominator, 1/self.nroot)

    def __repr__(self) -> str:
        return super().__repr__() + f"_{self.nroot}root"
```

Code A.15: Function that calculates the similarity between two games based on their details

```python
similarity = 0.0
similarity += difflib.SequenceMatcher(None, details.name, other_details.name).ratio() * self.weights["name"] if self.weights["name"] > 0 else 0.0
similarity += (1.0 -(abs(details.required_age -other_details.required_age) / 18.0)) * self.weights["required_age"] if self.weights["required_age"] > 0 else 0.0
similarity += self.weights["is_free"] *details.is_free *other_details.is_free
similarity += (1.0 -abs(details.controller_support -other_details.controller_support) / 2.0) * self.weights["controller_support"] if self.weights["controller_support"] > 0 else 0.0
similarity += self.weights["has_demo"] *details.has_demo *other_details.has_demo
similarity += max(1 -abs(details.price_usd -other_details.price_usd) / 70.0, 0) * self.weights["price_usd"] if self.weights["price_usd"] > 0 else 0.0
similarity += self.weights["mac_os"] *details.mac_os *other_details.mac_os
similarity += max(1 -abs(details.rating -other_details.rating) / self.rating_multiplier, 0) * self.weights["rating"] if self.weights["rating"] > 0 else 0.0
similarity += max(1 -abs(details.total_reviews -other_details.total_reviews) / max(details.total_reviews, other_details.total_reviews), 0) *self.weights["rating_count"] if self.weights["rating_count"] > 0 else 0.0
similarity += self.weights["has_achievements"] *details.has_achievements * other_details.has_achievements
if self.weights["release_date"] > 0:
    try:
        own_release_date = datetime.strptime(details.release_date, "%b %d, %Y")
        other_release_date = datetime.strptime(other_details.release_date, "%b %d, %Y")
        similarity += max(1.0 -abs((own_release_date -other_release_date).days) / 365.0, 0) * self.weights["release_date"]
    except: # Some games don't have a release date (\\N), some others have "Coming Soon" or even emojis
        pass
similarity += self.weights["coming_soon"] *details.coming_soon *other_details.coming_soon

return similarity / sum(self.weights.values())
```

B

Annex II: Results

When refering to relevant thresholds, as mentioned in Section 3.5, we will use the following notation: wt=0.30 means that we will only take into account games that the user has played more than 30% of their most played game. As well, we will use t=0.30 to mean that the LSH Ensemble is configured to return games with a similarity of 30% or higher, and w_{idf}=0.30 to mean that the IDF weight for tags is 0.30. We will use the same notation for all other thresholds.

All of our CBFs used a linearly normalized *PlayerGamesPlaytime* instance.

Combination	**P@5**	**P@10**	**P@20**	**R@5**	**R@10**	**R@20**	**Time (s)**
Cosine (Linear)	0.0540	0.0400	0.0312	0.0149	0.0218	0.0346	61,063.86
Cosine (Log)	0.0770	0.0575	0.0435	0.0214	0.0312	0.0480	61,105.11
Cosine (Square Root)	0.0596	0.0483	0.0369	0.0165	0.0264	0.0407	61,033.86
Pearson (Linear)	0.1954	0.1672	0.1281	0.0581	0.0983	0.1512	94,042.03
Pearson (Log)	0.2136	0.1753	0.1356	0.0632	0.1030	0.1587	95,043.25
Pearson (Square Root)	0.2056	0.1787	0.1389	0.0605	0.1048	0.1625	95,983.37
Raw (Linear)	0.0530	0.0401	0.0314	0.0146	0.0220	0.0348	65,261.46
Raw (Log)	0.0692	0.0528	0.0401	0.0191	0.0289	0.0438	65,443.07
Raw (Square Root)	0.0594	0.0479	0.0364	0.0164	0.0261	0.0399	65,674.90

Table B.1: Precision and recall results for CF, same as Table 4.3, without format. Parameters used: wt = 0.6, t = 0.8. User similarity followed by the normalization approach in parentheses.

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
t_{rel}=0.60, t_{lsh}0.60	0.3100	0.272	0.208	0.0710	0.1225	0.1879	6,420
t_{rel}=0.60, t_{lsh}0.80	0.3260	0.286	0.2125	0.0755	0.1289	0.1953	5,160
t_{rel}=0.80, t_{lsh}0.60	0.342	0.274	0.2110	0.0798	0.124	0.1896	7,020
t_{rel}=0.80, t_{lsh}0.80	0.348	0.277	0.2065	0.0813	0.1256	0.1861	5,760

Table B.2: Precision and recall for pearson game tag based CBF, for various ensembles, only for a smaller list of 100 users. t_{rel} denotes the minimum threshold for games to be included in the MinHash (to be considered relevant), t_{lsh} is the threshold for the LSH Ensemble index. We picked 0.60 and 0.80 because it was the fastest one and the one with the best results at 10 and 20. Time is approximated based on file creation times, as we lost our time results.

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
t=0.30, w_{idf}=0.00	0.0376	0.0306	0.0262	0.0109	0.0175	0.0302	13594.60
t=0.30, w_{idf}=0.15	0.0376	0.0320	0.0269	0.0108	0.0183	0.0304	14949.60
t=0.30, w_{idf}=0.30	0.0414	0.0340	0.0290	0.0117	0.0191	0.0328	11879.06
t=0.30, w_{idf}=0.60	0.0472	0.0382	0.0318	0.0137	0.0217	0.0364	9379.58
t=0.42, w_{idf}=0.00	0.0372	0.0307	0.0264	0.0108	0.0175	0.0303	10241.59
t=0.42, w_{idf}=0.15	0.0372	0.0319	0.0271	0.0107	0.0183	0.0307	9034.59
t=0.42, w_{idf}=0.30	0.0414	0.0340	0.0289	0.0117	0.0192	0.0328	8600.51
t=0.42, w_{idf}=0.60	0.0476	0.0382	0.0317	0.0138	0.0217	0.0361	9571.94
t=0.55, w_{idf}=0.00	0.0378	0.0307	0.0262	0.0110	0.0175	0.0302	3052.29
t=0.55, w_{idf}=0.15	0.0376	0.0322	0.0265	0.0108	0.0184	0.0302	2954.21
t=0.55, w_{idf}=0.30	0.0416	0.0344	0.0278	0.0117	0.0194	0.0313	2873.44
t=0.55, w_{idf}=0.60	0.0452	0.0372	0.0306	0.0129	0.0209	0.0348	2313.82
t=0.68, w_{idf}=0.00	0.0336	0.0298	0.0245	0.0095	0.0173	0.0281	963.79
t=0.68, w_{idf}=0.15	0.0340	0.0297	0.0255	0.0094	0.0169	0.0288	918.73
t=0.68, w_{idf}=0.30	0.0366	0.0306	0.0260	0.0100	0.0175	0.0295	865.00
t=0.68, w_{idf}=0.60	0.0394	0.0323	0.0248	0.0112	0.0183	0.0286	824.77
t=0.80, w_{idf}=0.00	0.0328	0.0288	0.0244	0.0091	0.0165	0.0276	850.73
t=0.80, w_{idf}=0.15	0.0334	0.0291	0.0248	0.0093	0.0164	0.0280	806.09
t=0.80, w_{idf}=0.30	0.0354	0.0298	0.0247	0.0096	0.0169	0.0278	752.47
t=0.80, w_{idf}=0.60	0.0392	0.0309	0.0235	0.0111	0.0175	0.0271	720.42
t=1.00, w_{idf}=0.00	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	69.07
t=1.00, w_{idf}=0.30	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	86.11
t=1.00, w_{idf}=0.60	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	68.21
wt=0.75, w_{idf}=0.00	0.0376	0.0306	0.0262	0.0109	0.0175	0.0302	23142.23
wt=0.75, w_{idf}=0.30	0.0432	0.0364	0.0304	0.0122	0.0205	0.0344	19974.60
wt=0.75, w_{idf}=0.60	0.0472	0.0381	0.0317	0.0137	0.0216	0.0363	20120.96
wt=1.00, w_{idf}=0.00	0.0360	0.0296	0.0243	0.0104	0.0170	0.0280	8675.47
wt=1.00, w_{idf}=0.30	0.0410	0.0334	0.0280	0.0117	0.0191	0.0322	8134.27
wt=1.00, w_{idf}=0.60	0.0460	0.0360	0.0286	0.0135	0.0207	0.0329	6528.09

Table B.3: Precision and recall for raw game tag based CBF. t is the threshold for the LSH Ensemble index, w_{idf} is the IDF weight for tags and wt denotes our tag $\rightarrow$ game dictionary approach with weights (no LSH Ensemble).

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
t=0.30, w_{idf}=0.00	0.0354	0.0286	0.0248	0.0104	0.0163	0.0284	2532.25
t=0.30, w_{idf}=0.30	0.0344	0.0298	0.0256	0.0098	0.0170	0.0291	2417.31
t=0.30, w_{idf}=0.60	0.0372	0.0315	0.0282	0.0103	0.0177	0.0319	2098.92
t=0.42, w_{idf}=0.00	0.0352	0.0287	0.0249	0.0104	0.0163	0.0285	2018.68
t=0.42, w_{idf}=0.30	0.0348	0.0299	0.0261	0.0099	0.0171	0.0297	1897.33
t=0.42, w_{idf}=0.60	0.0376	0.0316	0.0284	0.0105	0.0178	0.0320	1626.74
t=0.55, w_{idf}=0.00	0.0356	0.0298	0.0250	0.0105	0.0169	0.0287	748.52
t=0.55, w_{idf}=0.30	0.0348	0.0301	0.0262	0.0097	0.0171	0.0296	688.41
t=0.55, w_{idf}=0.60	0.0368	0.0326	0.0274	0.0101	0.0179	0.0306	549.00
t=0.68, w_{idf}=0.00	0.0318	0.0281	0.0232	0.0090	0.0163	0.0266	278.09
t=0.68, w_{idf}=0.30	0.0312	0.0288	0.0247	0.0085	0.0165	0.0279	259.20
t=0.68, w_{idf}=0.60	0.0342	0.0294	0.0247	0.0092	0.0163	0.0280	219.70
wt=0.75, w_{idf}=0.00	0.0350	0.0283	0.0245	0.0103	0.0161	0.0279	7707.61
wt=0.75, w_{idf}=0.30	0.0340	0.0294	0.0254	0.0097	0.0168	0.0289	7342.89
wt=0.75, w_{idf}=0.60	0.0362	0.0309	0.0273	0.0100	0.0174	0.0309	6437.81
wt=1.00, w_{idf}=0.00	0.0344	0.0279	0.0233	0.0101	0.0158	0.0267	2050.46
wt=1.00, w_{idf}=0.30	0.0336	0.0287	0.0238	0.0096	0.0164	0.0271	1814.81
wt=1.00, w_{idf}=0.60	0.0362	0.0289	0.0251	0.0101	0.0163	0.0284	1393.10

Table B.4: Precision and recall for cosine game tag based CBF. t is the threshold for the LSH Ensemble index, w_{idf} is the IDF weight for tags and wt denotes our tag $\rightarrow$ game dictionary approach with weights (no LSH Ensemble)

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
t=0.30, w_{idf}=0.00	0.0174	0.0156	0.0139	0.0050	0.0090	0.0158	2561.44
t=0.30, w_{idf}=0.30	0.0174	0.0156	0.0152	0.0052	0.0092	0.0175	2418.28
t=0.30, w_{idf}=0.60	0.0250	0.0227	0.0205	0.0072	0.0129	0.0235	2159.92
t=0.42, w_{idf}=0.00	0.0188	0.0157	0.0141	0.0054	0.0090	0.0161	2033.70
t=0.42, w_{idf}=0.30	0.0186	0.0157	0.0152	0.0055	0.0092	0.0174	1907.80
t=0.42, w_{idf}=0.60	0.0256	0.0230	0.0200	0.0074	0.0130	0.0229	1667.64
t=0.55, w_{idf}=0.00	0.0202	0.0181	0.0175	0.0056	0.0103	0.0205	748.65
t=0.55, w_{idf}=0.30	0.0202	0.0195	0.0179	0.0057	0.0113	0.0202	690.66
t=0.55, w_{idf}=0.60	0.0274	0.0240	0.0210	0.0076	0.0134	0.0235	561.59
t=0.68, w_{idf}=0.00	0.0218	0.0209	0.0181	0.0061	0.0123	0.0211	278.86
t=0.68, w_{idf}=0.30	0.0224	0.0200	0.0187	0.0064	0.0116	0.0216	259.92
t=0.68, w_{idf}=0.60	0.0240	0.0220	0.0191	0.0069	0.0131	0.0225	223.87
wt=0.75, w_{idf}=0.00	0.0170	0.0154	0.0139	0.0049	0.0089	0.0159	7810.68
wt=0.75, w_{idf}=0.30	0.0168	0.0146	0.0143	0.0050	0.0086	0.0167	7346.05
wt=0.75, w_{idf}=0.60	0.0234	0.0217	0.0196	0.0067	0.0123	0.0225	6624.96
wt=1.00, w_{idf}=0.00	0.0180	0.0155	0.0146	0.0049	0.0088	0.0169	2125.41
wt=1.00, w_{idf}=0.30	0.0170	0.0156	0.0144	0.0050	0.0092	0.0166	1823.94
wt=1.00, w_{idf}=0.60	0.0216	0.0220	0.0184	0.0064	0.0124	0.0210	1438.34

Table B.5: Precision and recall for pearson game tag based CBF. t is the threshold for the LSH Ensemble index, w_{idf} is the IDF weight for tags and wt denotes our tag $\rightarrow$ game dictionary approach with weights (no LSH Ensemble)

Combination	P@5	P@10	P@20	R@5	R@10	R@20	Time (s)
Details	0.0440	0.0334	0.0257	0.0134	0.0199	0.0310	21246.19
Developers	0.0606	0.0554	0.0467	0.0187	0.0343	0.0571	1397.43
Publishers	0.0570	0.0502	0.0402	0.0168	0.0301	0.0477	3021.03
Categories(t=0.30)	0.0362	0.0304	0.0251	0.0113	0.0184	0.0302	5114.46
Categories(t=0.42)	0.0370	0.0304	0.0251	0.0114	0.0185	0.0302	1869.48
Categories(t=0.55)	0.0242	0.0150	0.0082	0.0076	0.0094	0.0103	63.37
Categories(t=0.68)	0.0224	0.0141	0.0077	0.0071	0.0089	0.0099	61.14
Categories(t=0.80)	0.0222	0.0138	0.0075	0.0071	0.0087	0.0095	59.93
Categories(t=1.00)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	18.98
Genres(t=0.30)	0.0030	0.0035	0.0044	0.0007	0.0018	0.0049	4119.41
Genres(t=0.42)	0.0026	0.0031	0.0041	0.0005	0.0015	0.0045	2246.40
Genres(t=0.55)	0.0016	0.0022	0.0022	0.0003	0.0013	0.0027	255.79
Genres(t=0.68)	0.0016	0.0022	0.0022	0.0003	0.0013	0.0027	240.21
Genres(t=0.80)	0.0016	0.0022	0.0022	0.0003	0.0013	0.0027	235.48
Genres(t=1.00)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	19.08

Table B.6: Precision and recall for all other CBFs. Developers and publishers do not have a LSH index. Details uses a custom similarity function, defined in A.15. t is the threshold for the LSH Ensemble index.