MEF UNIVERSITY

GAME RECOMMENDATION SYSTEM FOR STEAM PLATFORM

Capstone Project

Serhan Bayram

İSTANBUL, 2021

MEF UNIVERSITY

GAME RECOMMENDATION SYSTEM FOR STEAM PLATFORM

Capstone Project

Serhan Bayram

Advisor: Prof. Dr. Semra Ağralı

İSTANBUL, 2021

MEF UNIVERSITY

Name of the project: Game Recommendation System For Steam Platform
Name/Last Name of the Student: Serhan Bayram
Date of Thesis Defense: 01/09/2021

I hereby state that the graduation project prepared by Serhan Bayram has been completed under my supervision. I accept this work as a "Graduation Project".

01/09/2021
Prof. Dr. Semra Ağralı

I hereby state that I have examined this graduation project by Serhan Bayram which is accepted by his supervisor. This work is acceptable as a graduation project and the student is eligible to take the graduation project examination.

01/09/2021

Prof. Dr. Özgür Özlük

Director
of
Big Data Analytics Program

We hereby state that we have held the graduation examination of __________ and agree that the student has satisfied all requirements.

THE EXAMINATION COMMITTEE

Committee Member	Signature
1. Prof. Dr. Semra Ağralı	………………………..
2. Prof. Dr. Özgür Özlük	………………………..

Academic Honesty Pledge

I promise not to collaborate with anyone, not to seek or accept any outside help, and not to give any help to others.

I understand that all resources in print or on the web must be explicitly cited.

In keeping with MEF University's ideals, I pledge that this work is my own and that I have neither given nor received inappropriate assistance in preparing it.

Name	Date	Signature
Serhan Bayram	01/09/2021	

EXECUTIVE SUMMARY

GAME RECOMMENDATION SYSTEM FOR STEAM PLATFORM

Serhan Bayram

Advisor: Prof. Dr. Semra Ağralı

SEPTEMBER, 2021, 35 pages

Increasing number of choices and competition in the markets, force companies to differ in services they provide to their customers. Offering better services have a positive impact on customer loyalty, and to do so, companies should understand their customers' interests and act accordingly. One popular method for this purpose is building recommendation engines to make personalized suggestions. In this project, collaborative filtering methods with implicit feedback are used to make recommendations to users of the Steam platform. The recommendation systems are built using two different matrix factorization techniques, Alternating Least Squares and Bayesian Personalized Ranking. Different models are created with implicit playtime data of the users and the results are evaluated by using Precision at k metric. Additionally, similar items that are offered by the models are analyzed. Results show that the models are considerably successful at finding personal choices and similar items. The best model finds the item in the libraries of 33% of the users.

Key Words: Recommendation Engine, Matrix Factorization, Collaborative Filtering, Alternating Least Squares, Bayesian Personalized Ranking, Implicit Feedback

ÖZET

STEAM PLATFORMU İÇİN OYUN ÖNERİ SİSTEMİ

Serhan Bayram

Proje Danışmanı: Prof. Dr. Semra Ağralı

EYLÜL, 2021, 35 sayfa

Seçeneklerin ve pazardaki rekabetin artması, şirketleri müşterilerine sundukları hizmetlerde farklılaşmaya zorlamaktadır. Daha iyi hizmetler önermenin müşteri sadakati açısından olumlu bir etkisi vardır ve şirketlerin bunu sağlayabilmesi için müşterilerinin ilgi alanlarını anlaması ve ona göre hareket etmesi gerekmektedir. Kişisel öneriler yapmak için öneri motorları geliştirmek bu amaca uygun yaygın bir yöntemdir. Bu projede, Steam platformu kullanıcılarına öneri yapmak için örtülü geri bildirimler ile işbirlikçi filtreleme yöntemleri kullanılmıştır. Öneri sistemleri iki farklı matris faktorizasyon tekniği olan Alternatif En Küçük Kareler ve Bayes Kişisel Sıralama ile kurulmuştur. Kullanıcıların örtülü oyun oynama süreleri verisi kullanılarak farklı modeler oluşturulmuş ve sonuçlar k değerinde Kesinlik metriği kullanılarak değerlendirilmiştir. Ek olarak, modeller tarafından önerilen benzer oyunlar incelenmiştir. Sonuçlar, modellerin hem kişisel tercihleri hem de benzer ürünleri bulmada dikkate değer ölçüde başarılı olduğunu göstermiştir. En iyi model, kullanıcıların %33'ünün kütüphanesinde olan oyunu bulmayı başardı.

Anahtar Kelimeler: Öneri Motoru, Matris Faktorizasyonu, İşbirlikçi Filtreleme, Alternatif En Küçük Kareler, Bayes Kişisel Sıralama, Örtülü Geribildirim

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In the last years the number of choices expanded tremendously with the increased accessibility to digital platforms. As Wu et al. (2021) mentions, this expansion in choices created an information overload, which brought challenges to both platforms and their customers. Users of these platforms spend notable time to reach the products they need or are interested in due to this overload. Reducing this time would create a significant value for the users, increase the customer satisfaction and make it possible for users to spend the remaining time to buy other products. These benefits are only some of the main reasons for the creation of recommender systems.

Recommender systems are the algorithms that produce suggestions to the users. These suggestions can be offered in various ways by the platforms for several reasons and can be based on user attributions or item attributions. In general, there are two main types of recommender systems: Collaborative Filtering (CF) and Content-Based Filtering (CBF). Adomavicius and Tuzhilin (2005) define CF as suggesting items that are chosen by other similar users and CBF as suggesting items that are similar to the previously chosen items of the user. Goldberg et al. (1992) mentioned CF for the first time in early 1990s. After that definition, lots of recommender models have been created by researchers.

Steam is one of the most popular online game distribution platforms (Steam Website, https://store.steampowered.com). Fenlon (2019) states that in 2019 there were 90 million monthly active users on the platform. Through the platform, users can reach free or paid games. Once a user buys a game, they get lifetime access to it. Providing lifetime access to the games positioned the platform as a virtual library. Since the users can access the games whenever they want, they buy games even if they do not plan to play them in near future. During a year, there are several discount events on the platform, in which the prices of the games decrease dramatically. Especially in these periods, users probably tend to purchase more games and enhance the diversity in their collections.

The main purpose of this study is to create a recommender system to offer games to users that they will be eager to buy. Such a system would increase revenue and loyalty since it chooses the right games for the users and saves their time. With the help of this system, the platform can create value for both the users and publishers of the games.

2. LITERATURE SURVEY

In this section, related works in the field of recommender systems are introduced. These works mainly focus on three different topics: Challenges of Recommender Systems, Feedback Types and Building & Evaluating Recommender Systems. In the first part, some difficulties that might appear during a recommendation engine building process and their possible solutions are given. In the second part, the details of implicit and explicit feedback, and their usage examples are provided. In the third part, several methods of building and evaluating recommender systems that are found in the literature are given.

2.1 Challenges of Recommender Systems

Building a recommendation engine has lots of challenges. Most of the recommendation engines offer the most popular items, however this causes non-popular items to become nearly invisible by users. For newly arrived items or users, where there is not enough data to measure popularity or profile users, a method should be defined to make these items visible for users. As Viljanen (2020) expresses, this problem is defined as *cold start*. Koren et al. (2009) mention that although collaborative filtering generally outperforms content-based filtering, it shows inadequate performance in such cases, and content-based filtering can be more useful. In real life it is better to handle multiple problems with the same recommender system. Burke (2002) implies that combining several recommendation approaches in a proper way would bring higher performance.

Koren et al. (2009) state that, especially in collaborative filtering systems, matrix factorization is very common. In this approach latent vectors of users and items are used to predict the unknown ratings. These vectors are created by using a huge user and item matrix which is expected to be highly sparse. Jung et al. (2004) and Fan et al. (2014) suggest that one of the major problems while building a recommender system is the sparsity of ratings since the rated items in the dataset have usually a very small proportion. In order to decrease sparsity, some items can be eliminated from the dataset. Reddy et al. (2020) suggests that removing items that have ratings below the mean should be excluded; hence, they do not make a notable contribution to the model performance. This approach might be useful while deciding on items to exclude.

Adomavicius and Tuzhilin (2005) indicate a possible problem with the content-based model as overspecialization, which is offering items that are very similar to the ones the user interacted with before. This can be thought as recommending glasses to a user who just bought a new one. Such a recommendation would make no contribution to the model. According to Kang and McAuley (2018) basically there are two approaches to make a recommendation to a user dynamically. The first approach is using data within a wider time range by Recurrent Neural Networks. The second approach is considering only more recent data to make offers, which performs well with less data, here Markov Chains can be preferred. They combine both approaches to build a more successful one. The proposed method considers long-term data to find relevant items and uses recent data to make the final decision. In the model, Dropout Regularization is used to prevent overfitting.

Pathak et al. (2017) address the benefits of creating personalized bundles to sell the items. This approach can also be an alternative solution to the cold start problem by bundling newly arrived games with popular ones. In the paper, to find the interest of users to the bundles, Bayesian Personalized Ranking is used. Also, to generate personalized bundles for the users, Greedy Algorithm is used.

2.2 Feedback Types

In order to create relationships and understand patterns between users or items, feedback data is needed. As Hu et al. (2008) state there are two types of feedback that are used in recommender systems, implicit and explicit feedback. Explicit feedback can be considered as the given ratings to the items. On the other hand, implicit feedback is more hidden in patterns, such as purchased items, visited pages, etc.

In real life, the feedback might not be very explicit to use directly in a model. Wan and McAuley (2018) mention the relationship between feedback signals. This approach can be used to discover the dependency between making a review and purchasing an item. This dependency might be helpful while deriving useful feedback from the original ones.

Transforming implicit features to create explicit feedback is another approach. Saaidin and Kasiran (2021) use prices and playtimes of games to calculate a rating score. For this purpose, the correlation between the playtime and the price has been calculated. After that, a rule-based approach is applied to find the ratings. Kamal et al. (2020) also use playtime of the games to calculate the ratings by applying a z-score approach for every user. After calculating the z-scores of the games, the ratings are assigned by some rules. Plunkett et al. (2016) use five equal percentiles of the playtime value to calculate the ratings.

2.3 Building & Evaluating Recommender Systems

Building a recommender system totally depends on the available data. As it is mentioned in the previous part, users might not explicitly share their opinions about products. Therefore, implicit feedback should be considered to create such a system. Hu et al. (2008) use Alternating Least Squares (ALS) with implicit feedback by defining the concept of confidence. As Victor (2017) explains, ALS is an optimization approach that can be used during matrix factorization to find the most similar representation of the initial user and item pairs. A user's preference can be easily understood through explicit feedback. On the other hand, as Hu et al. (2008) point out, it is hard to know whether a user likes or dislikes an item by inspecting implicit feedback. For instance, if a user does not have any implicit feedback on a specific item, he/she might not be aware of that item and this does not necessarily indicate that the user does not like that item. This is the idea behind the confidence term. It assumes that having more implicit feedback shows stronger bond between the user and the item, but not having any feedback also has a small impact. The effect of the implicit feedback can be arranged by using parameter alpha.

ALS algorithm tries to find a relationship between the user and item pairs as it is mentioned previously. Rendle et al. (2012) offer another solution for implicit feedback, called Bayesian Personalized Ranking (BPR), which is also a matrix factorization technique. Rather than using user-item pairs, it considers triplets which consist of a user and two items. This triplet holds the information of whether one item is preferred to the other item. This approach eventually helps to create rankings of items for users and makes it possible to make more personalized recommendations.

After building a recommendation engine, the performance of it should be measured. Krichene and Rendle (2020) explain several metrics such as Precision at k, Recall at k and Average Precision at k. At k represents the number of recommended items in all of these metrics. Cheuque et al. (2019) built game recommender models by using ALS and Factorization Machines methods and use Average Precision and Mean Average Precision metrics to evaluate the performance of these models. Also, they use Novelty and Diversity metrics to measure the distinctness of the recommendations.

3. ABOUT THE DATA

Data used in the project is gathered from the Julian McAuley's Recommender Systems Datasets Library as three different datasets with various information (Steam Dataset, https://cseweb.ucsd.edu/~jmcauley/datasets.html#steam_data). First dataset has the information of 88,310 users' owned games and playtimes of these games. Second dataset contains more detailed information about the 32,135 items in the platform such as name, release date, genre, and price. Third dataset contains 615 bundles' information. A bundle is a collection of several games and its price is cheaper than the individual sum of the games in it. This dataset is combined with the second dataset to gather information about more games.

3.1 Data Preparation

All of these datasets are in loose json format. Before making experiments, the data needs to be prepared. The preparation steps are as follows:

- Each data file is parsed, and each dataset is normalized.
- Duplicate records are eliminated.
- A game can belong to more than one genre and there is a genre array for each item. The first element in this array is considered as the game's genre.
- Records with null values are eliminated.
- In the datasets, there are some items that are not games but different types of software. The records with these items are excluded.
- For free games, some of the prices are entered in string format. They are changed as 0.
- In the first dataset, there are some games that do not exist in the second and third datasets. The records related to those games are excluded.
- All three datasets are merged to create two main datasets. One of them contains only paid games, and the other one contains both paid and free games.
- Users with only one item are excluded.
- Items with only one user are excluded.
- The playtime distribution is highly skewed, and some values are very high. Therefore, natural logarithms of the playtimes are added.

The paid dataset contains 3,382,945 rows and the other dataset contains 4,140,773 rows. Each row in the datasets corresponds to a user and user's owned item. For instance, if a user has 10 different games, there are 10 rows of that user.

3.2 Features

The explanations of the features that are used in the project are as follows:

- user: Unique id of a user
- item: Unique id of a game
- item_name: Name of a game
- playtime_forever: Playtime of a game by a user in hours
- single_genre: Genre of a game
- price: Price of a game
- playtime: Natural logarithm of playtime_forever

3.3 Exploratory Data Analysis

The main dataset has 65,945 unique users and 8,090 unique games. The paid dataset has 61,909 unique users and 7,587 unique games. These numbers indicate that 93,78% of the games are not free. The price distribution of the games can be seen in Figure 1. Since there are only 4 games that have higher prices than $100, they are excluded from Figure 1. Among the paid games 94.64% of the games have a price of less than $20.

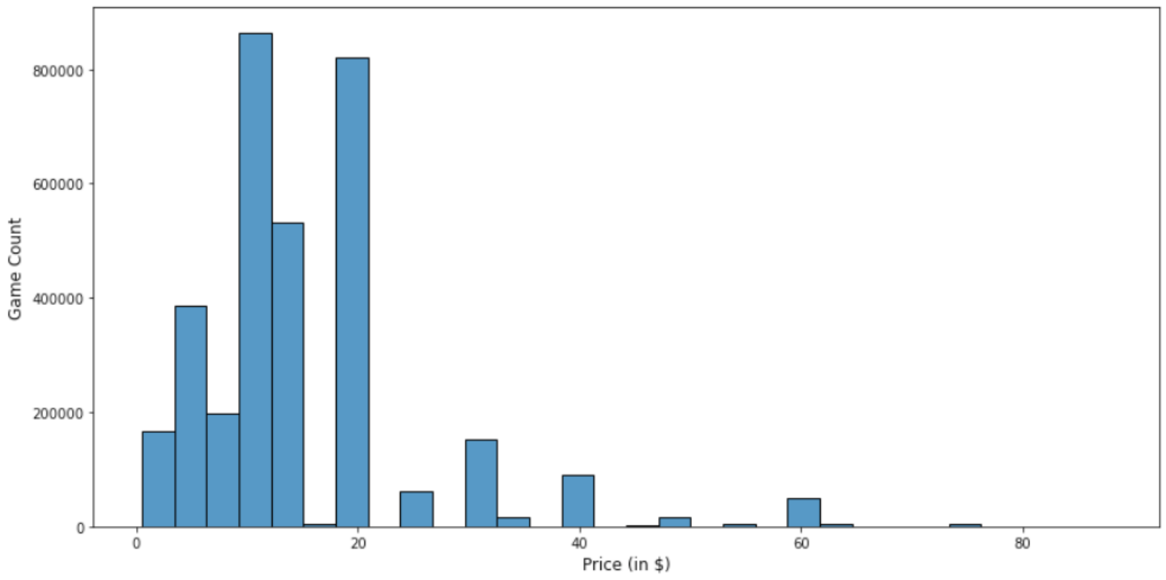


Figure 1: Price Distribution of the Games

The playtime distribution of the games is shown in Figure 2 considering both paid and free games. Only 2.59% of the rows contain playtime higher than 8,000 hours. So, Figure 2 shows at most 8,000 hours. 32.9% of the games in the libraries are not played even for an hour. This rate is 33.61% for paid games. This data supports the explanation given in the Introduction section as users pay for the games even if they do not plan to play them in near future.

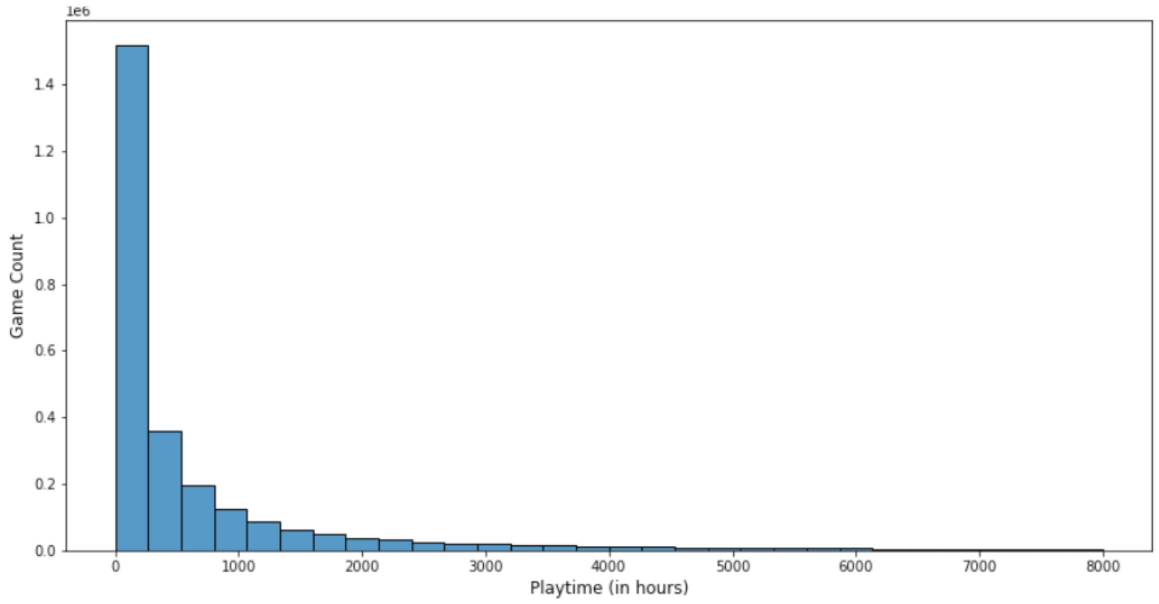


Figure 2: Playtime Distribution of All Games - Playtime Between 1 and 8,000 Hours

Figure 2 shows that the playtime values need to be transformed in order to achieve a normal distribution. Figure 3 shows the transformed playtime values by natural logarithm, again by considering both paid and free games. Since 0 has no logarithm calculation, 1 is added to all values and then the natural logarithm value is calculated.

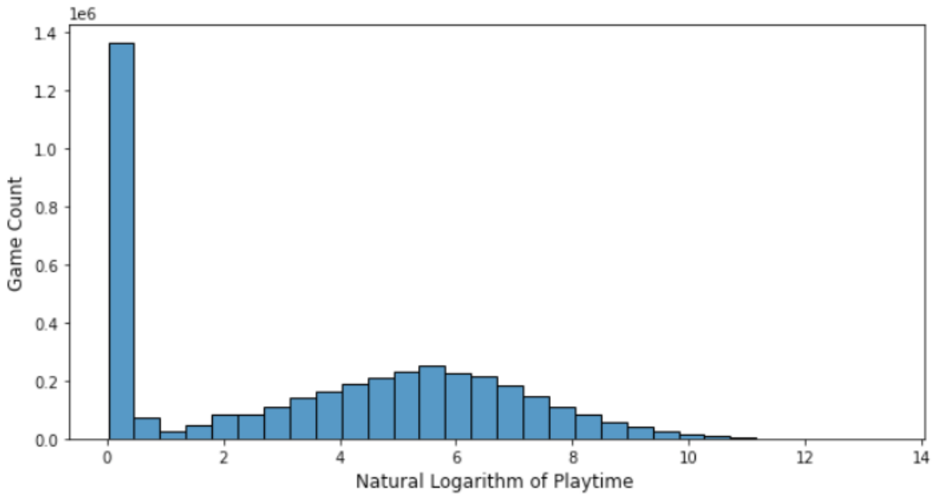


Figure 3: Natural Logarithm of Playtime Distribution of All Games

After the transformation, the distribution becomes really close to the normal if zero playtime records are overlooked.

Figure 4 shows the distribution of the number of owned games, considering free and paid games. Among 65,945 unique users, only 300 of them have more than 600 games. Hence, Figure 4 shows users who have at most 600 games.

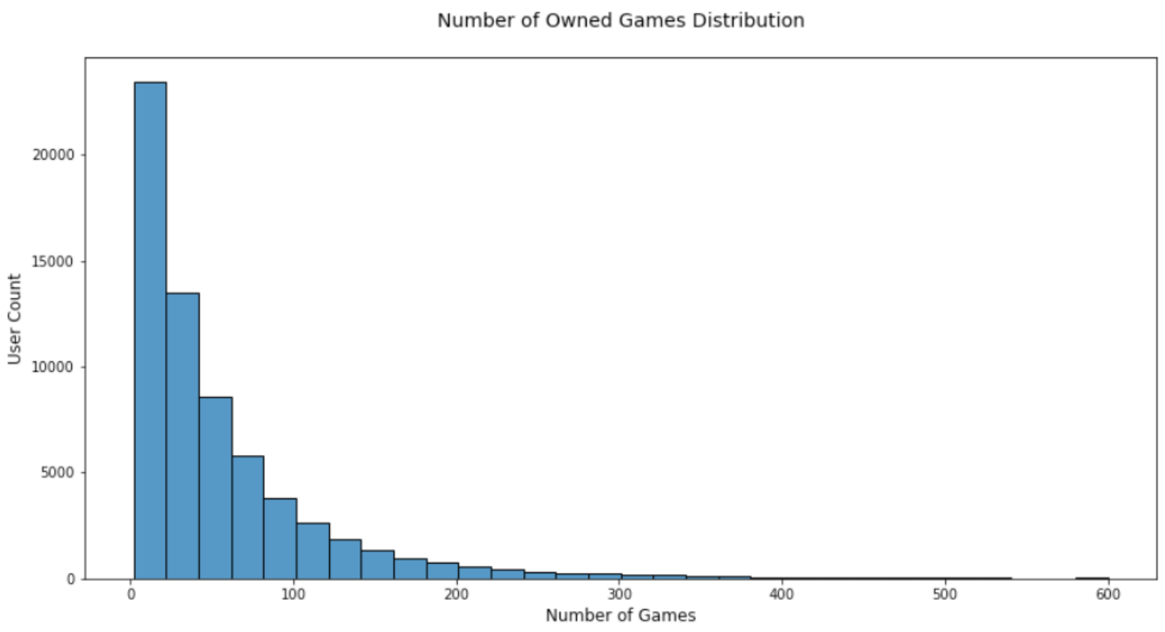


Figure 4: Distribution of Number of Owned Games

As expected, most of the users have less than 100 games. This fact actually is one of the main motivations of this project. There are lots of users who can buy new games if they are provided with accurate recommendations. 65.73% of the users have more than 20 games. This rate is 56.95% for the paid games. Thus, there is a huge potential with users who have less games. Moreover, this potential is also relevant for the ones with higher number of items because there are also users who have a lot more games.

4. PROJECT DEFINITION AND METHODOLOGY

In this project, game recommendations are made to the users by learning from their current library to estimate their possible preferences. As it is shown in the previous part, lots of users have small numbers of games, and most of the users do not play the games in their library. This circumstance can be considered as a tremendous opportunity for the Steam platform and the publishers. By suggesting correct games, it seems possible to make users buy new ones even if they do not plan to play them and increase the sales and revenue. More games will probably create more loyal customers, and eventually, the platform may benefit from this loyalty of the users.

It is not possible to give ratings to the games on the Steam platform. Users can only "Recommend" or "Not Recommend" items, and this is not an indicator of how much the item is liked or disliked. On the other hand, the platform knows how many hours a user played a specific game. Hence, implicit feedback seems more useful while determining a user's preferences.

In Section 2, two popular methods for building recommendation systems with collaborative filtering by using implicit feedback, ALS and BPR, are explained. In this project, both of these methods are used to create different models. There is a Python library, called "implicit", created by Ben Frederickson, which makes it really easy to build ALS and BPR models, to find similar games and to recommend items in really short time (Implicit Github Page, https://github.com/benfred/implicit).

ALS and BPR models have similar parameters which are as follows:

- Factors: Number of hidden factors to be discovered by the model
- Regularization: Regularization rate
- Iterations: Number of iterations
- Alpha: Parameter to adjust the level of confidence (only ALS)
- Learning Rate: Learning rate used during optimizing the model (only BPR)

In order to train models, the algorithms require a sparse item user matrix. Before creating this matrix, each username and game id is converted to numerical codes starting from 0 to the number of unique items and users. Also, it is assumed that, even if a user does not play a game, having that game in their library might indicate positive feedback.

Therefore, a constant value is added to transformed playtime values to signify this feedback. The models built require a sparse user item matrix to make recommendations to users.

The performances of the models are measured by using *Precision at k* metric. In order to measure the performance, train and test sets are derived from the cleaned and merged datasets. One train and test set is created by using the dataset with only paid games. The other one is created by using the dataset that contains both paid and free games.

Test sets contain one random item of all users. These items are excluded from the train sets. After the models built with the train sets, the models recommend games for all of the users. For instance, the recommendation engine suggests 10 items to a user. If one of these items is actually the excluded one, i.e., the item in the test set, then that recommendation batch is acknowledged as successful. These users can be labeled as correct users so to speak. *Precision at k* here is the number of correct users divided by total number of users. If there are 50 users in the dataset and *Precision at k* equals 20%, this means that the model found correct items for 10 of the users.

5. RESULTS

Using the methodology explained in the previous part, 747 different experiments have been performed to compare different parameters and algorithms with two datasets. As mentioned, one dataset contains only paid games and the other one contains both free and paid games.

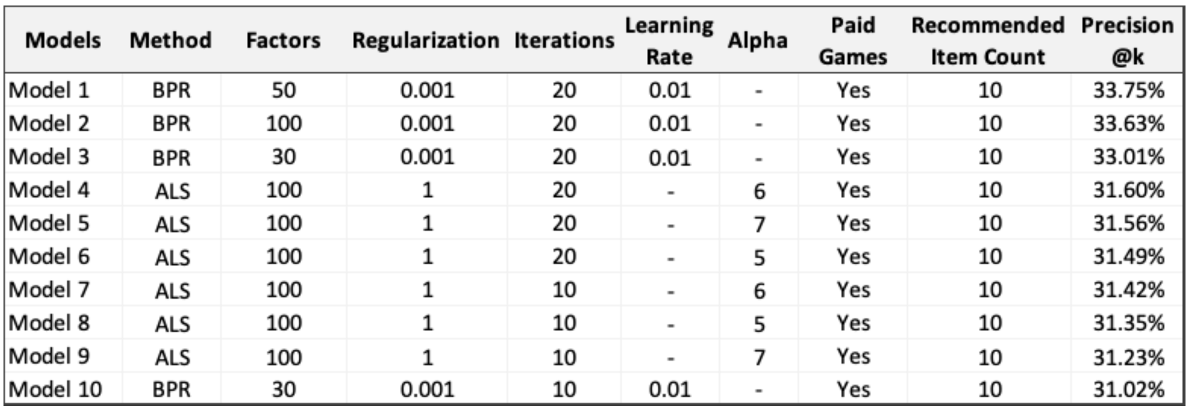
Each of the models are used to recommend games to users and their performance are defined with *Precision at k* metric. In this project, scenarios in which 5 or 10 items are recommended to the users are designed. This means k can be 5 or 10.

These experiments are completed with *implicit* version 0.4.4 and Python version 3.8.3.

5.1 Recommending Items

Among 747 experiments, models that show the best performance in terms of *Precision at k*, have the parameters and scores that can be seen in Table 1.

Table 1: Parameters and Precision at k Scores of the Best Models



Models	Method	Factors	Regularization	Iterations	Learning Rate	Alpha	Paid Games	Recommended Item Count	Precision @k
Model 1	BPR	50	0.001	20	0.01	-	Yes	10	33.75%
Model 2	BPR	100	0.001	20	0.01	-	Yes	10	33.63%
Model 3	BPR	30	0.001	20	0.01	-	Yes	10	33.01%
Model 4	ALS	100	1	20	-	6	Yes	10	31.60%
Model 5	ALS	100	1	20	-	7	Yes	10	31.56%
Model 6	ALS	100	1	20	-	5	Yes	10	31.49%
Model 7	ALS	100	1	10	-	6	Yes	10	31.42%
Model 8	ALS	100	1	10	-	5	Yes	10	31.35%
Model 9	ALS	100	1	10	-	7	Yes	10	31.23%
Model 10	BPR	30	0.001	10	0.01	-	Yes	10	31.02%

The best three models among all use the BPR algorithm. Only difference between these models is the number of factors. Model 1 and Model 2 have very close Precision values. However, using less factors would decrease the training and testing times. Therefore, if one of them should be selected, it would be Model 1 since it uses less factors to make predictions.

After those three models, ALS models show the best performance. These models have really close scores with different iteration and alpha values. Reducing iteration count also might be useful since it has a positive impact on training and testing times.

At 10th place, there is another BPR model with 30 factors and 10 iterations. Rest of the parameters are the same with other BPR models.

From the models in the list, to compare BPR and ALS, it can be said that BPR tends to require less factors, but ALS performs better with less iterations.

Table 1 reveals other significant points. All of the models above are built using only paid games. This means that including free games to the training dataset might decrease the success of the model. Actually, this case can be expected since free game definition is too wide. For instance, sometimes developers publish Beta, Demo or Early Access versions of the games. These versions usually are not completed games or just a small part of the original game. Therefore, games that suit the description above are probably not played too much, and in the model, playtime is taken into consideration.

Another important point is that recommendations with 5 items are not in the list. Again, an expected outcome, increasing the number of options will increase the probability of finding the correct game. This decision totally depends on the platform and this number can be adjusted very easily.

The precision scores of the models might vary since the test set is created randomly each time. However, this volatility would be really small which can be ignored. Therefore, these results can be assumed to represent the methods' success accurately.

Herewith, to evaluate the models as a whole, finding nearly 33% of the users' preferences could bring amazing benefits to the platform. It could boost the sales, enhance the loyalty and increase the platform's competition power with others in the market.

5.2 Finding Similar Items

Even though there are metrics to measure the success of recommendations, the topic sometimes becomes subjective. In order to understand how the model recommends games, Model 1, Model 4 and Model 7 are used to find the similar games of a given game. Model 1 is the most successful model with BPR, Model 4 is the most successful model with ALS and Model 7 is the most successful model with 10 iterations. These are the reasons behind this selection. In order to find similar games, a very popular strategy and simulation game Age of Empires II will be used. The results can be seen in Table 2, Table 3 and Table 4.

Table 2: Model 1 Similar Items

Game Name	Genre	Score	Owner Count
Age of Empires II: HD Edition	Strategy	1	8894
Age of Empires® III: Complete Collection	Simulation	0.903913	4612
Age of Empires II HD: The African Kingdoms	Strategy	0.687495	1162
Rise of Nations: Extended Edition	Simulation	0.668614	1601
Banished	Indie	0.473162	3871
Stronghold Crusader HD	Simulation	0.452918	763
Stronghold HD	Simulation	0.438628	682
Total War: ROME II - Emperor Edition	Strategy	0.429181	3660
SimCity 4 Deluxe	Simulation	0.425976	2070
Radiant Defense	Indie	0.421541	7

Table 3: Model 4 Similar Items

Game Name	Genre	Score	Owner Count
Age of Empires II: HD Edition	Strategy	1	8894
Age of Empires® III: Complete Collection	Simulation	0.922207	4612
Age of Empires II HD: The African Kingdoms	Strategy	0.79893	1162
Rise of Nations: Extended Edition	Simulation	0.503964	1601
Banished	Indie	0.32873	3871
H-Hour: World's Elite	Action	0.262713	8
Worms Clan Wars	Strategy	0.256265	1162
Supreme Commander: Forged Alliance	Strategy	0.219548	2904
Worms Revolution	Strategy	0.215995	3569
Worms Reloaded	Strategy	0.212841	2287

Table 4: Model 7 Similar Items

Game Name	Genre	Score	Owner Count
Age of Empires II: HD Edition	Strategy	1	8894
Age of Empires® III: Complete Collection	Simulation	0.803781	4612
Age of Empires II HD: The African Kingdoms	Strategy	0.789122	1162
Rise of Nations: Extended Edition	Simulation	0.507513	1601
Worms Armageddon	Strategy	0.286553	1729
Worms Clan Wars	Strategy	0.283191	1162
Banished	Indie	0.272245	3871
Worms Reloaded	Strategy	0.260656	2287
H-Hour: World's Elite	Action	0.253335	8
Supreme Commander: Forged Alliance	Strategy	0.237688	2904

First item in all of the tables is the game itself. Top 3 recommendations are the same for all models and are very successful. Two of them are the newer version and the other one is another known strategy game. Although the rest of the list differs a little bit, the game suggestions can be accepted as successful. Models usually find games from similar genres. The BPR model offers other popular games in that genre such as Stronghold Crusader and Total War: Rome II. A person who plays Age of Empires II would probably show interest in these games, too. Therefore, there is no problem to offer these games.

Also, the model suggests less popular games as well. Games with small numbers of owners are also shown as similar. In the literature review section, the problem "Cold Start" was mentioned where unpopular items are not suggested to the users. No special action is taken; however, these models offer such games as well. Nevertheless, the platform should define a strategy for games in this description to increase their recognition.

Unknown relationships between games or users might appear in detailed analyses by using the same approach applied in this section, manually analyzing counterparts of a specific game. These patterns can be used to create bundles, or this information can be provided to developers or publishers to adjust their games or marketing strategies accordingly. There is a lot to do with such an insight. Therefore, it would be really beneficial to think more comprehensively with a recommendation system.

6. CONCLUSION

Day by day more alternatives become available for consumers. In this ocean, the consumers spend considerable time to find the most suitable choices for themselves. Therefore, locating these options for the customers created a highly competitive field, recommender systems, for the companies. A successful recommendation system would ensure better position to a company in the market, as well as more sales and higher customer loyalty.

In this project, recommendation systems for the Steam platform are developed by using more than 60,000 users' game library and playtimes. The data is used to build collaborative filtering models with ALS and BPR methods. Both algorithms are used to predict games that are actually added to the libraries by the users and find similar games; and they showed considerably good performance on both operations.

The outcome of these models can be beneficial for the Steam platform to increase their sales and customer loyalty. By suggesting correct games, the platform would become an irreplaceable environment for its users. People who benefit from these recommendations would spend less time finding the games and more on playing.

As it is mentioned in the previous parts, the topic of recommendation can lead the companies to broader business areas. A strong recommendation system can be used to mine hidden information between items and users. This information might be really valuable and provide useful insights to both the platforms and publishers and may play a key role in shaping the gaming industry's future. Thus, the platform should leverage the power of such an engine.

APPENDIX A

```
#Reading .gzip Files
import pandas as pd
import numpy as np
import gzip

def parse(path):
   g = gzip.open(path, 'r')
   for l in g:
      yield eval(l)

def read_data(path):
   data = []
   gen = parse(path)
   while True:
      try:
         temp_dict = next(gen)
      except StopIteration:
         break
      data.append(temp_dict)
   return data

#First Dataset
user_items = read_data("Datasets/australian_users_items.json.gz")
item_df = pd.json_normalize(user_items, record_path = ['items'], meta = ['user_id'])
item_df = item_df[['user_id', 'item_id', 'item_name', 'playtime_forever', 'playtime_2weeks']]
item_df = item_df.drop_duplicates()
item_check = item_df.groupby(['user_id','item_id']).size().reset_index(name='counts')
item_check_list = item_check.loc[item_check.counts > 1].reset_index(drop = True)
```

```
    temp_index = []
    for index in range(0,len(item_check_list)):
      temp_index = item_df.loc[((item_df.user_id == item_check_list.loc[index,
'user_id']) &
                 (item_df.item_id == item_check_list.loc[index,
'item_id']))].index.to_list()
      temp_index.pop(-1)
      item_df = item_df.drop(temp_index)
    item_check =
item_df.groupby(['user_id','item_id']).size().reset_index(name='counts')
    item_count_check = item_df.groupby(['item_id']).size().reset_index(name =
"counts")
    exc_items = item_count_check.loc[item_count_check.counts == 1].item_id
    item_df = item_df.loc[~item_df.item_id.isin(exc_items)]
    user_count_check = item_df.groupby(['user_id']).size().reset_index(name =
"counts")
    exc_users = user_count_check.loc[user_count_check.counts == 1].user_id
    item_df = item_df.loc[~item_df.user_id.isin(exc_users)].reset_index(drop = True)
    item_df.isnull().sum()
    item_df.to_csv('Datasets/item_data.csv', index=False)

    #Second Dataset
    games = read_data("Datasets/steam_games.json.gz")
    games_df = pd.DataFrame(games)
    games_df = games_df[['id', 'app_name', 'genres','price']].rename(columns={'id':
"item_id", "app_name": "item_name"})
    games_df.isnull().sum()
    games_df = games_df.dropna().reset_index(drop = True)
    games_df.genres = games_df.genres.astype(str)
    games_df.genres = games_df.genres.str.replace('[', ")
    games_df.genres = games_df.genres.str.replace(']', ")
    games_df.genres = games_df.genres.str.replace('\", ")
    games_df = pd.concat([games_df, games_df.genres.str.split(',',expand=True)],
axis=1)
    for i in range(0, len(games_df)):
      if (games_df.loc[i, 0] != 'Free to Play') and (games_df.loc[i, 0] != 'Early Access'):
        games_df.loc[i, 'single_genre'] = games_df.loc[i, 0]
      else:
        games_df.loc[i, 'single_genre'] = games_df.loc[i, 1]
    games_df.single_genre = games_df.single_genre.str.replace(' Indie', 'Indie')
    games_df.single_genre = games_df.single_genre.str.replace(' Action', 'Action')
    games_df.single_genre = games_df.single_genre.str.replace(' Massively
Multiplayer', 'Massively Multiplayer')
    games_df.single_genre = games_df.single_genre.str.replace(' RPG', 'RPG')
    games_df.single_genre = games_df.single_genre.str.replace(' Sports', 'Sports')
    games_df.single_genre = games_df.single_genre.str.replace(' Racing', 'Racing')
    games_df.single_genre = games_df.single_genre.str.replace(' Strategy', 'Strategy')
    games_df.single_genre = games_df.single_genre.str.replace(' Simulation',
'Simulation')
    games_df = games_df.loc[~games_df.single_genre.isin(['Animation &amp;
Modeling', 'Utilities', 'Education', 'Design &amp; Illustration', 'Audio Production',
                               None, 'Video Production', 'Software Training',
'Accounting', 'Web Publishing', 'Photo Editing'])]
    games_df = games_df.drop(['genres'], axis=1)
    games_df = games_df.drop([0,1,2,3,4,5,6,7,8,9], axis=1)
    games_df = games_df.drop_duplicates()
    games_df = games_df.reset_index(drop=True)
    for i in range(0, len(games_df)):
      if type(games_df.price[i]) == str:
        games_df.price[i] = 0.00
    games_df.to_csv('Datasets/games_data.csv', index=False)

    #Third Dataset
    bundles = read_data("Datasets/bundle_data.json.gz")
```

```
bundle_df = pd.json_normalize(bundles, record_path = ['items'])
bundle_df = bundle_df[['item_id', 'item_name', 'genre', 'discounted_price']]
bundle_df = bundle_df.loc[bundle_df.item_id != '']
bundle_df = bundle_df.drop_duplicates()
bundle_df = bundle_df.drop(bundle_df.loc[bundle_df.item_id.duplicated()].index)
bundle_df = bundle_df.drop(bundle_df.loc[bundle_df.item_id.str.contains(",")].index)
bundle_df = bundle_df.reset_index(drop=True)
bundle_df = pd.concat([bundle_df, bundle_df.genre.str.split(',',expand=True)], axis=1)
for i in range(0, len(bundle_df)):
    if (bundle_df.loc[i, 0] != 'Free to Play') and (bundle_df.loc[i, 0] != 'Early Access'):
        bundle_df.loc[i, 'single_genre'] = bundle_df.loc[i, 0]
    else:
        bundle_df.loc[i, 'single_genre'] = bundle_df.loc[i, 1]
bundle_df.single_genre = bundle_df.single_genre.str.replace(' Massively Multiplayer', 'Massively Multiplayer')
bundle_df = bundle_df.loc[~bundle_df.single_genre.isin(['', 'Audio Production', 'Utilities', 'Education', 'Design & Illustration',
                                           'Software Training', 'Animation & Modeling', 'Web Publishing'])]
bundle_df.discounted_price = bundle_df.discounted_price.str.replace('$','')
bundle_df.discounted_price = pd.to_numeric(bundle_df.discounted_price)
bundle_df.item_id = pd.to_numeric(bundle_df.item_id)
bundle_df = bundle_df.drop(['genre',0,1,2,3,4,5,6,7], axis=1)
bundle_df.isnull().sum()
bundle_df.columns = ['item_id', 'item_name', 'price', 'single_genre']
bundle_df.to_csv('Datasets/bundle_data.csv', index=False)

#Creating main dataframes
raw_items = pd.read_csv('Datasets/item_data.csv')
raw_games = pd.read_csv('Datasets/games_data.csv')
```

```
raw_bundles = pd.read_csv('Datasets/bundle_data.csv')
raw_games_all = pd.concat([raw_games, raw_bundles.loc[~raw_bundles.item_id.isin(raw_games.item_id)]])
raw_items_all = raw_items.loc[raw_items.item_id.isin(raw_games_all.item_id)]
raw_items_all.columns = ['user', 'item', 'item_name', 'playtime_forever', 'playtime_2weeks']
raw_items_all = raw_items_all.merge(raw_games_all[['item_id', 'single_genre', 'price']], left_on='item', right_on='item_id', how = 'left').drop(['item_id'], axis = 1)
items_df = raw_items_all
items_df['playtime'] = items_df['playtime_forever'] + 1
items_df['playtime'] = np.log(items_df['playtime'])
items_df = items_df.drop(['playtime_2weeks'], axis=1)
item_count_check = items_df.groupby(['item']).size().reset_index(name = "counts")
exc_items = item_count_check.loc[item_count_check.counts == 1]['item']
items_df = items_df.loc[~items_df.item.isin(exc_items)]
user_count_check = items_df.groupby(['user']).size().reset_index(name = "counts")
exc_users = user_count_check.loc[user_count_check.counts == 1].user
items_df = items_df.loc[~items_df.user.isin(exc_users)].reset_index(drop = True)
items_df.isnull().sum()
items_df.to_csv('Datasets/model_data_all.csv', index=False)
items_df_paid = items_df.loc[items_df.price > 0]
item_count_check = items_df_paid.groupby(['item']).size().reset_index(name = "counts")
exc_items = item_count_check.loc[item_count_check.counts == 1]['item']
items_df_paid = items_df_paid.loc[~items_df_paid.item.isin(exc_items)]
user_count_check = items_df_paid.groupby(['user']).size().reset_index(name = "counts")
exc_users = user_count_check.loc[user_count_check.counts == 1].user
items_df_paid = items_df_paid.loc[~items_df_paid.user.isin(exc_users)].reset_index(drop = True)
items_df_paid.to_csv('Datasets/model_data_paid.csv', index=False)
```

```
#Modeling Part
model_data_all = pd.read_csv('Datasets/model_data_all.csv')
model_data_paid = pd.read_csv('Datasets/model_data_paid.csv')
import implicit
import scipy.sparse as sparse
import random
from sklearn import metrics

random.seed(0)
def prep_data(df, pt_add):
   df['playtime'] = df['playtime'] + pt_add
   df['user_id'] = df['user'].astype("category").cat.codes
   df['item_id'] = df['item'].astype("category").cat.codes
   user_item_count = df.groupby(['user_id']).size().reset_index(name = "counts")
   df = df.merge(user_item_count, on='user_id', how = 'left')
   test_df                                                                    =
df.groupby(['user_id'])['item_id'].apply(pd.Series.sample).reset_index(level=[0])
   test_df['check'] = 1
   train_df = df.loc[~df.index.isin(test_df.index)].reset_index(drop = True)
   print("Train   dataset   contains   {}   unique   items   and   {}   unique
users.".format(len(train_df.item_id.unique()), len(train_df.user_id.unique())))
   print("Train dataset total record count:", len(train_df))
   sparse_items   =   sparse.csr_matrix((train_df['playtime'],   (train_df['item_id'],
train_df['user_id'])))
   sparse_users   =   sparse.csr_matrix((train_df['playtime'],   (train_df['user_id'],
train_df['item_id'])))
   matrix_size = sparse_users.shape[0] * sparse_users.shape[1]
   owned_games = len(sparse_users.nonzero()[0])
   sparsity = 100 * (1 - (owned_games / matrix_size))
   print("The sparsity is {}%".format(round(sparsity,2)))
   return sparse_items, sparse_users, train_df, test_df
def build_als_model(sparse_item_df, factors, regularization, iterations, alpha):
        model_als = implicit.als.AlternatingLeastSquares(factors=factors,
regularization=regularization, iterations=iterations, calculate_training_loss = True,
random_state = 0)
        data = (sparse_item_df * alpha).astype('double')
        model_als.fit(data)
        return model_als

    def build_bpr_model(sparse_item_df, factors, learning_rate, regularization,
iterations, alpha):
        model_bpr = implicit.bpr.BayesianPersonalizedRanking(factors=factors,
learning_rate=learning_rate, regularization=regularization, iterations=iterations,
random_state = 0)
        data = (sparse_item_df * alpha).astype('double')
        model_bpr.fit(data)
        return model_bpr

    def show_similar_items(model, df, item_num, num_of_items):
        item_id = df.loc[df['item'] == item_num].iloc[0].item_id.item()
        similar = model.similar_items(item_id, N = num_of_items)
        items = []
        items_new = []
        itemnames = []
        genres = []
        scores = []
        for i in similar:
            idx, score = i
            game_id = df['item'].loc[df.item_id == idx].iloc[0]
            items.append(df['item'].loc[df.item_id == idx].iloc[0])
            items_new.append(df['item_id'].loc[df.item_id == idx].iloc[0])
            itemnames.append(df['item_name'].loc[df.item_id == idx].iloc[0])
            genres.append(df['single_genre'].loc[df.item_id == idx].iloc[0])
            scores.append(score)
        similars = pd.DataFrame({'item_id': items, 'new_item_id': items_new, 'item_name': itemnames, 'genre': genres, 'score': scores})
        owner_counts = df.groupby(['item']).size().reset_index(name = "owner_count")
        similars = similars.merge(owner_counts, how='left', right_on='item', left_on='item_id').drop(['item'], axis=1)
        return similars

    def make_recommendations(model, sparse_user_df, train_df, test_df, n_items):
        user_ids = []
        items_ids = []
        scores = []
        for u in train_df.user_id.unique():
          recommended = model.recommend(u, sparse_user_df, filter_already_liked_items = True, N = n_items, recalculate_user = False)
          for rec_pair in recommended:
            item_id, score = rec_pair
            user_ids.append(u)
            items_ids.append(item_id)
            scores.append(score)
        recommendations = pd.DataFrame({'user_id': user_ids, 'item_id': items_ids, 'score': scores})
        results = recommendations.merge(test_df, how='left', on=['user_id', 'item_id'])
        results.loc[results.check.isna(), 'check'] = 0
        precision = len(results.loc[results.check == 1]) / len(recommendations.user_id.unique())
        print("Precision @{}: {}%\n".format(n_items, round(precision * 100, 2)))
        results_2 = results.groupby(['user_id']).agg({"check": "max"}).merge(train_df[['user_id', 'counts']].drop_duplicates(), how='left', on=['user_id'])
        return results, results_2, precision

    sparse_item_df, sparse_user_df, train_set, test_set = prep_data(model_data_all, 0.01)
    sparse_item_df_paid, sparse_user_df_paid, train_set_paid, test_set_paid =
prep_data(model_data_paid, 0.01)

    #ALS Models
    iter_count = 0
    models = []
    model_names = []
    precisions = []
    for facs in [10,30,50,100]:
      for regs in [0.01, 0.1, 0.5, 1]:
        for iters in [5, 10, 20]:
          for alphas in [15, 30, 40]:
            model = build_als_model(sparse_item_df, factors = facs, regularization =
regs, iterations = iters, alpha = alphas)
            print("Factors: {} - Regularization: {} - Iterations: {} - Alpha:
{}".format(facs, regs, iters, alphas))
            result_df, prec = make_recommendations(model, sparse_user_df,
train_set, test_set, 10)
            models.append(model)
            model_names.append('model_als_' + str(iter_count))
            precisions.append(prec)
            iter_count += 1

    iter_count = 0
    models_als_5 = []
    model_names_als_5 = []
    precisions_als_5 = []
    for facs in [10,30,50,100]:
      for regs in [0.01, 0.1, 0.5, 1]:
        for iters in [5, 10, 20]:
          for alphas in [15, 30, 40]:
```

```
          model = build_als_model(sparse_item_df, factors = facs, regularization =
regs, iterations = iters, alpha = alphas)
          print("Factors: {} - Regularization: {} - Iterations: {} - Alpha:
{}".format(facs, regs, iters, alphas))
          result_df, prec = make_recommendations(model, sparse_user_df,
train_set, test_set, 5)
          models_als_5.append(model)
          model_names_als_5.append('model_als_' + str(iter_count))
          precisions_als_5.append(prec)
          iter_count += 1

  iter_count = 0
  models_als_10_paid = []
  model_names_als_10_paid = []
  precisions_als_10_paid = []
  for facs in [10,30,50,100]:
    for regs in [0.01, 0.1, 0.5, 1]:
      for iters in [5, 10, 20]:
        for alphas in [15, 30, 40]:
          model = build_als_model(sparse_item_df_paid, factors = facs,
regularization = regs, iterations = iters, alpha = alphas)
          print("Factors: {} - Regularization: {} - Iterations: {} - Alpha:
{}".format(facs, regs, iters, alphas))
          result_df, prec = make_recommendations(model, sparse_user_df_paid,
train_set_paid, test_set_paid, 10)
          models_als_10_paid.append(model)
          model_names_als_10_paid.append('model_als_' + str(iter_count))
          precisions_als_10_paid.append(prec)
          iter_count += 1

  iter_count = 0
  models_als_10_alpha = []
```

```
model_names_als_10_alpha = []
precisions_als_10_alpha = []
alphas_als_10_alpha = []
for facs in [50,100]:
    for regs in [0.01, 0.1, 1]:
        for iters in [10, 20]:
            for alphas in [5, 6, 7]:
                model = build_als_model(sparse_item_df, factors = facs, regularization = regs, iterations = iters, alpha = alphas)
                result_df, result_df_2, prec = make_recommendations(model, sparse_user_df, train_set, test_set, 10)
                models_als_10_alpha.append(model)
                model_names_als_10_alpha.append('model_als_alpha' + str(iter_count))
                precisions_als_10_alpha.append(prec)
                alphas_als_10_alpha.append(alphas)
                iter_count += 1

iter_count = 0
models_als_10_paid_alpha = []
model_names_als_10_paid_alpha = []
precisions_als_10_paid_alpha = []
alphas_als_10_paid_alpha = []
for facs in [50,100]:
    for regs in [0.01, 0.1, 1]:
        for iters in [10, 20]:
            for alphas in [5, 6, 7]:
                model = build_als_model(sparse_item_df_paid, factors = facs, regularization = regs, iterations = iters, alpha = alphas)
                result_df, result_df_2, prec = make_recommendations(model, sparse_user_df_paid, train_set_paid, test_set_paid, 10)
                models_als_10_paid_alpha.append(model)
                model_names_als_10_paid_alpha.append('model_als_paid_alpha' +
str(iter_count))
                precisions_als_10_paid_alpha.append(prec)
                alphas_als_10_paid_alpha.append(alphas)
                iter_count += 1

        als_10_outputs = pd.DataFrame({'model_names': model_names, 'models': models,
'precisions': precisions})
        als_5_outputs = pd.DataFrame({'model_names': model_names_als_5, 'models':
models_als_5, 'precisions': precisions_als_5})
        als_10_paid_outputs = pd.DataFrame({'model_names': model_names_als_10_paid,
'models': models_als_10_paid, 'precisions': precisions_als_10_paid})
        als_10_alpha_outputs = pd.DataFrame({'model_names':
model_names_als_10_alpha, 'models': models_als_10_alpha,
                              'precisions': precisions_als_10_alpha, 'alphas':
alphas_als_10_alpha})
        als_10_paid_alpha_outputs = pd.DataFrame({'model_names':
model_names_als_10_paid_alpha, 'models': models_als_10_paid_alpha,
                              'precisions': precisions_als_10_paid_alpha, 'alphas':
alphas_als_10_paid_alpha})

        als_10_outputs['factors'] = 0
        als_10_outputs['regularization'] = 0.0
        als_10_outputs['iterations'] = 0

        als_5_outputs['factors'] = 0
        als_5_outputs['regularization'] = 0.0
        als_5_outputs['iterations'] = 0

        als_10_paid_outputs['factors'] = 0
        als_10_paid_outputs['regularization'] = 0.0
        als_10_paid_outputs['iterations'] = 0
```

```
als_10_alpha_outputs['factors'] = 0
als_10_alpha_outputs['regularization'] = 0.0
als_10_alpha_outputs['iterations'] = 0

als_10_paid_alpha_outputs['factors'] = 0
als_10_paid_alpha_outputs['regularization'] = 0.0
als_10_paid_alpha_outputs['iterations'] = 0

for i in range(0, len(als_10_outputs)):
    als_10_outputs['factors'][i] = als_10_outputs.models[i].factors
    als_10_outputs['regularization'][i] = als_10_outputs.models[i].regularization
    als_10_outputs['iterations'][i] = als_10_outputs.models[i].iterations

for i in range(0, len(als_5_outputs)):
    als_5_outputs['factors'][i] = als_5_outputs.models[i].factors
    als_5_outputs['regularization'][i] = als_5_outputs.models[i].regularization
    als_5_outputs['iterations'][i] = als_5_outputs.models[i].iterations

for i in range(0, len(als_10_paid_outputs)):
    als_10_paid_outputs['factors'][i] = als_10_paid_outputs.models[i].factors
    als_10_paid_outputs['regularization'][i] =
als_10_paid_outputs.models[i].regularization
    als_10_paid_outputs['iterations'][i] = als_10_paid_outputs.models[i].iterations

for i in range(0, len(als_10_alpha_outputs)):
    als_10_alpha_outputs['factors'][i] = als_10_alpha_outputs.models[i].factors
    als_10_alpha_outputs['regularization'][i] =
als_10_alpha_outputs.models[i].regularization
    als_10_alpha_outputs['iterations'][i] = als_10_alpha_outputs.models[i].iterations

for i in range(0, len(als_10_paid_alpha_outputs)):
```

```
            als_10_paid_alpha_outputs['factors'][i]                                           =
als_10_paid_alpha_outputs.models[i].factors
            als_10_paid_alpha_outputs['regularization'][i]                                    =
als_10_paid_alpha_outputs.models[i].regularization
            als_10_paid_alpha_outputs['iterations'][i]                                        =
als_10_paid_alpha_outputs.models[i].iterations

        #BPR Models
        iter_count = 0
        models_bpr_10 = []
        model_names_bpr_10 = []
        precisions_bpr_10 = []
        for facs in [30,50,100]:
           for lr in [0.001, 0.01, 0.1]:
              for regs in [0.001, 0.01, 0.05]:
                 for iters in [5, 10, 20]:
                    model = build_bpr_model(sparse_item_df, factors = facs, learning_rate=
lr, regularization = regs, iterations = iters, alpha = 15)
                    print("Factors: {} - Learning Rate: {} - Regularization: {} - Iterations: {}
- Alpha: 15".format(facs, lr, regs, iters))
                    result_df,    prec    =    make_recommendations(model,    sparse_user_df,
train_set, test_set, 10)
                    models_bpr_10.append(model)
                    model_names_bpr_10.append('model_bpr_' + str(iter_count))
                    precisions_bpr_10.append(prec)
                    iter_count += 1

        iter_count = 0
        models_bpr_5 = []
        model_names_bpr_5 = []
        precisions_bpr_5 = []
        for facs in [30,50,100]:
```

```
        for lr in [0.001, 0.01, 0.1]:
            for regs in [0.001, 0.01, 0.05]:
                for iters in [5, 10, 20]:
                    model = build_bpr_model(sparse_item_df, factors = facs, learning_rate= lr, regularization = regs, iterations = iters, alpha = 15)
                    print("Factors: {} - Learning Rate: {} - Regularization: {} - Iterations: {} - Alpha: 15".format(facs, lr, regs, iters))
                    result_df, prec = make_recommendations(model, sparse_user_df, train_set, test_set, 5)
                    models_bpr_5.append(model)
                    model_names_bpr_5.append('model_bpr_' + str(iter_count))
                    precisions_bpr_5.append(prec)
                    iter_count += 1


    iter_count = 0
    models_bpr_10_paid = []
    model_names_bpr_10_paid = []
    precisions_bpr_10_paid = []
    for facs in [30,50,100]:
        for lr in [0.001, 0.01, 0.1]:
            for regs in [0.001, 0.01, 0.05]:
                for iters in [5, 10, 20]:
                    model = build_bpr_model(sparse_item_df_paid, factors = facs, learning_rate= lr, regularization = regs, iterations = iters, alpha = 15)
                    print("Factors: {} - Learning Rate: {} - Regularization: {} - Iterations: {} - Alpha: 15".format(facs, lr, regs, iters))
                    result_df, prec = make_recommendations(model, sparse_user_df_paid, train_set_paid, test_set_paid, 10)
                    models_bpr_10_paid.append(model)
                    model_names_bpr_10_paid.append('model_bpr_' + str(iter_count))
                    precisions_bpr_10_paid.append(prec)
                    iter_count += 1
```

```
bpr_10_outputs = pd.DataFrame({'model_names': model_names_bpr_10, 'models': models_bpr_10, 'precisions': precisions_bpr_10})
bpr_10_outputs['factors'] = 0
bpr_10_outputs['regularization'] = 0.0
bpr_10_outputs['learning_rate'] = 0.0
bpr_10_outputs['iterations'] = 0

for i in range(0, len(bpr_10_outputs)):
   bpr_10_outputs['factors'][i] = bpr_10_outputs.models[i].factors
   bpr_10_outputs['learning_rate'][i] = bpr_10_outputs.models[i].learning_rate
   bpr_10_outputs['regularization'][i] = bpr_10_outputs.models[i].regularization
   bpr_10_outputs['iterations'][i] = bpr_10_outputs.models[i].iterations

bpr_5_outputs = pd.DataFrame({'model_names': model_names_bpr_5, 'models': models_bpr_5, 'precisions': precisions_bpr_5})
bpr_5_outputs['factors'] = 0
bpr_5_outputs['regularization'] = 0.0
bpr_5_outputs['learning_rate'] = 0.0
bpr_5_outputs['iterations'] = 0

for i in range(0, len(bpr_5_outputs)):
   bpr_5_outputs['factors'][i] = bpr_5_outputs.models[i].factors
   bpr_5_outputs['learning_rate'][i] = bpr_5_outputs.models[i].learning_rate
   bpr_5_outputs['regularization'][i] = bpr_5_outputs.models[i].regularization
   bpr_5_outputs['iterations'][i] = bpr_5_outputs.models[i].iterations

bpr_10_paid_outputs = pd.DataFrame({'model_names': model_names_bpr_10_paid, 'models': models_bpr_10_paid, 'precisions': precisions_bpr_10_paid})
bpr_10_paid_outputs['factors'] = 0
bpr_10_paid_outputs['regularization'] = 0.0
```

```
bpr_10_paid_outputs['learning_rate'] = 0.0
bpr_10_paid_outputs['iterations'] = 0

for i in range(0, len(bpr_10_paid_outputs)):
  bpr_10_paid_outputs['factors'][i] = bpr_10_paid_outputs.models[i].factors
  bpr_10_paid_outputs['learning_rate'][i] =
bpr_10_paid_outputs.models[i].learning_rate
  bpr_10_paid_outputs['regularization'][i] =
bpr_10_paid_outputs.models[i].regularization
  bpr_10_paid_outputs['iterations'][i] = bpr_10_paid_outputs.models[i].iterations

model_outputs = pd.concat([als_10_outputs, als_5_outputs, als_10_paid_outputs,
bpr_10_outputs,
                bpr_5_outputs, bpr_10_paid_outputs, als_10_paid_alpha_outputs,
als_10_alpha_outputs]).reset_index(drop = True)

#Best 10 model
model_outputs.sort_values(by = 'precisions', ascending=False).head(10)

#Model 1 - Similars
show_similar_items(model_outputs.models.iloc[632], model_data_paid, 221380,
10)

#Model 4 - Similars
show_similar_items(model_outputs.models.iloc[709], model_data_paid, 221380,
10)

#Model 7 - Similars
show_similar_items(model_outputs.models.iloc[706], model_data_paid, 221380,
10)
```

REFERENCES

Adomavicius, G., & Tuzhilin, A. (2005). Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering, 17(6), 734–749. https://doi.org/10.1109/tkde.2005.99

Ahmad Kamal, A. S., Saaidin, S., & Kassim, M. (2020). Recommender System: Rating predictions of Steam Games Based on Genre and Topic Modelling. 2020 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS). https://doi.org/10.1109/i2cacis49202.2020.9140194

Burke, R. (2002). User Modeling and User-Adapted Interaction, 12(4), 331–370. https://doi.org/10.1023/a:1021240730564

Cheuque, G., Guzmán, J., & Parra, D. (2019). Recommender Systems for Online Video Game Platforms: the Case of STEAM. Companion Proceedings of The 2019 World Wide Web Conference. https://doi.org/10.1145/3308560.3316457

Fan, J., Pan, W., & Jiang, L. (2014). An improved collaborative filtering algorithm combining content-based algorithm and user activity. 2014 International Conference on Big Data and Smart Computing (BIGCOMP). https://doi.org/10.1109/bigcomp.2014.6741413

Fenlon, W. (2019, January 14). Steam now has 90 million monthly users. pcgamer. https://www.pcgamer.com/steam-now-has-90-million-monthly-users/.

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. Communications of the ACM, 35(12), 61–70. https://doi.org/10.1145/138859.138867

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative Filtering for Implicit Feedback Datasets. 2008 Eighth IEEE International Conference on Data Mining. https://doi.org/10.1109/icdm.2008.22

Implicit Github Page. Fast Python Collaborative Filtering for Implicit Feedback Datasets. Last Reached 03/August/2021. https://github.com/benfred/implicit

Jung, K.-Y., Park, D.-H., & Lee, J.-H. (2004). Hybrid Collaborative Filtering and Content-Based Filtering for Improved Recommender System. Computational Science - ICCS 2004, 295–302. https://doi.org/10.1007/978-3-540-24685-5_37

Kang, W.-C., & McAuley, J. (2018). Self-Attentive Sequential Recommendation. 2018 IEEE International Conference on Data Mining (ICDM). https://doi.org/10.1109/icdm.2018.00035

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix Factorization Techniques for Recommender Systems. Computer, 42(8), 30–37. https://doi.org/10.1109/mc.2009.263

Krichene, W., & Rendle, S. (2020). On Sampled Metrics for Item Recommendation. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. https://doi.org/10.1145/3394486.3403226

Pathak, A., Gupta, K., & McAuley, J. (2017). Generating and Personalizing Bundle Recommendations on Steam. Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval. https://doi.org/10.1145/3077136.3080724

Plunkett, B., Lin, B., Shi, S., & Painter, C. The Steam Engine: A Recommendation System for Steam Users. Retrieved July 12, 2021, from http://brandonlin.com/steam.pdf

Reddy, M. M., Kanmani, R. S., & Surendiran, D. B. (2020). Analysis of Movie Recommendation Systems; with and without considering the low rated movies. 2020 International Conference on Emerging Trends in Information Technology and Engineering (Ic-ETITE). https://doi.org/10.1109/ic-etite47903.2020.453

Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2012). BPR: Bayesian personalized ranking from implicit feedback. arXiv preprint arXiv:1205.2618.

Saaidin, S., & Kasiran, Z. (2021). Playtime - based vs Price-based Rating in Video Games Recommender System. 2021 IEEE 11th IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE). https://doi.org/10.1109/iscaie51753.2021.9431802

Steam Dataset. Recommender Systems and Personalization Datasets. Steam Video Game and Bundle Data. Last Reached 20/June/2021. https://cseweb.ucsd.edu/~jmcauley/datasets.html#steam_data

Steam Website. Steam. Last Reached 01/August/2021. https://store.steampowered.com

Victor. (2017, August 23). ALS Implicit Collaborative Filtering. Medium. https://medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe.

Viljanen, M., Vahlo, J., Koponen, A., & Pahikkala, T. (2020). Content Based Player and Game Interaction Model for Game Recommendation in the Cold Start setting. arXiv preprint arXiv:2009.08947.

Wan, M., & McAuley, J. (2018). Item recommendation on monotonic behavior chains. Proceedings of the 12th ACM Conference on Recommender Systems. https://doi.org/10.1145/3240323.3240369

Wu, L., He, X., Wang, X., Zhang, K., & Wang, M. (2021). A Survey on Neural Recommendation: From Collaborative Filtering to Content and Context Enriched Recommendation. arXiv preprint arXiv:2104.13030.