Personalized Game Recommendation System for Steam

Raghav Jajodia (rj1408) Bala Venkata Nithish Addepalli (bva212) Ieshan Vaidya (iav225)

Center for Data Science, New York University

Abstract

Steam is the largest digital distribution platform for PC gaming which houses games from huge publishers as well as from indie developers. A game recommendation system can increase the commercial revenue of the platform as well as connect gamers to niche games. In this project, we build and evaluate multiple recommendation models based on a rating implicitly derived from the playtime of a game. We developed two types of recommendation models that improve the same business objective in different ways. We can improve these models further by collecting more data as well as incorporating additional information like genre, publisher, price. We also present our analysis on data understanding and preparation to build these models.

1 Business Understanding

Steam is a digital distribution platform and marketplace where users can purchase and play games. Our goal is to develop a recommendation system which suggests top-k unplayed games to a user based on the user's personal gaming history and the preferences of similar users. This can improve sales by increasing conversion rates and decreasing the spends on marketing and promotional activities. This also attracts more users because good recommendation systems tend to increase users' engagement on the platform.

From a business perspective, we can set two different goals that a game recommendation system should achieve. One goal is to recommend top-k games to a user in decreasing order of liking. This implies that user is likely to rate i^{th} game higher than j^{th} game, if i^{th} game appears first in the recommendation of top-k games. Such a recommendation system places emphasis on the likings of the user and encourages a user to buy the top-k games thus increasing the business revenue. Recommendations from this model belong to section 'Games you might like'.

The second goal is to recommend top-k games in the order of confidence that user will buy the game. This means that if i^{th} game is ranked higher than the j^{th} game, then we are more confident that the user will buy the i^{th} game. Such a recommendation system places emphasis on whether or not a user buys a game or not, and hence recommendations from this model belong to section 'Other users bought these items'. This again leads to an increase in revenue.

While both the goals look similar, we treat them as separate business problems and build our target variable differently for them. In the first case, we build continuous rating models with the target variable being a **continuous** rating r_{ui} (rating of i^{th} game by the u^{th} user) while in the second case, we build **binary** models where with the target variable being $r_{ui} \in \{0, 1\}$ signifying whether user u has bought the game i. In data science terminology, the goal of the continuous case is to predict $\hat{r}_{ui}$ for any user u and $\forall\ i \in$ items and select top-k ratings. The goal of the binary case is to predict $\forall\ i \in$ items, the probability $\Pr(\text{Buy}_{ui} = 1)$ of the user u buying the game i and selecting the top-k probabilities for recommendation.

We model both the continuous and binary cases and present our results with appropriate evaluation metrics in this report.

2 Data Understanding

2.1 Data Collection

Steam provides a Web API interface [1] to collect game-level and user-level data. Every user on the platform is assigned a unique SteamID and similarly every application is assigned a unique AppID. These API calls return rich user-level and game-level features like game playtime of users, friends linked to a user, genre of the game, publisher of the game etc.

To build a recommendation system, we require well-defined ratings r_{ui}. However, Steam does not maintain explicit ratings for its content. Given that game playtime is usually a good indicator of a user's engagement in a game, we decide to use game playtime feature as an implicit rating. For the binary case, we require data indicating whether the user has bought the game or not. A playtime value of non-zero is equivalent to the user buying the game and thus we can derive the binary target

variable from the implicit ratings.

With the API call GetOwnedGames which takes SteamID as input, we collect the number of games owned by the user along with the playtime of every game owned by the user. However, there is no API call to collect SteamIDs and thus we resorted to scraping SteamIDs from a large discussion group on the Steam platform called 'Steam Universe' [2]. Naturally the users on this group don't represent the full population and we discuss introduction of biases in the next section. We scraped 50000 unique SteamIDs from the group webpage and then extracted the playtime data from the API.

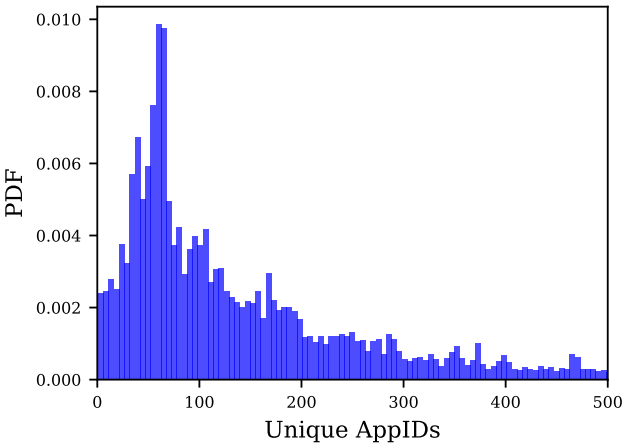

Figure 1: Distribution of AppIDs per user

2.2 Selection Bias

Steam Universe is a large group of Steam users for discussion about SteamOS and hardware. Given that users who subscribe to such groups are not casual gamers, we expected the user data to consist of dedicated gamers. Another source of bias that can be introduced is the group's preference to a specific choice of game genre or style which will skew predictions to such games. Since this is a discussion group for hardware, we do not expect any genre-preferential bias introduced here. However, testing for that is difficult. Another possible bias is region specificity. Steam is a digital platform and naturally a large percentage of the users on it are from the United States and thus there is demographic bias. While we don't collect location data of users, we expect the user-demographic of our collected data to be skewed towards the United States.

2.3 Data Analysis and Feature Understanding

An important characteristic for recommendation system data is the density / sparsity of the user x item rating matrix R_{ui}. A dense matrix carries more information that can used for recommendation. Given that a user is very unlikely to interact with a lot of games, we expect our matrix to be sparse. Figure 1 shows a distribution of unique AppIDs per user.

We observe that most of the users have few games in the library compared to the total unique AppIDs indicating that the matrix is sparse. We also observed the playtime distribution of most AppIDs is skewed to the left (with more users having a relatively smaller playtime). Figure 2 shows one such distribution.

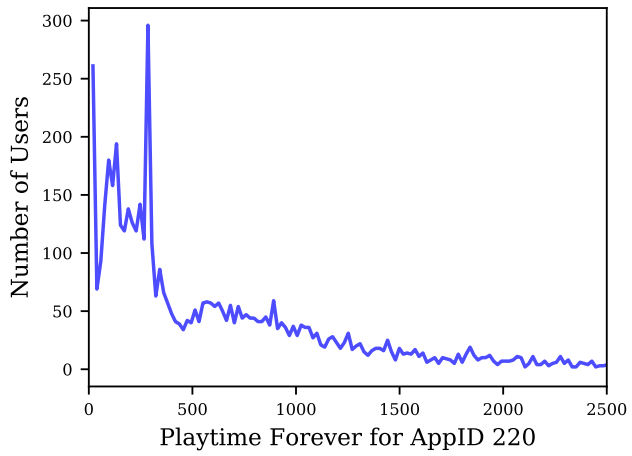

Figure 2: Playtime distribution for AppID 220

Figure 3 show the corresponding cumulative distribution function of the playtime for AppID 220.

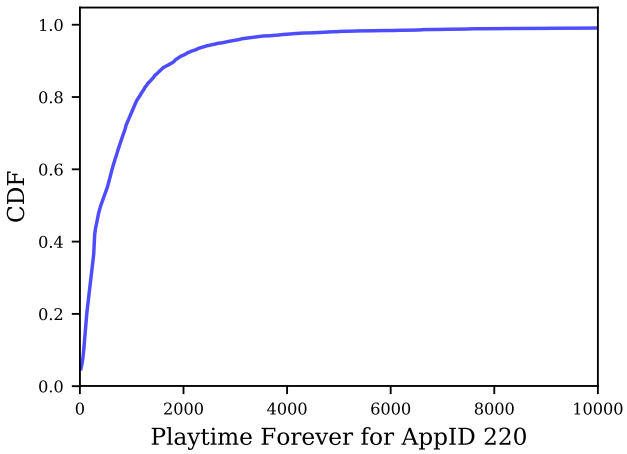

Figure 3: Playtime empirical CDF for AppID 220

The empirical cumulative distribution function also allows us to map any arbitrary playtime distribution to a 0-1 scale. We make use of this to construct our continuous rating scale. Additionally, this allows to bring games with diverse playtime distributions on the same footing.

3 Data Preparation

3.1 Data Cleaning

The dataset fetched by scraping the API had two issues :

1. Empty data - API call returned empty dictionary
2. Zero game count data - API call returned zero games owned

We remove such entries since they are non-informative from a modeling perspective (cold start problem - since model can't learn about a user who hasn't played any games). Removing such entries naturally omitted a lot of SteamIDs and left us with data in the form of tuples (SteamID, AppID, Playtime) where playtime is non-negative. We also decided to treat zero-playtime SteamID-AppID pair as equivalent to the user not having that game in his/her library and thus remove them from the data. This also helps us maintain smaller sized data without losing any information. This cleaning resulted in a user x item matrix with density 0.99%.

Sampling: We assume that games with a small playerbase are not useful for recommendation and thus discard them. Similarly, we assume that players that only small number of games in the library are also non-informative and we discard them. Also, this helps us in increasing the density of the matrix without losing much of the original data. Broadly, we define two criterias based on which we discard any data

1. Discard an AppID if it has less than c_1 unique players
2. Discard a SteamID if it has less than c_2 unique games played

We determine cutoffs c_1 and c_2 by the distribution of SteamID counts and AppID counts respectively. To determine c1, we plot the number of AppIDs against unique SteamIDs shown in Figure 4.

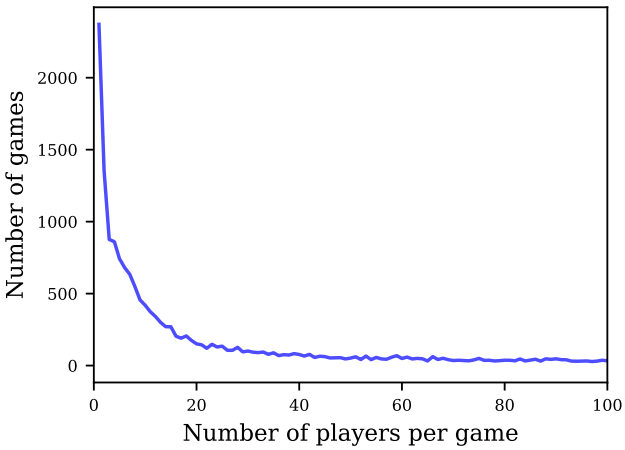

Figure 4: Data point (m, n) corresponds to there being n games that have exactly m players

We target increasing density without losing too much information. Based on the Figure 4, we choose the cutoff $c_1 = 10$ that best achieves this purpose. By removing these sparse games, the density of the new data improved to 1.65%.

To determine the cutoff c_2, we plot the distribution of AppID counts in Figure 5.

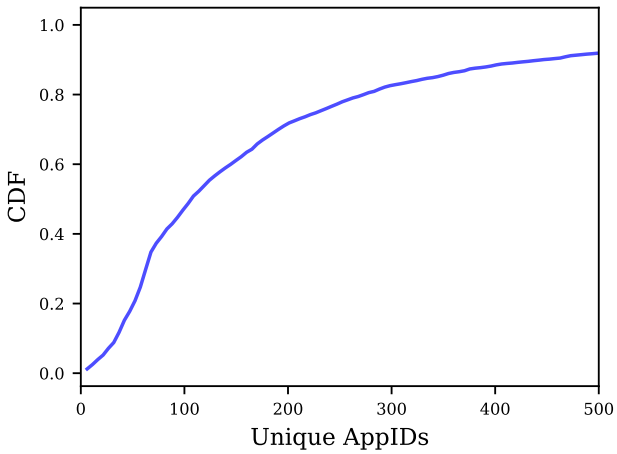

Figure 5: Empirical CDF of AppID counts (x-axis is cutoff at 500)

We determine c_2 to be 65, the reason being that the distribution of AppID counts starts to increase rapidly af-

ter 65 (users with number of unique games less than 65 are rare). We want to discard such users because they are quite sparse compared to the rest of the data and result in negligible data loss but noticeably increase the density. After this data cleaning process, we are left with 13,845 unique SteamIDs and 12,982 unique AppIDs with a density of 1.84 %. We use this data to build our recommender models.

3.2 Train Test Split

To tune the hyper-parameters and evaluate a model, we need to divide the data into train, validation and test sets. To ensure that every user has a fair representative of its data in the sets, we perform the splits in a leave-p-out fashion. Concretely, for every user we randomly sample 60 % of its data into train, 20 % in validation and the remaining 20 % in test. This way of performing the splits ensures that such scenarios do not occur where a user does not appear in train but does appear in test.

3.3 Feature Engineering

We observed that there is a lot of variation of playtime among different games. Given the diversity of games in terms of genres and game content, we expect such variation. For example, a story-based game which is typically completed in 20 hours will have its playtime distribution on a different scale to match-up based games where there is technically no completion time. To be able to treat such diverse games on the same footing, we transform the playtime to the same scale by using the empirical cumulative distribution function of playtime. For each game, we calculate empirical cumulative distribution of playtime and and transform it to 0-1 scale by evaluating it at that point. Empirical cdf is defined as

$$F_j(x) = \frac{1}{N} \sum_{x_i \in R_j} I_{\{x_i \leq x\}} \tag{1}$$

where R_j is the j^{th} column of R (corresponding to j^{th} app) and N is the number of non negative entries in R_j

For example, if the playtime of appid 220 ranges from 1 minute to 10000 minutes, then we will convert 1 minute to value 0 and 10,000 minute to value 1 (See the figure - CDF of PlaytimeForever for Appid 220)

4 Modeling and Evaluation

4.1 Recommendation Models

As discussed in section 1, typically recommendation models predict $\hat{r}_{ui}$ (rating that user u might give to item i), and then we pick top k $\hat{r}_{ui}$ for user u. Most popular techniques to do this include collaborative filtering based on user-user interactions, item-item interactions or hybrid approaches. In user-user collaborative filtering method, some popular techniques include finding Similarity between every pair of users, using methods like Cosine Similarity/Pearson Correlation, and then deriving the ratings for a particular user by aggregating the ratings from neighborhood. Other methods rely on matrix factorization, which we discuss in detail later. In the subsequent sections, we discuss a few continuous as well as binary models based on the techniques mentioned above.

4.1.1 Continuous Models

Baseline Model
We predict $\hat{r}_{ui}$ = average rating of game i as a baseline model. This is a fast and simple model which can be used to compare the performance of other matrix based factorization methods.

Collaborative Filtering
In these models, we make recommendations to a user based on the games played by similar users and select top-k games with highest similarity scores. We first compute the similarities between the users by methods like Pearson Correlation method, Cosine Similarity or Euclidean Distance between users.

The similarity score s_{ui} for a user-game pair is given by summing up the product of similarity with user u and rating for item i by all users. We then normalize these scores to obtain the final score based on which we pick the top-k items for recommendations. Normalized similarity score for a given u_{th} user and i^{th} item is given by

$$\text{Normalized Similarity Score}_{ui} = \frac{\sum_{u \in users} s_u * r_{ui}}{\sum_{u \in users} s_u} \tag{2}$$

where the term s_u refers to the similarity of the given user with the u_{th} user.

We can take the top-k items by the normalized similarity score and recommend to the user.

Matrix Factorization
The goal is to decompose user x item rating matrix R into two matrices P (users x latent factors) and Q (latent factors x items).We also assume that users and games have inherent bias (b_u and b_i respectively) and we treat this as parameter in our model. The equation for predicted rating becomes-

$$\hat{r_{ui}} = \mu + b_u + b_i + q_i^T p_u \tag{3}$$

where μ is global mean, b_u is bias of u^{th} user, b_i is bias of i^{th} item, q_i and p_u are i^{th} column and u^{th} row of Q and P matrix respectively. These parameters are obtained by minimizing the following loss function

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda(b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2) \tag{4}$$

λ being the regularization parameter.

We use SGD as optimization technique to find optimal parameters. We use surprise library [3] for the implementation of the model.

4.1.2 Binary Models

Baseline Models
For the binary case, we build two simple baseline models :

1. Recommend the top-k games based on total playtime
2. Recommend the top-k games based on total users

Collaborative Filtering
In extension to what has been discussed earlier, in continuous models subsection 4.1.1, we build the model in a similar way for the binary case where the term r_{ui} will be binary (indicating whether the person has played the game). We use Pearson Correlation and Cosine Similarity methods to compute the similarity scores.

We can also constrain the procedure of computing similarity scores by taking a minimum correlation/similarity threshold to improve results at times. We will discuss more about this and show the results of such models in detail in later sections.

Implicit Matrix Factorization
The idea behind implicit matrix factorization [4] is to trust the ratings with larger playtime more than the ratings with smaller playtime. We are more confident that a user likes a game if user has invested a lot of hours into that game and vice-versa. This is done by introducing a confidence term c_{ui} to the loss function (identical to the one discussed in continuous matrix factorization model without the bias and global mean terms) that adds a weight to the error term $(\hat{r}_{ui} - r_{ui})^2$ proportional to the playtime. We use the same confidence terms (linear and logarithmic) introduced by the authors which are defined as

$$c_{ui} = \begin{cases} 1 + \alpha \text{ (playtime)} & \text{Linear} \\ 1 + \alpha \log\left(1 + \frac{\text{playtime}}{\epsilon}\right) & \text{Logarithmic} \end{cases} \tag{5}$$

The loss function is optimized by alternating least squares method which provides a closed form update rule for the two parameters p_u and q_i.

4.2 Metrics

Continuous
We evaluate continuous models on RMSE (Root mean square error) and MAE (Mean absolute error). This is a good choice of metrics because ratings can range from 0-1, and RMSE/MAE are less if predicted ratings are close to actual ratings and vice versa.

Binary
A natural choice of metric for the binary case is precision@k which shows how good the top-k recommendations are. Precision@k is the proportion of recommended items in the top-k that are relevant. Here, relevant implies that the user has played the game. A high score of precision@k indicates that most of our recommendations are relevant which is inline with our objectives. We also consider recall@k to track the ratio of relevant recommendations we manage to make. The definitions of precision@k and recall@k are [5]:

$$\begin{aligned} \text{Precision@}k &= \frac{\#\text{ of recommended items that are relevant}}{\#\text{ of recommended items}} \\ \text{Recall@}k &= \frac{\#\text{ of recommended items that are relevant}}{\text{Total }\#\text{ of relevant items}} \end{aligned} \tag{6}$$

The denominator of recall@k is independent of k and thus we expect recall@k to increase as k increases. Similarly, we expect precision@k to decrease as k increases. We observe these trends in all the binary models trained.

4.3 Performance and Evaluation

4.3.1 Continuous Models

Baseline
The baseline model is to predict the mean game playtime of a game for the user as expected rating. For this case, we get RMSE = 0.2871

Collaborative Filtering
We implemented a collaborative filtering model for the continuous target variable using Cosine similarity method. We've observed an RMSE of 0.266 which is slightly better than the Baseline model.

Matrix Factorization
For matrix factorization method, we tune the hyper-parameters of the model with validation set. The hyper-parameters of the model are - (epochs, latent factors, learning rate and λ). By doing grid search on the space of these parameters, we get best RMSE of 0.2526 with the following hyper-parameters

$$\begin{aligned} \text{latent factors} &= 20 \\ \text{epochs} &= 20 \\ \text{learning rate} &= 0.05 \\ \lambda &= 0.02 \end{aligned} \tag{7}$$

Although this is not a significant improvement over baseline, we are able to show that this model performs better than baseline and cosine similarity based models.

We note the variation of each of the parameter on RMSE and MAE in following analyses.

We observe that Train and Test RMSE decreases with epochs, but the graph become almost flat on very high epochs $\sim$ 200 as seen in Figure 6

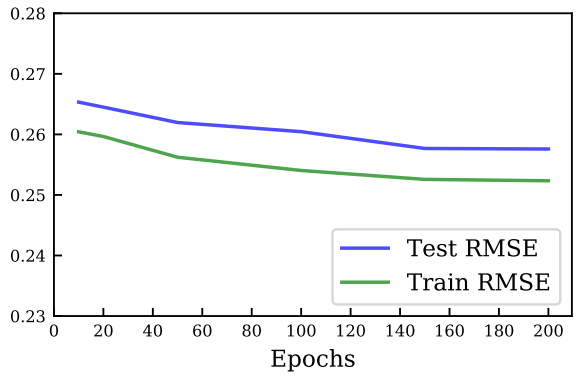

Figure 6: Variation of RMSE with epochs

We do not observe improvement in RMSE on increasing the latent factors. Infact, RMSE degrades slightly with increase in latent factors. This could possibly be because of over-fitting. Latent factors = 10 gives best results as seen in Figure 7.

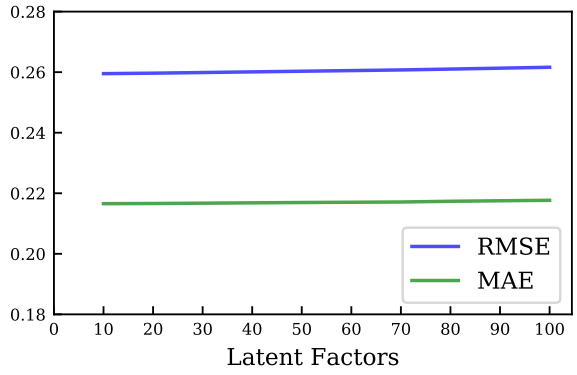

Figure 7: Variation of RMSE and MAE with latent factors

For default number of epochs (=20), optimal learning rate occurs at learning rate= 0.05 as seen in Figure 8.

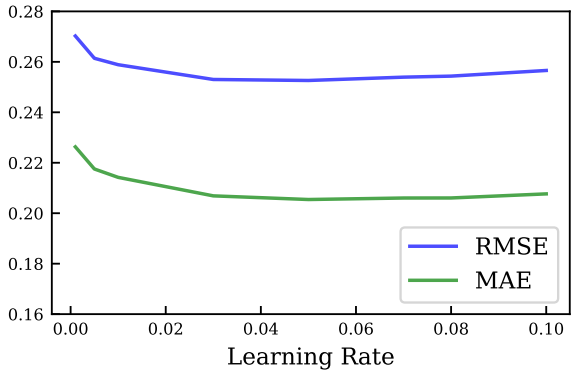

Figure 8: Variation of RMSE and MAE with learning rate

Regularization parameter = 0.07 gives nearly the best RMSE as seen in Figure 9. This is very close to the default regularization parameter (= 0.05)

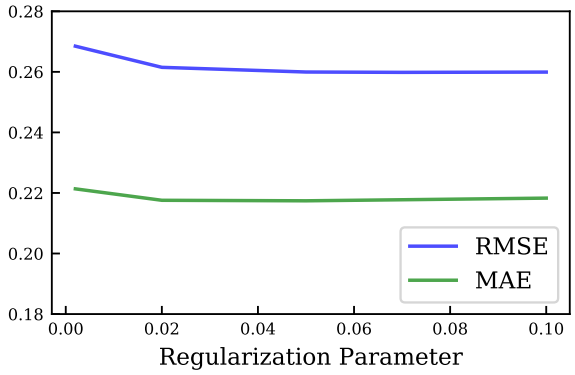

Figure 9: Variation of RMSE and MAE with Regularization

4.3.2 Binary Models

Baseline Models
Evaluating on test set, we observe the following precision@10 and recall@10 for the two defined baseline models:

Model	Precision@10	Recall@10
Baseline 1	0.0924	0.0352
Baseline 2	0.1086	0.0468

Collaborative Filtering
The performance of Pearson Correlation and Cosine Similarity based models in terms of the metrics decided above is as follows:

Model	Precision@10	Recall@10
Pearson Correlation	0.40	0.18
Cosine Similarity	0.40	0.19

We can further tune the performance of the model by constraining the type of users driving the selection of recommendations. Negative correlated users and less correlated can affect the score of a game in a negative way and hence trying to tune the model taking only the users having a correlation more than the threshold t. Hence for different values of correlation threshold, we evaluate the performance at a considered correlation threshold, referred to as performance@t, bench-marking it with the performance of taking overall users.

We perform a grid-search on the hyper-parameters - choice of similarity method and t - to identify the best hyper-parameters. Below is the plot[10] showing the variation of the performance of model for different values of correlation threshold and benchmark it with the original model's performance.

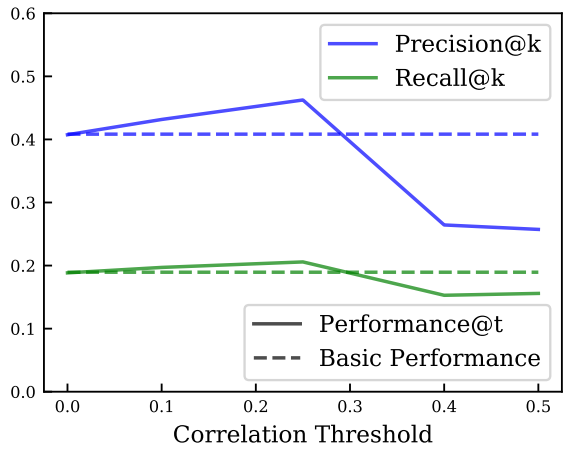

Figure 10: Variation of Precision@10 and Recall@10 with Correlation Threshold

From this analysis, our model is performing the best when we put a minimum threshold limitation of 0.25 on the user to user similarity. This does imply that the users with negative correlation are actually affecting the recommendations in a negative way. We also observe that the performance of the model goes down as increase the minimum correlation threshold as the number of samples based on which the model is predicting also goes down and hence the model isn't able learn all the users' preference clearly. Based on this analysis, we fix with the correlation threshold (t) as 0.25.

We also observe the affect on the model's performance with the number of recommendations being made. In the below plot [11] we see the variation of performance with respect to number of recommendations (k).

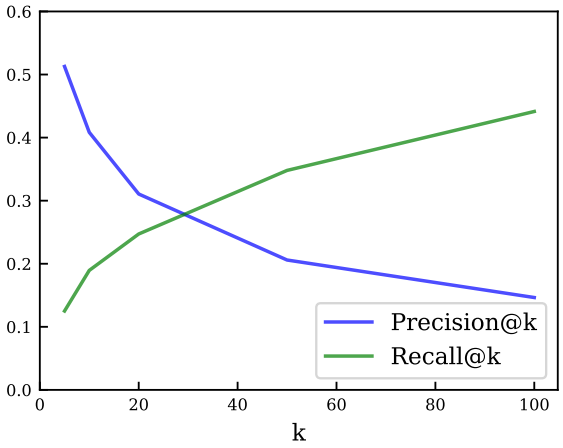

Figure 11: Variation of Precision and Recall with k

Discussion

Although both of the similarity methods are performing significantly better in comparison to the baseline model, there is still a lot of scope for improvement that can achieved by overcoming the drawbacks and issues of the similarity methods considered. Both of the implemented similarity methods can go wrong and fail [6] to correctly identify the correlation between users and may over or under report the similarity between users. Pearson correlation methods do not penalize or give weightage to proportion of common ratings in the data and can wrongly identify the correlation between two users. New Heuristic Similarity Model (NHSM) [7] tries to overcome most of the issues posed with the general most common similarity methods. It gives weightage to the proportion of common ratings between users, specifically penalizing the similarity with less proportion of items in common, and better approximates the similarity between users. We tried implementing the NHSM, but couldn't successfully complete it due to computational and time constraints.

Implicit Matrix Factorization We use the implicit library [8] for implementation of the model. We tune the hyper-parameters and then report the performance of the model on the test using the optimal parameters. The results are summarized in the next section.

Hyper-parameter Tuning : There are 4 hyper-parameters (latent factors, epochs, α, λ) to tune for the linear model and 5 (latent factors, epochs, $\alpha, \lambda, \epsilon$) for the logarithmic model. Due to computational constraints, we perform a coarse grid-search on the hyper-parameter space and fine tune them by evaluating on a validation set with the precision@k and recall@k metric (we fix $k = 10$ for tuning). The optimal hyper-parameters obtained for

the linear and the logarithmic model respectively were

$$\begin{aligned} \text{latent factors} &= 100, 100 \\ \text{epochs} &= 50, 50 \\ \alpha &= 0.01, 1 \\ \lambda &= 1, 1 \\ \epsilon &= \text{NA}, 1 \end{aligned} \tag{8}$$

The authors suggest that number of latent factors should be kept as large as possible within computational constraints. We observed that the performance starts to saturate at latent factors ≈ 90 as seen in Figure 12 and thus we fix them to be 100 (this analysis was performed for the linear model).

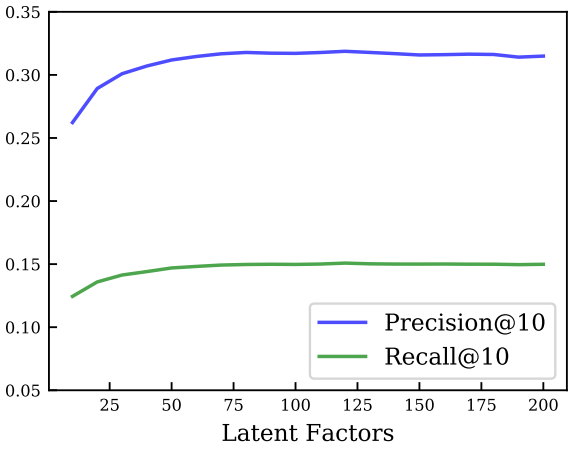

Figure 12: Precision@10 and Recall@10 vs Latent Factors

Since the optimization problem at any point of time is convex, increasing epochs can never increase the loss and should thus improve the model. However, we again observed that the performance saturated quickly as seen in Figure 13 (for linear model) and thus we fix epochs to be 50.

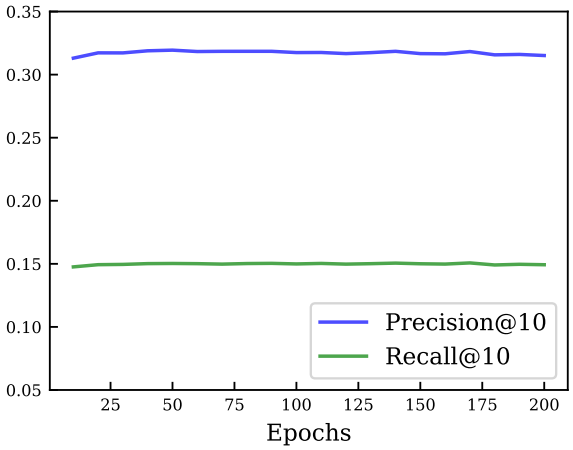

Figure 13: Precision@10 and Recall@10 vs Epochs

The authors observed that a value of $\alpha = 40$ performed best for their dataset. However, given that playtime can take very large values, we naturally observed that $\alpha = 0.01$ performed much better. Since logarithm naturally scales down the playtime, we found that $\alpha = 1$ performed the best for the logarithmic model.

Evaluation : We evaluate the two models using precision@k and recall@k metrics. Figure 14 show the performance of the linear and logarithmic models respectively against k.

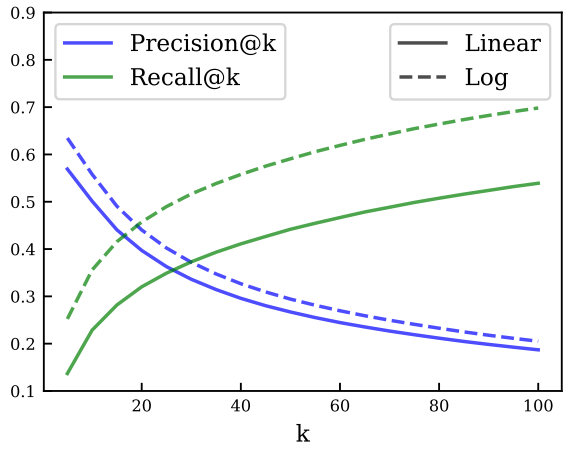

Figure 14: Precision@k and Recall@k vs k for linear model and log models

Learning Curve : During hyper-parameter tuning and evaluation, we observed that the test precision@k and recall@k is much higher than validation precision@k and recall@k. Since the test set is evaluated by training on larger data (more density, less sparsity), this suggests that the model performs better for denser matrices. We perform a learning curve analysis by varying the train set size

and evaluating on a fixed test set (since the recall@k depends on size of the evaluation set). We observe that the performance increases with density as seen in Figure 15. Additionally, we don't observe saturation on the full training set which suggests that getting more data (increasing density) to train would improve the performance further.

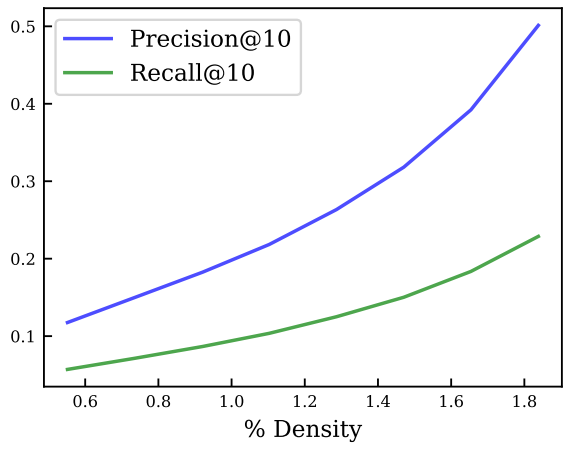

Figure 15: Learning curve for linear model

Discussion : Given that the inner product of two item feature vectors $q_i \cdot q_j$ gives the closeness of the two games, we found the most similar games for different types of games and observed interesting results. A game such as The Elder Scrolls V - Skyrim, which is an open-world, story rich, fantasy game had closest games near it to be Fallout 4, Witcher 3 which are similar open-world, story-rich games. Similarly for an indie, sandbox game like Terraria the model found that the most similar games were Portal 2, Garry's Mod. Based only on the playtime, the model is able to pick up the similarities of the games.

4.4 Results

The results of all the models are summarized below

Continuous Models

Model	RMSE
Baseline	0.2871
Cosine Similarity	0.2660
Matrix Factorization	0.2526

Binary Models

Model	Precision@10	Recall@10
Baseline 1	0.0924	0.0352
Baseline 2	0.1086	0.0468
Pearson Correlation	0.4625	0.2057
Cosine Similarity	0.4746	0.2089
Implicit Linear	0.5012	0.2287
Implicit Log	0.5580	0.2559

5 Deployment

Since we are treating binary and continuous cases are separate business problems, it makes sense to deploy both these models targeting recommendation of both cases. This can be done by displaying 2 carousels on the Steam website -
a) 'Games you might like' - For continuous case
b) 'Other users bought' - For binary case

We built our models on a very small subset of the population and thus deploying these models on the full data would face scalability issues. This can be mitigated by decreasing the frequency of training the models. For example, the models can be trained once a day or once a week based on constraints.

Ethically, this solution should not cause any problems because we don't utilize sensitive user information like gender, race as features for modeling our solution. One of the challenges in collaborative filtering is cold start problem. This will occur for a new user, when the model is not able to find similar users to the new user. Hence A/B testing is highly recommended for such cases.

References

[1] *Steam Web API*. https://partner.steamgames.com/doc/webapi_overview.

[2] *Steam Universe - Discussion about SteamOS, Beta Hardware, and Big Picture Mode*. https://steamcommunity.com/groups/steamuniverse.

[3] Nicolas Hug. *Surprise, a Python library for recommender systems*. http://surpriselib.com. 2017.

[4] Y. Hu, Y. Koren, and C. Volinsky. "Collaborative Filtering for Implicit Feedback Datasets". In: *2008 Eighth IEEE International Conference on Data Mining*. Dec. 2008, pp. 263–272. DOI: 10.1109/ICDM.2008.22.

[5] *Recall and Precision at k for Recommender Systems*. https://medium.com/@m_n_malaeb/recall-and-precision-at-k-for-recommender-systems-618483226c54.

[6] R. Bell, Y. Koren, and C. Volinsky. "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42 (Aug. 2009), pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263. URL: doi.ieeecomputersociety.org/10.1109/MC.2009.263.

[7] Haifeng Liu et al. "A new user similarity model to improve the accuracy of collaborative filtering". In: *Knowledge-Based Systems* 56 (2014), pp. 156–166. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2013.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0950705113003560.

[8] *Implicit - Fast Python Collaborative Filtering for Implicit Datasets*. https://github.com/benfred/implicit.

Appendix A Contributions

The sections on business understanding, data understanding, data preparation and deployment were based on discussions and inputs by all three of us. Additionally all of us worked on the write-up of the report. Specific contributions are noted below

Raghav (rj1408) : Data Collection (Scraping and API), Matrix Factorization Technique for Continuous Case

Nithish (bva212) : Collaborative Filtering for Binary and Continuous Case

Ieshan (iav225) : Data collection (Scraping and API), Implicit Matrix Factorization technique for Binary Case