Collaborative-Filtering-Based Prediction on Steam User-Game Data

By: Jonathan Tanoto

Abstract

Steam is a video game digital distribution by Valve. It has been of the most popular game platform that millions of gamers use around the world. That said, they have enormous data[1] regarding their user interactions as well as reviews on games. For this project, I used two datasets that consist of reviews of games per user, as well as user-game interactions. The predictive task I have chosen to pursue is to predict whether or not a given user has played a given game, by using the data given.

Introducing the Dataset

There are two datasets: user and item data, as well as reviews data.

User and Item Data

For this dataset, each entry of the data contains a certain user's information such as 'user_id', 'items_count', 'steam_id', as well as a dictionary of 'items' (games). For each entry of the games dictionary, it consists of 'item_id', and playtime data which shows how much the user has played the game. For this dataset, I only used the first 2000 users as it creates approximately 379,000 data points for my train/test data.

Reviews Data

For this dataset, it is also organized by users. Each entry is a specific user which entails his/her 'user_id', and a list of 'reviews'. Each of the 'reviews' entry consists of (the date it was) 'posted', 'item_id', 'recommend' (Boolean), and the 'review' text itself.

Data Pre-Processing

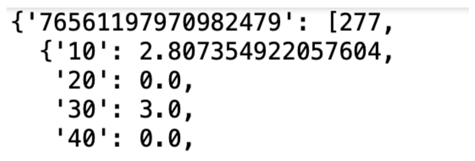
In order to start the predictive task, I needed to synthesize the data that I will be training and testing on. This was not given in the initial datasets that I chose. However, the User and Item dataset contains a collection of users with their individual collection of games. Therefore, I can create user-game pairs in the form of:

$$(user\_id, item\_id, 0/1)$$

However, the problem here is that I only have positive pairs that shows a user has played a certain game, no negative pairs. To solve this, I randomly sampled a game from a game set I created, that includes unique games in the data, that is not played by the user.
I do this for every positive entry which results in an equal balance between positive and negative pairs in my dataset.

User Data

I created a user database from the combination of the two datasets, extracting important features for each user that I may need later on in the project. For each user, I incorporated the features: 'steam_id', number of games owned, and a dictionary of hours_played[2] per game.



```
{'76561197970982479': [277,
  {'10': 2.807354922057604,
   '20': 0.0,
   '30': 3.0,
   '40': 0.0,
```

Figure 1. Sample entry of user data.

Game Data

Similarly, I created a game database for ease of accessing relevant information later on. For each game in the dataset, I incorporated the features: 'game_name', average hours played per game,

[1] http://deepx.ucsd.edu/public/jmcauley/steam/australian_user_reviews.json.gz http://deepx.ucsd.edu/public/jmcauley/steam/australian_users_items.json.gz

[2] Transformed with $\log_2(hours + 1)$ to improve scalability

and proportion of users who posted a review that recommended the game.

```
{'277630': ['Panzer Tactics HD', 0, 0],
 '214910': ['Air Conflicts: Pacific Carriers',
  6.068118208374362,
  0],
```

Figure 2. Sample Entry of game data.

Exploratory Data Analysis

I did an EDA on the two databases I created before. My EDA is separated into two parts: Games and User.

Games Trend

I used the Pandas library to quickly fetch some basic statistics for the game dataset. Below is a summary of the basic stats.

	Avg. Hours Played (transformed)	% Recommend
count	7865.000000	7865.000000
mean	5.098602	0.319519
std	3.152815	0.440584
min	0.000000	0.000000
25%	2.696159	0.000000
50%	6.057040	0.000000
75%	7.587323	0.888889
max	15.483658	1.000000

Figure 3. Game Database Basic Stats.

With the Avg. Hours Played variable, it seems to be right-skewed. This makes sense as it is highly probable for a user to own a game and have not put in any hours playing it, whereas there are some and only little amount of people who play

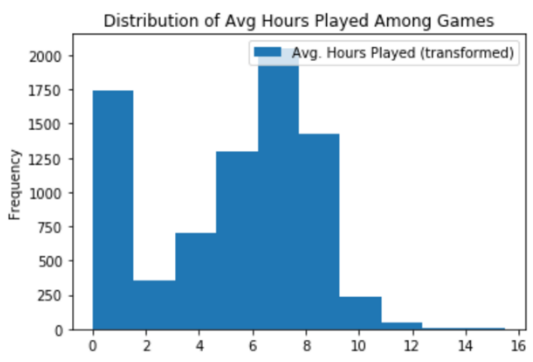


Figure 4. Distribution of Hours Played.

[3] 75th percentile

more than $2^7 - 1 \approx 190$ hours[3] for a single game.

The % Recommend variable seems to be right skewed as well. Although it peaks at around 0.0 - 0.1 which might be due to the fact that some games do not have enough reviews, so the percent-recommended for sparse games might be to the extremes (0 and 1). As a matter of fact, the 50th percentile is still at 0, which means that it is heavily skewed.

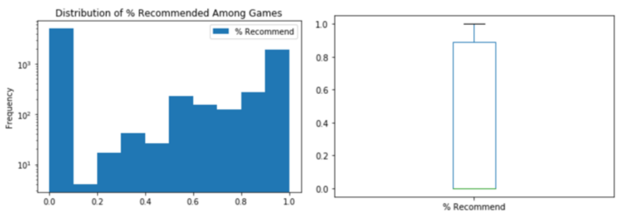


Fig 5. Distribution of % Recommend

Below is a scatter plot between the two variables. Although it seems random at first, there seems to be a positive correlation between the two. This means that as a game is played more, the % of recommendation increases. This makes sense as a good game will have high values for both variables—they have a positive correlation with each other.

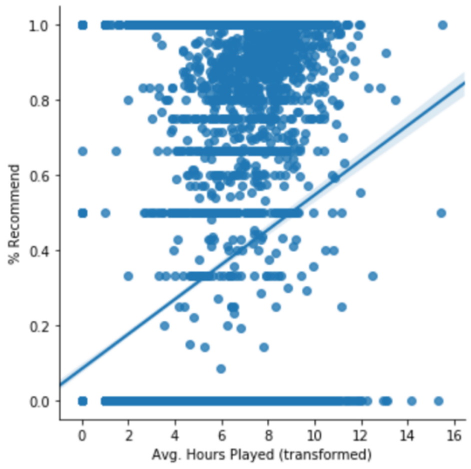
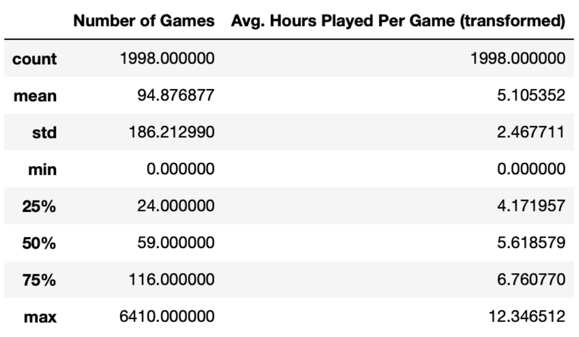


Figure 6. Scatter Plot of % Recommend vs. Avg. Hours (with Seaborn 'fit regression')

Users Trend

I used the Pandas library to quickly fetch some quick insights for the user dataset. Below is a summary of the basic stats.



	Number of Games	Avg. Hours Played Per Game (transformed)
count	1998.000000	1998.000000
mean	94.876877	5.105352
std	186.212990	2.467711
min	0.000000	0.000000
25%	24.000000	4.171957
50%	59.000000	5.618579
75%	116.000000	6.760770
max	6410.000000	12.346512

Figure 7. User Database Basic Stats

For number of games, it seems to be right skewed as it should be. Most people might even have less games than the mean (94)—which sounds like a lot. It is probably skewed by the outliers; the maximum is 6,410 games!

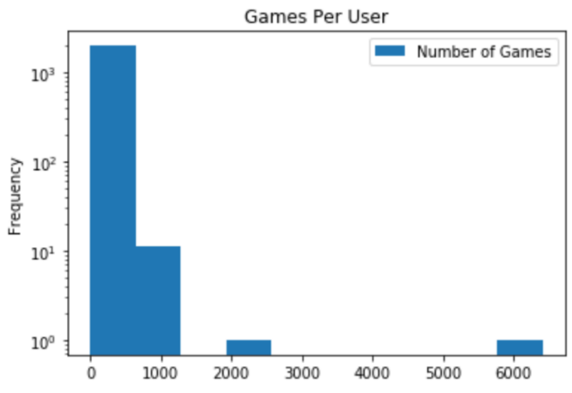


Figure 8. Distribution of Games per User.

For average hours per game, the trend seems to be normally distributed at around $2^5 - 1 = 31\ hours$ per game that users own. As usual, there is a peak at 0 which is completely normal since some user buy games and not play it.

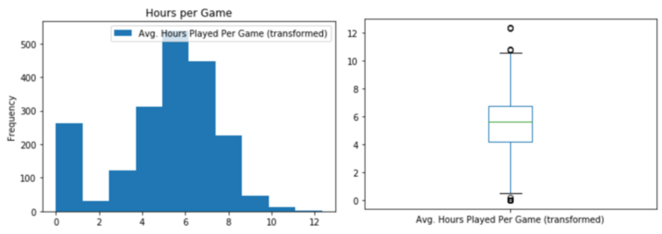


Figure 9. Distribution of Hours per Game

Below is a scatter plot comparing users' average hours played per game with the number of games they own. As seen, the regressor computes that the correlation can either be positive or negative, it is inconclusive. There is not much of a correlation we can expect between the two variables here since they are not exactly deterministic of one another.

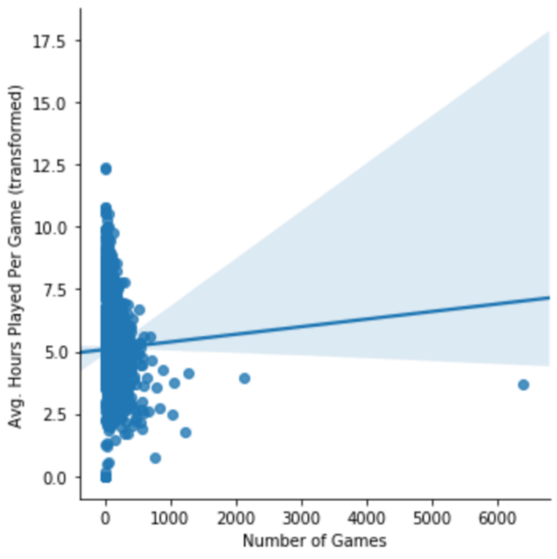


Figure 10. Scatter Plot of Avg. Hours per Game vs. Number of Games (with Seaborn 'fit regression')

Predictive Task Introduction

The predictive task I have chosen is to predict whether a user would play a certain game. The training data would have a balanced weight of labels 0 and 1 of people who have and have not played a certain game[4]. In order to engineer the features needed to be passed on to the machine learning models, I will be using a combination of variables from the User Database and the Game Database to provide some features that will prove helpful for the classification task. The prediction will be of the form:

$$f(game\ features, user\ features)$$
$$\downarrow$$
$$0\ (not\ played)\ or\ 1\ (played)$$

[4] from the negative samples synthesis I did.

The dataset I managed to construct has 379128 samples. I have divided the data into the following partitions:

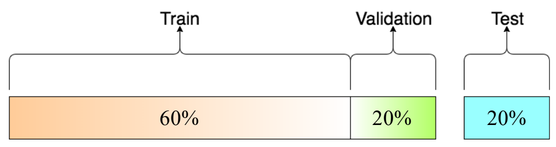


Figure 11. Train/ Validation/ Test Split Proportions.

Recommender Systems Functions

Jaccard:

The Jaccard similarity is used to compute the degree of similarity between two sets of items. The equation is as follows:

$$Jaccard(G_u, G_v) = \frac{|G_u \cap G_v|}{|G_u \cup G_v|}$$

get_max_sim(u, g):

[*Pseudo-code*] For the list of games G' also played by user *u*, for each game G'_i, compute the Jaccard Similarity of users who own game *g* and users who own game G'_i. Return the maximum similarity value out of all the games in G'.

This is an example of an item-based filtering where the metric shows the similarity between the users who play the games (items)

Feature Extraction

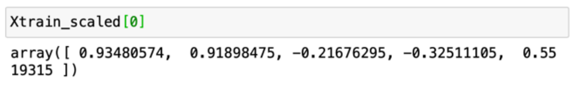
The features that I chose to represent the datapoints are as follows:

1) Get_max_sim(u,g): maximum similarity
2) % Recommend: Percentage of people who recommend the game
3) Game_hours: the average time spent on a game
4) Game_count: the number of games that the user has
5) User_hours: the average time user spends on a game

Notice that features 1-3 are game features, whereas features 4-5 are user features. It is crucial to implement features that represent both the item and the user respectively.

Next, I implement a StandardScaler transformation to the features in order to represent the features better and to prevent outliers from predicting wrong labels.

Finally, here is an example entry of the final features format:



```
Xtrain_scaled[0]

array([ 0.93480574,  0.91898475, -0.21676295, -0.32511105,  0.55
19315 ])
```

Figure 12. Example train datapoint feature representation.

Model

Baseline to Beat

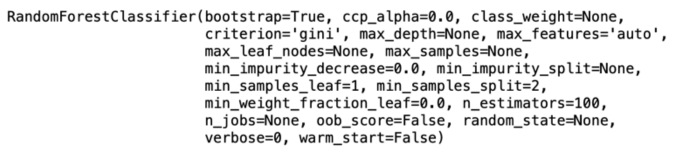
The Baseline for this predictive task is as follows. Make a count dictionary for each of the games that appear in the train data. If the game appears in the first half of the popular games list, then predict 1, otherwise predict 0. This trivial solution to the predictive task results in an accuracy of 0.6821.

The problem with this algorithm is that it is way too simple and trivial. Also, there should be more features incorporated into the prediction.

Summary of Models

Model	Accuracy on Test Set
Random Forest Classifier	0.843312
K Neighbors Classifier	0.815143
Linear SVC	0.744920
Logistic Regression	0.744797
Trivial Solution	0.682100

Random Forest Classifier



```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

I used the random forest classifier on the train data. Due to the large dataset and heavy computations, I skipped tuning the hyperparameters of the model. Apparently, it yielded the highest accuracy score.

K Neighbors Classifier

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=17, p=2,
                     weights='uniform')
```

Using GridSearchCV, I was able to fit the model using the training data and tune the 'n_neighbors' parameter of the model. The best score on the validation set came with the value of 'n_neighbors' being 17.

LinearSVC

```
LinearSVC(C=1e-05, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=10000,
          multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
          verbose=0)
```

I was able to use GridSearchCV to obtain the optimal value for the C parameter. The grid search reports that the best accuracy comes when the regularization parameter is .00001. Although it might make the model underfit the data, it seems that it works just fine. Although this strategy is not that optimal. On top of that, it was computationally extensive to use LinearSVC as the model for the predictive task.

Logistic Regression

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=10000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

On my previous predictive task on a similar dataset, logistic regression came up top for the highest accuracy score. This time, it seems to be beaten by the other models above. This is due to the difference in features that was used in my previous and current project. For this model, I was able to tune the hyperparameters for the model to gain the best accuracy on the validation set. The parameter that was tuned was the C parameter which was chosen to be 1.

Conclusion

Summary of Models

Model	Accuracy on Test Set
Random Forest Classifier	0.843312
K Neighbors Classifier	0.815143
Linear SVC	0.744920
Logistic Regression	0.744797
Trivial Solution	0.682100

From the table above, it is clear that the most accurate model that I implemented is the Random Forest Classifier with an accuracy of 0.843312 on the hidden test set.

Model Parameters

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

The parameters that I changed here was only the 'n_estimators' which relates to the number of trees in the 'forest'. This acts as some sort of a regularization parameter for the model. If 'n_estimators' is too large, then the model may be at risk of overfitting, on the other hand, if it is too small then the model might underfit. I used the validation set to tune the parameters to make sure that I am not overfitting to the train set.