NVIDIA Edge: Steam Game Recommendation Engine Report

Muf Tayebaly, Senior Project Lead

James Xue, Creative Project Manager

Abstract:

At NVIDIA, our course business is to empower gamers across the globe to be able to play their favorite games with the best performance hardware available. We are looking to create NVIDIA Edge, an online server based service for gamers to access the best GPUs in the world remotely and a low cost. The service will include a built-in library of games for Gamers to play at a subscription. We are hoping to determine the library of games to initialize our service with, as well as to recommend players new games as our service and library expands.

Objective:

We are looking to make use of the Steam Video Game dataset to better understand our gamers and determine what we can recommend.

The Steam gaming platform distributes games to its users and allows for multi-player management and a community for gaming. Game sales is an important revenue stream and some gamers are known to play the same game for long periods of time instead of branching out. With the increasing advertising platforms becoming available (Facebook, Instagram, Snapchat, etc..), costs of advertising can be pricey to gain new sales. Studies have shown (e.g. Waggener Edstrom Worldwide) that word-of-mouth is the primary leader in new game sales.

A recommender system can behave similarly to word-of-mouth where it can help identify new games for a user based on similar users or items. This can lead to more game sales by making the users aware of games they might be interested in without having to pay for expensive advertisements. If the recommendations prove to be successful with the user, it can even increase the validity for the users to continue to use the recommendations going forward to find new games. New sales creates increased revenue and the potential to decrease advertisements costs (if recommendations prove to sell more than advertisements) and therefore increasing profits. While advertisements are good to get the names known to larger audiences, especially those who are unfamiliar with the Steam platform, for existing users, recommendation systems can be very useful and what we will test within this project.

Recommendation systems can even strengthen the gaming community by getting users who are interested in similar games actually playing more of those games together and creating stronger relationships between gamers that can ultimately retain users in the long run.

We will use existing user and game data to prove we can make accurate (to some significance) recommendations of games that we already know the user is interested in since we will hold out known data.

Table of Contents

[Neighborhood Based Methods 3](#_Toc529368031)

[Overview 3](#_Toc529368032)

[Hyperparameter Effects: Neighborhood Size, Minimum Hours Played, Model Size 3](#_Toc529368033)

[Alternative Design Choices 3](#_Toc529368034)

[Case Study Evaluation 3](#_Toc529368035)

[Model Based Methods 4](#_Toc529368036)

[Overview 4](#_Toc529368037)

[Hyperparameter Effects: Neighborhood Size, Minimum Hours Played, Model Size 4](#_Toc529368038)

[Alternative Design Choices 4](#_Toc529368039)

[Case Study Evaluation 4](#_Toc529368040)

# Neighborhood Based Methods

## Overview

We choose to implement a Neighborhood Based Recommendation System by hand due to some customization required by our data. Let’s begin by covering our data and then move on to our methodology and customizations. The data we have consists of several columns: user, game, purchase, and playtime. **The challenge with our data is that it is extremely sparse and the playtime values greatly vary.** Out of the tens of thousand games available, most users play less than 10 games. **The playtime could vary between several hundred hours played to a few minutes.** On another level, we don’t have explicit ratings from the users. We can only implicitly infer their preferences from their hours played.

These challenges demanded several changes to how we measure performance.

First of all, due to the **sparsity of the matrices**, it was impractical and ineffective to remove the mean from the data when calculating predictions. The result of removing the mean is that we have a very, very, very significant number of negative hours played, leading to spurious and nonsensical predictions regarding hours played (many negative hours played predictions).

Secondly, we choose to create a **custom coverage metric** based on the catalogue coverage metric. Due to the sparsity of the matrices, **many of our similarities between users were actually zero**. This resulted in each user being recommended a very small number of games to play (most games had a predicted play time of zero). This means that it might not be effective to look at high ranking items but rather hours played. We set a hyperparameter, minimum hours played and determined the coverage on items that were recommended and predicted to be played over a certain number of hours. For example, for a minimum hour played of 1, we have a **coverage metric of 0.14**.

Thirdly, we created **custom ways to cross-validate the data**. Due to the sparsity of the matrix, it makes no sense to cross validate the one-hot-encoding itself. We cross validate the sample that we have into train, test, and validation. We create the one hot encoding on the entire set but then remove the values associated with test and validation. This cross validated one hot encoding was then used to determine similarity and predictions.

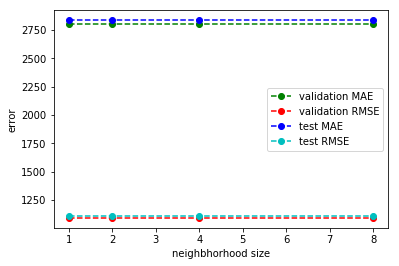
Fourth, we wanted to have unique sampling methods for the data. Due to the large number of players and games, creating a one hot encoding on the entire set was impractical and slow. Thus, we choose to sample k random games and also k top games to create separate

Lastly, we wanted to be able to choose our accuracy metric. We wanted to not rely on RMSE since our playtimes have large variations. Furthermore, we belive the MAE could be a good metric as to not overly punish differences from players who played 2000+ hours when compared to the average. Surprisingly, on our random 100 games subsample, **we found a validation MAE of 3938.3, validation RMSE of 1452.9, test MAE of 3417.9, and test RMSE of 2186.** This shows that most of the differences between predicted and true is less than 1 hours played (since RMSE is the sum of the square difference vs MAE is just the absolute difference). It is also promising that the validation and test are not too different.

## Hyperparameter Effects: Neighborhood Size, Minimum Hours Played, Model Size

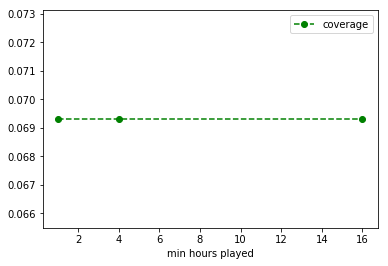
(i) Neighborhood Size:

This chart shows the difficulty of working with our data. Due to the sparsity of the matrix, the number of neighbors that have nonzero play times for a particular game is inherently very small. Furthermore, these small number of neighbors also tend to be very dissimilar from the user. As a result, changing the size of the neighborhood did not really affect our metrics. On the other hand, this does give a good comparison of the different metrics. MAE tends to be higher than RMSE, indicating small differences in play times on average (less than 1 hour).

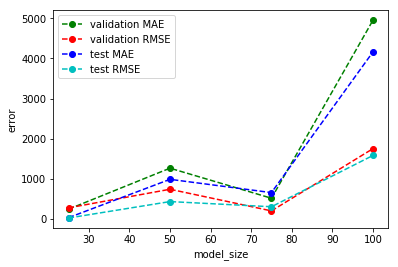
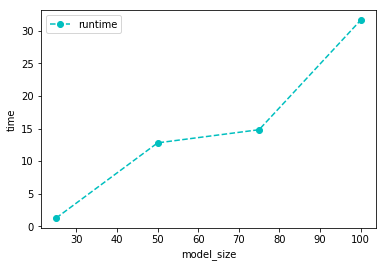


(ii) Min Hours Played and Coverage

This chart show how coverage changes, or rather, not changes with min hours played. This result and the previous result both show the difficulty of working with sparse data.



(iii) Size Effects



As shown by the chart above (where model size is the number of games to consider), size tends to have a significant effect on runtime and error metrics. The error metrics are surprising in that you expect them to increase as the number of samples explodes. However, you actually see a dip at 75 games, which is surprising. However, given that these are random samples, it is understandable. Furthermore, the validation and test errors are actually in line.

## Alternative Design Choices

**One of the challenges faced here is that we don’t have explicit ratings from users and we must rely on hours played to gain insight.** However, while someone who played a game for 1000 hours might appear to be very different from someone who played for 100 hours, we might want to categorize the hours into quartiles. In other words, **we create 4 buckets**, one for each of the quartiles and given values 1 to 4. This way, this might appear more similar to a rating system and individuals can be compared by their bucket rather than their absolute number of hours played.

**Another alternative design choice we can make is to choose the top users.** This might make sense business wise because these super-users might be the primary source of revenue. Furthermore, these users are much less sparse and share more in common with each other than a randomly selected sample of users. As a result, we can probably get better similarity measures using better user sampling.

Lastly, we can try techniques outside of neighborhood methods, which we will cover in the next section.

# Model Based Methods

## Overview

Since we heavily customized and hand built the entire preprocessing, prediction, and validation pipeline in the neighbors module, we can take a step back and see how existing packages can perform on our next phase: matrix factorization model.

Let’s begin with an alternative preprocessing flow. The Steam data was provided as a CSV with rows containing details of user-game purchased and hours played values. We preprocessed the data by creating 3 separate files: customers.csv (contains a list of unique customers and a new indexed ID), games.csv (contains a list of game names and a new unique game ID), and hours.csv (contains list of hours played for each customer and game using the new IDs). We used new IDs to help speed up processing and keep a good sense of customers and games.

Sampling data would be the important next step. For part 1, we decided to keep to a smaller data set as described in the instructions to gain better intuition on the data, testing and exploring. Since there was a lot of games that users did not really play and have data for, we decided to start with popular games so we filtered to games that were played by more than 50 users which gave 284 games. Out of those games we randomly sampled 90 games to use. Now that we had the games, we wanted to ensure we had the right users since there are a lot of users who would not have played any of those 90 games and therefore would not provide good recommendations with zero rows/columns. In order to work around this, we filtered the hours played list to only show the user-game hours played for the games in scope (from sample). This gives us a list of only customers who played our sampled games. To prevent hold-out issues between train and test data from holding out all of a customers hours played rating, we further filtered the customers unique list by those customers who had played more than 1 game. This decreases the likelihood of randomly holding out the rating for customer-game since that customer had only played that single game we held out. The sampling also maintains random seeds in case we need to replicate a run and can also specify new random seeds to get entirely new random data for cross-validations.

Now, it is important to discuss the training and testing. Using our sampled data of popular random games, customers who played more than 1 of the games sampled and the ratings list (customer-game hours played), we derived what our train and test data was by holding out data from the ratings (hours played) list. For example, one run we held out 10% of data for test and the remaining was used for training. We could then use the train data to create the sparse matrix and have the test data to measure accuracy.

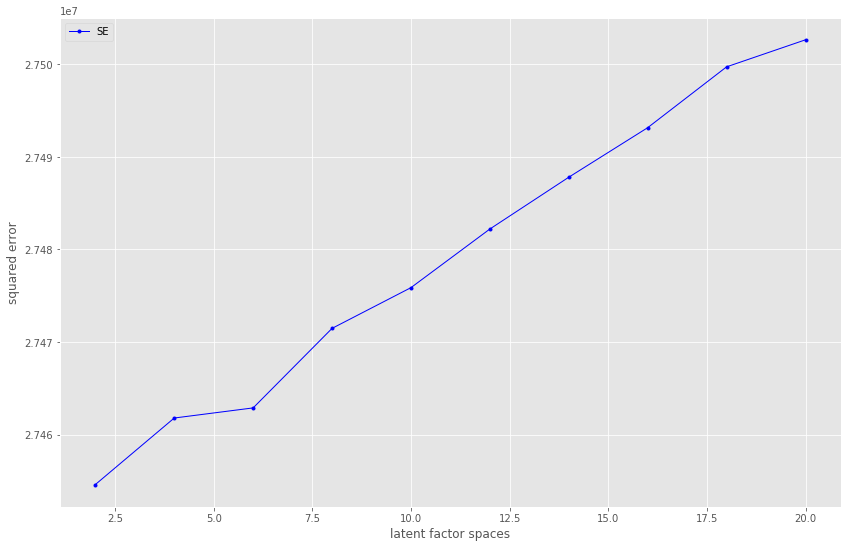
We finally get to our model for matrix factorization. We used the Implicit package as recommended in the project instructions so we could get a feel for how the data was being factored into latent spaces, our data is implicit, and the package runs quickly. Using the Alternating Least Squares method, we fit the model on the sparse matrix and got the U (customers) and V (games) factored matrices. The multiplication of these matrices gives us the estimated (learned) matrix R.

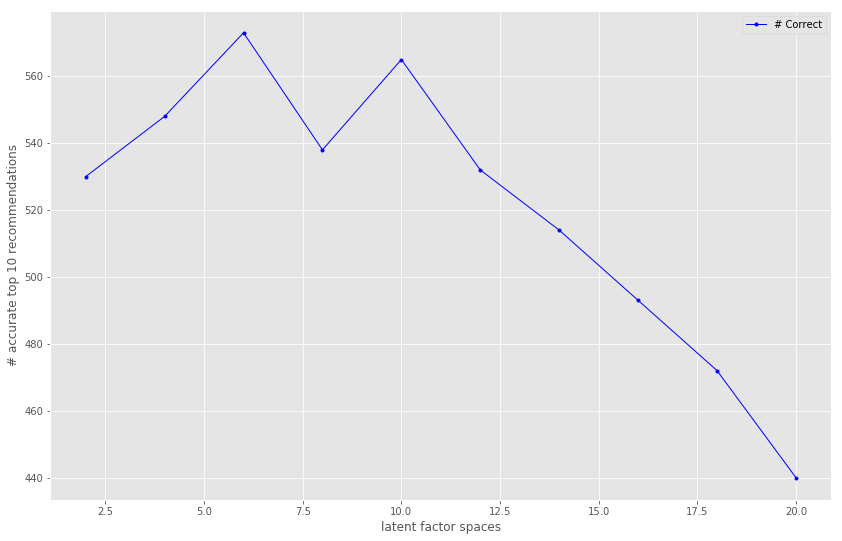
We tried using the squared error to measure accuracy from the estimated matrix R as compared to the test data we held out, but it looked like due to the large range of hours played (which we treated as a rating for the customer-game) the gap between the recommended and actual is high and thus gives a high squared error.

Another accuracy method which we thought worked better for this specific model was to compare the top 10 recommendations and if the test game actually played by the customer was one of the top 10 recommended games. This gave a good idea of whether or not the model was properly recommended the games to the users and we started seeing roughly 40-50% accurate recommendations.

## Hyperparameter Effects

We used the latent factor spaces as one of our parameters to see how it impacted the accuracy measure. Using the top 10 recommendation comparison measure, we were able to see a very small factor space was not as good as a medium size (2 factors as compared to 6 factors). As we went increased the factor spaces, we hit a maximum and then started seeing a decline in accuracy as the factors got too large. See the figure below which shows you which number of latent factor spaces worked better as compared to others.





## Going Foward

We should be able to use what we learned here to better expand on our models and recommendation techniques for the final project. We have implicit data of games purchased which can be brought into the picture and also help where we had large range in ratings (hours played) which could have impacted the recommendations. We did not use any side information which might help as well if we tried to pull some information on these games which can also help determine right factored spaces. Potentially creating our own Matrix Factorization algorithm for this particular dataset may also prove to work better given certain tweaks. Lastly, reviewing how the sample we used here as compared to other methods of sampling could show us something we did not notice.

# Case Study Conclusion: