39 Recommendation Systems in Games

Ben G. Weber

39.1 Introduction

Players have a large amount of content to choose from across a variety of different game platforms and marketplaces. Some game storefronts now offer thousands of games, and finding relevant content can often be challenging for players. One of the newer ways in which AI is being used is to help players discover content through recommendation systems. A successful recommendation system should suggest content that a player finds relevant and interesting, and that the player would not have otherwise discovered. In games, recommendation systems can be applied to a variety of tasks outside of selling content, including matchmaking systems and dynamic difficultly adjustment (Medler 2008).

The goals for a recommendation system vary based on the system user. For a player, a common motivation for browsing content suggested by a recommender is to find games that closely match the player's preferences. Recommendations can be useful for finding similar titles that do not fit well into existing categories or genres. Another motivation for using recommendation lists is to discover new content that the player would not have otherwise considered. For platform owners that manage a storefront, a common goal is to improve the monetization of players on the system. This can include improving conversion rates, from viewing the store to making a purchase, increasing total daily sales on the platform, and increasing sales for a specific type of content. For game developers, the motivation is to get featured on the recommendation system, to drive additional sales and player engagement.

This chapter will introduce some of the algorithms used to implement recommendation systems and provide examples of systems being used in games. Next, it provides source code examples for setting up a recommendation system using Java, Scala, R and SQL. It concludes by discussing how to test a recommender in offline and online experiments, and options for deploying a recommendation system.

39.2 Recommendation Algorithms

A variety of algorithms are available for suggesting content to players. The best algorithm to use will depend on the number of items in the marketplace, how quickly the marketplace changes, the number of users interacting with the service, and the richness of telemetry collected by the game and platform. An approach that works well for a small, curated game storefront may not transfer well to a marketplace with thousands of titles, millions of users, and constant updates. Additionally, algorithms that use gameplay data to find more relevant content for players, such as a player's favorite class or items in an RPG title, may be too computationally expensive to use in practice.

There are three common types of approaches used for implementing recommendation systems. The first is content-based filtering, which uses information about a specific item, such as game genre, to find related items. The second is collaborative filtering, which uses common purchase patterns across players to make suggestions. The third is model-based approaches that predict a player's likelihood to purchase in order to suggest content. Hybrid approaches can also be used that integrate multiple algorithms.

39.2.1 Content-Based Filtering

Content-based filtering uses only information about an item to suggest related items. This is usually handled through metadata about the item, such as game genre or other item tags. For example, if a user decides to download *Battlefield 4*, a content-based recommendation system might suggest related shooter titles, such as *Battlefield Hardline*. The Steam recommendation system behaves like a content-based system, because it usually suggests content within the same sub-genre. One of the benefits of a content-based system is that it does not need any prior data about the user in order to make suggestions. These systems do not suffer from the cold-start problem, since they do not try to make inferences about users. One of the challenges in using content-based filtering is that accurate metadata or tags need to be created and maintained for the entire catalog.

39.2.2 Collaborative Filtering

Collaborative filtering uses data collected from users and transactions to make inferences about which content users will find relevant. It is therefore making recommendations for you based on the preferences of other users, rather than the genres of the games that you have played. One of the benefits of collaborative filtering is that it can be used to reduce the amount of metadata that needs to be maintained about items in the catalog. For example, instead of specifying a genre for every game on a storefront, you can use data collected from players to determine which games are related.

The collaborative-filtering approach can be used for both item ratings and item rankings. In an item rating system, the goal of the recommender is to predict how a user would rate a game that the user has not yet rated. A system that can accurately model a user's ratings can suggest new games to try by predicting the unrated games the user would rate highest. This is a form of explicit data collection, since the user is assigning scores to items. In an item ranking system, the goal of the recommender is to suggest content that the user is most likely to interact with next. Rather than assign specific scores to items, these types of systems provide an ordered list of content that the user may find relevant. This is a form of implicit data collection, since the user is interacting with the content but not assigning a specific score. Netflix is an example of an item rating system, since it predicts how users will rate new movies, whereas Amazon is an example of an item ranking system, since it suggests related items but does not show explicit scores.

Collaborative filtering can either directly model the relationships between users and items, often referred to as *neighborhood methods*, or indirectly model these relationships using inferred variables (*latent factors*). For direct relationships, user-to-user or item-to-item based approaches can be used. For indirect relationships, solving for the latent factors is treated as a matrix factorization problem and alternating least squares (ALS) is commonly used (Ryza et al. 2015). Apache Mahout provides libraries for user-based and item-based collaborative filtering (Anil 2010), whereas Apache Spark provides an implementation using ALS.

In user-based collaborative filtering, the goal is to find a set of users with similar preferences to the user that needs recommendations. Items that similar users purchased or interacted with are used to generate a list of recommendations. Items from users that are more similar to the user that needs recommendations are given more weight than less similar users. The algorithm works as shown in Listing 39.1. It computes a weighted average for each of the items rated by similar users and returns the items with the highest weighted averages.

Listing 39.1. Pseudocode for User-Based Collaborative Filtering.

```
For every other user V
    Compute the similarity S between U and V
    For every item I rated by V
        Add V's rating for I weighted by S to I's avg. weight
Return the top rated items
```

Item-based collaborative filtering uses a similar approach, but builds recommendations by computing item similarities rather than user similarities. Pseudocode for this algorithm is shown in Listing 39.2. When a user needs a list of recommendations, the system uses prior item ratings and retrieves similar items based on a similarity measure.

Listing 39.2. Pseudocode for Item-Based Collaborative Filtering.

```
For every item I
    For every item J already rated by U
        Compute the similarity S between I and J
        Add U's rating for J weighted by S to I's avg. weight
Return the top rated items
```

In item-based collaborative filtering, item similarity is usually computed based on the overlap of users that interacted with both items I and J. This approach differs from content-based filtering, which computes item similarity based on metadata. This algorithm was used to implement one of the previous versions of Amazon's recommendation system (Linden et al. 2003).

Both item-based and user-based collaborative filtering use similarity measures to select items to recommend. One of the similarity measures that can be used by both algorithms is the *Tanimoto coefficient* (Anil 2010). For user similarity, the coefficient is defined as the overlapping number of items purchased by both users (intersection) over the total number of items purchased by the users (union). It returns a value of 1 for users with the exact same preferences and 0 for users with no overlap. Different similarity measures are useful for different types of recommendations: Pearson correlation and Euclidean distance are commonly used for explicit feedback (item ratings), whereas the Tanimoto coefficient and Log Likelihood are frequently used for implicit feedback (item rankings) (Anil 2010). It's best to experiment with different similarity measures on test data to evaluate which measures work best.

Alternating least squares is another algorithm used for collaborative filtering. It is a model-based approach in which the goal is to associate user and item relationships through latent factors rather than directly representing these associations. Latent factors are variables that are not directly observable, but assumed to have influence on users' preferences for content. This approach uses two matrices: One where each user has *k* latent variables, and a second where each item in the catalog has *k* latent variables. ALS is used to solve for these variables, and the resulting matrices can be used to predict how users will rate new content (Koren et al. 2009). This approach is highly scalable and was used by the winning entry in the Netflix prize

39.2.3 Model-Based Filtering

Predictive models can also be used to recommend content for players. One approach that can be used is modeling a user's likelihood to purchase a game, which can be represented as a classification problem. In order to create a recommendation list, a separate classifier needs to be created for each game, and the outputs of the classifiers are used to generate a sorted list of games for a user. One of the benefits of this approach is that if an eager model is used, such as logistic regression (Hastie et al. 2001), then recommendations can be computed very quickly for a user. Some of the drawbacks of this approach are that it requires building a classifier per game in the catalog and may require significant offline training.

In practice other models are often used in order to scale to a large item catalog. One example of a model-based recommendation system used in practice is the Xbox recommender (Koenigstein et al. 2012). This system uses Bayesian inference with a bilinear model, where each user and item is represented as a vector and the inner product of a user vector and item vector predicts the user's affinity for the item.

Another way models can be utilized for recommendations is by identifying different segments, or cohorts of users. For example, one segment of players may prefer RPG titles and purchase a large amount of RPG games and DLC, whereas another segment of players may prefer free-to-play FPS titles. If a model can accurately predict segments for players, then these segments can be used to recommend different content to different groups of players. This approach is often combined with content-based filtering or handcrafted rule sets for a small, curated game catalog.

39.2.4 Algorithm Selection

Depending on the target deployment environment, a variety of different options may be available for implementing a recommendation system. When choosing what approach to use, it is useful to ask the following questions:

1. Is the system generating item ratings or item rankings?
2. How large is the item catalog?
3. Is the metadata for items well maintained?
4. Is there a massive user base with tens of millions of users?
5. Should the recommender include gameplay-specific events?

If the goal of the system is to generate predicted ratings for content, then collaborative is usually the best approach. For item rankings, several options can be well suited. If the item catalog is small, then content-based filtering may be the best approach if accurate and up-to-date metadata is available for the item catalog, such as tags that describe a game. Otherwise, if the catalog is small then model-based approaches such as a classifier per game may be useful. For large item catalogs, collaborative filtering can be useful. In the case of Amazon, with a massive user base, an item-based algorithm was used (Linden et al. 2003). For large user bases where additional information about player behavior should be used for recommendations, such as the favorite class of a player in RPGs, user-based collaborative filtering can be used. ALS provides an approach that can scale to a variety of use cases.

The recommendation algorithm used for the in-game marketplace in *EverQuest Landmark* is user-based collaborative filtering (Weber 2015). One of the unique challenges faced by this title is that user-generated content, published through the Player Studio program, can be sold in the game's marketplace. This resulted in a large item catalog where limited metadata are available to describe items. Additionally, one of the goals for the recommendation system was to incorporate gameplay-based metrics, such as the amounts of different resources collected by a player, when making suggestions for content. User-based collaborative filtering works well for this approach, because the similarity metrics used can incorporate additional features about players, and this approach does not require accurate metadata about items.

39.3 Building a Recommender

There are a variety of open-source tools that can be used to prototype and deploy a recommendation system. This section will present examples for generating a recommended item list using Mahout in Java (Anil 2010), MLlib in Scala (Ryza et al. 15), recommenderlab in R (Hahsler 2011), and directly in SQL. Each of these libraries has evaluation metrics that can be used to measure the performance of a recommender, such as precision and recall metrics. Some of these libraries are better suited for prototyping different system configurations, whereas some are also suitable for deployment in a production system.

39.3.1 Java: Apache Mahout

Mahout is a machine learning library implemented in Java that provides a variety of collaborative filtering algorithms (Anil 2010). Using this library, it is possible to quickly evaluate a variety of recommendation system configurations by combining different algorithms and similarity measures. Mahout can be used on a single machine for prototyping, or deployed to a cluster for a production system. Daybreak Games used this library to test out different recommendation system configurations for the marketplace in *EverQuest Landmark* (Weber 2015).

Mahout implements user-based collaborative filtering with a *UserNeighborhood* class that specifies how similar a user needs to be in order to provide feedback for item recommendations. An example script that generates five game recommendations for user 101 is shown in Listing 39.3. This example uses the Tanimoto similarity measure to find the similarity between users, which computes the ratio of the number of shared games (intersection) over the total number of games owned by the players (union). The script loads game purchase data from a CSV file, which lists game purchases as tuples of User ID and Game ID. This file is used as input to a data model which is then passed to the recommender object. Once a recommender object has been instantiated, the *recommend* method can be used to create a list of game recommendations for a specific user. In this example, the top five game recommendations are retrieved for the user with ID 101. The import statement in this script shows a common directory shared by these classes, to find the fully-qualified class names readers should refer to the Mahout documentation.

Listing 39.3. User-Based Collaborative Filtering with Mahout (Java).

```
import org.apache.mahout.cf.taste.*;

DataModel model = new FileDataModel(new File("Games.csv"));
UserSimilarity s = new TanimotoCoefficientSimilarity(model);
UserNeighborhood neighborhood = new
    ThresholdUserNeighborhood(0.1, similarity, model);
UserBasedRecommender recommender = new
    GenericUserBasedRecommender(model, neighborhood, s);
List recommendations = recommender.recommend(101, 5);
```

An example of item-based collaborative filtering with Apache Mahout is shown in Listing 39.4. The main difference in this example is that a different recommender object is instantiated, and *UserNeighborhood* is not specified. The item-based recommender provides the same *recommend* method that can be used to generate game recommendations. One of the useful classes that Mahout provides not shown in these examples is *RecommenderEvaluator*, which provides functionality for computing the recall and precision of a recommender.

Listing 39.4. Item-Based Collaborative Filtering with Mahout (Java).

```
DataModel model = new FileDataModel(new File("Games.csv"));
ItemSimilarity s = new LogLikelihoodSimilarity(model);
ItemBasedRecommender recommender = new
    GenericItemBasedRecommender(model, s);
List recommendations = recommender.recommend(101, 5);
```

39.3.2 Scala: Apache Spark

One of the tools becoming more popular for building recommendation systems is Apache Spark. In fact, many of the single-machine algorithms available in Mahout are being deprecated in favor of Spark. In addition to Mahout, Spark provides a built-in library called MLlib which includes a collection of machine learning algorithms. Currently, ALS is the only implementation of collaborative filtering available in MLlib (Ryza et al. 2015). Although Spark supports multiple languages, the example in this section uses Scala.

A Scala example using MLlib to perform user-based collaborative filtering is shown in Listing 39.5. This script first runs a query to retrieve game purchases in a UserID, GameID tuple format, and then transforms the data frame into a collection of ratings that can be used by the ALS model. Implicit data feedback is being in this example, which is why the *trainImplicit* method is used instead of the *train* method. The input parameters to the train method are the game ratings, the number of latent features to use, the number of iterations to perform for matrix factorization, the lambda parameter which is used for regularization, and the alpha parameter which specifies how implicit ratings are measured. Once the model is trained, the *recommendProducts* method can be used to retrieve a recommended list of games for a user. In this example, five games are retrieved for the user with ID 101.

Listing 39.5. User-Based Collaborative Filtering with MLlib (Scala).

```
import org.apache.spark.mllib.recommendation._

val games = sqlContext.sql("
    select UserID, GameID from GameOwenership
    group by UserID, GameID")

val ratings = games.rdd.map(row =>
 Rating(row.getInt(0), row.getInt(1), 1)
)

val rank = 10
val model = ALS.trainImplicit(ratings, rank, 5, 0.01, 1)
model.recommendProducts(101, 5)
```

39.3.3 R: recommenderlab

If you are more comfortable programming in R, then the *recommenderlab* package provides a great framework for testing out different recommendation systems (Hahsler 2011). An example using this package for user-based collaborative filtering is shown in Listing 39.6.

Listing 39.6. User-Based Collaborative Filtering with recommenderlab (R).

```
install.packages("recommenderlab")
library(recommenderlab)

matrix <- as(read.csv("Games.csv"),"realRatingMatrix")
model <-Recommender(matrix, method = "UBCF")
games <- predict(model, matrix["101",], n=5)
```

The package is available on the CRAN repository and can be installed using the standard *install.packages* function. Once loaded, the package provides a *Recommender* function which takes a data matrix and recommendation method as inputs. In this script, the data matrix is loaded from a CSV file, and the method used is user-based collaborative filtering (UBCF). The *predict* function is then used to retrieve five items for user 101.

39.3.4 SQL

If you have purchase history stored in a database, another option for prototyping a recommendation system is to use SQL directly. This approach can be computationally expensive to use, but can be useful for spot-checking a few results for sampled data. It is also useful in situations where pulling data to a machine running Spark or R is slow or expensive. An example using SQL to perform user-based collaborative filtering is shown in Listing 39.7. In this example, the inner query computes the Tanimoto coefficient between users by finding the ratio in overlapping games divided by total number of games purchased. The outer query returns an average score for each retrieved game. The result set of this query is five game recommendations for user 101.

Listing 39.7. User-Based Collaborative Filtering in SQL.

```
select u. UserID, v. GameID, avg(Tanimoto) as GameWeight
from (
    select u. UserID, v. UserID V_ID,
        count(distinct u. GameID) Overlap,
        Overlap/(u. NumGames + v. NumGames - Overlap) Tanimoto
    from Purchases u
    Join Purchases v
     on u. GameID = v. GameID
    where u. UserID = 101
    group by u. UserID, v. UserID, u. NumGames, v. NumGames
) u
Join Purchases v
    on V_ID = v. UserID
group by u. UserID, v. GameID
order by GameWeight desc
limit 5
```

39.3.5 Evaluating Recommenders

The scripts in this section have provided examples of how to retrieve game suggestions for a specific user. One of the ways to evaluate the quality of a recommender is to use a qualitative approach, in which the output of the recommender is manually examined for a small group of users. Another approach is to use the built-in evaluation metrics included in the different libraries. For example, recommenderlab and MLlib provide functions for computing receiver-operating characteristic (ROC) curves which can be used to evaluate different system configurations.

When evaluating a recommender, it is also a good practice to compare the performance of the recommendation system to other handcrafted approaches, such as a top sellers list. One of the metrics used to evaluate the recommendation system for *EverQuest Landmark* was a holdout experiment, where a single item is removed and the goal of the recommender is to identify the held-out item in as few suggestions as possible. This enabled different Mahout configurations to be tested against hand-authored rule sets. Also, recommender evaluation should not be limited to the prototyping stage. Once put into production, the system should be compared against control groups, such as a top sellers list and other recommendation system configurations.

39.4 Deploying a Recommender

One of the challenges in building a recommendation system for a game is deploying the system so that it can retrieve item recommendations in near real time. A common method for achieving this level of responsiveness is setting up offline and online phases. In the offline phase matrices are precomputed as part of a batch process performed daily, and in the online phase a web service performs a lookup from the most recently precomputed matrix. Using this approach avoids the need to evaluate models in real-time when building game suggestions. Another approach is to set up a streaming recommendation system that computes items for each user in near real-time. For example, Spark can be configured in a streaming mode where new batches of users are evaluated every second. In this configuration, the model needs to be responsive enough to ensure efficient retrieval of recommendation lists. Another approach is to implement the logic for collaborative filtering on the game server. This is similar to the streaming approach, but the response is handled directly by the game server rather than through a recommendation library.

Once a recommender has been deployed, it is useful to measure the performance of the system versus a control group, such as a top sellers list. This can be done through A/B testing where the majority of users interact with the recommendation system, and a holdout set of users serve as the control group and receive suggestions from a top selling items list.

39.5 Conclusion

Recommendation systems have transformed how users discover content and many games are now using these systems in practice. This chapter has provided an overview of common algorithms for building recommendations, provided examples for setting up a system in different languages and environments, and discussed options for deploying a system for a game. Recommendation systems can help improve monetization in a game or marketplace, and also have the potential to help players discover new games to play and find novel content that they would not have otherwise discovered.

References

Anil, R., Owen, S., Dunning, T., and Friedman, E. 2010. *Mahout in Action*. Greenwich, CT: Manning Publications

Hahsler, M. 2011. Recommenderlab: A framework for developing and testing recommendation algorithms. Technical Report. https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf (accessed June 19, 2016).

Hastie, T., Tibshirani, R., and Friedman, J. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer.

Koenigstein, N., Nice, N., Paquet, U., and Schleyen, N. 2012. The Xbox recommender system. In *ACM Conference on Recommender Systems*, Dublin, Ireland, pp. 281–284.

Koren, Y., Bell, R., and Volinsky, C. 2009. Matrix factorization techniques for recommender systems. *Computer*, 42(8): 30–37.

Linden, G., Smith, B., and York, J. 2003. Amazon.com recommendations: Item-to-Item collaborative filtering. *IEEE Internet Computing*, 7(1): 76–80.

Medler, B. 2008. Using recommendation systems to adapt gameplay. In *Discoveries in Gaming and Computer-Mediated Simulations: New Interdisciplinary Applications*, ed. R. E. Ferdig. Hershey, PA: IGI Global, pp. 64–77.

Ryza, S., Laserson, U., Owen, S., and Wills, J. 2015. *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. Sebastopol, CA: O'Reilly Media.

Weber, B. 2015. Building a recommendation system for EverQuest landmark's marketplace. *GDC Talk*. http://www.gdcvault.com/play/1022431/Building-a-Recommendation-System-for (accessed June 19, 2016).