Team Members: Brian Choi, Misun Ryu, Dharmesh Panchal, Andrew Le
Advisors: Julian McAuley, Ilkay Altintas

Building Recommender Systems for Video Games on Steam

Final Report

Abstract

Recommendations are a major part of companies' business today, helping guide customers toward products that they will eventually want to buy. As the largest video game distribution platform on PC systems, Steam similarly relies on recommendations to improve user engagement on their platform. Our group used data gathered from Steam's website to build a recommender system that can accurately predict which games a user should and will buy next. Our model seeks to emulate recommender systems commonly seen on platforms such as Netflix, Amazon, and Spotify to provide interesting and varied recommendations as well as open a deeper understanding of the users and games on Steam itself.

Our data pipeline ingests data containing user's purchase histories and item descriptive summaries to train numerous recommender models targeting different user purchase motivations. Our models mainly use collaborative filtering with a regression component utilizing descriptive features to form predictions in the form of scoring and ranking. The resulting models are used in an interface that given user or game input, outputs recommendations ranging from a single overall list to genre-based lists and item-to-item lists. The interface also provides visualizations exploring user and item preferential patterns and general statistical analysis of the entire data set.

Using Bayesian Personalized Ranking, we were able to create a model that successfully predicted user preferences, using video game ownership as a target variable. We additionally used dimensionality reduction and clustering to further explore the relationship between users and games. Substituting the classic Bayesian Personalized Ranking algorithm with the Weighted Approximate-Rank Pairwise loss function gave us a substantial improvement in predictive power and we were able to further improve performance with additional feature selection and data filtering. We also cover analysis of our model's response to the cold start problem and how well the hybrid content-collaborative model compares in performance to the pure collaborative filtering model.

Introduction

Challenge

Recommender systems are a cornerstone of modern machine learning and data science. With applications ranging from commerce to social media to government, the effects of recommender systems can be seen in countless places. As the name suggests, recommender systems are models that predict a rating or preference for an item given a user. Differing from traditional machine learning models, recommender systems do not fall neatly into a supervised or unsupervised framework but rather uses techniques from both to achieve its goals. Much of the modeling in recommender systems center around interactions between "users" and "items" (in the case of our project, video games) so the two terms will be used liberally throughout the remainder of the report.

As one of the clearest use cases today, Netflix uses a variety of recommender system models to provide its users with movies and shows they are most likely to enjoy. This plays largely in to Netflix's business model of not only providing an easy and convenient streaming service but also actively tailoring its content so that users are likely to continually engaged. Similarly, Amazon uses a variety of recommender system models to provide users with items they are most likely to buy. While Amazon has become a huge all-in-one online shopping place over the past two decades, its catalogue is huge and users may not always know what to buy. In fact, users may not even be aware of the existence of an item they may want to buy. For Amazon's bread and butter e-commerce department, recommender systems play a huge role in increasing revenue and user engagement and discovery.

Our project is focused on the recommendation of videogames on Steam, a digital video game distribution platform developed by Valve Corporation. Originally built to support integration of its first party software, Steam has evolved in to an all encompassing platform for third party videogames, covering nearly every video game released on personal computers (Windows, Mac, Linux). By providing a simple way for users to buy, download, and store their games across multiple machines, Steam has become the #1 video game distribution platform for PCs users. Much like iTunes for music, once a user has bought a video game on Steam, they can forever access it through their account. Player and game data on the platform are tracked religiously, providing rich resources for modeling user behavior and video game engagement.

Goal

Our goal is to build a robust recommender system on Steam data that not only accurately predicts video games that users are likely to buy but also us understand patterns in user behavior and item description better. Along with our final client facing product which will provide recommendations given a user, we are including a heavy pre and post-modeling data analysis component to our project that will lead to insight in to the data set itself. Some of the main questions we intend to answer are:

- How well can user purchase patterns in our data set be predicted?
- In what ways can we shape our data to maximize likelihood of video game purchase?
- Many games are bought but never played. While it is important to recommend games that users will ultimately play, how can we simultaneously recommend games that users will and also maximize revenue for the parent company? In what ways can these two questions be simultaneously answered?
- How can we best recommend video games to new or inactive users? This is also known as the cold start problem.

- Are we able to find logical and insightful groupings of games using collaborative filtering? For example, will be able to see certain genres of games (FPS, RPG, action, puzzle, etc.) grouped together purely by looking at who buys those games?

We also addressed these additional ancillary questions regarding our experiences with the project:

- What are some limitations of our data set and model? What are some questions we aren't able to answer using only what was available to us?
- What are some technical elements of our model that we need to consider and work around? Examples include hyperparameter tuning, scaling and data gathering.
- What are some of the latest research done in the field of recommender systems and how can we apply them to our project?

Related Work

Our work builds on top of previous work done in recommender systems. The Netflix Prize Problem was a pivotal turning point in recommender system research and was a major inspiration for our work. The BellKor team used matrix factorization to greatly improve Netflix's rating algorithm, opening a new wave of development in the field. Bayesian Personalized Ranking (BPR), developed by Rendle et. al., was another major development in the field of recommender systems and will also be heavily referenced in this report. We will discuss the Netflix Prize Problem and Bayesian Personalized Ranking in greater detail later in the report.

Our advisor, Professor Julian McAuley, specializes in recommender systems at the University of California, San Diego and guided us on the theoretical underpinnings of our project, especially relating to our understanding and implementation of Bayesian Personalized Ranking. Professor McAuley has published a number of papers on recommender systems and has a repository of data sets that can be used for recommender system research. The repository of data sets can be found here: https://cseweb.ucsd.edu/~jmcauley/datasets.html. A full list of Professor McAuley's publications can be found here: https://cseweb.ucsd.edu/~jmcauley/. We found one of his papers, "VBPR: Visual bayesian personalized ranking from implicit feedback", of particular interest to our work. In that paper, Professor McAuley combined Bayesian Personalized Ranking with a Convolutional Neural Network to create a hybrid model that combined collaborative filtering with visual features from convolution. The paper influenced our work on building a hybrid model.

Finally, as the #1 platform for games on PC, Steam already has a recommender system. Lacking benchmarks from Steam, we are not necessarily trying to replicate or improve on what Steam already has implemented but instead building something similar using our own methodology to understand more about the platform itself and the gaming industry. We do use what Steam currently has implemented as inspiration.

Team Roles and Responsibilities

- Brian Choi - Product management, Programming, Research
  - Brian managed the team and was responsible for setting up external meetings with Professor McAuley and internal meetings between team members. In the early stages, he focused on reading literature and researching methodologies to help the team gain an overall greater understanding of the problem at hand. He helped in some of the early work on data preprocessing and data pipeline and also contributed to the early work in building the model in Tensorflow. He later mostly contributed to developing the final model, outputs, and analysis used in the final deliverable. In particular, he focused on the analysis of the different models, cold start users, and PCA of genres.
- Dharmesh Panchal - Research, Development, Visualization
  - Dharmesh worked on data processing, researching, and testing new potential methods for building model. He also worked on the prototype recommendations that used LightFM. He analyzed item-item similarity model to understand how the level of playing time affects the results. In addition, he helped design some of the presentations and the Tableau dashboard used to illustrate the games recommended by the model. He scraped the game data for use in the deliverable.
- Andrew Le - Data analysis, Programming, Visualizations
  - Andrew worked on data analysis, configured AWS and installed the necessary software for team. He also helped resolved other sysadmin issues caused by AWS environment and coordinated between team UCSD computer to resolve any EC2 issues. He also worked on data extraction, processing and analysis. He programmed intermediate data models and used them to provide analysis plots such as bar/histogram/scatter charts, PCA plots. He later assisted in providing input data and other research to use on Tableau dashboard.
- Misun Ryu - Data engineering, Programming, Visualizations
  - Misun focused on data processing, researching, building a hybrid model and creating a dashboard for a visualization of the final recommender model. In the early stage, she worked to pre-process steam datasets to analyze and understand the details of each feature to make data ready to build recommender models. She built a hybrid model by engineering user and item features from the dataset and execute the models to compare the performance and analyze the result. At a later stage, she focused on processing the recommender model outputs and create a dashboard to present the game recommendation for users and the game recommendation by the game selection effectively.

Data Acquisition

Data Sources

The data was provided by Professor Julian McAuley on his website (https://cseweb.ucsd.edu/~jmcauley/datasets.html#steam_data). The data consists of item_user data, review data, item metadata, and bundle data. For the purposes of our project, we only primarily used the item_user and item metadata with review data used sparingly to augment the item metadata, particularly to derive a more precise review score for each game. The bundle data was interesting and a different path we could have taken our project on but ultimately was not incorporated in to model.

Name	**Description**	**Storage**	**Size**
Australian_user_reviews	User review of games of Steam. Scrapped from user profiles.	Flat File	25,799 entries 26,628 KB size
Australian_user_items	User profiles of Steam users. Describes users and the games they own.	Flat File	88,310 entries 541,170 KB size
Bundle_data	Bundle metadata. Lists all currently existing bundles/discounts on Steam.	Flat File	615 entries 817 KB size
Steam_games	Game metadata on Steam	Flat File	32,136 entries 19,735 KB size

The data is composed of the following fields:

Steam_reviews
- Username (string): Username of the reviewer
- Product_id (numeric): ID of game
- Page (numeric): The page # the review appears on.
- Page_order (numeric): Unknown for now… possibly the way the pages were sorted.
- Text (string): Main text of review
- Hours (numeric): Number of hours played by user.
- Recommended (bool): Gives True/False recommendation of game
- Date (date): Date review posted
- Products (numeric): Unknown for now… possibly the number of games the user owns.
- Early_access (boolean): Whether the user had early access to the game (can play before the official release date)

Steam_games

- App_name (string): App name for game
- Title (string): Full title of the game
- Item_id (string/numeric): ID of game
- Url (string/url): Link to main page for game
- Review_url (string/url): Link for the game review
- Genres (list): Categor(ies) of the game
- Tags (list): Categor(ies) of the game. Similar to genre but functions as hashtags
- Release_date (date): Date when game was released
- Discount_price (numeric): Discounted price of game
- Price (numeric): Listed price of game
- Specs (list): specifications, description of game
- Early_access (bool): Whether the user had early access to the game (can play before the official release date)
- Developer (string): creator of the game
- Publisher (string): publisher of game
- Metascore: a score of game

User_reviews

- User_ID (string/numeric): Foreign key linking to user_items
- User_URL (string/url): Hyperlink to user's profile page
- Review (list): List of all reviews user has ever posted
  - Item_id: Foreign key of the game linking back to reviews
  - Last_edited (string/date): Gives the date when the review was last edited
  - Posted (string/date): Gives the date when the review was last posted
  - Recommend (bool): Gives True/False recommendation of game
  - Review (string): Text of the review
  - Helpful (numeric): Gives the number of people that found the review helpful
  - Funny (numeric): Gives the number of people that found the review funny

User_items

- User_id (string): Primary key of the user
- Steam_id (string/numeric): Steam ID number. Sometimes same as user_id but not always. Steam_id appears to always be numeric.
- User_url (string/url): Link to steam user profile
- Items (list): Games owned by user
  - Item_id (string/numeric): ID of game
  - Item_name (string): Title of game
  - Playtime_forever: Number of minutes played since purchase
  - Playtime_2weeks: Number of minutes played in the last two weeks

Bundle_data

- Bundle_id (numeric): ID of the game bundle

- Bundle_discount (numeric/percentage): Percentage discount rate of the bundle
- Bundle_price (numeric/price): Listed price of bundle
- Bundle_final_price (numeric/price): Final price of bundle
- Bundle_name (string): Name of bundle
- Bundle_url (string/url): URL for the bundle games
- Items (list): List of games that included in the bundle
    - Item_id (string/numeric): Foreign key linking back to items primary key..
    - Item_name (string): Title of game included within bundle
    - Discounted_price (numeric/price): Discounted price of game within the bundle.
    - Item_url (string/url): URL to main page of game
    - Genre (string): Category of games included in bundle. Perhaps should be list.

Data Collection

The original data set was scraped and collected by Professor Julian McAuley with the help of his former and current graduate assistants, Rajiv Pasricha and Wang-Cheng Kang. The primary data file (user-item file) was gathered by first using the Steam API to obtain the IDs of regional users (Australian) and their html user profiles were then parsed to obtain the games they own and how many hours they have played it all time and in the last two weeks. Note that because of our data set is static, we have a lack of temporal data beyond two fields playtime_forever and playtime_2weeks. While we were able to make extensive use of the playtime_forever in our modeling, the lack of a true temporal component, data points collected on the playtime_2weeks over a period of weeks, months, or years, made it hard to do true time series analysis in our component. Time series were therefore not a large part of our project.

Steam_games and steam_reviews were scraped using a modified scrapy script found here (https://github.com/prncc/steam-scraper). Much like how user_data was obtained, the crawler parsed profiles of video games on Steam and their user reviews and collected them in the form of JSON files. Professor McAuley's website has additional review data and item metadata beyond the original set obtained by Rajiv Pasricha but unfortunately lacks the user_item we need to expand our collaborative filtering model. Wang-Cheng Kang helped us understand how this data was obtained and gave us tips toward using the script to collect more data with the same schema.

For this project, we ended up not utilizing more data. We devoted some time to building a web crawler to obtain more data but switched the focus to analysis after running in to multiple roadblocks. We successfully scraped more data for the steam_games and steam_reviews data sets but was not able to obtain more data for user_items, the most important data for our model. We leave additional data collection as future work for this project if continued. While we would have liked to generalize our model to a larger data set, we felt we were still able to achieve our stated goals with the data set we had available.

Data Pipeline Diagrams

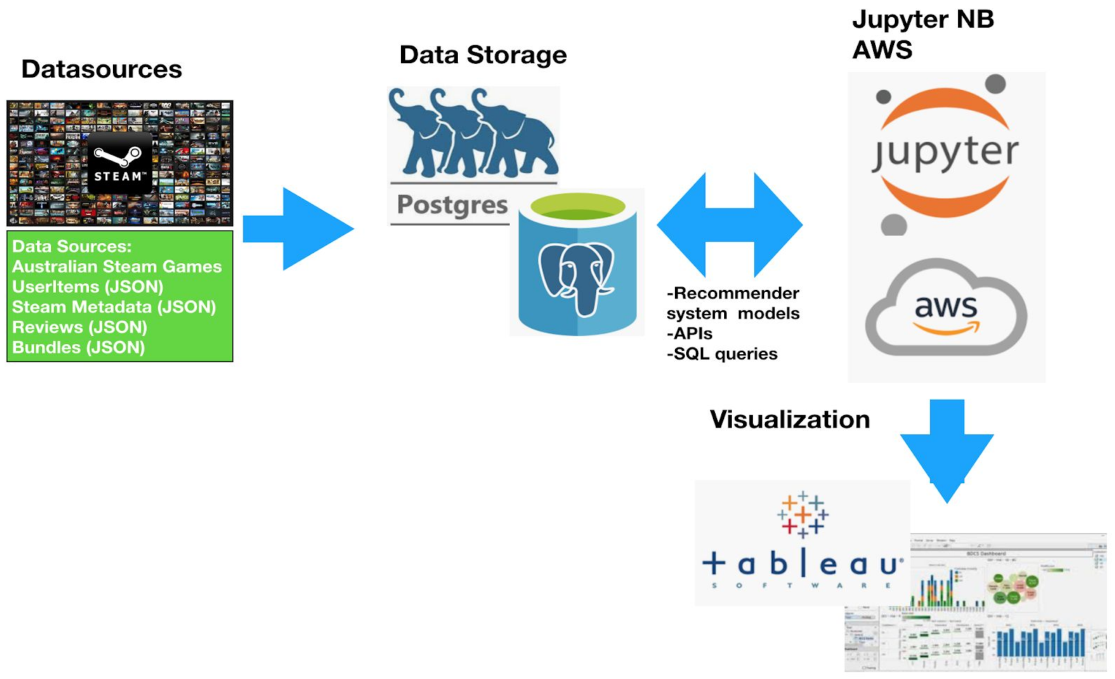


Figure 1: High level description for Steam recommender systems starting from raw data sources and ending at our Tableau visualization user interface

		Environment	*Process Data*	*Derive Models*	*Analyze Results*	*Visualization*
Datasource						
Python		Jupyter notebook version 3.6 installed on EC2 m5ad.12xlarge used to import/parse JSON data.	Create user-item matrix. Pre-process data joining user-item data to game metadata	Find optimal hyperparameter using 5-fold cross validation. Train final model.	Perform post-training analysis of model on: -Cold start problem -Games bought vs played -BPR vs WARP -Hybrid Model	
PostgreSQL		PostgreSQL v11.0 installed on EC2 m5ad.12xlarge used to import data and store temporary tables.		Save performance of hyperparameter sets using		

Figure 2: Detailed description of each process in our pipeline

Setup for Data and Analytics Environment

Our initial work was done on local machines. Since our data is largely medium sized, it could be reliably preprocessed and loaded to memory on most computers. Additionally, because the data was mostly hierarchical, JSON files were appropriate and sufficient for storing and reloading the data when needed. We initially considered using MongoDB or Postgres to store our JSON files but because of our data was not large enough where we needed to worry about storage issues or querying times, we did not pursue this option.

We later created EC2 instances for us to build and run our models on. The data was stored similarly on EC2 using flat files. Because the file sizes were generally small and loading and querying our data was more efficient using the JSON module and Pandas methods, we kept the flat file format of our work. While we did set up a SQL database on our server to work with, as later explained it was mainly used for hyperparameter validation. We did not use AWS for our implementation in Tensorflow, explained in more detail in the section below. A powerful instance for deep learning would have significantly helped with some of the troubles we had with our Tensorflow implementation but we felt LightFM in combination with EC2 offered a much stronger solution. LightFM offers natural, easy to use CPU multiprocessing ability making it synergize with the large EC2 instances we created very well.

We created three instances to work with, a t2.micro for testing and learning the ins and outs of AWS EC2, a m5ad.4xlarge for most of our work leading up to the final two weeks, and a m5ad.12xlarge for all of our work in the final week. The m5ad.4xlarge has 16 CPUs available for computation while the m5ad.12xlarge has 48 CPUs available. The m5ad.4xlarge also uses 2 x 300 NVMe SSDs while the m5ad.12x large uses 2 x 900 NVMe SSDs allowing much faster work for moving data in and out of memory.

The m5ad.4xlarge instance sufficed for most of our work. In the figure below, we can see that while training time significantly decreases as CPUs are added, it is not decreasing linearly as naively expected. As learned in previous big data and database courses, parallel processing often has bottlenecks, particularly in data flow, that keeps run time from improving linearly. The figure shows that training time improves 3.6x as we move from 1 CPU to 20 CPUs which is still a significant improve but we found utilizing 10 CPUs under the m5ad.4x large system was fast enough for our work. However, the large m5ad.12x large did allow multiple users to multiprocess their model training which proved important in the final two weeks of the project.

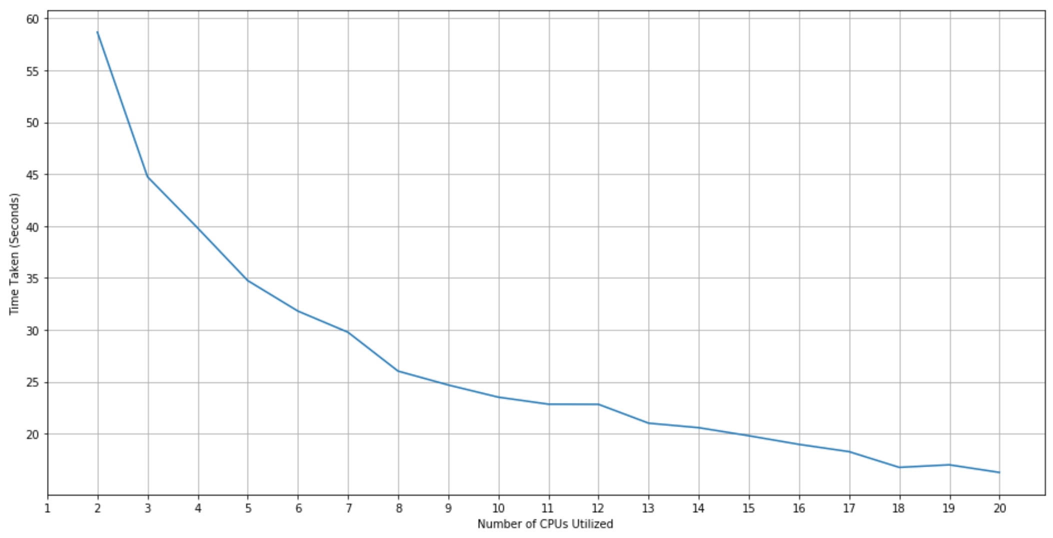


Figure 3: Plotting train time vs number of CPUs.
Models were trained on 15 epochs with the WARP loss function.

Data Preparation

Issues

Although data as provided by Professor McAuley came as JSON files with minimal problems, the data needed to be reformatted before working on them. Our early work focused on transforming the text such that it could be easily read by Python's JSON module. We wrote a function to quickly rewrite the data in a native JSON format that could be easily read. The bulk of the data remained but the function cleared the vast majority of issues with special characters, spacing, and quotes vs double quotes that we were encountering early on.

The data itself needed to be extensively cleaned. While the user-item data was fairly clean, the steam_games data had a lot of missing or unclean fields that needed to be fixed or filtered out completely. Steam_games's item_id field served as the primary key for all joining. Because app_name does not always match between australian_user_items and steam_games, item_id is used as a common join field between the two tables. In many cases, item_id was missing altogether in which case we did not consider the game at all. The price field also required fixing, especially to distinguish free games from games that require cost. Both fields are important in our analysis methods but are often missing from the game metadata.

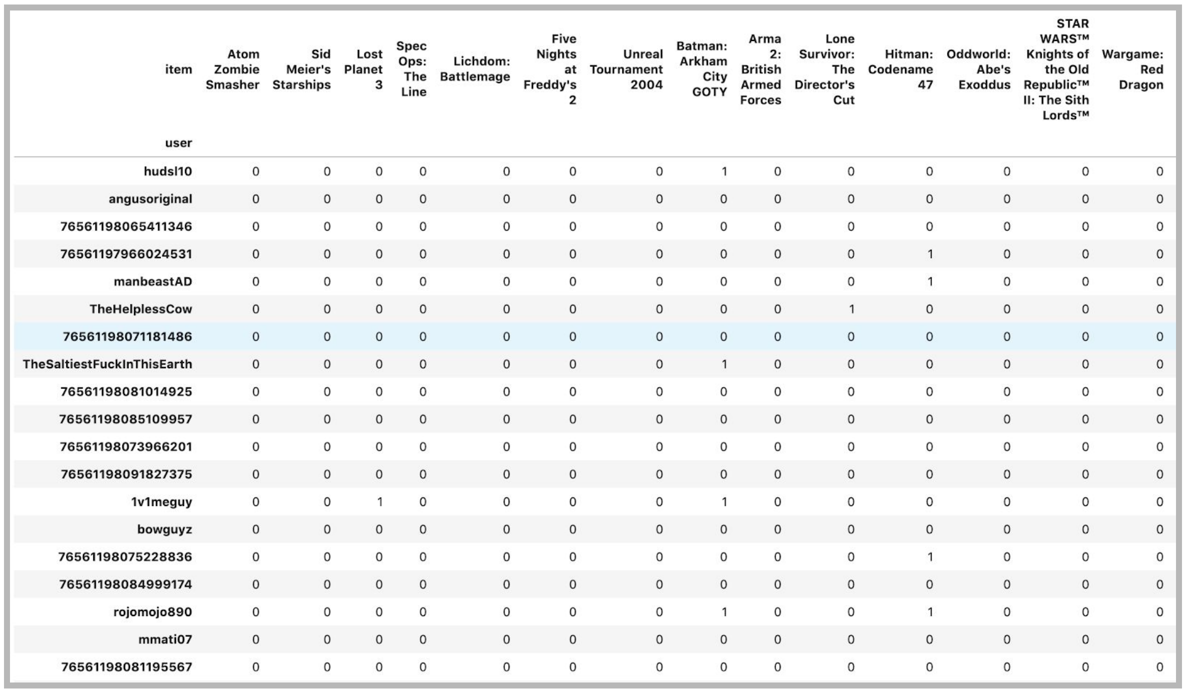
Games with multiple editions are sometimes listed as different items in our data set, effectively splitting the user-item interaction of that game across two or more largely disjoint set of users. While listing multiple editions does not often happen, it occurs frequently for games that are popular enough to gain Game of the Year (GOTY) or Director's Cut editions. These popular games often provide a lot of information to a collaborative filtering model so splitting its user base will negatively impact the model. Some examples of popular games with GOTY editions include Batman: Arkham Asylum, Batman: Arkham City, Deus Ex, Fallout 3, Plants vs

Zombies, and Unreal Tournament. Fallout 3 is an especially important example because the non-GOTY edition and GOTY edition are respectively ranked 320 and 122 in the top 1,000 games by user ownership. Fallout 3 by total ownership would easily break top 100 if combined. The data set and our model consider the two games essentially separate and while they will contribute similarly in the model, we believe the total predictive power of the item is reduced compared to when they are combined as one game. Luckily, the vast majority of games are only listed as a single edition and this is a relatively rare problem in our data set.

Downloadable content, or DLC for short, were originally thought to be a potential problem with the data set. DLCs have become a large part of the industry as video game publishers have aimed to maximize profits by through post-release DLCs DLCs are commonly simple add-ons to other games and normally serves to extend a game's storyline, gameplay, or cosmetics. During EDA, we believed the listing of DLCs would similarly split a game's ownership or playtime, causing a similar issue as GOTY editions but on a much larger scale. However, on further analysis, our user-item data set did not include DLCs as games owned and all playtime for a game and its DLCs are consolidated to its main game. We were able to significantly reduce pre-processing and joining of games and their DLCs.

Data Transformation/Pre-Processing

The user-item or interaction matrix is the most integral data structure for our model. The user-item matrix is an m_u by n_i matrix where m_u is the number of users and n_i is the number of items in the system. The matrix describes the interactions between users and items, in our case whether or not a user owns a particular game. Starting with the australian_user_items data, we parse the JSON file to produce a list of all user-item interactions then pivot the table to generate the user-item matrix.



item	Atom Zombie Smasher	Sid Meier's Starships	Lost Planet 3	Spec Ops: The Line	Lichdom: Battlemage	Five Nights at Freddy's 2	Unreal Tournament 2004	Batman: Arkham City GOTY	Arma 2: British Armed Forces	Lone Survivor: The Director's Cut	Hitman: Codename 47	Oddworld: Abe's Exoddus	STAR WARS™ Knights of the Old Republic™ II: The Sith Lords™	Wargame: Red Dragon
user														
hudsl10	0	0	0	0	0	0	0	1	0	0	0	0	0	0
angusoriginal	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561198065411346	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561197966024531	0	0	0	0	0	0	0	0	0	0	1	0	0	0
manbeastAD	0	0	0	0	0	0	0	0	0	0	1	0	0	0
TheHelplessCow	0	0	0	0	0	0	0	0	0	1	0	0	0	0
76561198071181486	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TheSaltiestFuckInThisEarth	0	0	0	0	0	0	0	1	0	0	0	0	0	0
76561198081014925	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561198085109957	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561198073966201	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561198091827375	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1v1meguy	0	0	1	0	0	0	0	1	0	0	0	0	0	0
bowguyz	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561198075228836	0	0	0	0	0	0	0	0	0	0	1	0	0	0
76561198084999174	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rojomojo890	0	0	0	0	0	0	0	1	0	0	1	0	0	0
mmati07	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76561198081195567	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4: Example of user-item matrix. Note sparsity of matrix.

We filtered the number of games in our model to the top 1000 most popular games among users. This is done for performance and aesthetic reasons. From exploratory data analysis, the percentage of users who own games past the top 1000 games becomes significantly sparser. When looking at other similar recommender systems on commercial websites such as Amazon, Netflix, and Steam itself, it is extremely uncommon for a rare or unpopular game to be recommended. Because Steam only deals in video games, products which are largely of the same use (entertainment) and appealing mainly to the same demographic, we felt a top 1,000 list of games would make the make sense. The sparsity of games past the top 1,000 will be covered in more detail in the EDA section of this paper. Additionally, because every game in the system will need to be trained with its own embedding, filtering to only the top 1,000 games out of 10,497 games greatly improves training time as well as helps the model reach convergence as games with sparse data are avoided.

As mentioned in the issues subsection, not every game listed in australian_user_items can be joined to the steam_games table. We require the game metadata from the steam_games table for the final output so we filter all games in australian_user_items where the item_id is missing from steam_games. Because the steam_games data was scraped sometime after the australian_user_items and australian_user_reviews data, some games may have been discontinued or removed. This is commonly seen with some extremely popular games such as Elder Scrolls V: Skyrim which was removed for the Game of the Year Edition (essentially the same game but listed as a separate item). As a result, some games that users own in Australian_user_items may not appear. Between all games in Australian_user_items and steam_games, 1,760 out of 10,497 were mismatched. This was not a problem for our model as we adapted by taking the next most popular game if one in the top 1000 was missing in the join.

We also filtered out free games for our final model. This topic will be covered in more detail in the findings section but was an essential step that was inserted late in to our data preprocessing pipeline. Early on we did not filter out free games but we began noticing that certain free games dominated the recommendations as well as games owned. This skewed our model and degraded the overall performance of our model. One of our stakeholders is Valve itself which wishes to improve revenue on its platform. Downloading a free game comes at no cost to the user and dilutes the list of recommendations which can ultimately push a user to actually spend money on the service. The lack of cost to the user meant that free games fail to meaningfully contribute a collaborative model which relies on users' personal tastes. We had to parse the steam_games price field to convert certain strings to price equal to zero which we then filtered in the following data preprocessing steps.

Feature Selection

Feature selection played a secondary but important role in our model. The primary feature we used was game ownership by user which served as the interactions for our collaborative filtering model. In addition to a collaborative filtering model, we also built a hybrid model which is a combination of collaborative filtering and content-based recommendation model. By building a hybrid model, we could have a benefit from two kinds of recommendation models and was able to compare the performance with a pure collaborative filtering model. To build a hybrid model, below features are selected and processed to build a content for users and items.

- User features
    - Playhour_forever: As Playtime_forever is recorded in minutes, the data is converted by 10 hour unit to make the feature simple.
    - Playhour_2weeks: As Playtime_2weeks is recorded in minutes as well, this is converted to hours played.
- Item (game) features
    - Genres: Most games are categorized in several genres at the same time; hence all genres are flattened to generate a binary matrix for all games. We trained and tested our model with the top 1000 games, and these games includes 18 types of genres.
    - Price: price data is rounded to a dollar to make the data simple. If price does not exist for a game, the price is set to zero and regarded as a free game.
    - Release date: This data indicates when the game is released, and it is converted to a year to make the data simple.
    - Playhour_forever: The median of playtime_forever is calculated from all users who owned the game to add a playhour for a game.
    - Playhour_2weeks: In the same way as Playhour_forever, this is calculated from playtime_2weeks of all users who own the game.
    - Metascore: Initially this was considered as an item feature; however, metascore is missing for more than 50% of the games that we are dealing with; hence this feature is dropped and not used for a hybrid model.

Analysis Methods

Exploratory Data Analysis

The exploratory data analysis stage of our work focused on discovering basic statistics about the game and how they might shape the questions we will eventually answer for our project. During EDA, we did not have a specific data set in mind and explored user_item and steam_games equally to find out as we can about our data. In particular, we looked at games owned, playtime by users for each game, and game release dates.

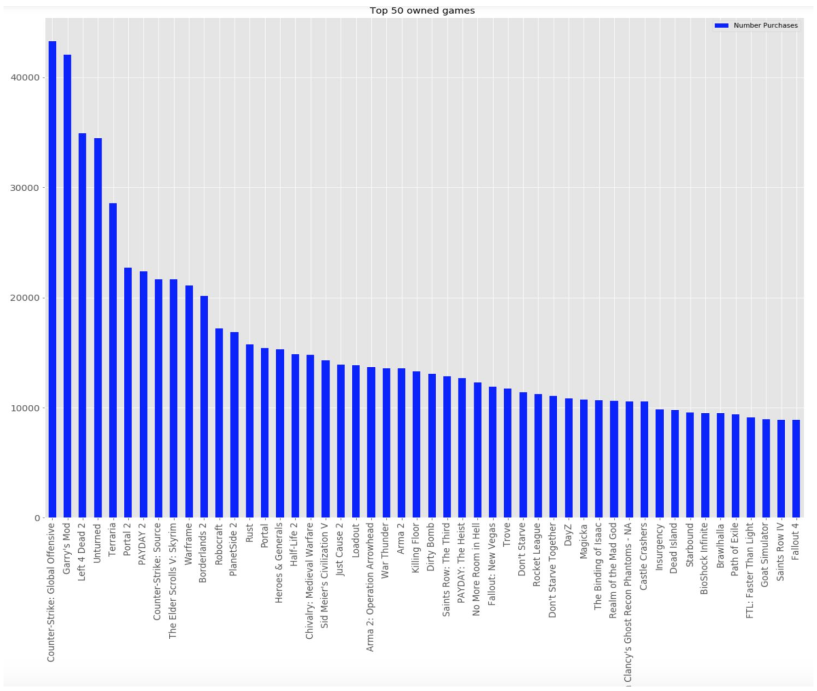


Figure 5: Top 50 games owned/bought by all users

The above figure shows the top 50 owned games by users considering all genres. We notice a higher peak for the top 10 games considering purchasing frequency. These games generally have [‘Action’, ‘Adventure’] as their Genres or Tags which reflected gamer’s trend. Our later PCA analyses show these 2 genres are more popular kinds. We then conducted further analyses to see whether these 50 games have the same trends considering “PlayTime2Weeks” and “PlayTimeForever” parameters in the subsequent figures.

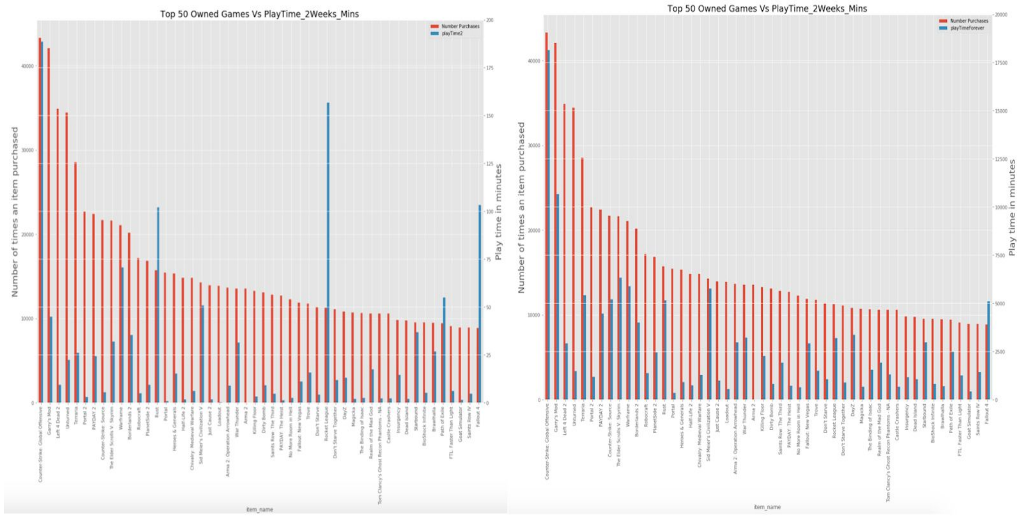


Figure 6: Comparison between PlayTime_2Weeks (left) and PlayTime_Forever(right) for the top 50 games mentioned in the previous figure.

However, the play frequencies (2 weeks or ever play) didn't follow the same trend as purchasing game frequency. Looking at the below figure below, we see that popular games are not necessarily the most played. Some of the games included in the histogram are free meaning they could be downloaded and not played at all. With Playtime_2weeks, some games are simply old and not played anymore, leading to further decline in their popularity. PlayTime_Forever shows a more "agreeable" trend than "PlayTime_2Weeks" when looking at the top 10-20 games, showing. These results can be biased because gamer's play trend is various attributing to different characteristics (e.g. game type, game difficult levels…). In particular, multiplayer games have much more replayability than single players, leading to longer play time. These findings led us to consider factoring in playtime in our model in some way.

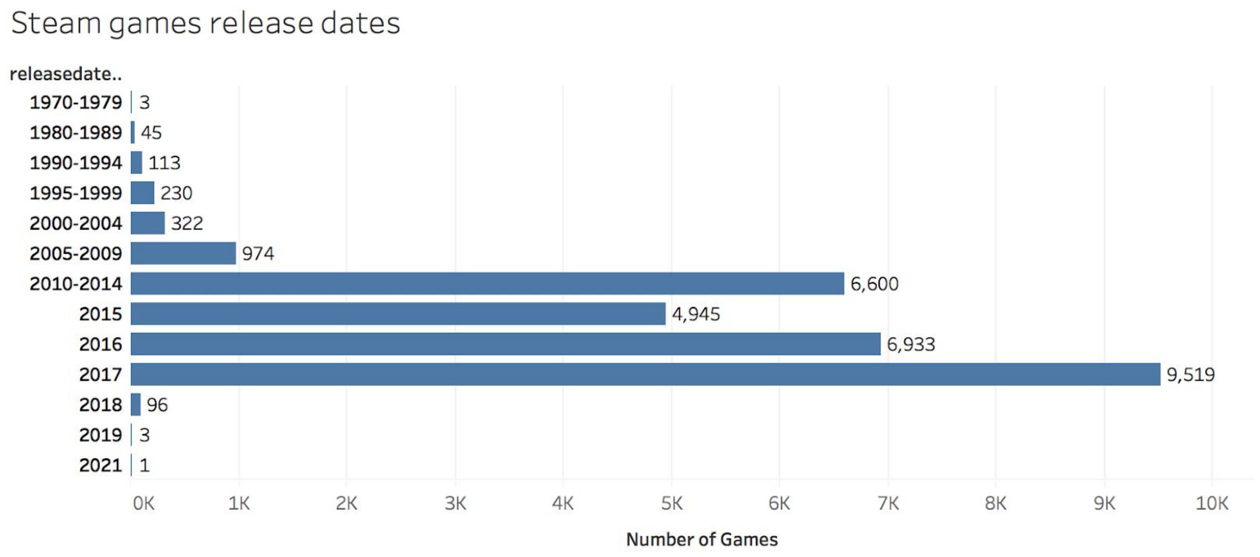


Figure 7: Game release dates

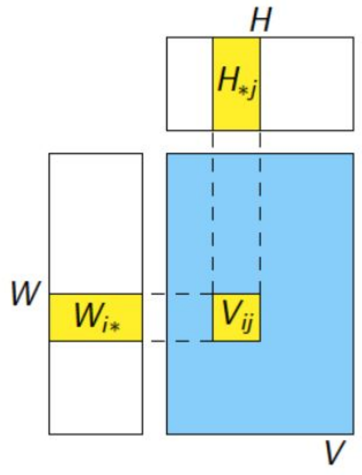
The majority of our games were released between 2010 and 2017. As mentioned in the data pre-processing and issues sections, there was a disconnect between the user_item data and the steam_games data. The steam_games data was scraped much later than the user_item data, resulting in some games from the user_item data set to be missing from the steam_games data set. In particular, our user_item data set only has games up to the end of 2015, making our user_item data set a few years behind. Nevertheless, this did not eventually hamper our progress for the project. Because of the large range of release dates for games on the platform, we eventually used release date as a part of our hybrid model.

Model Background

Our model primarily uses collaborative filtering to predict games that users are most likely to buy. Collaborative filtering is a recommender system technique that predicts a user's behavior and preferences based on other users' behaviors and preferences. For example, even if a user John_Doe has not seen a particular movie, his preference for that movie can be derived by examining other users who have seen that movie and additionally enjoy other movies that John_Doe also enjoys.

The user-item, or interaction, matrix is the backbone of most standard collaborative filtering models and that is the primary data structure we work with. The user-item matrix is a simple matrix representing user interactions with users in the rows, items in the columns, and that user's rating or interaction with the item as the cell's value. In our model, the user-item matrix represents whether or not a user owns a game as 1 or 0 respectively. With the user-item matrix, we can apply matrix factorization to then assign a score for every user-item pair beyond the 0 and 1s in our original data matrix.

Matrix factorization is one of the most widely used techniques in collaborative filtering today. Drawing on linear algebra concepts of matrix decomposition and deeply related to singular value decomposition, matrix factorization algorithms (in the context of recommender systems), generally aim to decompose an interaction matrix into two lower dimensional matrices U and I representing the users and items of the system. Similar to singular value decomposition, these lower dimensional matrices U and I contain latent factors or embeddings, vectors that largely or fully describe the hidden features for each user and item in the system. For example, by decomposing a user-item matrix into a (n x 30) user matrix and (m x 30) item matrix, we are essentially describing each user as a linear combination of 30 numbers. This technique has applications in many fields but proves to be especially powerful in recommender systems as the decomposition allows us to study the relationship between users and items in a purely quantitative, analytical way.



Figures from Gemulla et al. (2011)[33]

One of the best known applications of matrix factorization in recommender systems is the Netflix prize challenge. In 2006, Netflix started an open challenge asking competitors to build a collaborative filtering algorithm that best predicts user ratings for films, with a grand prize of one million dollars. No additional information was provided beyond a user's rating for a movie. As a user-item matrix, the data set would be mostly sparse with most user/movie ratings missing. Note that a missing value in this case would not be interpreted as a 0 but rather as null or no data. One of the most publicized solutions to the Netflix prize, led by the BellKor team at AT&T Labs, used a matrix factorization algorithm called alternating least squares to train embedding for every user and item. A user's rating for a movie can be predicted by very simply taking the dot product of that user's embedding with that item's embedding (with the help of bias terms!). By training the embeddings to predict existing ratings, their dot product can further be used to predict that do not yet exist. BellKor eventually combined their work with Big Chaos, an Austrian team, to win the grand prize, achieving a 10.06% improvement over Netflix's own collaborative filtering algorithm.

In the years following, many other matrix factorization algorithms were developed to solve other problems in recommender systems and collaborative filtering. Our work focused on Bayesian Personalized Ranking (BPR) and the related Weighted Approximate-Rank Pairwise

(WARP) which looks to solve the one class (binary) recommender system problem. In our case, we are looking to predict whether or not a user will buy a game and our interactions are defined by whether or not a user currently owns a game. Our user-item matrix will have a 1 assigned to that user-item pair and a 0 otherwise. In contrast to the Netflix prize problem, none of the interactions are missing and the user-item matrix is fully defined for all cells. However, because the goal is to predict which games that a user currently does not own and is most likely to buy in the future, the 0s in our user-item matrix are similar in function to the missing values of the Netflix rating data set.

In BPR, a score is similarly calculated by taking the dot product of a user's embedding and an item's embedding. After training, the dot product should generate a real number value for that user-item pair which can then be used for further ranking in recommendation. For example, we can find which item to recommend for John_Doe by simply taking the dot product of his user embedding and all other items in the system and then sort by which game has the highest score. BPR is trained by sampling a random positive example and a random negative example from the same user and then trying to maximize the probability that i is preferred over j using an iterative optimization algorithm such as stochastic gradient descent. The gradient is taken with respect to the weights of ɣ_u, ɣ_i, and ɣ_j and then updated with each iteration. The WARP loss functions modifies this by only updating ɣ_u, ɣ_i, and ɣ_j if (ɣ_u*ɣ_i - ɣ_u*ɣ_j) is negative during sampling. In other words, WARP will only update the weights if it finds a violation during training.

$$p(i \text{ is preferred over } j) = \sigma(\gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j)$$

Figure 8: From Julian McAuley's CSE258 slides showing BPR objective. Sigma refers to sigmoid function. Gammas are embeddings/latent factors for users and items. I and j are positive and negative item examples.

For more detail on the mathematical foundation of Bayesian Personalized Ranking, please read the paper "BPR: Bayesian Personalized Ranking from Implicit Feedback" by Rendle et. al. (2009) with link in references.

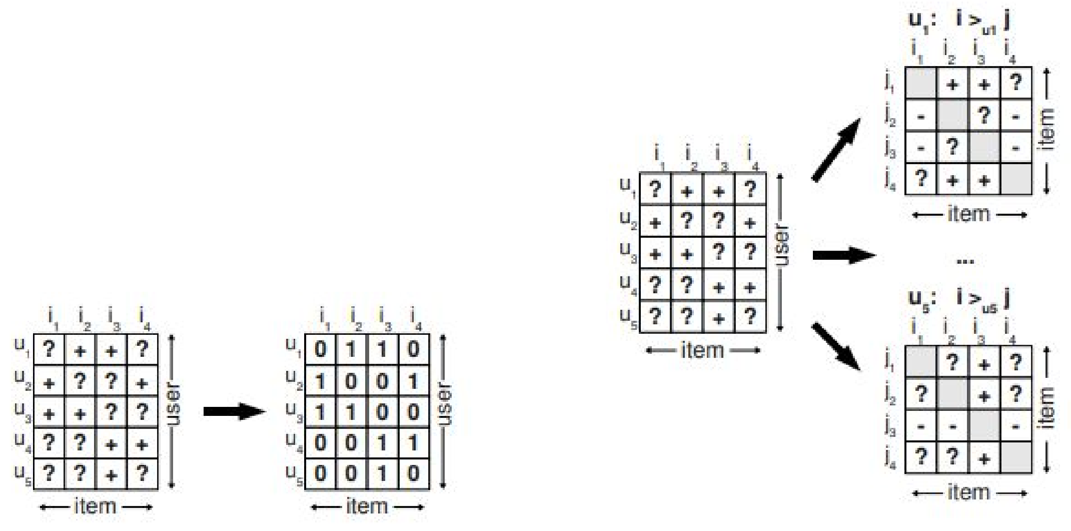


Figure 1: On the left side, the observed data S is shown. Learning directly from S is not feasible as only positive feedback is observed. Usually negative data is generated by filling the matrix with 0 values.

Figure 2: On the left side, the observed data S is shown. Our approach creates user specific pairwise preferences $i >_u j$ between a pair of items. On the right side, plus (+) indicates that a user prefers item i over item j; minus (–) indicates that he prefers j over i.

Figure 9: From Rendle et. al (2009) showing motivation behind BPR

For item-item recommendation, similarity scores based on distance are commonly used to measure the similarity of the games against another one. In our model, we apply cosine similarity within the interaction matrix for each item across all users. Once a score is computed for all items, it is subtracted from 1 to produce a distance that is used for ranking the similarity of the game to the selected item.

$$\cos(\theta) = \frac{\mathbf{M_1} \cdot \mathbf{M_2}}{\|\mathbf{M_1}\| \|\mathbf{M_2}\|} = \frac{\sum\limits_{i=1}^{n} M_{1i} M_{2i}}{\sqrt{\sum\limits_{i=1}^{n} M_{1i}^2} \sqrt{\sum\limits_{i=1}^{n} M_{2i}^2}}$$

Figure 10: Formula for Computing Cosine Similarity Scores of Each Item

Implementation

We focused on two libraries for building our model, Tensorflow and LightFM. Tensorflow was the initial approach we used both to allow ourselves a larger degree of control and customizability and a learning experience for Tensorflow and machine learning engineering. We also found a lack of available libraries and technologies in our early surveys on par with Scikit-Learn, Keras, Tensorflow, Pytorch, and other popular frameworks/libraries for traditional machine learning and neural network. It was therefore necessary to begin our project doing a ground up implementation using Tensorflow. After meetings with Professor McAuley and deciding on Bayesian Personalized Ranking as our learning algorithm of choice, we found several libraries specifically built for BPR. In particular, LightFM and Spotlight caught our eyes with Spotlight being slightly newer implementing some state of the art work done on deep recommender systems. We settled on LightFM because it was professionally built by Lyst and provided a more complete API for our project. However, we continued to work on our initial implementation in Tensorflow until we ran into performance and speed bottlenecks, after which we switched our project to completely using LightFM.

We implemented Bayesian Personalized Ranking using Tensorflow building off code written by Dongxu Duan on Github. Dongxu Dong's BPR repository can be found here: https://github.com/dongx-duan/bpr. Compared to our later work using LightFM, our work with Tensorflow required heavy data preprocessing and data reshaping. Whereas LightFM had many functions automating interaction matrix construction, user/item indexing, metric evaluation, and mini-batch sampling, we were forced to manually code and debug many of these processes ourselves. This was a major time sink during our early work and was a major reason we later switched to LightFM. While we were initially able to build a convincing model using Tensorflow, with performance comparing extremely similarly to LightFM when comparing the same data set and hyperparameters, our implementation would take roughly two hours to converge running on GPU vs LightFM converging in less than two minutes on 20+ CPUs. This was a major factor in our decision to switch fully to LightFM on top of quality of life issues we were facing with our implementation. We suspect that while deep learning libraries such as Tensorflow and PyTorch are exceptional for building neural networks, their computational framework are not optimized for algorithms such as Bayesian Personalized Ranking.

LightFM is a hybrid recommender system for specifically solving the one class recommendation problem. Built by Maciej Kuja for Lyst, the source code is written in C and optimized for recommendation using matrix factorization. Its full documentation can be found here: https://lyst.github.io/lightfm/docs/home.html. Like Scikit-Learn, Keras, and PyTorch, LightFM is built using the familiar workflow of define, fit, and predict, making it easy to pick up for users experienced with other machine learning libraries. It also offers multi CPU processing on both model training and metric evaluation, making it ideal for running on large EC2 instances. One of the main issues with our Tensorflow implementation was the long time it took to evaluate the test set despite significant time towards optimizing the operation. LightFM was optimized for test set evaluation and multi processing for evaluation made the process much simpler. One of our main motivations for continuing work on Tensorflow after discovering LightFM is to further develop the hybrid content-collaborative model. However, digging deeper in to LightFM's documentation, we found that LightFM already has in-built functionality for hybrid modeling, pushing us to adopt LightFM as our machine learning library of choice for the remainder of the project.

Analytical Workflow

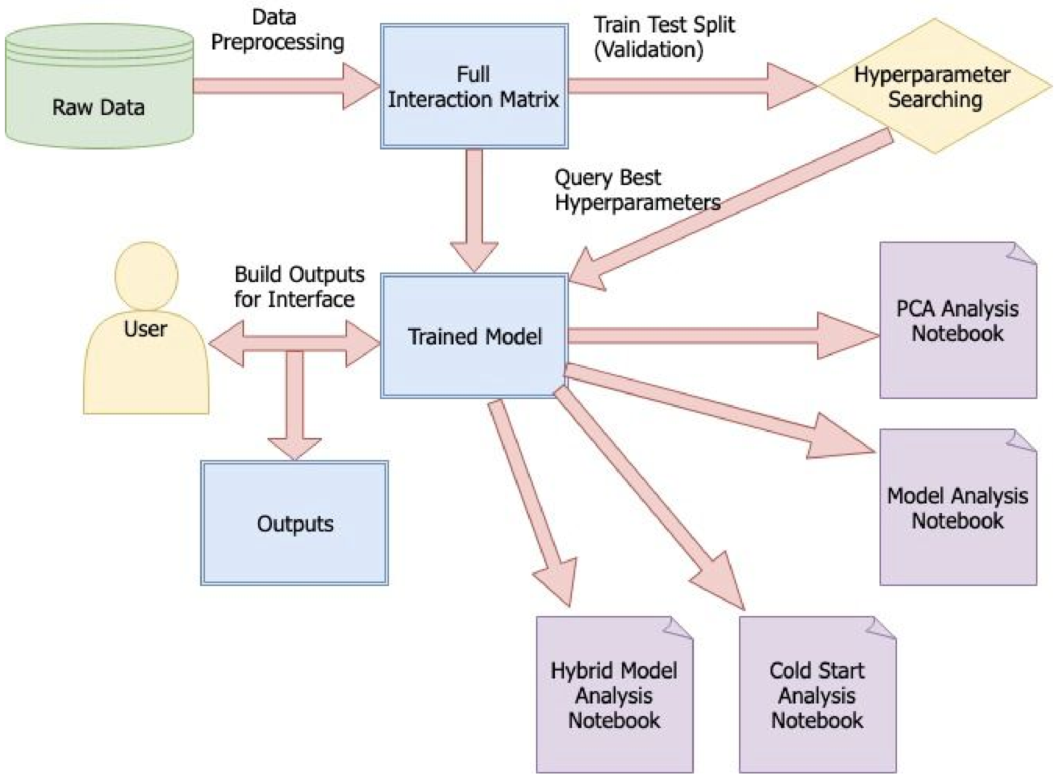


Figure 11: Rough outline of analytical workflow. After training our model we perform various analysis on separate notebooks

After preprocessing our data and tuning our model, we train the final model using the best set of hyperparameters. Using the trained model, we then perform a series of analysis to finalize our findings. Each of these notebooks contribute directly to the major findings in this report. Outputs are generated using cookbook code borrowed and modified for use with LightFM. The cookbook can be found here: https://github.com/aayushmnit/cookbook. Even though we made many changes to the structure of our data pipeline over the course of the last two months, the general analytical workflow has remained the same.

Solution Architecture, Performance, and Evaluation

Evaluation

Our models were primarily evaluated using AUC. AUC-ROC is normally defined as the area under the curve of the receiver operator characteristic and is commonly used method to evaluate the overall performance of a classifier. Most simply, AUC tells how much a model is able to distinguish between different classes and has a wide range of interpretations depending on its application. As applied to recommender systems, in particular binary class recommender systems, the AUC is equal to the probability that a randomly chosen positive example will rank higher than a randomly chosen negative example. For our data, AUC measures the probability that our model ranks an owned game higher than a not-owned game.

Other metrics for recommender systems exist including precision and recall. Precision and recall take on a slightly different definition as commonly seen in classification tasks and for the purposes of this project, will not be the major focus for model evaluation. When applied to recommender systems, precision and recall respectively measure the fraction of known positives in the first k positions of a ranked list and the number of positive items in the first k position divided by the number of positive items in the test period. These metrics are sometimes problematic for our sparse data set and sparser test set as some users have very few interactions to calculate precision and recall from. For this project, we will not be using precision and recall in the evaluation of our models. Finally, reciprocal rank is used to examine how well the models handle the top ranked games. Reciprocal rank measures 1 divided by the rank of the highest ranked positive example, showing how many unowned games are recommended over owned games in the test set.

For evaluating how well our models perform, we used a train test split, splitting 20% of the total data set to 80% train and 20% test. The training set is then further split into 80% train and 20% validation for cross validation. Our findings, examined in greater detail later, showed robust performance resistant to smaller data sets. Despite cross validation only training with 64% of the original data set, we found performance comparable to the final model trained with the full data set.

Model Tuning

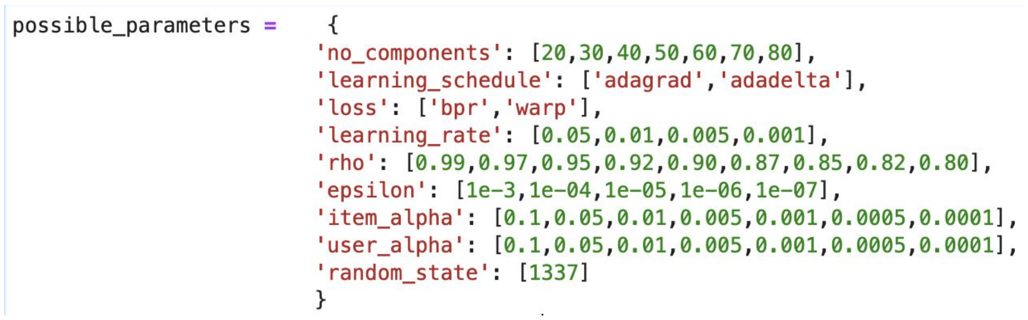
Hyperparameter searching was a large part of optimizing our performance. Poor choice of hyperparameters resulted in huge degradation of performance, making selection of proper set of hyperparameters extremely important. The best performing set of hyperparameters had an average AUC of 0.909 over cross validation while the worst performing set of hyperparameters had an average AUC of 0.377. With LightFM, there were five important hyperparameters to tune:

- Loss - Loss function used for training. Only BPR and WARP considered.
- No_components - Number of components/dimensions for user/item embeddings.
- Learning_rate - Optimizer's learning rate. Controls how fast an optimizer converges. Small learning rate might provide closer to optimal result but take longer to converge. Large learning rate might not give optimal weights or fail to converge altogether.
- Item_alpha - L2 regularization strength for item embedding's weights.
- User_alpha - L2 regularization strength for user embedding's weights.

LightFM is only programmed to run AdaGrad and AdaDelta. Because gradient optimization algorithms generally has the same end goal, the choice between AdaGrad and AdaDelta was not extremely important. That said, if AdaDelta is chosen, two additional hyperparameters need to be tuned:

- Rho - Moving average hyperparameter. Also known as the decay factor.
- Epsilon - Conditioning hyperparameter. Also known as the fuzz factor (see Keras)

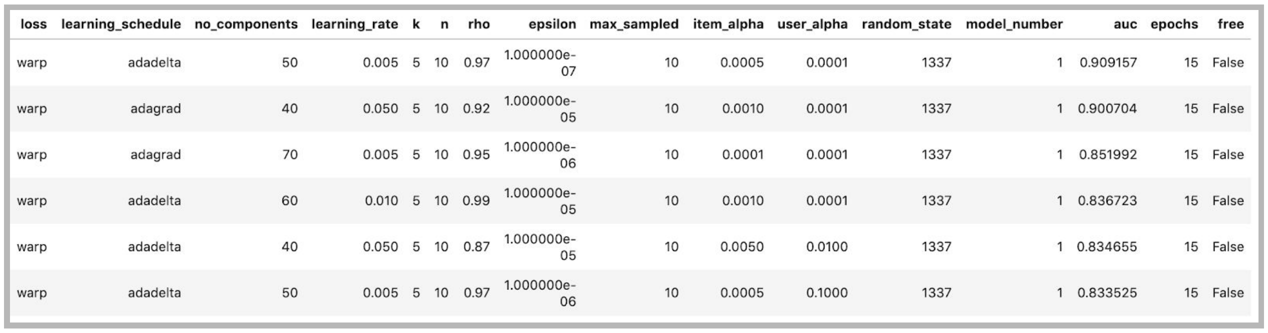
For each of these 9 parameters (8 listed above plus the choice of optimizer), we set a list of values to search over. The hyperparameter values to search are listed in the figure below and altogether, there were 246,960 possible combinations. There were far too many combinations to run a full grid search over so random search over the grid was used instead, searching for merely 200 iterations. Bergstra et. al. (2012) showed that a purely random trials are more efficient for finding the optimal hyperparameters than grid search. While we did not use a pure random search as certain numerical hyperparameters such as our regularizers and learning rates are still discrete, the relatively small number of iterations we ran still showed a huge difference between the best performing set of hyperparameters and worst performing set of hyperparameters and we felt confident that our hyperparameters were close to optimal for the final model.



```
possible_parameters =    {
                        'no_components': [20,30,40,50,60,70,80],
                        'learning_schedule': ['adagrad','adadelta'],
                        'loss': ['bpr','warp'],
                        'learning_rate': [0.05,0.01,0.005,0.001],
                        'rho': [0.99,0.97,0.95,0.92,0.90,0.87,0.85,0.82,0.80],
                        'epsilon': [1e-3,1e-04,1e-05,1e-06,1e-07],
                        'item_alpha': [0.1,0.05,0.01,0.005,0.001,0.0005,0.0001],
                        'user_alpha': [0.1,0.05,0.01,0.005,0.001,0.0005,0.0001],
                        'random_state': [1337]
                        }
```

Figure: List of parameter values used in hyperparameter search

We used 5-fold cross validation for model tuning to combat overfitting. Running the algorithm over 20 CPUs on our EC2 instance, a full cross-validation on a single set of hyperparameters took roughly 90 seconds on average, the time largely dependent on the learning rate. The mean AUC for each iteration is calculated and then saved along with the parameters to a SQL database on our instance. See figure below for example of the validation_metrics table we created. The use of SQL to store our hyperparameters was convenient and resistant to failure from crashes on the notebook or server. For training the final model, we simply query for the best performing hyperparameter set by AUC. Note additional fields below for random_state, model_number, free, and epoch. We used the same random_state for all processes to ensure non-random results on validation. Model_number and free refer to the data set filtering mentioned in the Data Preparation and Analysis Methods sections of this paper.



loss	learning_schedule	no_components	learning_rate	k	n	rho	epsilon	max_sampled	item_alpha	user_alpha	random_state	model_number	auc	epochs	free
warp	adadelta	50	0.005	5	10	0.97	1.000000e-07	10	0.0005	0.0001	1337	1	0.909157	15	False
warp	adagrad	40	0.050	5	10	0.92	1.000000e-05	10	0.0010	0.0001	1337	1	0.900704	15	False
warp	adagrad	70	0.005	5	10	0.95	1.000000e-06	10	0.0001	0.0001	1337	1	0.851992	15	False
warp	adadelta	60	0.010	5	10	0.99	1.000000e-05	10	0.0010	0.0001	1337	1	0.836723	15	False
warp	adadelta	40	0.050	5	10	0.87	1.000000e-05	10	0.0050	0.0100	1337	1	0.834655	15	False
warp	adadelta	50	0.005	5	10	0.97	1.000000e-06	10	0.0005	0.1000	1337	1	0.833525	15	False

Figure 12: SQL query showing top 6 performing set of hyperparameters. WARP is the winning loss function in every set.

Picking the right number of epochs is the final part of tuning our model. To determine the right number of epochs for our final, we trained our models one epoch at a time during validation to find when validation AUC begins decreasing. The optimal number of epochs is also different when comparing BPR vs WARP so comparisons between the two during model tuning must be made with that mind. The following figures are plotted using the best set of hyperparameters for BPR and WARP and respectively and done on the training set with free games filtered out. In the figure below, the train and validation AUC are plotted for each epoch. The train AUC is expectedly continually increasing with each epoch but the validation AUC maximizes around 12-15 epochs and begins decreasing from there, indicating overfitting past 15 epochs.

The optimal number of epochs are clearly different comparing Bayesian Personalized Ranking to WARP. Whereas WARP peaks around 15 epochs and begins overfitting past that point, BPR immediately begins decreasing in performance starting from the first epoch. Interestingly, the train AUC decreases along with the validation AUC in the BPR example while typically a training metric in most machine learning models would continually increase or stabilize. This is especially true in classic overfitting scenarios where the train metric would increase while the validation/test metric would decrease as seen in the WARP plot. While WARP is simply a modified version of BPR in implementation, the ability of WARP to only update embeddings if it detects an incorrect label helps prevent overfitting as seen in the BPR case. In either case, from the small number of epochs needed to reach optimal performance in both WARP and BPR, it can be seen that our data set is highly sensitive to overfitting and extreme care must be taken to avoid it.

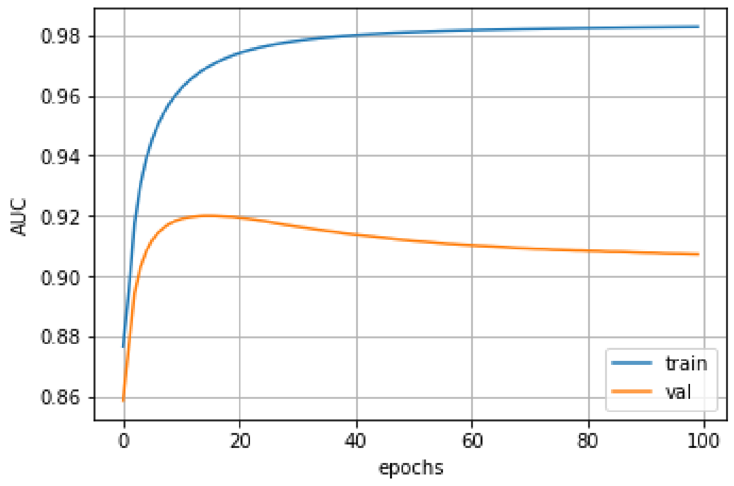


Figure 13: Train vs Validation AUC for WARP

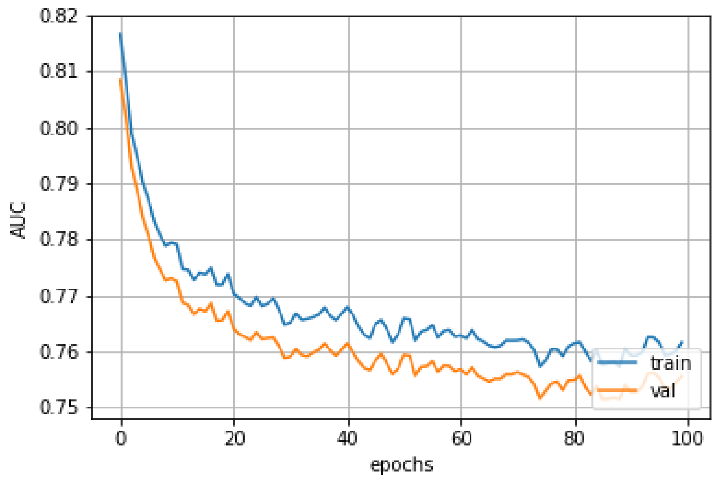


Figure 14: Train vs Validation AUC for BPR

Scalability

Scalability was not a major focus for our project because of the moderate size of our data sets. We had hoped to scrape more data and expand the size of our project but left that work for the future. Most of the scalability elements of our project came down to running hyperparameter searching on AWS and trying to find the optimal hyperparameters for our model. LightFM's support of multi CPU processing allowed us to train our models over many hyperparameter sets very quickly, giving us results that would have otherwise not been possible on local machine.

Scalability of our results for our dashboard was important. We wanted to improve the responsiveness of our dashboard which was running slowly with the size of our output JSON files. We had to scale down the size of our output files to lower the load on Tableau Desktop.

AWS Budget

We efficiently kept our Steam project within allowance ($2000). Specifically, we spent $470 as of today setting up our instances. Because, we decided to deploy PostgreSQL databases and Jupyter Notebook on the same tier this eliminated the need to have multiple servers for parallel processing. In the beginning, our team set up a couple micro servers to parse and analyze smaller datasets which didn't factor much into the overall usage.

Jupyter notebook and PostgreSQL on AWS are user friendly when implementing our analysis models. We experienced latency issues at times but these issues were solved by selection of higher performance instances. We migrated servers from time to time when project progresses to accommodate larger datasets and faster data processing. We monitored our usage to keep the cost under control.

The three main instance types we used and its costs were:

- t2.micro - $0.0116 per hour
- m5ad.4xlarge - $0.824 per hour
- m5ad.12xlarge - $2.724 per hour

Findings and Reporting

Filtering Out Free Games

We made the decision to filter out free games in our data set late in our project. When examining the games in our data set, we found that the top 1000 games contain far too many free games. While removing the free games in our interaction matrix ultimately came down to a business level decision, we wanted to evaluate the performance between a model trained using an interaction matrix with free games vs a model trained using an interaction with no free games.

The models are trained on different data sets, making it hard to directly compare performance without variance playing a role. Because free games form a major part of many players' libraries, the top 1000 games that would be present in each data set are very different, with only 820 out of 1000 common games shared by the two. In order to compare the two models, we trained and tested both models using a random 80/20 train test split and ran this 30 times each to reduce variance. Even then, there is a possibility of variance playing a role in their mean AUC.

We found that by filtering out free games, we achieved a moderate improvement in AUC. The data set with free games kept had a mean AUC of 0.908 over 30 runs compared to 0.914 for the data set with free games removed. This convinced us that removing free games adds more meaningful information to the model's understanding of user preference. All further analysis and final model was performed by removing free games from the data set.

Predicting on Bought vs Played

In addition to filtering out free games in our model, we left open several options for the interaction matrix. There are 2,741,617 positive interactions (games owned by users) in our post-processed interaction matrix out of 65,972,000 possible positive interactions. The interaction matrix can be seen to be fairly sparse with only 4.15% of total user-item pairs being positive. Running BPR on this, the objective is clearly to build a model optimizing games that users will buy. However, many users spend money on games but never play it. A whopping 816,848 games that are owned have not even been opened with 0 minutes played all-time. An additional 297,392 games have never been played more than 30 minutes, giving a grand total of 1,114,240 games that have never been played for more than 30 minutes. This represents over 40% of all games that users own.

This presents a problem with our model as we do not want to recommend games that users will end up not playing. Sometimes these games are bought in a bundle or on sale and shelved for future play but are simply forgotten about. In either case, we want to build a model or an ensemble of models that will try to focus on games that users will actually play. This is difficult as the two are separate objectives, one focusing on pure revenue and the other focusing on user satisfaction.

We built four separate models, all using LightFM and the same hyperparameters, and tested how well they performed against one another's test sets. Each model differs purely on the interaction matrix and which user-item pairs are positive vs zero. The interaction matrices of models 2-4 are subsets of model 1. The models are:

- Model 1 - No filters applied. Same interaction matrix from post-processing. 2,741,617 positive interactions
- Model 2 - Only user-game pairs where the game has been played for more than zero minutes. 1,924,769 positive interactions.
- Model 3 - Only user-game pairs where the game has been played for more than thirty minutes. 1,627,377 positive interactions.
- Model 4 - Only user-game pairs where the game has been played for more than the 10th percentile played time by game. Similar to models 2 and 3 except the playtime filter value is different for each game. Some games, in particular multiplayer games, are played much more than others so a variable filter is used to account for that. 1,888,716 positive interactions.

		Test Set			
		No Filter	Only games played > 0 minutes	Only games played > 30 minutes	Only games played > 10th percentile/game
Model	No Filter	0.9151	0.9076	0.9106	0.9061
	Only games played > 0 minutes	0.881	0.9102	0.9179	0.9084
	Only games played > 30 minutes	0.8623	0.9015	0.9168	0.8995
	Only games played > 10th percentile/game	0.8812	0.9105	0.9185	0.9089

Figure 15: Table showing each model's AUC vs test data set.
We trained models using various interaction matrices, labeled model 1-4 top to bottom and then tested how well they did against another model's interaction matrix. Model 1 focuses purely on game ownership while models 2-4 focus on games that players buy and actually play.

Expectedly, model 1 clearly performs better for its own test data set. Models 2-4 perform significantly worse on model 1's test data set compared to model 1. Model 2-4 similar perform better on their own data set than model 1 although model 1's performance is only slightly worse than models 2-4 on their own data set. Interestingly, over multiple iterations, model 4 performs slightly better than model 2 and model 3 on their own data sets, although the difference is not very significant. The performance of model 1 on model 2-4's data vs the performance of models 2-4 on model 1's data set indicate model 1 is able to train a model that works well on both on maximizing revenue and recommending games that users will actually play.

Despite model 2-4 only containing 59-70% of the total positive examples from model 1, they were still able to achieve strong performance on model 1's data set and even stronger performance on their own data set. There was an expectation that reducing the training data by such a significant amount would severely degrade the model's performance but our experimentation showed that was not the case. While the additional data from model 1 clearly pushes it to strong performance across the board, the drop in performance from removing up to 40% of the data was largely minimal, showing the robustness of our model when tuned.

However, the worse performance of models 2-4 on model 1's data set without the vice versa being true indicates a weakness in the training data of models 2-4. An experiment was conducted randomly splitting model 1's data by 40% so the size is comparable to model 3's, the smallest data set of the four, and compare its performance across the four interaction matrices.

		Test Set			
		No Filter	**Only games played > 0 minutes**	**Only games played > 30 minutes**	**Only games played > 10th percentile/game**
Model	**Random Filter to 71% of Original Size. Same as Model 2.**	0.903	0.8969	0.9005	0.8952
	Random Filter to 60% of Original Size. Same as Model 3.	0.8946	0.8888	0.8925	0.8871
	Random Filter to 69% of Original Size. Same as Model 4.	0.9024	0.8963	0.8998	0.8946

Figure 16: Interaction Matrix randomly filtered to same sizes as models 2, 3, and 4.

Even with training data that is randomly filtered, performance is still fairly good across the board. While the random filtering generally performs worse on model 2-4's test sets, they still perform better on model 1's test set compared to model 2-4's, further showing the robustness of our model despite data removal. Our choice of model performs well despite choice of interactions. Still, because model 1 has great performance across the board and simultaneously recommends games that players are likely to buy and play without needing play data described, we chose model 1 as our final model for output.

Cold Start Problem

The cold start problem is a well known and common issue seen in recommender systems, in both content-based and collaborative filtering. Because recommender systems rely on information about users and items to begin building a model, new users and items will usually not have enough information or interactions for the model to predict well on. This problem can be seen in nearly any recommender system where when a new user is registered, the recommended items will usually be very generic and not catered towards that users' tastes yet. The problem also presents itself with new items as the system is unable to recommend or place it on a ranked list without users having previously interacted with it. A pure collaborative filtering system usually fails at attacking this problem properly.

For the purposes of this project, cold start users are loosely defined as any user owning a small number of games. No particular number is used as a strict definition of cold start in our model but our analysis will focus users who own at least one game since AUC cannot be measured for users who own zero games. In other words, no new or fully cold users will be included. In some figures below, users with five or fewer games will be considered cold for illustration purposes.

Cold start items will not be a part of our analysis for several reasons. Firstly, our data is static and new games will not be added to the data set. As mentioned in the data transformation subsection, we are only recommending games within the top 1,000 most popular games. While the 1000th most popular game, Beyond Divinity, has significantly fewer interactions than the top 5-10 most popular games, it is still owned by 620 users, moving it out of our loose definition of cold. It should still be noted that Beyond Divinity differs from Counter-Strike: Global Offense, the most popular game, by a factor of 70. By comparison, the hottest user in our data set owns 935 out of the top 1,000 most popular games while the coldest users only own 1. Finally, since recommendations are mainly made on a user-to-item basis, with user_id as the input variable, focusing on optimizing performance for cold start users is a much more important task and the one we chose to focus our attention on.

Surprisingly, we found that our model performs extremely well for cold start users. The hybrid collaborative+content-based model is supposed to alleviate some issues associated with the cold start problem but we found a pure cold start model still performs much better for cold users than hot users. The line chart below plots the AUC as a function of user coldness. A pure collaborative filtering is model is used in the chart, with no additional features included beyond user-item interactions. The same test set is used as in the final model with test set evaluation done by querying out users by the number of games they have in the training set and testing on them only. The x-axis represents users with n number of games in their training set. For example, if a user owns 6 games and 5 of those games are used for training, he/she will be plotted as 5 while the remainder will be used in the test set.

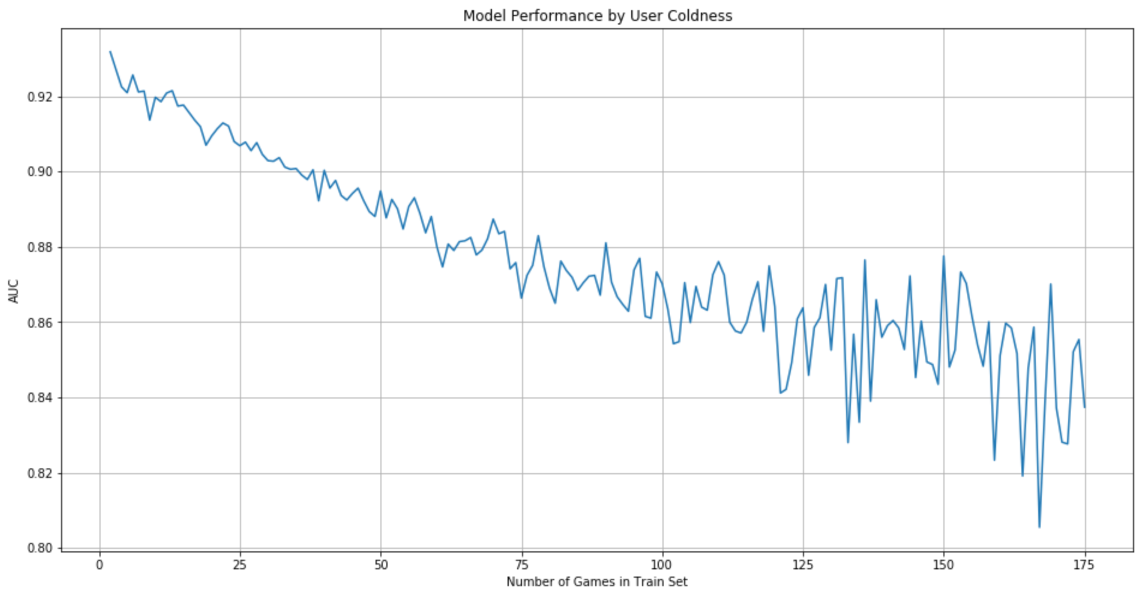
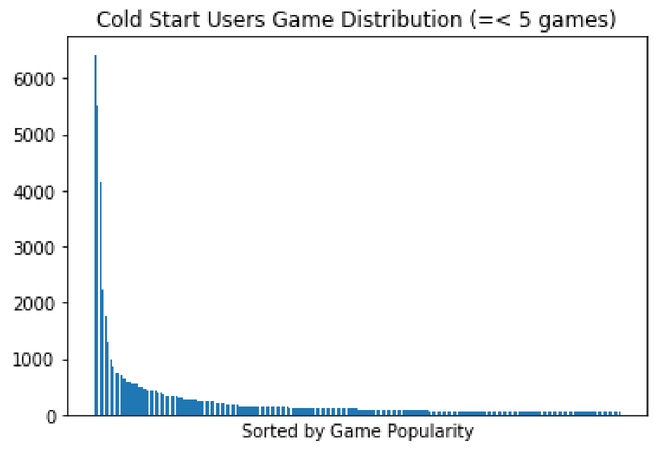


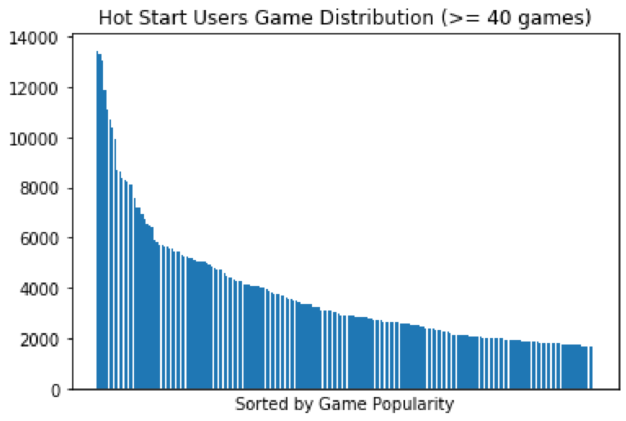
Figure 17: Cold Start Users vs AUC on test set.

Users were queried by the number of games they owned in train set. Results get noisy as user hotness increases. Number of users to test dwindles as 125+ games owned is rare.

The figure clearly shows a downward trend in AUC as users own more games. This result is extremely counterintuitive and goes against our initial hypothesis that recommending games for hot users should be significantly easier than recommending games for cold users. The most surprising finding the figure shows is that the model recommends best for users with only 2-3 games in their data set. Despite variance possibly playing a role, the difference between users with 2-5 games and users with 100+ games is a 6-7% in AUC, very significant for modeling purposes.

This finding improves our understanding of the data set and perhaps reveals a major part of the nature of the video game industry. Video games are much less affected by personal tastes like fashion or automobiles and users tend to initially buy some extremely popular games such as Counter-Strike: Global Offensive, Terraria, Half Life 2, and Portal 2 before branching off into more specialized genres or tastes. The following figures illustrate the difference when comparing cold start users vs hot start users. Cold start users' owned games tend to concentrate on some of the most popular, forming a strongly exponential shape. Hot start users tend to have a much more diverse set of games with a much flatter curve. Recommending games becomes a more difficult task as most hot users already own the most popular games and recommender models will have to discover additional features to properly model them.





Figures 18: Cold vs Hot Start User Game Distribution.
The top 200 games owned are plotted for both cold and hot start users. The distribution for hot start users is much flatter, indicating games are distributed much more evenly. Cold users tend to own the most popular games which test well in our model.

Principal Component Analysis

Principal component analysis was used to visualize the item embeddings in our model and how they can potentially describe or not describe genres, specs, and other tags that define a game in its metadata. One of the most powerful aspects of collaborative filtering is the ability to automatically discover preferences for genres despite not using it as an input. We first trained our model using the optimal hyperparameters with free games removed and no additional filters, using 20 components for the embedding. Note that the number of components we chose to run our model is different from what we choose to train our final model. We found that training our model with a lower number of components and then running PCA results in higher percentage of variance explained for the first two components which helps with the following visualizations. The figure below shows the test AUC of the model by number of components trained. The AUC begins dropping sharply once the model falls below 10 components.

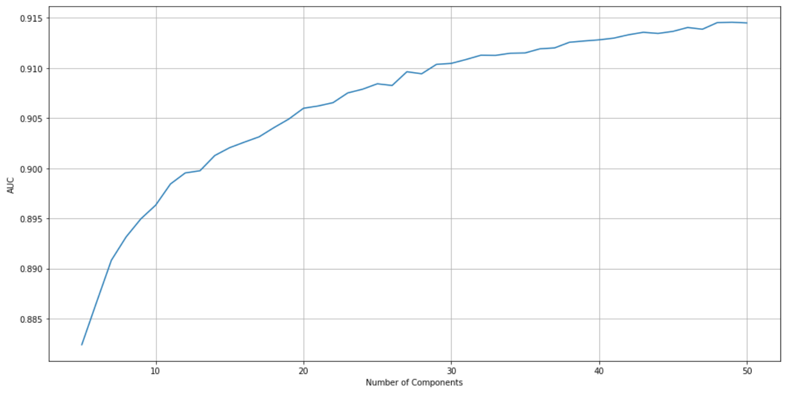


Figure 19: AUC by number of components when training model

We wanted to use enough components that our model is still well represented but still retains as much information as possible after running AUC. Embedding to 20-50 dimensions and then taking the first 2 principal components is radically different from training the model to embed on 2 dimensions. Whereas BPR in 50 dimensions followed by PCA gives a partial explanation of a complete model, BPR in 2 dimensions attempted to give a complete model using only 2 dimensions. For this reason, we did not train our model to only embed on 2 dimensions and then visualize and opted to use PCA instead. In the figure below, the top 2 principal components only explain 22% of the total variance but we were still able to derive meaningful visualizations and analysis from plotting them.

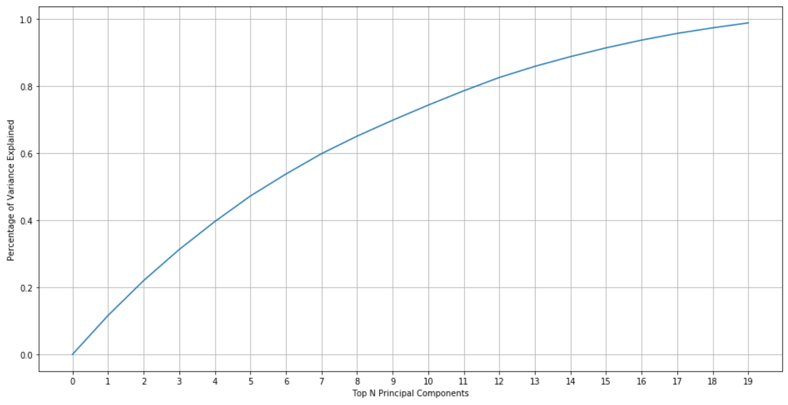


Figure 20: Percentage of Variance Explained by Taking Top N Principal Components

We chose a handful of genres and specs and plotted their coordinates with respect to their top 2 principal components. Some genres showed a clear trend in the space they occupy in the reprojected space while other genres showed no such trend. Genres are defined by the game publishers and each game can have multiple genres. Similarly, when talking about specs, single player games can also be multiplayer. These genre labels are explored in the visualizations below. Because of genres are not mutually disjoint, strictly defining a game as action, role playing, puzzle, etc. is difficult and broader genres are difficult to generalize to a single region.

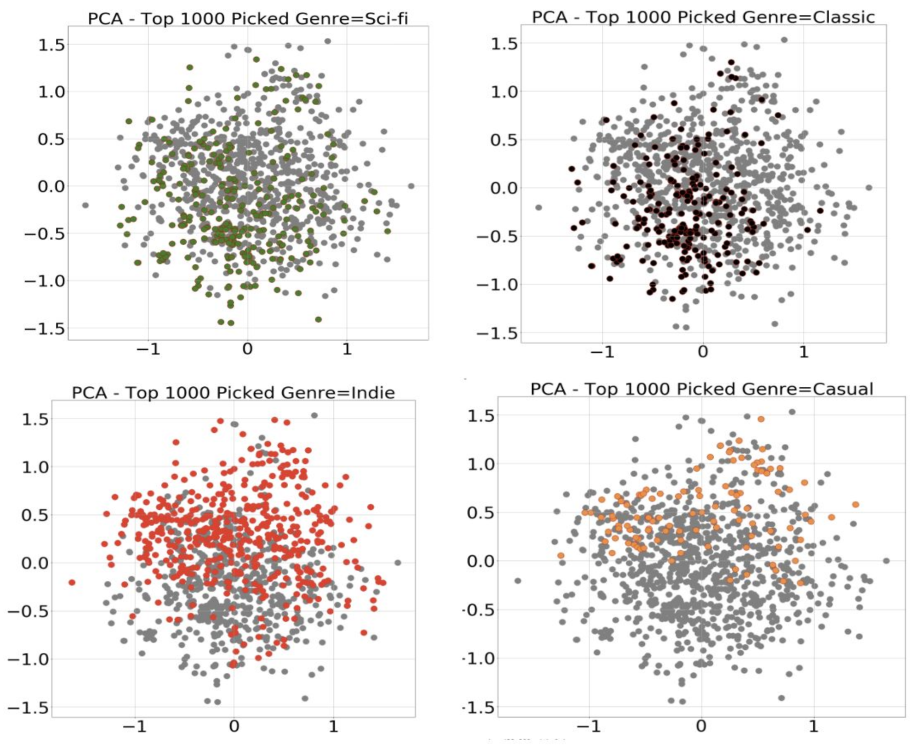


Figure 21: Principal Component Analysis (PCA) on Sci-Fi, Classic, Indie and Casual genres for top 1000 selected games. These cases showed distinct directional pattern whereby games were distributed mostly in the same cluster.

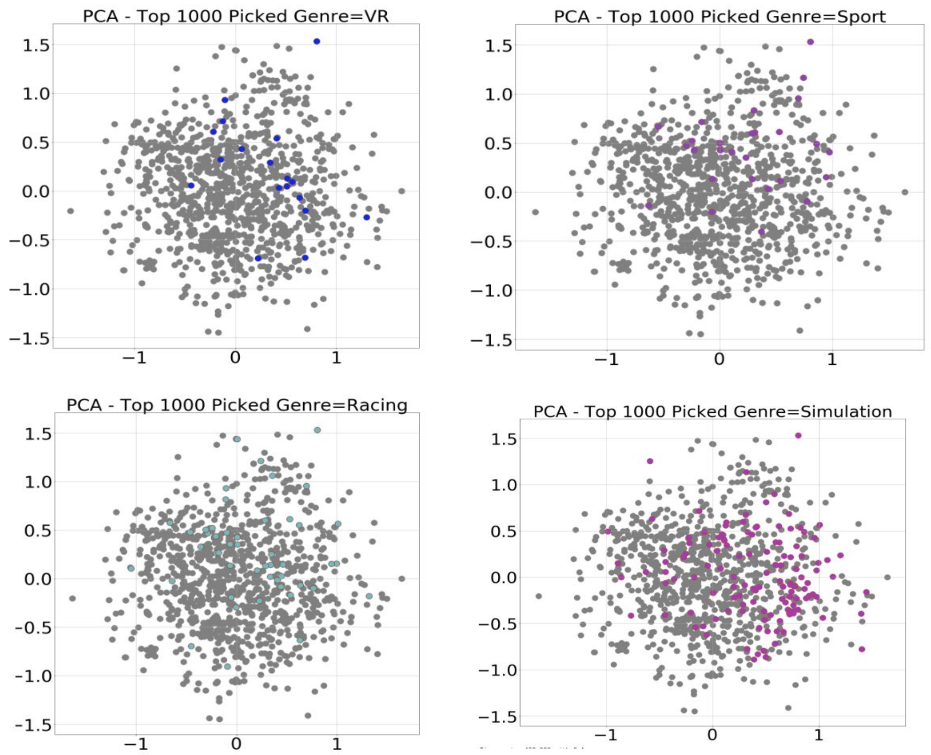


Figure 22: Principal Component Analysis (PCA) on VR, Sport, Racing and Simulation genres for top 1000 selected games. Similar to above figure except these genres are even more specialized with fewer games per genre.

The genres in the first figure above are good examples of genres that are specialized enough to see a trend after PCA transformation. In particular, sci-fi and classic games occupy a general region to the left side of the space, casual games occupy a general region on the upper half of space, and indie games occupy a general region on the upper half of the space. Games have 2.5 genres on average so it is very difficult to say that genres, as defined by the publishers, have natural clusterings that can be found using traditional distance based metrics. However, from PCA, we still find that genres, particularly ones that a little more specialization like classic and casual do show a general pattern that emphasizes the genre modeling function of BPR.

In the second figure below, we see the same pattern with genres that are even more specialized. Some of these genres such as VR, Sport, and Racing find their own small clusters in the space. Much fewer games are labeled under these genres and their tagging in this genre indicates a more specific type of game. In these cases, PCA works even better as a way to visualize genres. Note that there are still outliers. With only 22% of variance explained in the top 2 coordinates and human labeling involved, genre modeling using PCA is still not perfect.

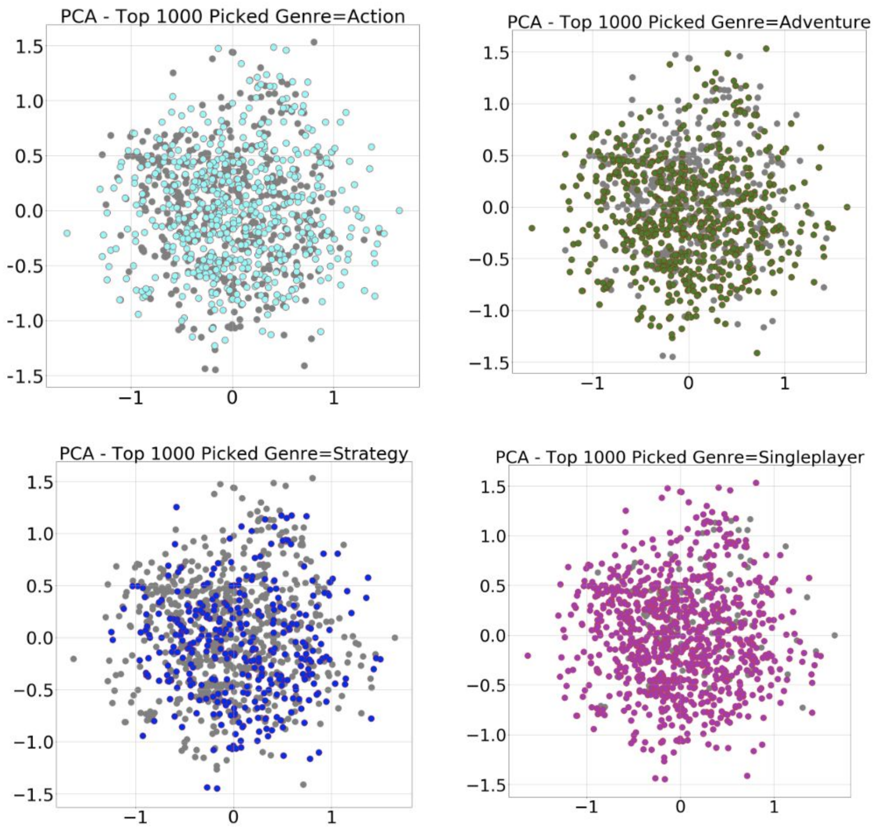


Figure 23: Principal Component Analysis (PCA) on Action, Adventure, Strategy and Single-Player genres for top 1000 selected games. These cases showed congested pattern due to majority of games were categorized as these genres which occupy majority of datapoints.

These genres show no discernible trend when plotted. The genres picked here: single-player, action, strategy, and adventure, are extremely broad and encompasses a huge variety of games. Out of these genres, only strategy shows a slight preference toward the left region of the space. This sheds a little insight to the way publishers choose to tag their games. Often times, generic genres such as action and adventure simply do not give enough information to define a game. In the steam_games data set, 11,325 out of 32,135 games contain the genre “action”, 8,243 games contain the genre “adventure”, 27,577 games contain the spec “single-player”, and 6,597 contain the genre “strategy”.

Hybrid Model

Our main focus is to build a recommender model based on collaborative filtering, and we were able to gain pretty impressive performance by carefully selecting our datasets and tuning hyperparameters on each model that we created. As we mentioned from Analysis and Implementation section earlier, we heavily utilized LightFM library to build our recommender model with collaborative filtering method, and one additional powerful features that LightFM also provides is a capability to build a hybrid model easily which is a model built by the combination of collaborative filtering and content based method. By taking advantage of this feature in LightFM, we were able to extend our model and build a model to experiment more data and compare their performance. We generated several combinations of user and item features which are described in Data Preparation and Feature Selection earlier and built several models to compare its performance.

Feature matrices creation

Since our dataset is public, it does not include any private user data besides playtime for games that each user own. Hence, we decided to focus on game item features only to generate content based features to build a hybrid model.

- Genres: This feature is added as a default item feature. Binary matrix is created to indicate the game is categorized by which genres and matrix is normalized to calculate weights.
- Price and Released Year: These features are added with genres to add more game characteristics.
- Median total playhour, Median playhour in recent two weeks: These features are also added with other features to experiment to see these features add any performance gain.

LightFM provides an interface to create a feature matrix by feeding a list of items and feature parameters in pre-defined format; hence we took advantage of this functionality and built a feature matrix using LightFM library interface initially. Since our features don't belong to one category and some features are not directly related to each other, we also tried manually create a feature matrix by manipulating and normalizing parameters for each feature separately and ran the model to test the performance.

After creating n by m feature matrix which n is a number of games and m is a number of features, the feature matrix is normalized for each game to re-generate weights for features of each game.

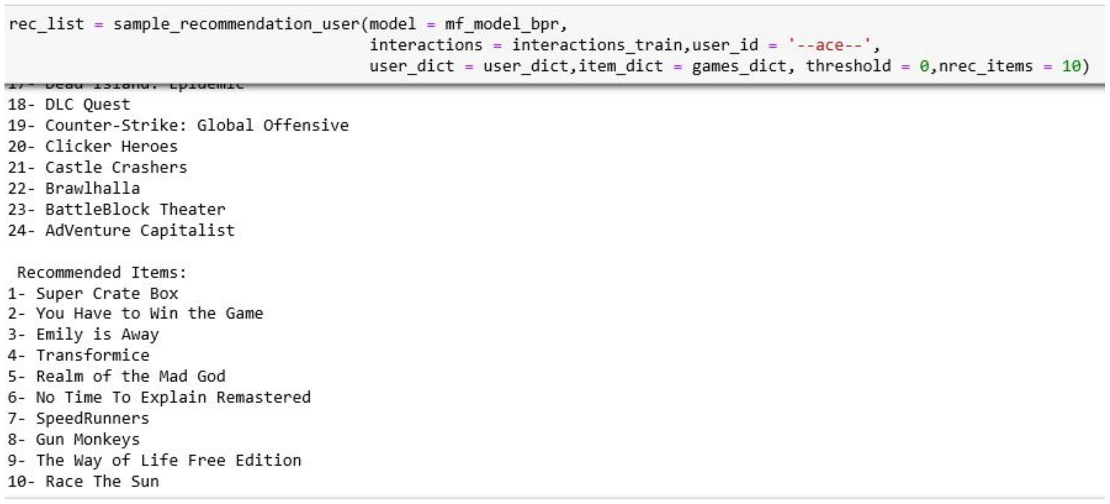
Performance

Initially, we had native assumption that more related user-item features will boost the model performance when it is compared to a pure collaborative filtering model because it adds more characteristics to the training and test data to complete the model. By trying several combinations of feature selections and matrix creation methods that we described above, we found that the performance varies in each combination and methods and its gap is large. The model trained with the feature matrix built by LightFM library produced lower AUC scores to compare with the results from a collaborative filtering model with the same configuration parameters such as loss function, learning rate, and L2 penalties; hence, we focused on building the feature matrix manually by normalizing each feature values separately and normalizing again by each game to calculate weights of each feature by the game. By adding and processing all selected features including game genres, price, released year, and playhours to build the feature matrix,, we were able to achieve similar AUC scores with the same model configurations.

Since we tried to build a hybrid model at a later stage of the project, we have not had a chance to search hyperparameters to finalize the best working hybrid model. Since this hybrid model with user or item features will not be the same model with the collaborative filtering model even if we run them with the same training and test interactions, it will be important to find hyperparameters for hybrid model also to finalize the best working model. This will be our future research items to improve our model performance and create more findings to explain the model behavior.

Model Outputs

For our user-item recommendations, we output the top 20 ranked games for each user. Each ranking is generated using both WARP and BPR model. The ranking model produces a score value for each game via collaborative filtering. The scores for all recommended games are then sorted in descending order and the top 20 high scoring games are recommended to the user as shown in the Figure below. (Note that Figure shows top 10 in this case)



```
rec_list = sample_recommendation_user(model = mf_model_bpr,
                                      interactions = interactions_train,user_id = '--ace--',
                                      user_dict = user_dict,item_dict = games_dict, threshold = 0,nrec_items = 10)
```

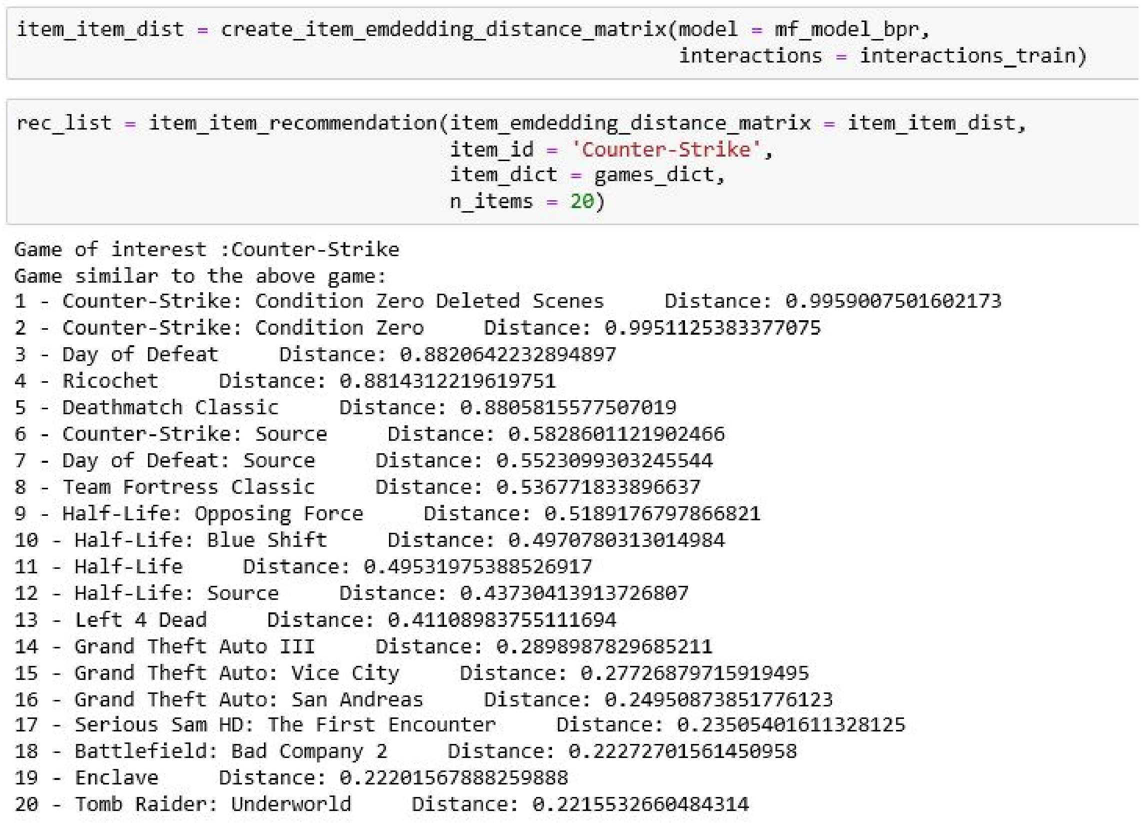
```
17- Dead Island: Epidemic
18- DLC Quest
19- Counter-Strike: Global Offensive
20- Clicker Heroes
21- Castle Crashers
22- Brawlhalla
23- BattleBlock Theater
24- AdVenture Capitalist

 Recommended Items:
1- Super Crate Box
2- You Have to Win the Game
3- Emily is Away
4- Transformice
5- Realm of the Mad God
6- No Time To Explain Remastered
7- SpeedRunners
8- Gun Monkeys
9- The Way of Life Free Edition
10- Race The Sun
```

Figure 24: Shows the User-Item Recommendations Using the BPR Model

For our item-item recommendations, we output the top 20 ranked games for each item similar to the item selected using again both WARP and BPR model. For each item, a distance is computed via cosine similarity and then sorted in descending order as shown in Figure. The top 20 games with the highest distance scores are recommended.

Both user-item and item-item recommendations were output as JSON files which were then converted to a flat table for ingestion in Tableau. The transformation of the output files as JSON files to flat files improved performance for the dashboard and overall made the post-output data munging much easier.



```
item_item_dist = create_item_emdedding_distance_matrix(model = mf_model_bpr,
                                                       interactions = interactions_train)
```

```
rec_list = item_item_recommendation(item_emdedding_distance_matrix = item_item_dist,
                                    item_id = 'Counter-Strike',
                                    item_dict = games_dict,
                                    n_items = 20)
```

```
Game of interest :Counter-Strike
Game similar to the above game:
1 - Counter-Strike: Condition Zero Deleted Scenes     Distance: 0.9959007501602173
2 - Counter-Strike: Condition Zero     Distance: 0.9951125383377075
3 - Day of Defeat     Distance: 0.8820642232894897
4 - Ricochet     Distance: 0.8814312219619751
5 - Deathmatch Classic     Distance: 0.8805815577507019
6 - Counter-Strike: Source     Distance: 0.5828601121902466
7 - Day of Defeat: Source     Distance: 0.5523099303245544
8 - Team Fortress Classic     Distance: 0.536771833896637
9 - Half-Life: Opposing Force     Distance: 0.5189176797866821
10 - Half-Life: Blue Shift     Distance: 0.4970780313014984
11 - Half-Life     Distance: 0.49531975388526917
12 - Half-Life: Source     Distance: 0.43730413913726807
13 - Left 4 Dead     Distance: 0.41108983755111694
14 - Grand Theft Auto III     Distance: 0.2898987829685211
15 - Grand Theft Auto: Vice City     Distance: 0.27726879715919495
16 - Grand Theft Auto: San Andreas     Distance: 0.24950873851776123
17 - Serious Sam HD: The First Encounter     Distance: 0.23505401611328125
18 - Battlefield: Bad Company 2     Distance: 0.22272701561450958
19 - Enclave     Distance: 0.22201567888259888
20 - Tomb Raider: Underworld     Distance: 0.2215532660484314
```

Figure 25: Shows Item-Item Recommendations of Top 20 Using the BPR Model

Visualization/Dashboard

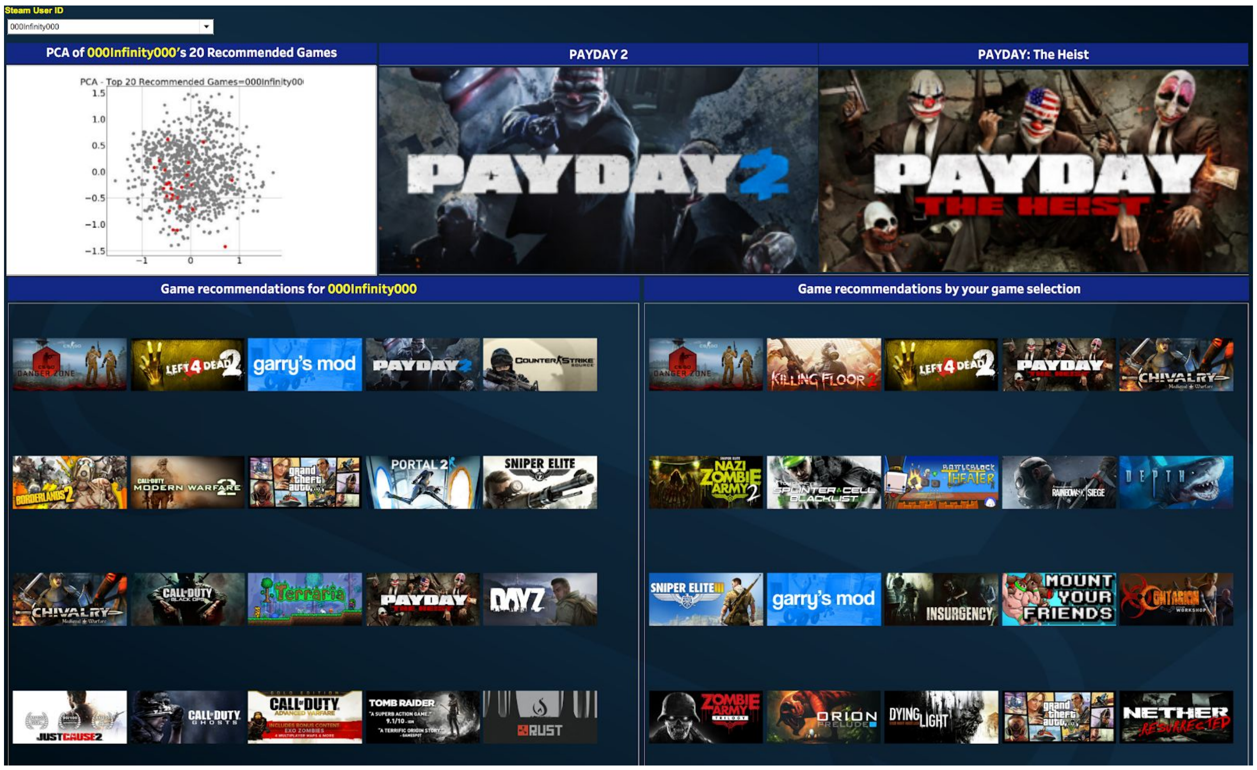
Based on the output from our best recommender model, we built a dashboard using Tableau to represent our game recommendations to users. In our dashboard, we display the top 20 game recommendations for each user (user-item) and also the top 20 game recommendations that are similar to other games (item-item). The user-item recommendations are displayed via drop down filter for each user. The item-item recommendations are displayed when selecting a game from the user-item recommendation (See Figure below). The recommendations for the games were precomputed via Python and then migrated over to Tableau to display on the dashboard. Also, we display the genre preference of each user using a bubble plot.

The image titles of the recommended games were scraped from the Steam website through a simple Python script and imported into Tableau to show the game recommendation. Some preprocessing was involved to link each image to the game title. The image titles gives the user a concrete picture of what type of games to recommend, adding to the effectiveness of the visual.

In the dashboard, each user-item recommendation is customized per user via dropdown filter. The dashboard shows the top 20 game titles on the bottom left. The recommendations were precomputed using the WARP model for games played for more than 10th percentile play time. When title can then be displayed on the top center when hovering the mouse over it.

The item-item game titles shown on the bottom right and can be represented whenever one selects a user-item recommendation. It shows all of the games that would be similar to the selected recommendation using cosine similarity model. A larger image of the title is also displayed on the top right. The two panels thoroughly highlight which games to recommend to the user, which contribute to the expressiveness of the dashboard.

Finally, we generated a PCA scatter plot that portrays the top 1000 popular games in grey color and overlay the top 20 recommended games for each user in red color and displayed it on the top left of the dashboard. From PCA figure, we can see how the top-20 recommended games are positioned and clustered out of all top 1000 popular games.



Steam User ID
000Infinity000
PCA of 000Infinity000's 20 Recommended Games
PCA - Top 20 Recommended Games=000Infinity00
PAYDAY 2
PAYDAY: The Heist
Game recommendations for 000Infinity000
Game recommendations by your game selection

Figure 26: Tableau Dashboard of Steam Game Recommender Systems: the top-20 user-item recommendations on the bottom left, the top-20 item-item recommendations on the bottom right, the selected title from the top-20 recommended games are shown on top middle, the selected title from the top-20 item-item recommendations on the top right, PCA scatter plot of the top-20 recommended game and the top-1000 popular games on the top left.

Conclusion

While the main focus of our project was on building a powerful and robust recommender system, we also wanted to put a heavy emphasis on data analysis and understanding every facet of the data and model. Bayesian Personalized Ranking is a transparent algorithm that allowed us to use the model and its outputs and take our analysis in many different directions as seen in the findings section. Our findings were especially focused on the questions posed in the introduction. We felt we succeeded in our goals for the project and our analysis was in-depth, insightful and can serve as a guide for stakeholders and analysts alike.

Through exploration of several potential models, we discovered the simplest ones worked best for our data set. While we tried various filtration and data manipulation techniques, the most simple collaborative model gave us the best results. The cold start problem remains one of the most interesting and pressing problems facing our data set but we felt confident in our ability to address it with the model we have in place. The hybrid content-collaborative model we built did not improve performance as we had hoped but we want to keep researching hybrid approaches and potentially discovering from informative features to improve the performance of our models.

We built our dashboard in Tableau showing the top 20 games to recommend to the user based on the WARP model. The dashboard provides customized user-item recommendations for each user. For each game recommended, an item-item recommendation for games that are similar to the recommended title. The selected games are also shown as a larger image on the top middle and right. Also, a bubble plot representing genre of games that the user owns is shown on the top left. All of this information provide a comprehensive assessment on which type of games would best suit the user and thus add to the expressiveness and effectiveness of the dashboard.

References

[1] Robert M Bell, Yehuda Koren, Chris Volinsky. The BellKor 2008 Solution to the Netflix Prize. AT&T Labs Research. 2008.

[2] R. Bell and Y. Koren, and C. Volinsky. “The BellKor solution to the Netflix Prize”. http://www.netflixprize.com/assets/ProgressPrize2007_KorBell.pdf. 2007.

[3] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner and Lars Schmidt-Thieme. BPR: Bayesian Personalized Ranking from Implicit Feedback. UAI 2009

[4] Maciej Kula. Metadata Embeddings for User and Item Cold-start Recommendations. RecSys 2015

[5] Tong Zhao, Julian McAuley, Irwin King. Improving Latent Factor Models via Personalized Feature Projection for One Class Recommendation. CIKM, 2015

[6] Ruining He, Julian McAuley. VBPR: Visual bayesian personalized ranking from implicit feedback. AAAI, 2016

[7] Wang-Cheng Kang, Julian McAuley. Self-attentive sequential recommendation. ICDM, 2018

[8] Mengting Wan, Julian McAuley. Item recommendation on monotonic behavior chains. RecSys, 2018

[9] Apurva Pathak, Kshitiz Gupta, Julian McAuley. Generating and personalizing bundle recommendations on Steam. SIGIR, 2017

[10] Nicolas Usunier, David Buffoni, and Patrick Gallinari. Ranking with ordered weighted pairwise classification. In Proceedings of the 26th Annual International Conference on Machine Learning, ICML ’09, pages 1057–1064, New York, NY, USA, 2009. ACM.

[11] James Bergstra, Yoshua Bengio. Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research 2012

Appendices

A. DSE MAS Knowledge Applied to the Project
    a. Python for Data Analysis
        i. Project was completely coded in Python, using all the knowledge learned in this class.
        ii. Extensively used Pandas, Numpy, Scipy, and Scikit-Learn in project.
        iii. Explored web crawling and data scraping to expand scope of project.
        iv. Most plots in report generated by Matplotlib.
    b. Probability and Statistics
        i. Applied PCA to item embeddings for further analysis. PCA was a major part of our analysis and findings
        ii. Informative projections played a large role in understanding recommender systems. Emphasis on linear algebra was crucial to understanding theoretical background of matrix factorization/collaborative filtering.
    c. Database Management Systems
        i. Set up PostgreSQL server on EC2 instance.
        ii. Used PostgreSQL database to store hyperparameter search results.
    d. Machine Learning

        i. Understanding metrics, in particular AUC, played a huge role in evaluating and expressing performance of our models.
        ii. Used cross validation to optimize performance of our metrics
        iii. Understanding loss functions, gradient descent/optimization, and logistic regression essential to understanding Bayesian Personalized Ranking.
    e. Data Analysis Using Hadoop and Spark
        i. Early implementation done in Tensorflow. Tensorflow was essential to early work on our project.
        ii. Understanding of multiprocessing and its applications to this project.
    f. Data Integration & ETL
        i. Knowledge of building data flow diagram.
        ii. Data preprocessing was an important thing taught and was a key part of our data pipeline.
        iii. Helped understand data integration and how parts of a data science project should fit together in a data pipeline.
    g. Beyond Relational Databases
        i. Raw data came in the form of JSON.
        ii. Early on explored ideas of storing data in NoSQL databases.
    h. Data Visualization
        i. Principles of data visualization factored heavily into design choices for our visualization.
        ii. Used Tableau to build dashboard. Final project prepared us well for this.

B. Project Resources
    a. Project Library Archive: https://ezid.cdlib.org/id/doi:10.6075/J0FQ9TZM
    b. Data Sources: https://cseweb.ucsd.edu/~jmcauley/datasets.html#steam_data
    c. Project source code repository: https://github.com/bcc008/UCSD_Steam_Capstone