Machine Learning Engineer Nanodegree

Capstone Project Report: Building a Game Recommender System

Robert Lizatovic

September 30, 2019

I. Definition

Project Overview

Recommender systems attempt to predict a “rating” or a “score” users would give to items, and based on these predictions recommend new items to new or existing users[1]. Recommender systems are quite ubiquitous today and many enterprises are utilizing them regularly to offer more personalized and targeted content to their customers (i.e. Amazon, Netflix, Spotify etc.). Research in recommender systems is very active and broad with different types of general approaches employed and many different types of algorithms developed and used[2]. The classical approaches to developing recommender systems include collaborative filtering (CF) and content-based filtering (CBF) methods, as well as various hybrid techniques combining the two.

This work focuses on developing and testing several algorithms for building a game recommender system. Similar to a movie or a book recommender, the primary function of this system would be to predict whether a particular user would enjoy playing a particular computer game and then suggest to users games that they might like. To develop such algorithms, I used previous user-game ratings and additional game characteristics obtained from a dataset collected from Steam (a platform for distributing, playing, and discussing computer games)[3]–[5]. The recommender system would function by first predicting the ratings (in the case of Steam game reviews, the ratings are a binary: “recommend” - 1 and “not recommend” - 0 labels) that players would give to games they haven’t interacted within in the past, and then constructing a ranked list of games to recommend based on the predicted ratings.

Problem Statement

A fully fledged recommender system in production is a complex machinery that needs to take many things into account in addition to the accurate modelling of user-item preferences (such as for example not recommending items the user has already interacted with in the past, changing recommendations from time to time etc). The primary focus of this work is on developing the first part of a recommender system: the algorithm that predicts user-item ratings.

The problem this work tries to solve is thus the following: given previous user-game ratings and general game characteristics, predict the rating of any given user-item pair (not observed in the training set). To accomplish this, I first carefully separated the full ratings dataset into a training and a test set in such a way that all users and all games are represented in both sets (more on this below). This is necessary for all of the algorithms employed as they need to build a representation of each user and game in order to model the interaction properly. The models are built on the training set and subsequently evaluated on the test set for performance comparisons.

Metrics

For evaluating model performance, here I use the **mean squared error** (**MSE**) between the predicted and actual user-game ratings:

$$MSE = \frac{1}{N}\sum_{ij}\left(r_{ij} - \widehat{r}_{ij}\right)^2$$

Where $\widehat{r}_{ij}$ and r_{ij} are the predicted and actual rating of user *i* for game *j* and *N* is the total number of ratings available in the dataset.

As the ratings are given as a binary “recommend”/”not recommend”, they are first encoded as 0/1 values. Although the targets are themselves binary values, we want our recommender system to be able to rank the predictions so that top-k games may be suggested to each user. In order to accomplish this, the predictions are outputted as continuous values and the general method of regression is used to compute them.

II. Analysis

Data Exploration

For this project I have used the player-game reviews dataset collected from the Steam gaming platform[3]–[5]. The dataset is comprised of two text files:

- A player-game reviews file that contains the ratings (binary “recommend”/”not recommend”), and reviews of players for various games distributed and sold on Steam, in addition to other metadata.
- A game information file that contains information about the games themselves such as title, genres, specifications, pricing, metascore, publisher etc.

Besides the explicit ratings given by the users, the dataset contains a plethora of additional information that can provide further context for modelling player-game preferences. These are for example the general game information data (genres, specifications, metascores etc.) which can be used to learn better representations of how players interact with games[6].

Each line of the player-game reviews file is a JSON-like string containing the user ID and a reviews array (which contains all the reviews given by this user as nested JSON objects. A sample (for a single user ID) is provided below:

{'user_id': '76561197970982479', 'user_url': 'http://steamcommunity.com/profiles/76561197970982479', 'reviews': [{'funny': '', 'posted': 'Posted November 5, 2011.', 'last_edited': '', 'item_id': '1250', 'helpful': 'No ratings yet', 'recommend': True, 'review': 'Simple yet with great replayability. In my opinion does "zombie" hordes and team work better than left 4 dead plus has a global leveling system. Alot of down to earth "zombie" splattering fun for the whole family. Amazed this sort of FPS is so rare.'}, {'funny': '', 'posted': 'Posted July 15, 2011.', 'last_edited': '', 'item_id': '22200', 'helpful': 'No ratings yet', 'recommend': True, 'review': "It's unique and worth a playthrough."}, {'funny': '', 'posted': 'Posted April 21, 2011.', 'last_edited': '', 'item_id': '43110', 'helpful': 'No ratings yet', 'recommend': True, 'review': 'Great atmosphere. The gunplay can be a bit chunky at times but at the end of the day this game is definitely worth it and I hope they do a sequel...so buy the game so I get a sequel!'}]}

Each line of the game information file is a JSON-like string containing key:value pairs. An example (for a single game ID) is provided below:

{u'publisher': u'Kotoshiro', u'genres': [u'Action', u'Casual', u'Indie', u'Simulation', u'Strategy'], u'app_name': u'Lost Summoner Kitty', u'title': u'Lost Summoner Kitty', u'url': u'http://store.steampowered.com/app/761140/Lost_Summoner_Kitty/', u'release_date': u'2018-01-04', u'tags': [u'Strategy', u'Action', u'Indie', u'Casual', u'Simulation'], u'discount_price': 4.49, u'reviews_url': u'http://steamcommunity.com/app/761140/reviews/?browsefilter=mostrecent&p=1', u'specs': [u'Single-player'], u'price': 4.99, u'early_access': False, u'id': u'761140', u'developer': u'Kotoshiro'}

In both cases, the formats are not truly JSON, but strings of python code that can be executed and initialized via the *eval()* python built-in function.

The initial summary of the available dataset is presented in **table 1.** Here we can see that there are many more users than there are games and that the rating density (defined as the fraction of available user-game ratings over all possible user-game pairs) is very low (which is typical of recommender system data).

Table 1: Initial summary of the user-game ratings dataset.

# of users	# of games	# of available ratings	Ratings density (%)
25458	3682	59305	0.0633

Exploratory data analysis (EDA) revealed that most of the ratings are positive, as can be seen in **figure 1.**

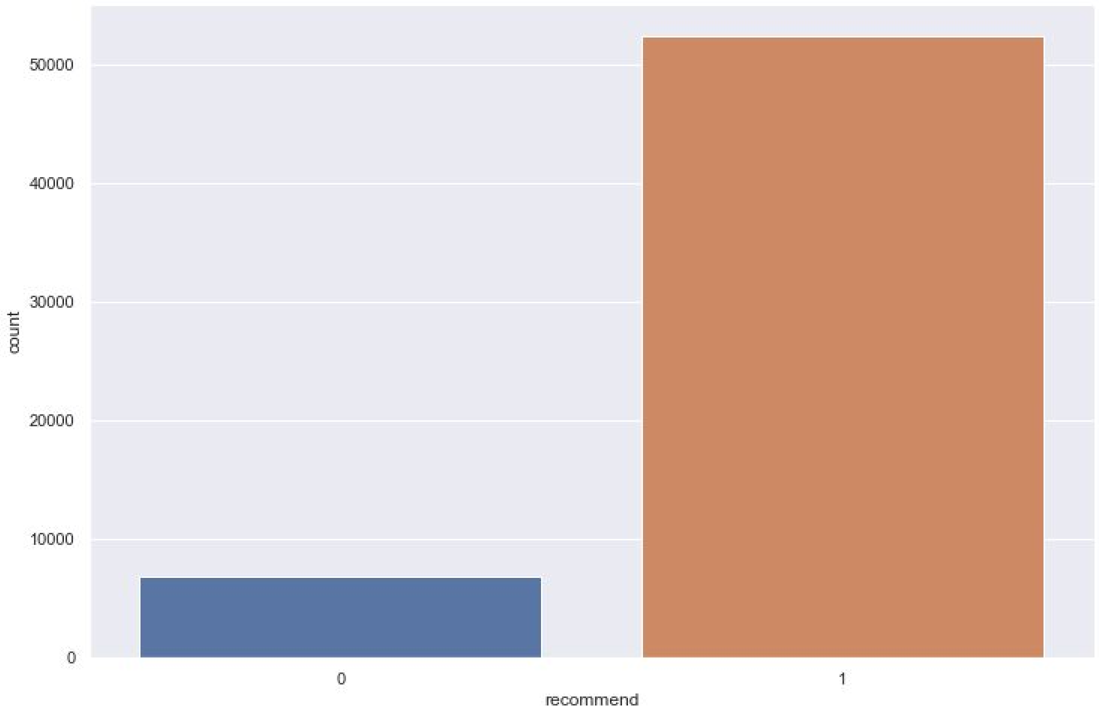

***Figure 1: The distribution of positive and negative ratings in the full ratings dataset.** 0 - negative ratings (11.52 %), 1 - positive ratings (88.48 %).*

Furthermore, as is characteristic of this type of data, the distribution of rating counts over the available items is extremely skewed, with a small fraction of games receiving a lot of ratings, and the majority of games receiving only a few or no ratings at all (**figure 2: top**). The situation is similar for users, but not as extreme (**figure 2: bottom**).

The additional game data mentioned above requires some preprocessing before it can be used in the modelling stage. Namely, with the exception of a few numerical features (such as the price and metascore), most of the available game features are categorical and multi-valued. For example, a game can belong to multiple genres (i.e RPG, Adventure, FPS etc) and possess multiple specifications (i.e. platformer, 3D, first-person, single-player, multi-player etc). These need to be extracted and *featurized* by multi-label encoding. Moreover, not all games have available metadata, meaning that these missing values must be imputed somehow before feeding it into the machine learning algorithm.

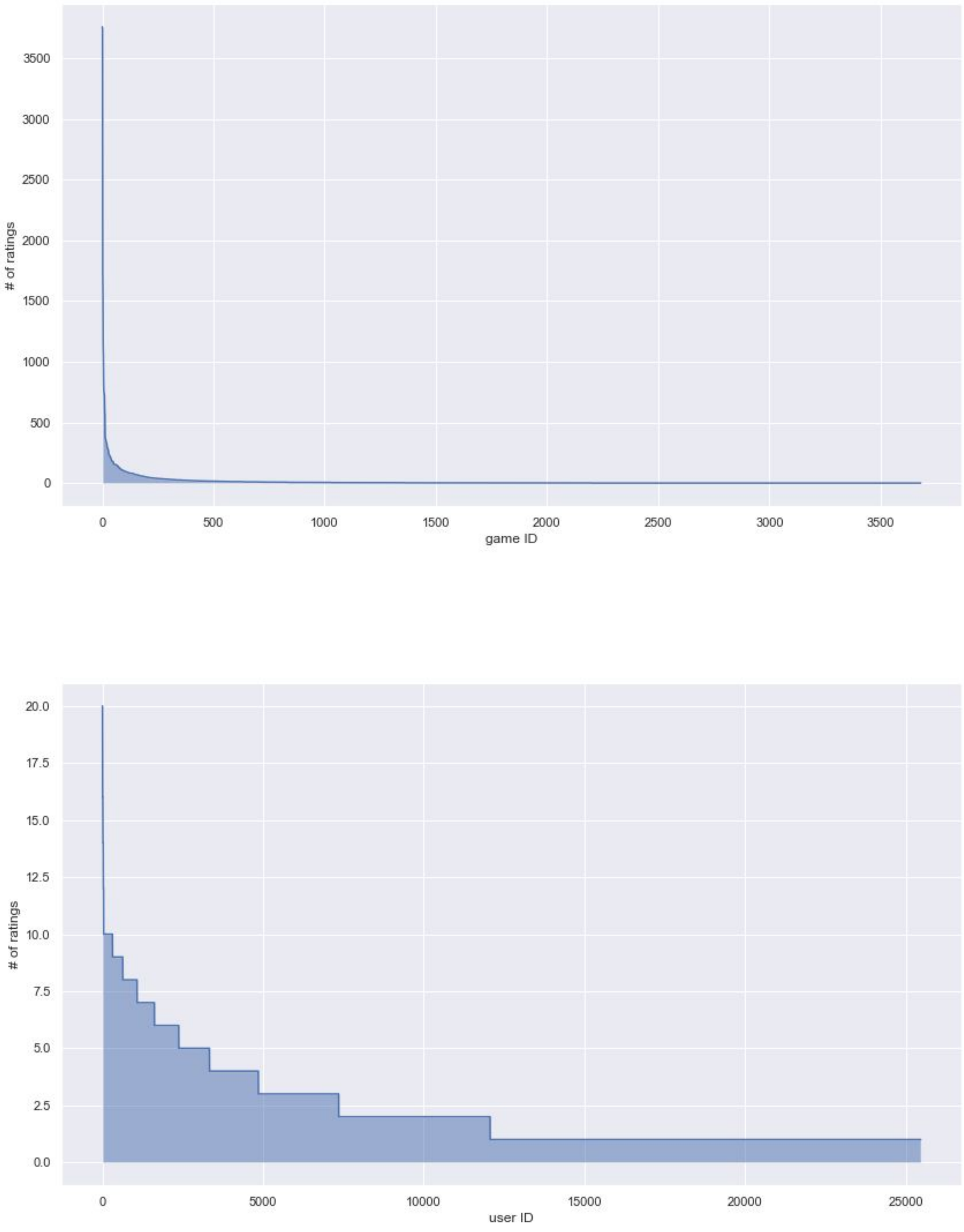

Figure 2: Distribution of available ratings per game/user ID. *Top: the distribution (counts) of available ratings per each game ID (sorted in descending order). Bottom: the distribution (counts) of available ratings per each user ID (sorted in descending order).*

Algorithms and Techniques

The primary purpose of all the algorithms developed here was to learn a parametrized model of how players interact with and rate games on Steam. This has been done by using the previous user-game ratings data (in a CF-like approach), game attribute/content data (in a CBF-like approach), as well as combining the two (in a hybrid-like approach). All of the models developed here try to predict how well a particular user will like a particular game (on a 0 to 1 scale) that he/she hasn't interacted with before.

Benchmark models

To establish a baseline, two benchmark models have been used in this project. The first is a simple baseline model that looks at overall ratings of each user, each game, and the general ratings of all games in the training set to predict specific user-game ratings (https://surprise.readthedocs.io/en/stable/basic_algorithms.html) The second is a model-based CF algorithm that attempts to learn latent representations of users and games in a lower-dimensional feature space based solely on user-game ratings. This was accomplished using matrix factorization via the SVD algorithm (https://surprise.readthedocs.io/en/stable/matrix_factorization.html). This algorithm is a standard practice in recommender system development and is therefore a good benchmark to test new models against.

Content-based filtering algorithm (CBF)

A custom CBF algorithm was developed and tested. To predict a rating for a particular user-game pair, it does the following:

1. First the game-game similarity matrix is computed via the Pearson correlation coefficient between the encoded game features extracted from the game data file.
2. Then the training dataset is inspected for games rated by the user of interest and those ratings are collected.
3. The rating for the item of interest is then calculated as a weighted average of that users ratings, where the weights are the similarity scores between the game of interest and the games this user has previously rated.
4. If no data is available for the game of interest (some data is missing), then the average rating of that user is used as the prediction instead.

This model is memory-based and does not really utilize machine learning to obtain the predictions, but it is a good model to test nevertheless.

Models based on neural nets

Several models based on neural nets have been developed and tested in this work. Here I describe the general principles used (for specific details about the architecture/parameters, see **Methodology**). The output of each network is a scalar (bounded between 0 and 1) - the predicted rating. All networks are trained by minimizing a regularized loss function - MSE between the predicted and real ratings.

Matrix factorization using neural nets (model A): This model is very similar to the benchmark SVD algorithm except it uses the general framework of neural nets (with stochastic gradient descent) to obtain the user-game embeddings. Each unique user and game are represented as regularized feature vectors (embeddings) of the same size, whose dot product is the predicted rating. The idea behind this model is to test how well a neural net can perform the same task the SVD benchmark algorithm is doing, before considering more complex architectures.

Deep learning of user-game interactions (model B): This model is similar to **model A**, except it does not compute the dot product between the user-game embeddings. Instead it concatenates the two vectors and then passes them through several fully-connected (FC) layers before finally outputting the rating. All the weights are regularized to avoid overfitting. The idea behind this model is that it would be able to model non-linear relationships between users and games.

Modelling user interactions with game metadata (model C): This model is similar to **model A**, except instead of using game embeddings, it computes the predicted rating as a dot product of an embedded user vector and game vector derived from the available game metadata. The idea behind this model is that the features derived from the game metadata are potentially more relevant than the autoencodings found in the embedded game layer.

Deep learning of user interactions with game metadata (model D): This model is similar to **model B**, except it uses the available game metadata instead of the game embeddings. User embeddings are concatenated with game metadata before being passed on to the deeper FC-layers of the network. The idea behind this model is that it would be able to model non-linear relationships between user and various game features and therefore arrive at better predictions.

Deep learning of user-game interactions with game metadata (model E): This model is similar to **model B** and **model D**. It uses 3 inputs: user embeddings, game embeddings, and the additional game metadata. All 3 inputs are concatenated and passed through a deep FC-network that outputs the predicted rating. The idea behind this model is that it would utilize all of the available information and model interactions between users, games, and game features in a complex, non-linear fashion.

III. Methodology

Data Preprocessing

The data available for this project comes in the form of two files: the user-game ratings and the game features data. Both files have been obtained from jjjj et al.

User-game ratings preprocessing

The data was formatted into a table format where each row contains the unique user ID, item ID (game), and the rating itself (which is a binary label of “recommend” or “not recommend”). The ratings were first converted into numerical values of 0 (not recommend) and 1 (recommend).

To lessen the effect of the extremely skewed distribution of ratings (**figure 2**) and enable the downstream modelling stages, the dataset was truncated by extracting a 3-core (a subset in which no user rated less than 3 items and no item was rated by less than 3 users). This was necessary due to the fact that all unique users and items must be present in the training set in order for the algorithms employed here to work. If a user rated

only one game for example, and that rating ended up in the test set, the model would not be able to find the necessary embeddings for that particular user ID and would not be able to make a prediction.
After preprocessing, the ratings dataset was split into train/test subsets in a stratified manner using a 75:25 ratio (i.e 25% of each users' ratings were held out in the test set and each user had at least 1 item in the training and test sets). Care was also taken so that no unique game ID was accidently completely dropped from the training set during the split. The algorithms developed here necessitate the presence of every unique user and game ID at least once in the training set in order to be able to construct the embedding matrices.

Game metadata preprocessing

The game metadata file contains several features about the games sold on Steam. These features include the game title, price, metascore (aggregate score based on many users' reviews), general sentiment (positive, negative, mixed etc.), genres (Action, Adventure, RPG etc.), specifications (single-player, multi-player, MMORG, FPS etc.), game publisher, game developer etc. Unique game ID:s were used as join keys to merge this dataset with the ratings dataset.

Of the mentioned features, only two are numerical: price and metascore. These were normalized using MinMax scaling to a 0 - 1 range as part of preprocessing. The sentiment feature contains 7 categorical values ranging from "*Overwhelmingly Negative*" to "*Overwhelmingly Positive*". These were mapped onto a linear scale of -3 to +3 to reflect the inherent ordinal nature of these categories (i.e "*Very Positive*" > "*Mostly Positive*"). The "*early_access*" feature is boolean and was mapped to 0/1. Two additional features were selected to include in the metadata dataset: genres and specs. As each contained a list of categorical values, these were encoded using a multi-label encoder (each available category was transformed into a feature of its own with values 0 or 1).

All of the missing values were imputed using feature mean values after merging with the ratings dataset. The games without any available metadata therefore only contained the feature mean values.

Implementation

All models developed in this work were trained on the training set (75% of the available ratings) and evaluated on the test set (25% of available ratings) using MSE as the metric. All models (save for the pure CBF algorithm) were regularized to a smaller or greater extent (see below).

Benchmark models

The benchmark models were implemented using the *surprise* python package (https://surprise.readthedocs.io/en/stable/index.html). The baseline model was fit on the training set using default parameters. Similarly, the matrix factorization algorithm (SVD) was applied on the training set using default L2 regularization strength of 0.02 and 10 latent factors for users and games.

Neural net models

All neural net models were developed using the Keras API with a Tensorflow backend configured to execute on the CPU. To monitor network training and avoid overfitting, 20% of the training set was used for validation. All networks were trained using a regularized loss function - MSE, and the training performance was monitored using MSE on the validation set. All models used the *Adam* optimizer with differing learning rates and were trained for 50 epochs using a batch size of 32. Callbacks were configured to reduce the learning rate upon reaching a plateau (no change in the validation set MSE for 5 epochs), and stop training altogether after 10 epochs if no change in observed.

Model A uses the unique user/game ID:s to create two embedding layers, whose dot product produces the predicted user-game ratings. Several embedding sizes were tested ranging from 8 - 64. Similarly, several regularization strengths were tested ranging from 10^{-4} to 10^{-1}. The final model contained an embedding of size 16 for both users and games. The embedding weights were initialized using *glorot* (truncated normal) initialization and the loss function was regularized via the L2 norm of the embedding vector weights (λ=0.05). Before outputting the final prediction, the dot product is passed through a batch normalization layer (to center it around 0) and capped to a 0 - 1 range using a sigmoid activation function. These last two steps were essential for obtaining stable training and reasonable performance.

Model B uses the unique user/game ID:s to create two embedding layers and concatenate them before passing them to FC layers. Several embedding sizes were tested ranging from 8 - 64. Several FC hidden layers were tried having sizes of 16, 32, and 64. The final model contains 2 hidden layers of size 32. The output of each layer (save the first layer with concatenated user-game embeddings) is batch normalized before passing it through the activation function (ReLu). For regularization, all weights, biases, and embeddings are regularized via the L2 norm. Different regularization strengths were tested (final model has a λ of 0.05). Dropout was used to further avoid overfitting (with drop rates of 0.2 to 0.5 tested). A dropout "layer" was placed at the output of each hidden layer in the network (save for the last hidden layer). The final model had a dropout rate of 0.5. The final model used the Adam optimizer with a learning rate of 10^{-4}.

Model C uses the unique user ID:s to create the user embeddings layer (size 62). The item input layer is composed of the 62 features extracted from the game metadata file. The dot product of these two layers is the predicted user-game ratings. Several regularization strengths were tested ranging from 10^{-6} to 10^{-1}. The embedding weights were initialized using *glorot* (truncated normal) initialization and the loss function was regularized via the L2 norm of the embedding vector. Before outputting the final prediction, the dot product is capped to a 0 - 1 range using a sigmoid activation.

Model D uses the unique user ID:s to create the user embeddings layer and concatenate it with a game input layer consisting of the 62 features extracted from the game metadata file. The resulting layer is then passed down to deeper FC layers. Several embedding sizes were tested ranging from 8 - 64 (final model contained 16). Several FC hidden layers were tried having sizes of 16, 32, and 64. The final model contains 1 hidden layer of size 64. The output of the hidden layer is batch normalized before passing it through the activation function (ReLu). For regularization, all weights, biases, and embeddings are regularized via the L2 norm. Different regularization strengths were tested (final model has a λ of 0.05). Dropout was used to further avoid overfitting (with drop rates of 0.2 to 0.5 tested). A dropout "layer" was placed at the output of each hidden layer in the network. The final model had a dropout rate of 0.5. The final model used the Adam optimizer with a learning rate of 10^{-5}.

Model E uses the unique user/game ID:s to create two embedding layers for users and items. The model also uses the game metadata features as a third input layer of size 62. All three input layers are concatenated before being passed through the deeper FC layers. Several embedding sizes were tested ranging from 8 - 64 (final model has 16). Several FC hidden layers were tried having sizes of 16, 32, and 64. The final model contained 3 hidden layers of sizes 32, 64, and 32. The output of each layer (save the first layer with concatenated inputs) was batch normalized before passing it through the activation function (ReLu). For regularization, all weights, biases, and embeddings are regularized via the L2 norm. Different regularization strengths were tested (final model has a λ of 0.001). Dropout was used to further avoid overfitting (with drop rates of 0.2 to 0.5 tested). A dropout “layer” was placed at the output of each hidden layer in the network (save for the last hidden layer). The final model had a dropout rate of 0.5. The final model used the Adam optimizer with a learning rate of 10^{-4}.

Refinement

During the development of optimal neural net models, many architectures and parameters were tried and tested. A notable fact during refinement was that regularization strength played a major role in obtaining good models. Setting λ to an optimal value was often a deciding factor of whether the model will overfit or underfit. Furthermore, the learning rate - ɑ, was extremely important for obtaining stable learning. In many cases the parameter had to be reduced significantly in order to obtain stable training. Dropout and batch normalization generally improved the results by decreasing overfitting and speeding up training. In **model A**, batch normalization was critical for obtaining reasonable predictions by centering the dot product of user-item embeddings before passing it through the sigmoid activation.

IV. Results

Data Preprocessing

The results of the data preprocessing steps can be seen in **figures 3** and **4**. The final dataset statistics can be read from **table 2**. As a result of extracting the 3-core from the original dataset, many unique users and games were dropped, and the entire set was reduced to about 56 % of its original size. This did not however change the distribution of the positive and negative ratings as can be seen in **figure 3**.

Table 2: Summary of the 3-core truncated user-game ratings dataset.

# of users	**# of games**	**# of available ratings**	**Ratings density (%)**
6906	1346	33292	0.3582

Analysis of the train-test split results indicated that no unique users or items were completely dropped from the training set and that each user had at least one rating in the test set. Similarly, the split did not affect the distribution of positive and negative ratings, with the ratio remaining nearly the same.

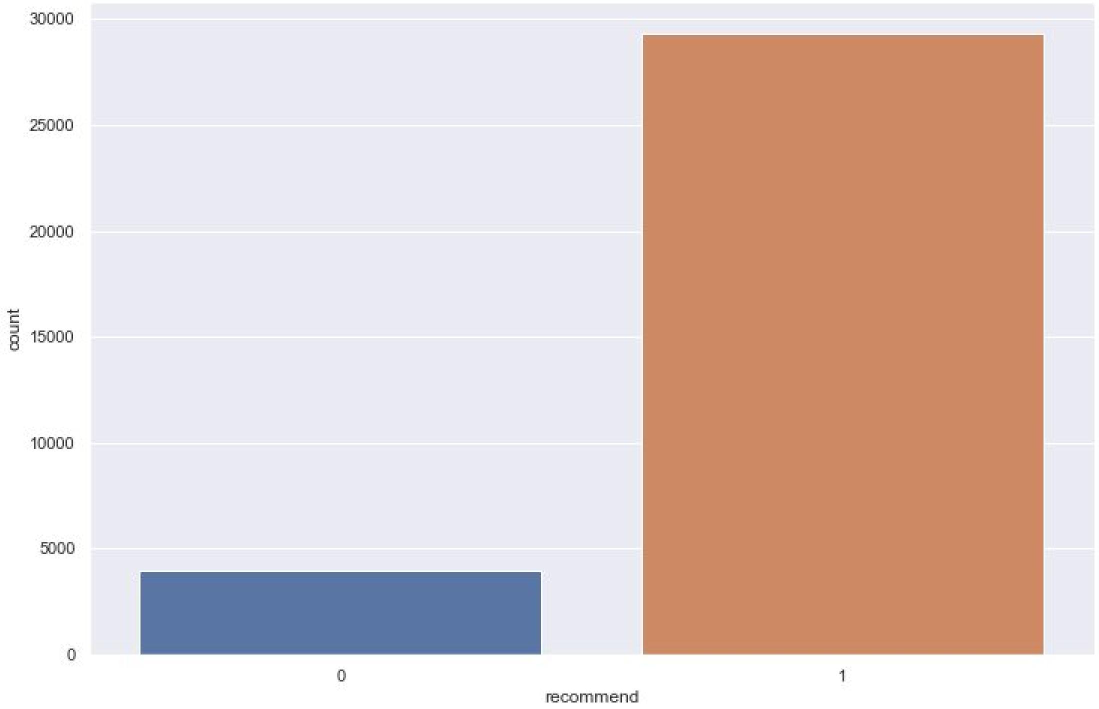

Figure 3: The distribution of positive and negative ratings in the 3-core truncated ratings dataset. *0 - negative ratings (11.89 %), 1 - positive ratings (88.11 %)*

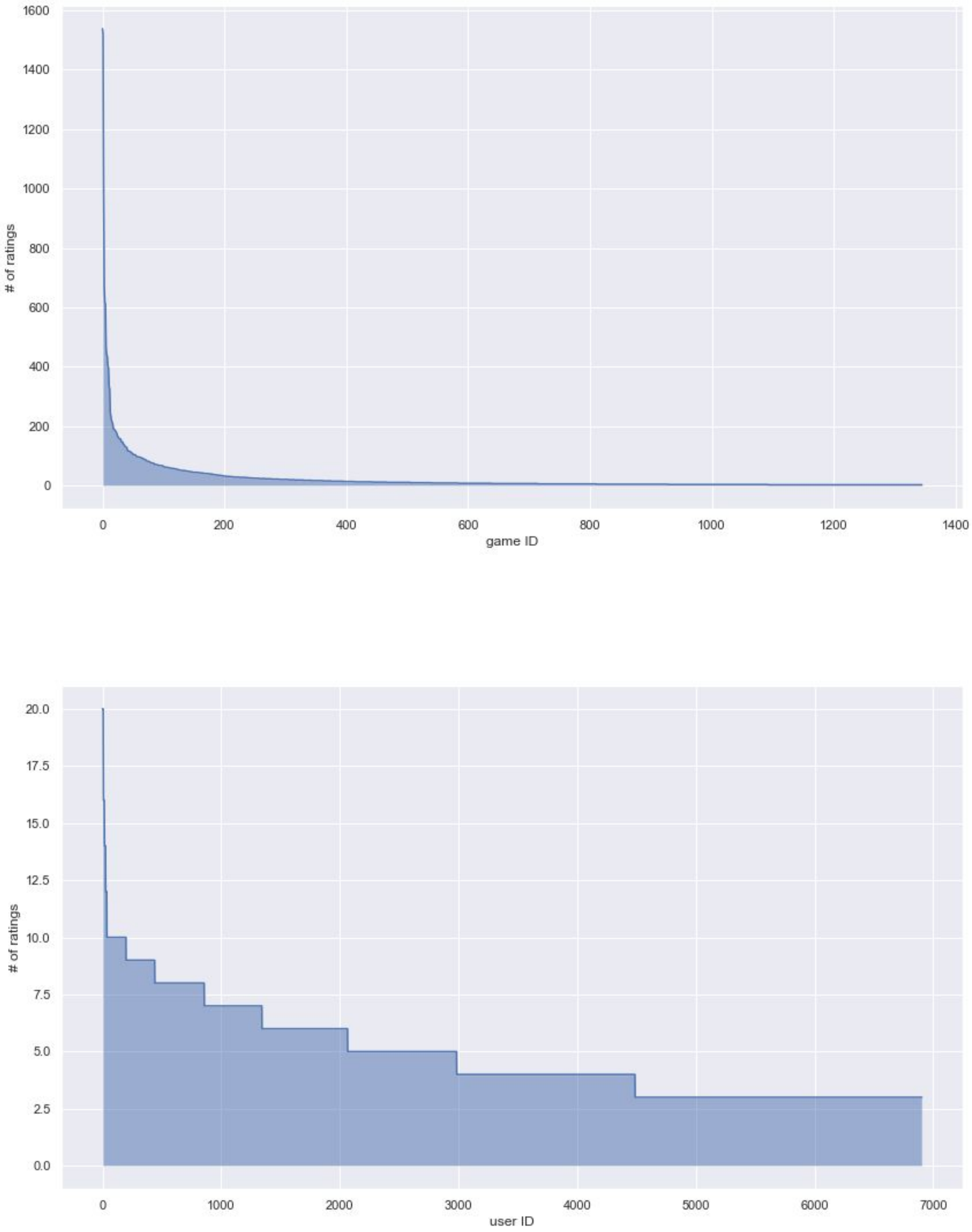

Figure 4:* *Distribution of available ratings per game/user ID in the 3-core truncated dataset. *Top: the distribution (counts) of available ratings per each game ID (sorted in descending order). Bottom: the distribution (counts) of available ratings per each user ID (sorted in descending order).*

Model Evaluation and Validation

The model that achieved the highest performance on the test set was **model B**: **deep learning of user-game interactions.** This model maps unique user/game ID:s to two embedding matrices that act as autoencoders, automatically learning meaningful features of user-game interaction based solely on user-game ratings data (and not considering the additional game metadata). Utilizing a deep architecture (with 2 hidden FC layers), this model is capable of modelling non-linear relationships between the inferred user-item embeddings, thus giving it greater representational power. The final performance on the test set was MSE=0.09.

During training, this model (and all the other neural net models) was prone to overfit, and a substantial tuning of hyperparameters such as regularization strength and dropout rate were necessary in order to achieve good performance. Furthermore, reducing the learning rate was the key factor in obtaining stable learning that resulted in low MSE on the validation set. The final architecture of **model B** can be seen in **figure 5**.

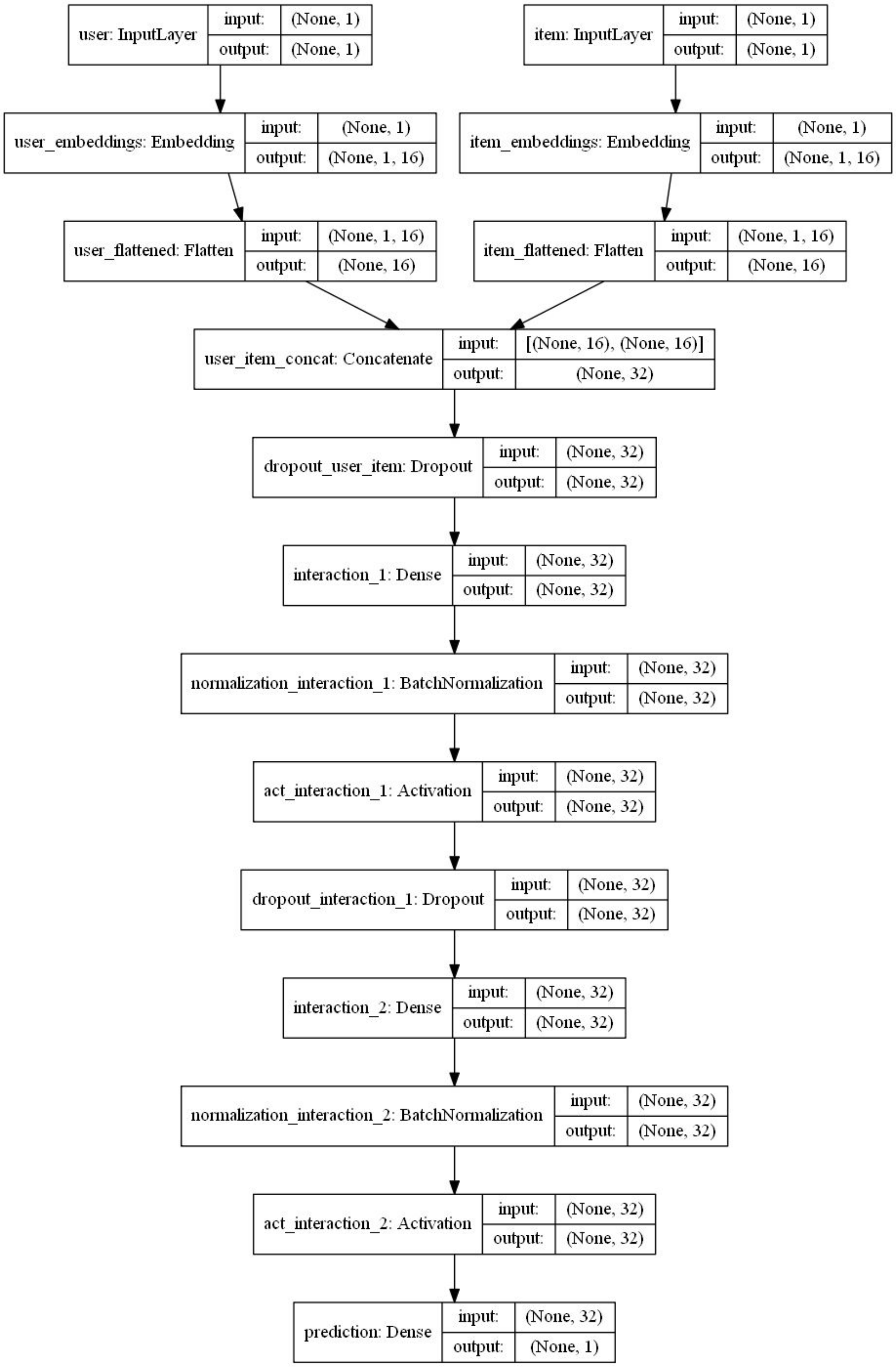

Figure 5: Final model B architecture. *The output of the prediction layer is passed through a sigmoid activation to cap the prediction to a 0-1 range. The hidden layers use the ReLu activation function.*

Justification

Although **model B** performed well on the test set, it could still not beat the SVD benchmark model of the *surprise* python package (which achieved an MSE of 0.086 on the test set). Based on this, I would not select **model B** as the best approach for solving the problem of predicting user-game ratings, at least not in its current form.

V. Conclusion

Free-Form Visualization

Here I present and summarize the overall project workflow (**figure 6**) as a graph of stages (operations) and intermediary data/models. The process begins with importing, cleaning, and formatting the two input files: the user-game ratings and the game metadata.

A 3-core is then extracted from the full ratings dataset, before a train/test split is made. The features of the game metadata set are encoded.

Models are built using the training set and in some cases the game metadata as well, During model training, 20% of the data is used for cross-validation to monitor overfitting. A model is produced and evaluated on the test set, and a final score is calculated.

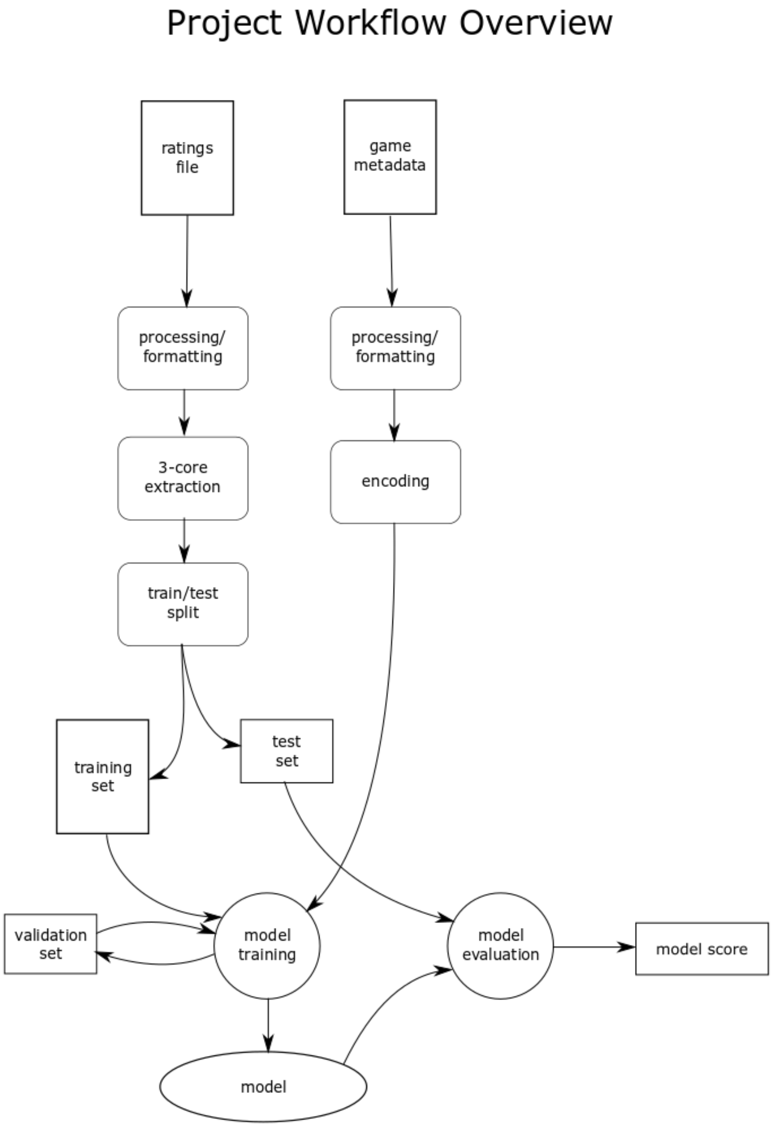

Figure 6: Project workflow overview*. This figure summarizes the overall project workflow: data preprocessing, encoding, splitting into train/test sets, model building with cross-validation, and final model evaluation (scoring)*

Reflection

In this work, I built and tested several machine learning algorithms for a game recommender system. These algorithms are trained on previous user-game ratings and attempt to predict how a particular user would rate a particular game that he/she hasn’t interacted with before. The dataset for building the recommender system was obtained from Steam, a platform for distributing and playing computer games.

There are many particularities regarding recommender systems that sets them apart from other machine learning tasks. One of the most challenging aspects of this project was in how the data needs to be processed. Due to the extremely skewed nature of user-item ratings (**figure 2**), the ratings matrix is extremely sparse and for many users/items there is just not enough data to build a meaningful representation of how users interact with games. To alleviate the sparsity problem, I extracted a 3-core which resulted in a somewhat more dense dataset (though still quite sparse).

A second aspect that was difficult was how the dataset should be split into training and test sets. All of the algorithms used in this work necessitate that all unique user and game ID:s are present in the training set (and preferably also in the test set). This is due to the fact that the only information we have about the users is their unique ID:s. As such, the only way their preferences can be modelled is through embeddings that try to represent each user as a vector of real numbers. Therefore, it is not enough to simply randomly split the dataset into train/test sets: a lot of consideration is necessary in order to ensure the obtained sets are compatible with the problem at hand.

Improvement

The reason why none of the models developed in this project could beat the benchmark models could be many-fold. *Surprise* is a refined package for building recommendation systems and its algorithms have been optimized and tuned to work for such use-cases. It is quite likely that further experimentation with the models developed here and their learning parameters would result in a performance better than that of the benchmark models.

It is also interesting to observe that the addition of game metadata did not improve any of the model's performance. The reasons could be many, perhaps the extracted game features are simply not relevant and only result in additional noise in the data. Perhaps a different network architecture would result in better performance. Some of the games had no available metadata, and in those cases the feature mean values were used. It is possible that this introduced a lot of noise in the system. Therefore, obtaining the full metadata set for all games in the ratings dataset would most likely be beneficial.

References

[1] 'Recommender system - Wikipedia'. [Online]. Available: https://en.wikipedia.org/wiki/Recommender_system. [Accessed: 21-Aug-2019].

[2] 'Toward the Next Generation of Recommender Systems'. [Online]. Available: https://dl.acm.org/citation.cfm?id=1070611.1070751. [Accessed: 21-Aug-2019].

[3] 'Item recommendation on monotonic behavior chains'. [Online]. Available: https://dl.acm.org/citation.cfm?id=3240369. [Accessed: 24-Aug-2019].

[4] '[1808.09781] Self-Attentive Sequential Recommendation'. [Online]. Available: https://arxiv.org/abs/1808.09781. [Accessed: 24-Aug-2019].

[5] 'Generating and Personalizing Bundle Recommendations on Steam'. [Online]. Available: https://dl.acm.org/citation.cfm?id=3077136.3080724. [Accessed: 24-Aug-2019].

[6] 'TopicMF'. [Online]. Available: https://dl.acm.org/citation.cfm?id=2893874. [Accessed: 22-Aug-2019].