Steam User Score Classification for Video Games

Project Report

Team 8

presented by
Alexander Weiß
Christian Marheine
Lars Hoffmann
Nick Weber
Simon Beckmann

submitted to the
Data and Web Science Group
Prof. Dr. Christian Bizer
University of Mannheim

May 26, 2017

Contents

1 Application Area and Goals

In this work we will analyze a large collection of game data from the digital game distribution platform Steam. Besides being a game distribution platform for PC, Mac and Linux, Steam additionally offers services like digital rights management, multiplayer gaming and social networking. Alongside the online-shop at `http://store.steampowered.com` it offers a downloadable client to install purchased games on the local client, chat with friends, synchronize your game progress in a cloud storage and a variety of other functions. With its over 125 million registered accounts, the platform is considered to be the largest digital distribution platform for PC gaming. In 2013, Steam was estimated to have a 75% market share[1].

Today, many online-shops offer product ratings based on reviews of customers who bought an item. Steam also offers such a rating system based on recommendations, which they call *User Score*. Any user of the platform can make a recommendation on every game they bought. It can either be positive or negative. A simple way of calculating the user score can be achieved by using the following formula:

$$\%UserScore = \left(\frac{Rec_{pos}}{Rec_{pos} + Rec_{neg}} \right) * 100$$

This formula demonstrates the basic principle of the user score. It should be noted that in reality the calculation differs slightly. For game developers it may be interesting to get insights on the most valuable properties of a game, which lead to a good user score. Properties that may influence the user score could be genre, supported languages, price, release date, developer or publisher. Therefore, our aim for this team project is to find good models to predict the user score of unreleased games based on their respective properties. If we find a reasonable good classifier model, we can conclude that specific properties of games lead to a specific user score.

2 Gathering, Structure and Size of the Data Set

To build up our dataset, we searched for different web services, which offer information about video games sold on Steam. Initially, we thought about using the promising dataset at `http://steam.internet.byu.edu`. This dataset was gathered as part of the work by [O'Neill et al., 2016] by using the Steam Web API.

[1] `http://www.bloomberg.com/news/2013-11-04/valve-lines-up-console-partners-in-challenge-to-microsoft-sony.html/`

As our objective shifted away from building a recommender system, we searched for new data resources. As a result, the *Steam Web API*, *SteamSpy* and *SteamDB* turned out to be appropriate resources to get the game information needed. Since all required information was not available at only one of these sources, we chose to combine their provided data to create our dataset. In the following, the gathering of data at the different sources is described in more detail.

The Steam Web API is a set of official APIs provided by Valve, the corporation behind Steam. It is a RESTful web service, which provides an enormous amount of information about Steam's social user network, gaming statistics and the Steam market for video games. It offers different HTTP endpoints to request information from the Steam database that is returned in a JSON format. Since our interest was to get information about video games, only two of the endpoints were used.

1. Endpoint: http://api.steampowered.com/ISteamApps/GetAppList/v0001/
2. Endpoint: http://store.steampowered.com/api/appdetails?appids=[AppID]

The first endpoint was used to gather all available App-IDs. Unfortunately, these IDs are also used for irrelevant apps such as trailer, music or movie entities. Therefore, these IDs had to be filtered to only get game entities. Additionally, we were only interested in full games. Therefore, game related apps such as demos or Downloadable Content (DLC) had to be filtered as well. The second endpoint was used to crawl information about the qualified App-IDs. All attributes gathered from this can be found in Figure 1.

After the initial crawl the user score was missing. To cope with this issue it was decided to crawl the Steam shopping site directly. This is done by visiting `http://store.steampowered.com/app/[AppID]` with the respective ID as URL parameter and then parsing the returned HTML document. Inside the individual HTML body of each site exists a HTML child, which contains information about the number of positive and negative recommendations. This in turn can easily be parsed with a HTML parser. The parser was used to extract the number of positive and negative recommendations to calculate the user score. Only after the crawl we discovered missing values for all games that had an age rating of 18. This was due to an age check before entering the site. Bypassing this age check automatically would have been a non-trivial task. To complement the missing values, a new source with the same up-to-date information was searched. Therefore, SteamDB[2] was chosen. This is an unofficial site which aggregates and visualizes data available on Steam. The missing values were extracted as before by parsing the retrieved HTML document. In our initial crawl the price of each game was not

[2] `https://steamdb.info`

Attribute	Type	Description
appid	Number	ID of the entitiy
name	String	Name of the entity
type	String	Values: ”game”,”movie”,”demo”,”dlc”
required_age	Number	Age Rating
release_date	String	Release date of the entity
is_free	Boolean	Indicates if the game is free to play
controller_support	String	Indicates if controllers are supported
detailed_description	String	HTML String with a long description of the game
about_the_game	String	HTML String
short_description	String	HTML String with description of the game
supported_languages	String	HTML String of supported languages
header_image	String	URL to image
website	String	URL to the game website
developer	Array of String	Developers of this entity
publisher	Array of String	Publishers of this entity
currency	String	Currency
inital	Number	Price when it was released
final	Number	Current price
discount_percentage	Number	If discounted, the discount percentage
packages	Array of String	Array of package IDs.
platforms	Object	Boolean values of supported OS (Windows, Mac, Linux)
categories	Array of Number	Array of category IDs
genres	Array of String	Array of different genres
recommendations	Number	Total number of recommendations

Figure 1: Steam API attributes

in a standardized format. SteamSpy[3] is another unofficial site that provides additional information to Steam data as well as its own API to gather the data. We used this API to gather all game prices (in US$) and some other information. However, some of these attributes had to be discarded later on.

In the end, the gathered data consisted of 13,103 game records with 42 attributes each. This data was used for further pre-processing as described in the next section. test

3 Pre-Processing

After all the necessary data was gathered, the single stored CSV files were joined together for further processing in RapidMiner. Before using classifier algorithms for data mining, it was necessary to pre-process the gathered data. In the following, the steps in pre-processing will be described.

[3] `https://steamspy.com`

The first step was to **remove all unnecessary attributes** from our data set. Initially it contained several pure HTML code attributes, which did not increase the information content. An example is the *detailed_description* attribute (cf. Figure 1). It contains a long HTML string with the description of the game. We also removed attributes containing URLs and all obsolete price attributes of our initial crawl (cf. Section 2). Additionally, we discarded the attributes *owners* and *final_review_count* during the data mining stage. Not only did it improve the performance, also *owners* and *final_review_count* were unrealistic attributes for unreleased games.

Next we had to **split attributes** containing more than one information into multiple columns. Some attributes were stored as arrays or strings like the supported languages of a game. In the example of languages, the attribute was a comma-separated string concatenation of supported languages. Out of these we generated a new binary attribute for each important language (e.g., English, German, Spanish) found in the string concatenation by using the *Generate Attribute*-operator with function expressions as seen in Figure 2.

```
if(contains([supported_languages],"English"), true,false)
```

Figure 2: Function Expression in RapidMiner

After splitting the attributes we had to **replace missing values** for all unknown attribute values. For attributes of type *boolean*, only the value *true* was existent. Instead of having value *false*, the values were missing and therefore *unknown*. The same problem also occurred for other non boolean attributes. We used the *Replace Missing Values*-operator in RapidMiner in order to assign the respective default value to attribute values.

Now that we have assigned default values to some attributes, there were still instances with missing attribute values. Therefore, we **filtered examples** for records with missing attribute values for which no default value could be assigned.

After having checked the data for further inconsistency, the next step was to **remove duplicates** in the data. This was achieved by using the *Remove Duplicates*-operator with the attribute *name* as selection filter.

Attributes like the release date were formatted in every conceivable way, thus **standardization** was required. Therefore, the release date was split into release year and month. Both attributes were generated by using the *Generate Attribute*-operator as already mentioned earlier. In this case we used an expression to find years between 1998 and 2017 in the string and generated the *release_year*-attribute.

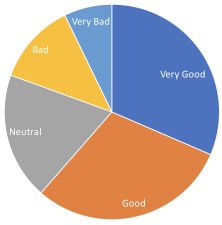
In the same way we generated the *release_month*-attribute by looking for substrings such as *'May'*, *'June'*, and so on.

Now that the data was cleaned up, 28 of the initial 42 attributes were left. Seven of these 28 attributes were numerical and the remaining ones were nominal. Before starting the data mining stage, we had to further transform the data in order to use it as input for the algorithms. This transformation process will now be elaborated in more detail.

As described in the previous section (cf. Section 1), our goal is to build a classifier able to predict user scores for unseen instances. The user score is a percentage for the proportion of users that positively rated a game (i.e., recommended the game). It is obvious that the raw user score is not suitable as a class label since every possible user score would be treated as a single class. Therefore, we discretized the data by using the *Discretize by User Specification*-operator in RapidMiner. This operator enables to change numerical attributes into user-specific classes. Thereby, the numerical user score is changed to a nominal attribute. For algorithms which can only handle nominal or binomial attributes as input, we also discretized if needed. Our initial mapping produced five classes that can be discerned from the following table. This is based on *metacritic*'s[4] mapping. Soon we observed that

User Score	5 Classes	4 Classes	3 Classes
89%-100%	Very Good	Very Good	Good
74%-89%	Good	Good	
49%-74%	Neutral	Neutral	Neutral
19%-49%	Bad	Bad	Bad
0%-19%	Very Bad		

(a) User Score Discretization



(b) Class Distribution

Figure 3: User Score Discretization and Class Distribution for five Classes

only a very small fraction of data belongs in the category *Very Bad* (cf. Figure 3b) and so we decided to use one *Bad* category for all games with a rating below 50%. This reduced the multiclass classification problem to four classes. Later on we also tried three classes but focused on four classes. The discretization of the user score for each number of classes can be found in Figure 3a.

Tests in RapidMiner revealed the most predictive attributes through weighting attributes by information gain. As a result, the most predictive attributes were *developer* and *publisher*. In the meantime we did our first classification attempts

[4] http://www.metacritic.com/about-metascores

in RapidMiner to optimize and fine-tune the data. It was soon discovered that there was a relatively big proportion of game developer and publisher with only one single game release. The used classifiers had problems in classifying a lot of games because their developer and publisher were never seen before. The remaining attributes had only a fraction of the predictive power, thus lots of games were misclassified. Therefore, we built a second data set containing only games that had at least n other games ($n = 3$ turned out to be a good number) released by its respective developer or publisher. This was realized by aggregating developer and publisher, counting their number of appearances and filtering all examples with less than n appearances. The number of games in our data set decreased significantly from $11,460$ (after all the previous pre-processing steps) to just $3,471$ examples. On the other hand, this increased accuracy in some cases by roughly 10%. We agreed to use both data sets and also tried different numbers of classes for future predictions. This produced six different data sets we are going to use for each classifier. The previously exemplified aggregation process is labeled as '+' in Section 4.2 Figure 5.

4 Data Mining

4.1 Classification

Now that we have finished the pre-processing of data, we started with data mining. We decided to try and evaluate multiple classifiers. The different approaches for each tested classifier will be elaborated in the following. We start with the classifiers that we discarded relatively fast and why so.

The *ID3*-operator learns an unpruned Decision Tree from nominal data only. Because it is hard to find the optimal discretization for every attribute and the results were not very promising, we decided to discard ID3 and focus on other classifiers. Another classifier we tried to apply was Random Forrest. This classifier randomly generates decision trees and chooses the best one as a final model. Due to this fact, it cannot reach a better performance than a optimized decision tree so we discarded this classifier as well. Other considered classifiers were Neural Nets and Deep Learning. One of the issues regarding both classifiers is that it took too much time, when combined with the *Optimize Parameter*-operator, to create a model for our dataset . Despite this, neural nets cannot handle binomial attributes.

In the following, we will focus on the more promising classifiers that we tested in depth.

Decision Tree: The *Decision Tree*-operator (DT) in RapidMiner uses a more flexible algorithm than ID3, CHAID or C4.5 for example. Before building the tree,

we discretized numerical attributes like the price. Furthermore, we filtered some attributes with the use of weighting by information gain (see kNN), to remove attributes that do not contribute to a better performance. Despite this, we used the *Optimize Parameter*-operator to find the best values for specific parameters we considered as important. For example, we searched for the optimal splitting criterion and came up with GainRATIO except for one time where the Gini index criterion performed better. An issue we encountered during testing was that if we applied pre- or post-pruning, the DT was most likely classifying every single example as one class. This was definitely not the scope of expectations. Therefore, we decided to drop the parameter of pruning and not apply it. After dropping this parameter the DT was giving us comprehensible results.

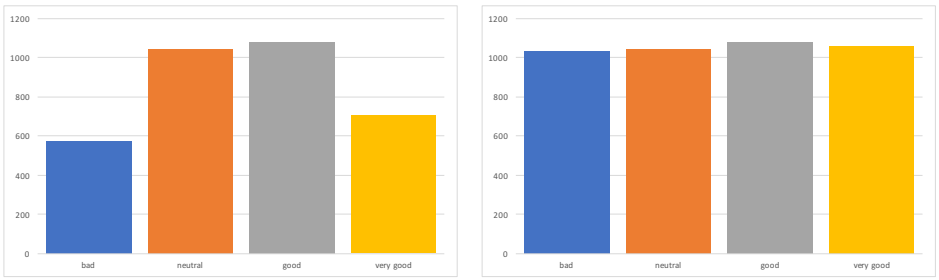
Multiway Decision Tree: The normal DT only supports binary splits, which makes splits on numerical data repetitively. A lot of times the trees are getting very deep and are hard to understand for humans. Therefore, we used the *Decision Tree (Multiway)*-operator (MW-DT). MW-DTs tend to give more accurate results in some use cases as well as provide a more human understandable tree. We used the *Optimize Parameter*-operator to find the best settings for the MW-DT. Furthermore, we encountered the same problem as with the normal DT (cf. Decision Tree) and dropped the parameter of pruning.

***k*NN:** The k-Nearest-Neighbor (kNN) classifier performed better than the previous decision trees. For kNN to be meaningful, it is important to normalize numeric attributes, find an appropriate k, balance the data and avoid the curse of dimensionality (e.g., avoid too many dimensions for small sized training data)[Jiang et al., 2007]. The k was determined by the *Optimize Parameter*-operator in the range $1 \leq k \leq 50$ and the balancing is described in Section 4.2. By using the *Weight by Information Gain*-operator in RapidMiner, the most unpredictive attributes were discovered. Attributes with too little information gain were discarded and in turn increased the accuracy by roughly 3%. In the end 10 to 15 (depending on the data set) attributes were used for classifying.

Naive Bayes: The (naive) independence assumption between attributes of the naive Bayes classifier is rarely true in real-world applications. Nevertheless and against our expectations, the naive Bayes classifier performed best. Before learning the classifier, we discretized numerical attributes and selected a subset of all attributes with the use of weighting by information gain (see kNN). In order to cope with the zero-frequency problem, *laplace correction* was used in RapidMiner.

4.2 Evaluation

To estimate a model's performance we used cross-validation. The data was split into k subsets of equal size and these were generated using stratified sampling. The number of subsets k was determined by the *Optimize Parameter*-operator in the range $2 \leq k \leq 50$. Before training the classifiers, we balanced our data in the training panel inside the cross-validation operator. In the example of four classes, our training data consisted of roughly 1,000 neutral and good games and of about 600 bad and very good ones (Note: these numbers are dependent on the used k in cross validation). The class distribution before and after balancing the data is visualized in Figure 4. Each of the six data sets have a different adjusted balance.



(a) Unbalanced Data (b) Balanced Data

Figure 4: Class Distribution for Unbalanced and Balanced Data

To measure performance we use a single precision, recall, F_1-score and accuracy value for the entire confusion matrix. It should be noted that there are two ways to calculate averaged precision and recall for multiclass problems: *Macro- and Microaveraging*. Macroaveraging simply computes the average over all classes and therefore gives equal weight to each class. Because of this property and the fact that we have a much varied class distribution, we chose Macroaveraging to calculate performance measures. It is calculated as:

$$Measure_{macro} = \frac{1}{n} \sum_{i=1}^{n} Measure(c_i)$$

where *Measure* is either precision or recall, n is the number of classes and c_i is class i. Therefore, we first calculated precision and recall for each single class and then average these. For further information about macro- and microaveraging we recommend [Manning et al., 2008]. With this macroaveraged precision and recall, the F_1-score and accuracy can be calculated as usual. The performance of each classifier and their respective setting can be found in Figure 5.

		Precision			Recall			F_1-Score			Accuracy		
# classes		3	4	5	3	4	5	3	4	5	3	4	5
MW-DT	+	44.4	34.4	29.0	45.9	36.2	30.9	44.5	34.6	29.5	48.3	34.5	32.0
	-	40.8	32.1	26.7	41.3	32.8	28.5	40.5	32.0	26.2	43.6	32.1	27.5
Decision Tree	+	45.3	35.1	30.7	47.8	36.6	33.5	44.9	35.2	31.3	49.1	35.1	32.9
	-	41.3	32.1	28.3	42.8	32.8	31.6	39.7	32.0	28.1	45.0	32.1	28.7
kNN	+	49.8	41.7	**37.7**	49.9	41.6	37.9	49.9	39.5	**37.8**	**55.0**	41.6	**40.0**
	-	42.3	34.0	29.3	42.3	34.0	29.2	42.3	34.0	29.2	46.9	34.4	32.2
Naive Bayes	+	**51.0**	**44.7**	36.4	**53.0**	**46.3**	**40.4**	**51.2**	**45.0**	37.0	54.9	**44.6**	39.2
	-	45.9	36.8	31.6	47.0	38.4	34.4	44.9	35.9	30.0	47.7	35.8	30.4

The '+' and '-' next to the classifiers name indicates whether the aggregation process (as described at the end in Section 3) was used (+) or not (-).

Figure 5: Evaluation Results in percent

4.3 Conclusion

Over the course of this project we analyzed a large collection of game data. We tried to build a classifier able to predict user ratings for games, based on their attributes. Therefore, we gathered game data from different resources, joined the data into one data set and used preprocessing steps to build the final training set. By training different classifiers, we hoped to find a reasonable good classifier model that can predict game ratings based on their properties. We focused on the most promising classifiers and presented their settings as well as their evaluation results.

As can be seen in Figure 5, there was not a single classifier able to predict user scores with a satisfying performance. Overall the Naive Bayes classifier performed best but was closely followed by the kNN classifier. Each column in the table has one bold written entry, indicating the best performance for the respective setting. It can be seen nicely, that all best performances for each number of classes were with the aggregated developer and publisher setting (cf. Section 3). It is also obvious that the performance of each classifier increases with decreasing number of classes. Out of interest we tried a discretization of the user score into two classes (good and bad) and even then the performance was at a moderate 65% (F_1-score). As the binary classification would be an over simplification of our initial goal and has several other problems (most of the ratings reside in the middle neutral range), we did not further pursue this. Due to the large amount of attributes video games can have, alongside with the fact that some properties of video games can not be captured (e.g., the overall hype for a game), makes it very difficult for classifiers to predict the correct user rating. Our results show that the prediction is not very dependent on numerical- or nominal data. Much more it seems that the needed information to create a good model lies in textual form and personal opinions of the users. We are confident, that with the use of text analysis methods, the classification performance could have been improved. By conducting surveys among players of

video games, further interesting attributes could be revealed. Already early on we searched for similar data mining projects to compare against. Unfortunately, there were only a small number of scientific publications on video game related data mining projects. However, there were plenty of publications on the somewhat similiar task of movie rating prediction. [Armstrong and Yoon, 2008] investigated to which extent a movie's average user rating can be predicted. They concluded with

> *"Given the nature of the data we are trying to learn, it is not surprising that a more significant increase in prediction accuracy [...] was not achieved. Many of the reasons people like or dislike a movie depend upon factors that [...] cannot be captured (or are subjective), such as how intriguing a plotline is; or are difficult to encode as a feature, such as the plot description."*

This supports our estimation and could be a major factor for the poor performance of the tested classifiers. It would be possible to achieve a better performance by over simplifying the problem (e.g., above mentioned two-class problem) but this would have no benefit for real-world applications. With this work we provided a good foundation for predicting video game ratings. It would be interesting to further look into the problem and build on this work in the lecture *Text Mining*.

References

Nick Armstrong and Kevin Yoon. Movie rating prediction. Technical report, 2008.

L. Jiang, Z. Cai, D. Wang, and S. Jiang. Survey of improving k-nearest-neighbor for classification. In *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, volume 1, pages 679–683, Aug 2007. doi: 10.1109/FSKD.2007.552.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

Mark O'Neill, Elham Vaziripour, Justin Wu, and Daniel Zappala. Condensing steam: Distilling the diversity of gamer behavior. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 81–95, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4526-2. doi: 10.1145/2987443.2987489. URL `http://doi.acm.org/10.1145/2987443.2987489`.