Greedy Recommendation Model for Collectible Card Games

Gustavo Cornejo Juanita Fernández Francisco Jorquera

Abstract

Building optimal decks in collectible card games is a challenging combinatorial problem. Players must evaluate individual cards based on their attributes while accounting for synergies and interactions within the deck and the overall context of the game. This paper introduces a recommendation system that employs a greedy algorithm to maximize the probability of winning against a random opponent in Hearthstone. Experimental results indicate that this approach significantly outperforms alternative methods, although it incurs higher computational costs. These findings underscore the effectiveness of algorithmic solutions in improving decision-making processes in deck-building strategies. Code is available at this repository: https://github.com/gustavo-cornejo/RecSist-Proyecto.

1. Introduction

A prominent example of collectible card games (CCGs) is Hearthstone, a free-to-play online game developed by Blizzard Entertainment and released in 2014. In 2016, the game had more than 40 million registered players worldwide.

In Hearthstone, players build decks of 30 cards anchored by a hero that has 30 health points and a unique ability. Matches are played in a one-on-one format, where players take turns playing cards and attacking their opponent. The goal is to reduce the opponent's health to zero.

The total pool of cards is of 5,000, and given the vast number of potential combinations and strategies, generating decks with consistently high win rates is particularly challenging. Each strategy requires substantial experimentation and adaptation to dynamic game play scenarios.

This work proposes a recommendation system that draws on data from previous battles to guide players in constructing optimal decks. The system approximates deck win rates and suggests essential modifications, enhancing the efficiency and effectiveness of deck-building strategies.

2. Related work

Research on deck recommendation systems for CCGs is limited, but with notable contributions such as the Qdeck system proposed by Chen et al. (Chen et al., 2018). This system employs reinforcement learning and probabilistic methods, such as Markov processes, to recommend decks efficiently. The Qdeck system has demonstrated impressive scalability and deck quality, and its algorithm is reproduced as part of this study.

Hearthstone's research has primarily focused on analyzing game play data. For instance, Mora et al. (Mora et al., 2022) explores archetypes prevalent in the game using datasets of thousands of decks contributed by the player's community. This study uses similar datasets under the title "History of Hearthstone" for analysis.

Other research has addressed win rate estimation, such as the AAIA'18 competition organized by Silver Bullet Labs and Knowledge Pit during the FedCSIS 2018 conference.

3. Datasets

Two datasets are used in this study. "History of Hearthstone" was sourced from HearthPwn, this dataset contains over 200,000 decks submitted and rated by Hearthstone players. Each entry includes crafting cost, hero class, card composition, average player rating, and upload date. Analysis indicates a balanced distribution of decks across hero classes, with no class having fewer than 30,000 registered decks, as shown in Figure 1.

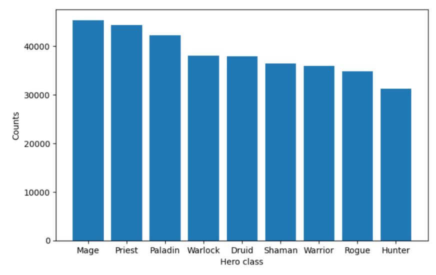


Figure 1. Number of decks for each class.

The second dataset "Hearthstone decks", was obtained from HSReplay, with over 700 decks. Each deck includes card composition and win rates based on at least 1,000 matches. The data serves as training and validation input for deck's win rate predicting machine learning algorithms.

4. Methodology

This recommendation model will be changing an initial deck to maximize its probability of winning a match against a random opponent. Generaly, this problem is expressed as:

$$\begin{aligned} &\max f(A) \\ s.t. \quad &|A| = D \\ &A \subset P \end{aligned}$$

Where D represents the size of a deck, in Hearthstone's case $D = 30$. P represents the pool of all cards, with cardinality $N := |P|$. And A is the deck we want to recommend.

There are to main issues. The first one is the combinatorial nature of the problem, the scalability is really low and it makes it very hard to create real time recommendations. The second problem is that the function f is unknown, leading us to the task of approximating it. In this study we address both as stated below.

To approximate f, this study trains neural networks using the datasets mentioned earlier. The architectures used for win-rate prediction are illustrated in Figure 2.

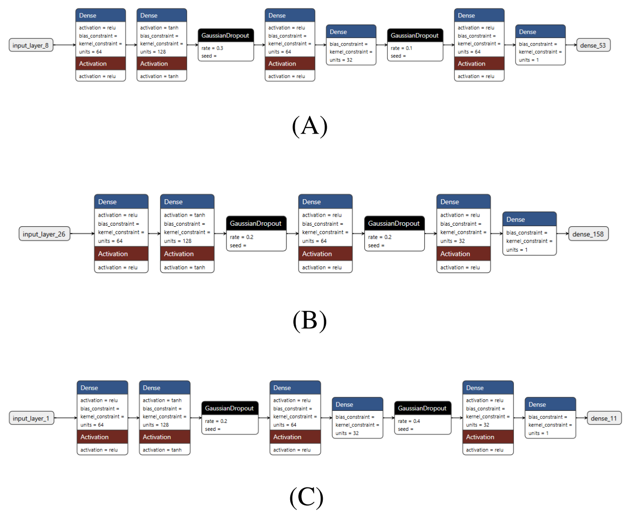


Figure 2. Here we present three slightly different architectures used for the win-rate prediction.

We refer to the models in Figure 5 (A), (V), (C) as NNv1, NNv2 and NNv3 respectively.

For deck modification, we propose the algorithm DeckRec based in the Qdeck algorithm (Chen et al., 2018) that is adapted to iteratively replace cards to maximize win-rate while minimizing computational costs.

Algorithm 1 DeckRec (A : deck; P : pool of cards)

```
for i in range(D) do
    Obtain the combination $(i, j)$ with $i \in A$ and $j \in P \setminus A$
    such that maximizes the win rate of $(A \setminus \{i\}) \cup \{j\}$
    $A \leftarrow (A \setminus \{i\}) \cup \{j\}$
end
return $A$
```

This is a greedy algorithm, that maximizes the win-rate at each iteration. It is assumed that the maximum of iterations needed is 30, in the worst case we have to modify the whole deck. A weakness of this algorithm is that it still considers too many combinations. Therefore, to make this search over a much smaller set of combinations we propose a mixed method.

First, we have to understand the BestSimilar(D, δ) of the deck D as the deck in our database with the highest rating between all the decks with δ or less cards of difference with the deck D. Now, we use this to create the following algorithm.

Algorithm 2 DeckRecWTemplate (A: deck; P : pool of cards; δ : difference, γ : fixed cards)

```
$A$ = BestSimilar(A,$\delta$)
for i in range(D - $\gamma$) do
    Obtain the combination $(i, j)$ with $i \in A$ and $j \in P \setminus A$
    such that maximizes the win rate of $(A \setminus \{i\}) \cup \{j\}$
    $A \leftarrow (A \setminus \{i\}) \cup \{j\}$
end
return $A$
```

This algorithm works under the idea that, since there are common cards between the original deck and a highly rated one, there most be some cards that work good together and therefore it may not be necessary to change the full deck in order to get a good win-rate.

Finally, we tested the proposed algorithms, DeckRec and DeckRec with Template, by comparing their performance in the win-rate function against the results obtained by Random, Most Popular, and Best Similar algorithms.

5. Results

5.1. Neural Networks

For the training of the neural networks we have the following results:

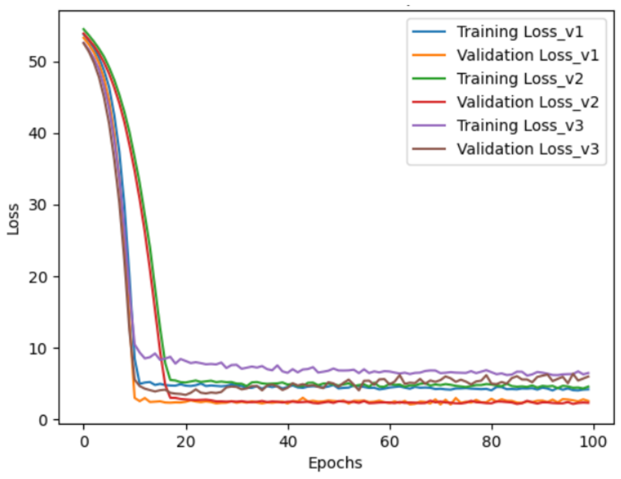


Figure 3. Loss function over epoch for all

To compare the different models we have tables with the statistics for three classes of win rate [WR]. The network separates the win rate into three classes: poor, mid and high. We can then identify the best model by analyzing which one minimizes the overlapping between them. When the statistical values are well separated by class we have a good model.

	Poor WR	Mid WR	High WR
Mean	52.41	52.46	53.49
SD	1.14	0.93	1.58
75%	53.12	52.96	53.91
Max	55.22	54.39	56.80

Table 1. Summary statistics for NNv1 model

	Poor WR	Mid WR	High WR
Mean	50.99	51.33	51.79
SD	1.60	1.25	1.01
75%	51.69	52.09	52.58
Max	53.92	53.41	53.47

Table 2. Summary statistics for NNv2 model

	Poor WR	Mid WR	High WR
Mean	50.99	51.33	51.79
SD	1.60	1.25	1.01
50%	51.31	51.55	51.83
75%	51.69	52.09	52.58
Max	53.92	53.41	53.47

Table 3. Summary statistics for NNv3 model

By examining the tables, we can conclude that the first and third models outperform the second. Specifically, the third model provides better separation of the three classes. However, the first model achives a better distinction between the mid and high classes. Since we are primarily interested in comparing decks with high win rates, we ultimately chose the first model.

5.2. Win rates

Although other metrics of the algorithms, such as rating, ranking, novelty and diversity, can be compared, we believe that analyzing the win rate increase and execution time is much more relevant. These metrics are directly aligned with the ones used in the proposal by Chen, et al.(Chen et al., 2018), which serves as the foundation for our approach.

First, we compare the performance of all the algorithms mentioned with the win-rate function. The methods are Random [R], Most Popular [MP], Best Similiar [BS], DeckRec and its variation DeckRecWTemplate [DeckRecT]. The table shows the increase factor of the win-rate of the generated deck compared to the initial deck.

	R	MP	BS	DeckRec	DeckRecT
Mean	1.13	0.98	0.95	1.51	1.51
SD	0.53	0.24	0.20	0.35	0.36
Min	0.59	0.69	0.62	1.06	1.05
Max	1.88	1.27	1.20	1.93	2.00

Table 4. Win-rate increase [%] by model using NNv3

	R	MP	BS	DeckRec	DeckRecT
Mean	1.09	0.97	0.99	1.27	1.27
SD	0.24	0.16	0.23	0.21	0.21
min	0.74	0.80	0.65	1.03	1.04
max	1.42	1.13	1.43	1.51	1.50

Table 5. Win-rate increase [%] by model using NNv1

As seen, the two proposed algorithms generate, on average, a much higher increase factor in the win-rate compared to the results achieved by Random, Most Popular, and Best Similar.

Since the proposed DeckRec with Template algorithm relies on the BS implementation with the variable parameter δ, we tested the behavior of the increase in the win-rate factor by varying δ, as shown in the table.

	25	**20**	**15**	**10**
Mean	1.04	1.02	0.98	1.00
SD	0.33	0.38	0.33	0.00

Table 6. Win rate increase [%] by δ in BS

It can be observed that with a very low δ, it is not possible to generate a new deck with Best Similar, as no better deck would be sufficiently similar. Additionally, it is evident that allowing decks with greater card variability is beneficial for increasing the win-rate.

Finally, we compare the two proposed algorithms against each other. To do this, we evaluated their performance in scenarios with the modified variable N, specifically for the values 30, 40, and 50. We compared their performance in terms of the win-rate achieved and the time, in seconds, taken for the recommendation.

Model	**35**	**40**	**50**
DeckRec	1.52	1.46	1.24
DeckRecT	1.68	1.70	1.46

Table 7. Win rate increase [%] by N

Model	**35**	**40**	**50**
QDeckRec	31.90	106.48	199.66
QDeckRecT	34.12	88.45	145.82

Table 8. Execution times [s] by N

It is clear that the proposed DeckRec with Template algorithm significantly outperforms the original DeckRec, as it consistently achieves better recommendations in terms of win-rate increase and also results in a significant reduction in time.

Finally, we compared the effectiveness of the algorithm originally proposed by Chen et al.(Chen et al., 2018), where the following key results were obtained.

Training	**Search Time**	**Func. Calls**	**Win Rate**
Wall Time	**Wall / CPU Time**		
1 day		20K	1.64
2 days	0.38 sec / 9.63 sec	41K	1.88
3 days		62K	1.93

Table 9. Training results for different time intervals.(Chen et al., 2018)

Considering that the training time is much longer than ours, the win-rates obtained are not directly comparable, but the search times they achieve are noteworthy.

Conclusions

The implementation of a recommendation system for collectible card games proved to be a challenging task. The solution should consider multiple factors, such as game mechanics, deck-building strategies and user preferences, while combining all possible decks.

The main achievement was the identification of two algorithms that consistently optimize decks, leading to an increase in their win-rate. Furthermore, these algorithms propose precise modifications, striking a balance between minimal changes and significant performance improvements. The ability to maintain this balance is crucial, as it ensures that the recommendations remain both practical and impactfull in real-game scenarios.

However, it is important to acknowledge the dependency of this study on the win-rate function, whose effectiveness is conditioned by the quality of the data used. Although, an official and sufficiently large dataset was considered, certain inherent issues may still affect the results.

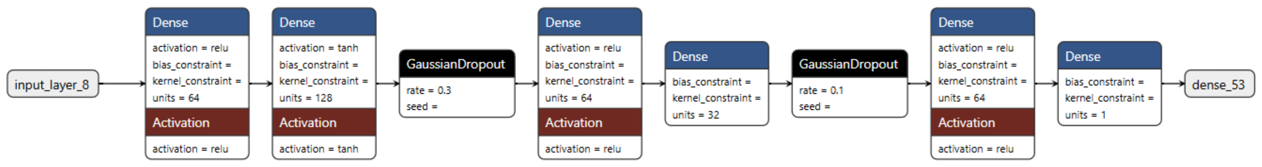
Our primary concern lies in the deck scoring system. Since decks are rated by up votes, once a user gives their vote to a deck, it is unlikely to be removed, even if the deck's performance deteriorates after a game update. This creates a potential issue, as outdated decks may mislead the recommendation process, introducing inaccuracies and biases during training. These factors could significantly impact the reliability of the system over time.

It is also important to highlight the time-related issues with the proposed algorithms. While the mixed approach of DeckRec with Template achieved significantly better performance in terms of time, it still fell short when compared to the results proposed by Chen et al. (Chen et al., 2018). This gap highlights the ongoing challenge of optimizing the trade-off between time efficiency and model performance, which remains a key area for future improvement.

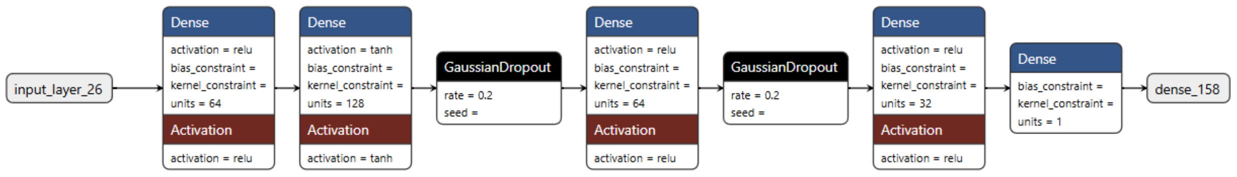
References

Chen, Z., Amato, C., Nguyen, T.-H. D., Cooper, S., Sun, Y., and El-Nasr, M. S. Q-deckrec: A fast deck recommendation system for collectible card games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, Maastricht, Netherlands, 2018. IEEE Press. doi: 10.1109/CIG.2018.8490446. URL `https://doi.org/10.1109/CIG.2018.8490446`.

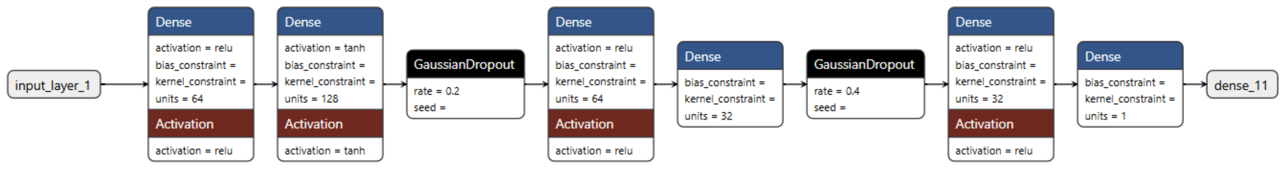
Mora, A. M., Tonda, A., Fernández-Ares, A. J., and García-Sánchez, P. Looking for archetypes: Applying game data mining to hearthstone decks. *Entertainment Computing*, 43:100498, 2022. doi: 10.1016/j.entcom.2022.100498. URL `https://www.sciencedirect.com/science/article/pii/S1875952122000222`.


input_layer_8
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
Dense
activation = tanh
bias_constraint =
kernel_constraint =
units = 128
Activation
activation = tanh
GaussianDropout
rate = 0.3
seed =
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
Dense
bias_constraint =
kernel_constraint =
units = 32
GaussianDropout
rate = 0.1
seed =
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
Dense
bias_constraint =
kernel_constraint =
units = 1
dense_53

(A)


input_layer_26
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
Dense
activation = tanh
bias_constraint =
kernel_constraint =
units = 128
Activation
activation = tanh
GaussianDropout
rate = 0.2
seed =
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
GaussianDropout
rate = 0.2
seed =
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 32
Activation
activation = relu
Dense
bias_constraint =
kernel_constraint =
units = 1
dense_158

(B)


input_layer_1
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
Dense
activation = tanh
bias_constraint =
kernel_constraint =
units = 128
Activation
activation = tanh
GaussianDropout
rate = 0.2
seed =
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 64
Activation
activation = relu
Dense
bias_constraint =
kernel_constraint =
units = 32
GaussianDropout
rate = 0.4
seed =
Dense
activation = relu
bias_constraint =
kernel_constraint =
units = 32
Activation
activation = relu
Dense
bias_constraint =
kernel_constraint =
units = 1
dense_11

(C)