Universiteit Leiden
The Netherlands

Opleiding Informatica

Predicting the outcome of the card game Klaverjas

Marnix Zwetsloot

Supervisors:
Jan N. van Rijn, Frank W. Takes, and Jonathan K. Vis.

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2020

Abstract

Klaverjas is a card game where two teams of two players compete to earn the most points. The goal of this thesis is to improve upon an existing machine learning model, for predicting the outcome of such a game. The model that was created to this end can much more efficiently generate Klaverjas games as data for a machine learning model alongside overall improved results. The way this was accomplished was by implementing an active learning algorithm. This algorithm repeatedly analyzes a large dataset of games and selectively generates the most effective Klaverjas games to add to the model. Using this method we measured an accuracy of 83.9%. This thesis provides both a machine learning model that can be used to make Klaverjas predictions and an algorithm that could possibly be used to improve this and many other machine learning models.

Contents

1 Introduction **1**

2 The Game Klaverjas **3**
2.1 The Rules of Klaverjas 3
2.2 Klaverjas in our Research 4

3 Related Work **5**

4 Background **6**
4.1 Alpha-beta search 6
4.2 Machine learning 7

5 Method **9**

6 Experiments **11**
6.1 Regression 11
6.2 Sampling bias 13
6.3 Active Learning 13
6.4 Auto-sklearn 17

7 Conclusions and Further Research **21**

References **22**

1 Introduction

The goal of this thesis was to predict the outcome of the game of Klaverjas. This was to an extent already done in the previous research [vRTV18]. Our goal was to improve upon these results.
Klaverjas is a card game where two teams of two players compete to gather the most points. After a number of rounds the team with the most points wins the match. Klaverjas is usually a game with hidden information; players can not see each others cards. The version of Klaverjas that will be used for this research has complete information about the game state. In previous work it was shown that optimizing for a single round with complete information has a search space manageable for an exhaustive search algorithm.
One of such algorithms is alpha-beta pruning which assumes optimal play for all players to compute a theoretical best move. It also reduces computation time significantly compared to minimax, which does the same but does not apply pruning, bringing average computation time to 1.5 minutes per game on average.
Unfortunately there are exceptions with a much longer computation time of around 45 minutes for a single game, which would be too slow for real-time predictions. In order to make faster predictions we can train a classifier using data generated using a machine learning algorithm. This method has an accuracy of about 88%[1]. Improving from there is challenging. Improvements could be made by increasing the amount of data the algorithm used to build the model. Theoretically adding all possible configurations to the model should give perfect performance.
Essentially, this comes down to creating a mapping between configuration and evaluation. Unfortunately this is not viable since there are far too many configurations. One of the things explored in this thesis is how efficient adding more data to the model is.
Other analyses done involve predicting a point score rather than determining only the winner, potentially providing even more detailed predictions. Something that could then be used to make multi-round predictions.
Classification is a machine learning technique that predicts a categorical attribute, in our case a Boolean value whether a given team won or lost. This data was obtained by calculating the outcome of many games using the alpha-beta algorithm. In general, a larger dataset means better performance, but alpha-beta will start taking a lot of CPU time. In order to use this time as effective as possible we want to selectively choose configurations to add that will give the most improvement. The algorithm proposed in this thesis is to detect properties where performance is low and specifically add configurations that contain these properties.
Our contributions are the following:

1. We perform a regression experiment, which shows out-of-the-box results of regression are insufficient.

2. An experiment analyzing sampling bias, in order to determine if overfitting is at some point an issue and finding a point of diminishing returns.

3. We propose an active testing methodology, this contains a program that will selectively add only the most efficient configurations and compares it to random sampling. Even though the

[1] While the earlier presented method obtained 88% accuracy, this was achieved on a non-uniform sample. When applying the earlier selected method on a uniform sample, it achieved a performance of 81,8%.

improvement was very small, overall we had a percentage wise increase in effectiveness, in this experiment going from 0,095% for random sampling to 0,349% for selective sampling.

4. Finally, an implementation of auto-sklearn.

The remainder of this thesis is organized as follows. In Section 2, we explain Klaverjas and the specific rule sets used for this research. In Section 3, we discus all used sources and work related. In Section 4 we discuss alpha-beta, machine learning and how they were used. In Section 5 the methods created to make the experiments possible are explained. In Section 6 the results of the experiments are presented and discussed. Finally in Section 7, we draw conclusions about the overall research.

2 The Game Klaverjas

Section 2.1 will explain the rule set of the version of klaverjas used. Section 2.2 will explain what part of this rule set we applied our experiments on.

2.1 The Rules of Klaverjas

There are many different versions of Klaverjas. The one used for this paper is: 'Rotterdams verplicht'. Klaverjas is always played with two teams of two players. All four players are assigned a direction, north, east, south and west. The teams are always north and south against east and west.

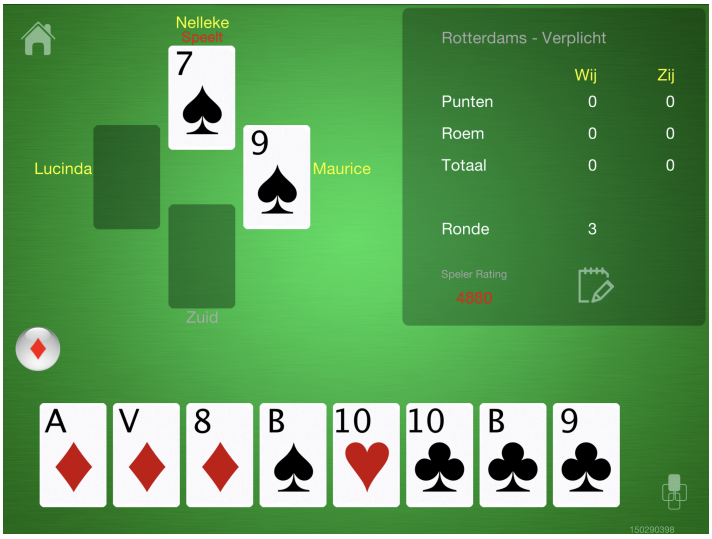

Figure 1: Image of a game of klaverjas, with imperfect information (image from: www.computeridee.nl/downloads/games/klaverjassen-lite-nooit-meer-mensen-tekort/).

The game uses 32 of the cards of a French deck of cards. The ace, king, queen, jack, 10, 9, 8 and 7 of the four suits. Every round each player receives 8 cards, so the whole deck is dealt at the start. Every game of Klaverjas consists of 16 games where all players deal 4 times. The person left of the dealer determines the trump suit.
Every trick each player starting with the person who decided the trump suit, will play one card, then the other players also play a card in clockwise order. Every game has 8 tricks. The person with the card of the highest rank wins that round and receives points for his team determined by all the cards played. The rank of a card is shown in Table 1 with the rank going from top to bottom (rank and point value are different). The person who wins a trick starts the next trick. A trump card always beats a non-trump card and a card of a different suit than the beginning card always loses. Although you always have to play the starting suit if you can and otherwise if you can play a trump card you also have to. The points earned in every round is always the points of all cards played added up as noted in Table 1.
After all players played all their cards, if the point total earned by the team starting the round is lower than that of the opposing team, they are 'nat', which means that all the points earned in this

non-Trump card	points	Trump card	points
ace	11	jack	20
10	10	9	14
king	4	ace	11
queen	3	10	10
jack	2	king	4
9	0	queen	3
8	0	8	0
7	0	7	0

Table 1: Value of cards in Klaverjas.

round go to the opposing team. There is also the possibility to gain bonus points called meld, the following ways to earn extra points exists:

- If a team won every single round it is called 'pit', which means that this team gains an additional 100 points.
- The last round is always worth 10 points.
- 3 cards of the same suit in a row are played is worth 20 meld points.
- 4 cards of the same suit in a row are played is worth 50 meld points.
- 4 cards of the same face are played is worth 100 meld points.
- King and queen of the trump suit are played is worth 20 meld. this does stack with 4 card or 3 card

Note that if one of the meld points is a subset of a previous one you only gain the highest one.
At the end of the game, the points of all individual tricks is summed up and the team with the most points wins the game.

2.2 Klaverjas in our Research

For this research we have limited what parts of the game we looked at. Most importantly we only analysed Klaverjas without hidden information, this means that all players show their cards to each other. The reason for this is that this way we have complete information about the starting position of the game, this has very important implications for the experiments done.
We have only looked at a single round, so from every player having 8 cards to every player having played all their cards, trying to get a score as high as possible on this single round.
We only worked with games where we play the first card, although not choose the trump suit. Which has the effect that the choice of trump suit is built into the starting position of the game, which reduces the size of the search tree, without losing generality.

3 Related Work

This paper is build upon previous research: [vRTV18]. The research consists of the application of the alpha-beta algorithm for the specific version of Klaverjas as mentioned in Section 2. The source for this exact rule set can be found in [Par08].
In order to apply the active learning algorithm described in this thesis it is necessary to have an algorithm to compare to in place, it is important that this algorithm is either perfect or very accurate an advised algorithm would be any that is based on traversing the game tree, these are for a number of card games already created, see for example [VDHUVR02], [vRTV16].
The algorithm used for this project is alpha-beta you can be read more about this algorithm in chapter 4. For a more general analysis of the general alpha-beta algorithm see [KM75], [Pea82].
Finally the machine learning work explained in chapter 5 was also in place. For a very successful implementation of machine learning in games see AlphaGo [SHM+16], [SSS+17].
The experiment described in section 6.4 uses the auto-sklearn package [FKE+15] which is an extension of the well known Python package scikit-learn [PVG+11]. auto-sklearn is an algorithm that will automatically apply a number of optimization algorithms from scikit-learn to your model in order to find the best one.

4 Background

In Section 4.1 we will discuss the alpha-beta algorithm used throughout this thesis. In Section 4.2 we will discuss the original machine learning models, used as a starting point for this thesis.

4.1 Alpha-beta search

The alpha-beta algorithm is applied on Klaverjas without hidden information. Without this property you would have to take guesses on what cards the other players would have. Given two different distributions of cards between the other players, it is possible you need to apply two different strategies, Proving that without this condition applying the perfect strategy is not always possible. Using Klaverjas without hidden information we no longer have any elements of chance, making it possible to find if there is a guaranteed winning game progression. Alpha-beta does this by assuming 'perfect play' from all players and does basically a brute-force search by working out the decision tree of the game.
Alpha-beta does some pruning of subtrees in case it can guarantee it will never provide a better result. Finally the alpha-beta algorithm will return who will win the game given perfect play from all players, where suboptimal play from opponents will only give an opportunity to score even better.
Unfortunately, there is a downside to alpha-beta, the game tree of Klaverjas even in the current configuration is very large. In the worst-case scenario every trick ever player can in the first trick choose to play any one of eight cards, next trick every player can choose to play any 1 in 7 up to the last card where there are no choices anymore. This means this game tree has $8^4 \cdot 7^4 \cdot 6^4 \cdot 5^4 \cdot 4^4 \cdot 3^4 \cdot 2^4 \cdot 1^4 = 1.64 \cdot 10^{18}$ leaves, and is $(32 - 4) + 1 = 29$ plies deep. -4 Since the last trick you do not have a choice but play your last card and $+1$ for the first node that keeps the 8 subtrees together.
Luckily for most games, alpha-beta doesn't take too much time traversing the tree since the tree is not as large as mentioned above and it can prune subtrees. It is not as large because most of the time players do not have the maximum number of cards they can play as used in the example above. For example, if you have only six cards left in your hand but only one of the color played you have to play it. In this case rather than six options you have only one option. Pruning can be done if the current subtree the algorithm is looking at can no longer result in a better outcome than a previously found outcome.
In most cases calculating one game takes roughly 1.5 minutes. However there are a couple configurations where it has to traverse a deep search space and in these cases the computation time can get up to 45 minutes. And a single configuration that even took about 4 days.

4.2 Machine learning

The second part in the original paper [vRTV18] was an introduced machine learning algorithm. Alpha-beta takes in some cases a significant amount of time to compute the outcome. Machine learning was introduced in an attempt to keep similar results with less computation time.
In order to construct most of the existing machine learning models you need data to train and test on. This was not initially available so the existing alpha-beta algorithm was used in order to generate data for this. The data used would be represented as a distribution of all cards to the 4 players, or in other words an initial game state, along with the score alpha-beta assigns this game. Instead of generating data randomly, 981,541 equivalence classes were manually constructed, in an attempt to more evenly represent the diversity of possible games than a random sample would. One game for each of these classes was then generated.
The specific algorithms used were decision trees and random forests. Figure 2 shows an example of how a decision tree looks, though created from a small subset of 500 games in order to keep the model small.

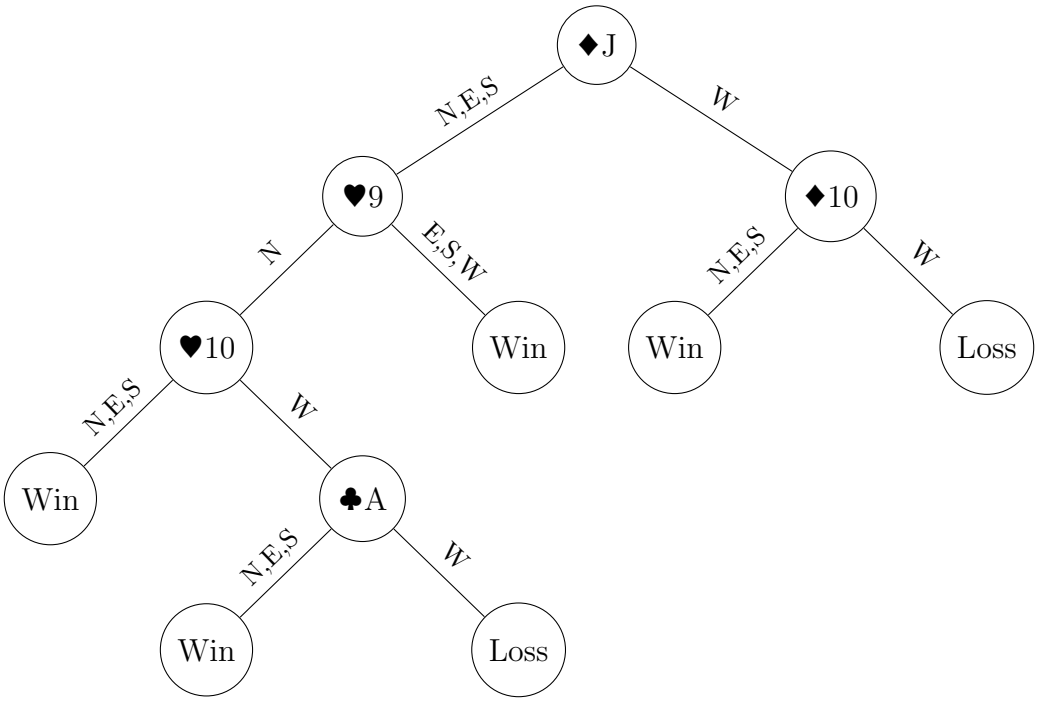

Figure 2: Example Decision Tree Classifier.

In Figure 2, the nodes show cards and the edges show players. Once a prediction about a certain games needs to be made you can determine what path to follow based on the possession of certain cards. For example if West has the jack of diamonds you go to the right. Finally in the leaves you see the prediction for the outcome of the game.
Besides card possession as predictor, also handcrafted rules were used as a separate experiment. Handcrafted rules are rules intuitively useful in the game, for example: How many aces does your team have?

Feature	Decision Tree	Random Forest
Card ownership	82.44%	88.16%
Crafted rules	88.02%	91.98%

Table 2: Accuracy machine learning from paper [vRTV18].

The second algorithm used for machine learning was random forests. The random forest algorithm is a combination of multiple decision trees which get generated with different random states. All decision trees give a prediction, and then the most predicted result is selected by the algorithm as the final result. For this project random forest 64 was used, meaning majority vote of 64 decision trees. Table 2 shows all results.

5 Method

In this section we describe the main contribution of this work the active learning approach that aims to improve the dataset to create better models for predicting Klaverjas, specifically by finding better games to add to a the dataset. The following method was used to generate these games:

1. Generate a random set of games
2. Predict outcome with current model
3. Compare these with actual results
4. Construct new decision tree that predicts what games are going to get an incorrect prediction
5. Analyse this tree for properties that indicate a game will get an incorrect prediction
6. Generate new games using the derived properties
7. Calculate actual results for new games using Alpha-Beta
8. Add new games to current model
9. Repeat from step 1 (either until performance plateaus or until time constraints)

These 9 steps are further elaborated below:

step 1 In order to generate random games we generate a random number between 0 and 99,561,092,450,391,000 using the `numpy.random.randint` function. This value gets converted into a game state using an algorithm provided in [vRTV18]. Then for all these games we run alpha-beta (provided by the same paper) in order to find the actual outcome and append it to the game.

step 2 We use `sklearn.ensemble.RandomForestClassifier` [PVG+11] to create a random forest 64 classifier to predict the outcome for the games generated in step 1.

step 3 We convert the actual outcome provided by alpha-beta from step 1 to a Boolean value representing a win or loss. Then we compare this with the predicted outcome of step 2, and get a new Boolean value for the prediction of each game, where true represents a game was correctly classified by the machine learning algorithm and false represents a game was incorrectly classified.

step 4 We use `sklearn.tree.DecisionTreeClassifier` [PVG+11] to train a decision tree classifier that predicts the values of step 3. Giving us a decision tree that can give predictions of whether a given game will be correctly evaluated by the original model.

step 5 Now we traverse the decision tree classifier from step 4 and find all rules that were part of a path that led to incorrect predictions. For each unique path a set of all these rules is stored.

step 6 New games are created from scratch that have all rules of one of the paths of step 5 enforced, with the remainder of data randomized. For each rule at least one game is created, however a maximum value can be provided to say roughly how many games you want to create. This number will be divided by the number of unique paths found so every unique path will have the same number of games created, closest to the chosen maximum.

Step 7 The new games from step 6 get their actual outcome added using alpha-beta.

step 8 Set of games from step 7 is added to the original model.

step 9 These steps can be repeated however often you want, larger steps can take better advantage of multithreading, but smaller steps can theoretically respond faster to the new dataset, possibly providing higher efficiency.

6 Experiments

Sections 6.1 through Section 6.4 discuss the different experiments done for this thesis. Experiment 1 is a regression implementation of the machine learning model. Experiment 2 is an analysis of the size of the dataset used for the training of the machine learning model. Experiment 3 is a performance analysis of the implementation of the algorithm discussed in Section 5. Finally Experiment 5 is an implementation of the auto-sklearn package along with the algorithm from Section 5.

6.1 Regression

We wanted to improve the existing machine learning model from [vRTV18] by making more detailed predictions about a given game by predicting a score instead of a Boolean value. Alpha-beta originally already returned a score, which for previous experiments was then turned into a Boolean value. In order to create a regression model the score value was kept instead.
Next the models needed to be replaced, since decision tree classifier and random forests are as the name suggests classifiers. The most straightforward models to replace these are decision tree regressor and random forest regressor. These work very similar to their classifier variants with the key difference that instead of having a set number of classes as prediction the prediction can be any natural number.
Figure 3 shows how well this most straightforward implementation for regression performed. The decision tree regressor used is trained on the dataset of about 500,000, and tested on 1,000 games. The reason for only 1000 games is that Figure 3 will become overpopulated if more data would be plotted. Predicted result is what Machine Learning predicted and actual result is was alpha-beta returned. The line is used as an indicator where you would want the data points.

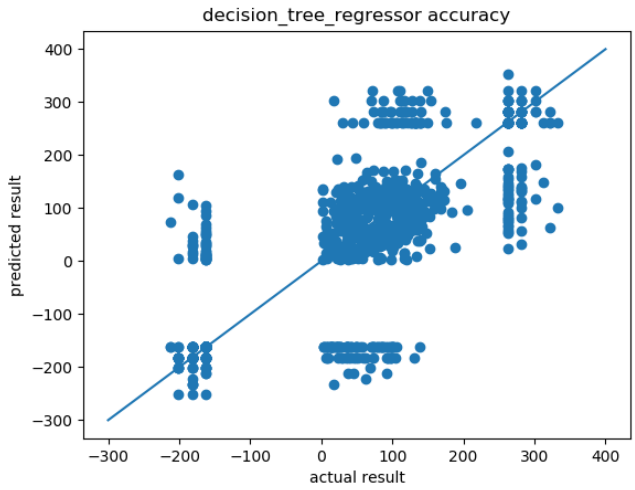

Figure 3: Accuracy of a decision tree regressor.

The most notable result here is that the data points seem to be clustered in 7 groups. These are very well explainable using the rules of the game.

1. center: are predicted wins which were indeed wins
2. bottom-left: are predicted losses which were indeed a loss
3. middle-left: are predicted wins which were actually a loss
4. lower-middle: are predicted losses which were actually a win
5. upper-middle: are predicted wins with 'pit', which actually didn't score pit
6. top-right: are predicted wins with 'pit' which indeed did score 'pit'
7. middle-right: are predicted wins without 'pit' which actually did score 'Pit'.
8. top-left: notice there are no data points here, would be predicted wins with pit but actually a loss.
9. bottom-right: notice here are no data points either, would be predicted losses which were actually a win with 'pit'.

Only cluster 3 and 4 are predictions that did not predict winner correctly, which makes sense since this should be only about 17.5% of the data. (See card ownership decision tree classifier performance in Table 2 of Section 4.2.) But then also cluster 5 and 6 have significant errors since the difference of 'pit' is an immediate error of 100 points plus or minus an additional error. And then even in the correct clusters the exact point values can still be off by quite a lot.
Figure 4 shows the size of these errors, by introducing a tolerance which a point is allowed to be off from its target, if within this tolerance the point is evaluated as correct and outside of it incorrect. Accuracy now represents percentage of correctly classified games given this tolerance. Here, the inadequate performance for regression becomes more clear. If we take 90% accuracy as acceptable we have a tolerance of about 200.
We can explain the bend in the graph around accuracy of 0.75 the same way we did in Figure 3. Performance per tolerance slows down because the lower negative scores do not exist. then after about 100 tolerance of slower increase in accuracy we start including the barely lost games in the tolerance and the slope increases again. Then it slopes down again when only outliers can get included in the tolerance.

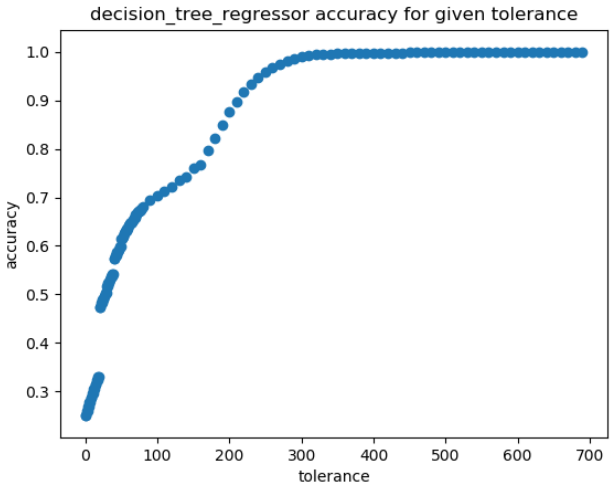

Figure 4: Accuracy of a decision tree regressor within a tolerance.

6.2 Sampling bias

Since the regression problem did not show potential, we from here on aim to solve the classification problem, this way evaluation is simpler. The size of the dataset is one of the aspects that can be optimized to increase performance. The dataset is a set of games of Klaverjas with their results. More data is not always better since decision trees are definitely prone to overfitting.
Figure 5 shows you can see the size of the dataset against the performance, the final sets are the equivalence class set and a completely random generated set of the same size. The equivalence class set is determined by a set of classes constructed to contain all different properties based on the rules of Klaverjas. Both starting with one game and adding games this training set. All results are an average over 10 iterations where the same dataset was used every time. The 10 iterations had 10 independent testset of constant size to test on.

Figure 5 shows that with confidence can be concluded that over fitting has not taken place yet. Both datasets show a learning curve (note that the scale is logarithmic). The most likely reason for this is that as a game is added it is more likely already classified correctly adding nothing new to the model. Interestingly the equivalence classes do not seem to give an improved result over a completely random set. A possible reason for this is that the equivalence classes include disproportionately many of the rare cases, that in completely random sampling will almost never occur. which makes the model over accommodate for accurate results on these cases while decreasing performance on the more common cases.

6.3 Active Learning

The trainnig set used for this experiment was chosen based on results of the second experiment. A program is created that will automatically perform the entire process shown in Figure 6 that only needs to be provided with parameters.

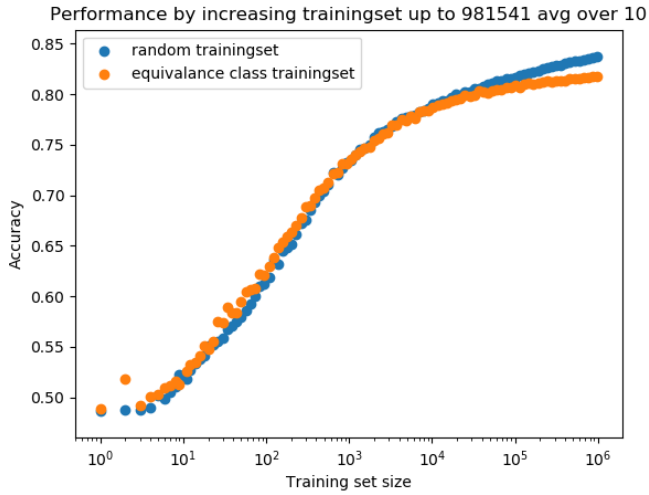

Figure 5: accuracy random forest based on size of the training set.

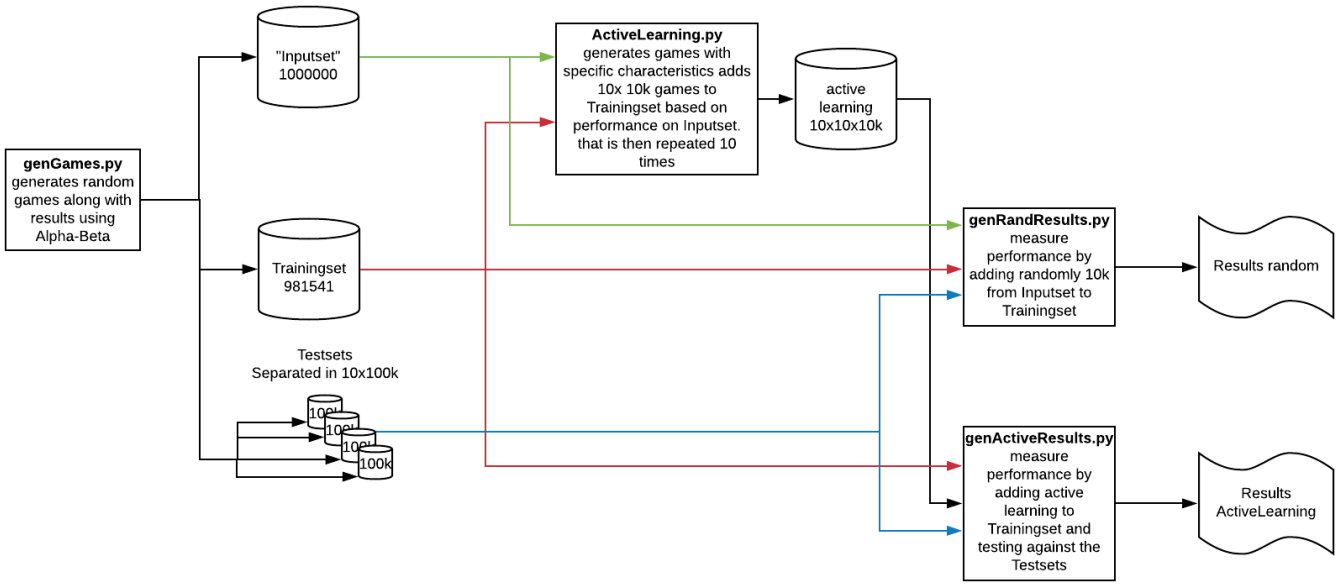

Figure 6: Dataflow for active learning experiment.

The cylinders represent datasets of games of Klaverjas. An entry of a game of Klaverjas is the ownership for every card and a Boolean value provided by alpha-beta representing the correct prediction for this game. Each value on the dataset represents how many games were in this dataset for this specific experiment.

- Inputset is used for two different purposes, it is used as an intermediate evaluation to generate a different dataset in ActiveLearning and is directly added to the original set in genRandResults.
- Trainingset is the base set from which both random and active learning start with.
- Testsets are used to evaluate performance of both random and active learning, the same exact set is used for both in all iterations. For a new run a new test set is used.

The four squares represent Python programs written for the purpose of this experiment.

- `genGames.py` randomly samples games of Klaverjas and their results will be calculated by the preexisting alpha-beta algorithm. These are dumped to all CSV files mentioned above.
- `ActiveLearning.py` executes the algorithm described above, using Trainingset as the starting point and using Inputset data for step 1 in the algorithm. Then dumps the new games from step 6 to a new CSV file.
- `genActiveResults.py` evaluates the performance of the results of active learning by adding them to the Trainingset and evaluating against random test sets sampled from Testset.
- `genRandResults.py` evaluates the performance of a equivalent amount of randomly sampled games from input set add to Trainingset and evaluating against the exact same sets sampled from Testset.

Finally, the wave papers are text files with lists of results, which if configured to do so will automatically also be saved as a plot.
The arrows represent where the data goes in the graph so all raw data gets initially generated by the genGames file. ActiveLearning uses the data from input set as an evaluation set explained in Section 5, then creates a new set from scratch. The genResults files determine performance of both random and active learning. Note that the same data from inputset is used to generate random results, with the important distinction that genRandResults uses this data directly to add to the training set to evaluate performance, where genActiveResults only indirectly uses this data by using it for the initial evaluation in ActiveLearning. Both sets uses the exact same test sets for the same steps.
We want to assess whether the active learning approach would improve classification performance over enhancing the dataset with random samples. The active learning approach is the main contribution of this work. Table 3 shows the results from the experiment described above. The dataset column represents the number of games in the training set. The first row contains the exact same randomly sampled set of games for both the random and active columns. The games added to the random column were also randomly sampled and the games in the active column were determined using the method described in Section 5. The random and active columns represent the

dataset	random	active
981541	0.836151	0.836151
991541	0.836489	**0.836997**
1001541	0.836499	0.837531
1011541	0.836331	0.837654
1021541	0.836807	0.838315
1031541	0.836736	0.838237
1041541	0.836847	0.838651
1051541	0.836251	0.838328
1061541	0.837025	0.838559
1071541	0.836769	0.838794
1081541	**0.836942**	**0.839068**

Table 3: Final results of the active learning experiment.

accuracy scored on the same independent testsets. Finally the accuracy scores in both random and active columns are averages over 10 runs of the same experiment.
Note that the experiments of Sections 6.2 and 6.3 use identical training set though the test set for Section 6.2 was $10 \cdot 10^4$ and here of size $10 \cdot 10^5$ so the end point of the dataset analysis is not necessarily the same as the first datapoint in this experiment. Table 3 and Figure 7 show the same results but plotted with their standard deviation.

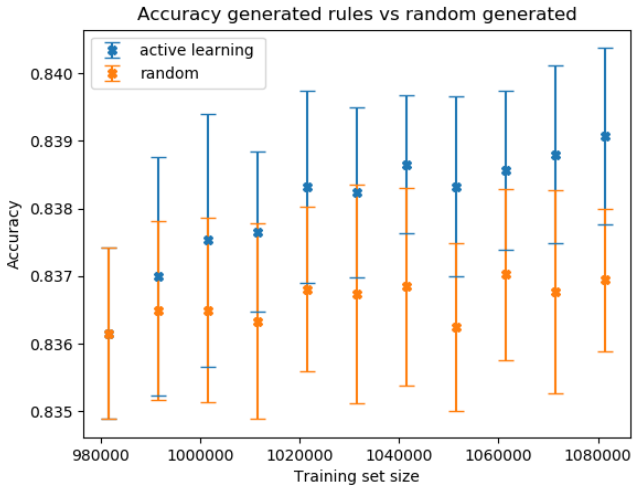

Figure 7: Final results of the active learning experiment plotted with standard deviation.

The following conclusions are drawn from the results:

- Active learning seems to perform slightly better than random. (although significantly if considered proportionally, roughly 3.7× effectiveness over the 10 iterations)

- Neither active learning nor random seem to be reaching a plateau on performance yet, which is not surprising since from 980,000 to 1,080,000 is only 10% increase in data.
- The first iteration of active learning seems considerably more effective than consecutive runs. A possible reason for this may be that the first model still has all missing crucial games for the 1 million where hopefully as many as possible can be covered for. Where for the next 9 iterations there is increasingly less 'holes' in the systems to patch.

Figures 8 and 9 show decision trees that are part of the model. Note that these are not the random forest that give the previously mentioned results but rather the models that were purely created and evaluated in order to determine what data to generate next. Also for the sake of readability the model is only shown up to depth 5 while the original trees had depth 7. Though depth was variable so not necessarily all trees in this experiment had depth 7.

6.4 Auto-sklearn

Scikit-learn was used in order to apply the original models like decision tree classifier, and random forest 64. Though these original models where chosen somewhat arbitrarily. In order to better evaluate the functionality of the active learning process and possibly improve performance even more we attempted to apply auto-sklearn which tries to automatically find the best classifier (or regressor) along with its optimal parameters. We repeated the experiment described in active learning again however we replaced The random forest 64 with auto-sklearn. Also the results in Figure 10 only shows an average over 6 runs instead of 10.

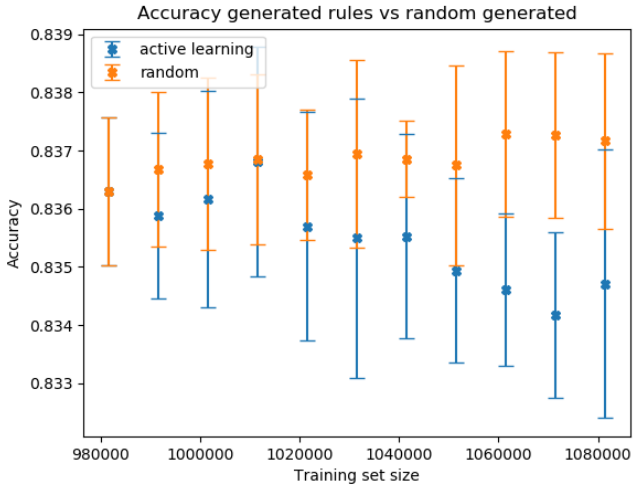

Figure 10: Results of the auto-sklearn experiment.

As we can see, performance is significantly lower than the previous experiment for both random and active learn. Which is strange since auto-sklearn should perform at least as well as Random Forest 64 since auto-sklearn should also attempt to use random forest 64. We suspect that the algorithm simply does not have enough time to find any decent model, since auto-sklearn is based

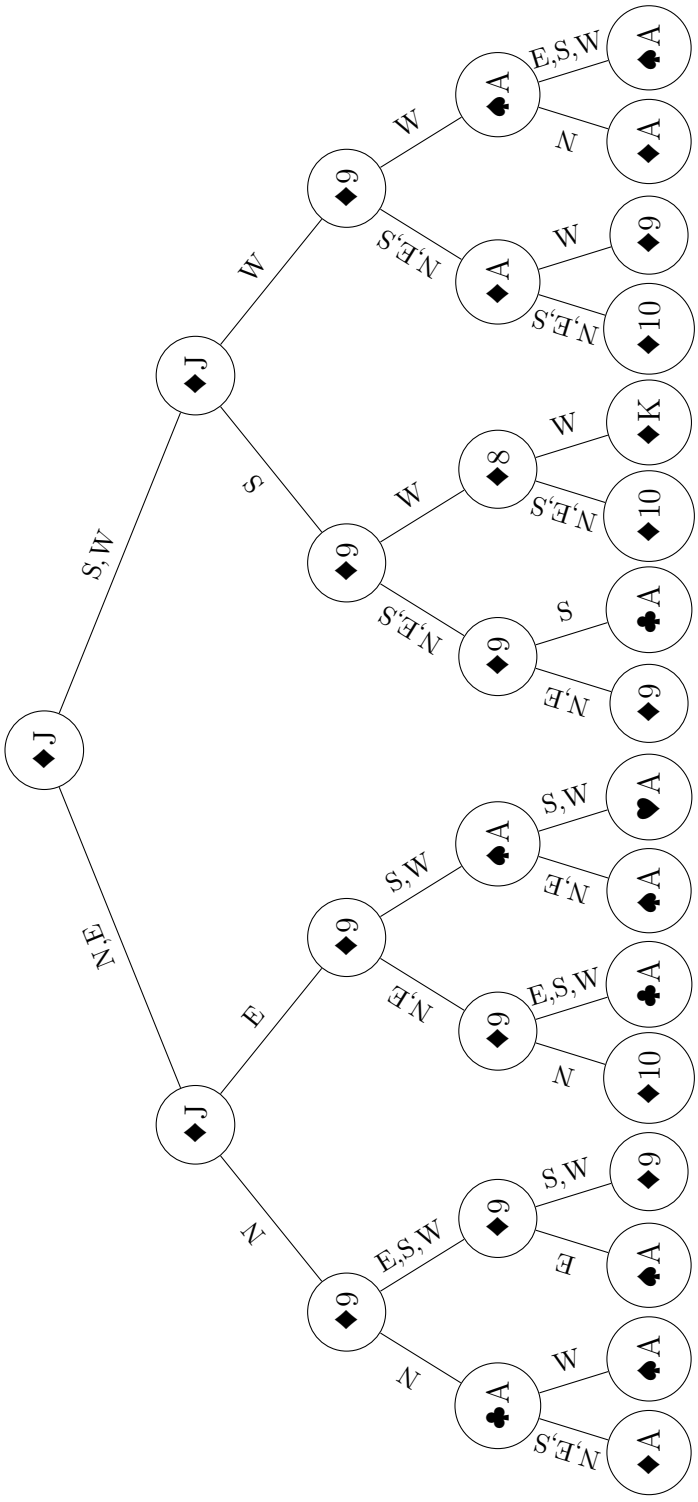

Figure 8: Decision tree of iteration 1 run 1.

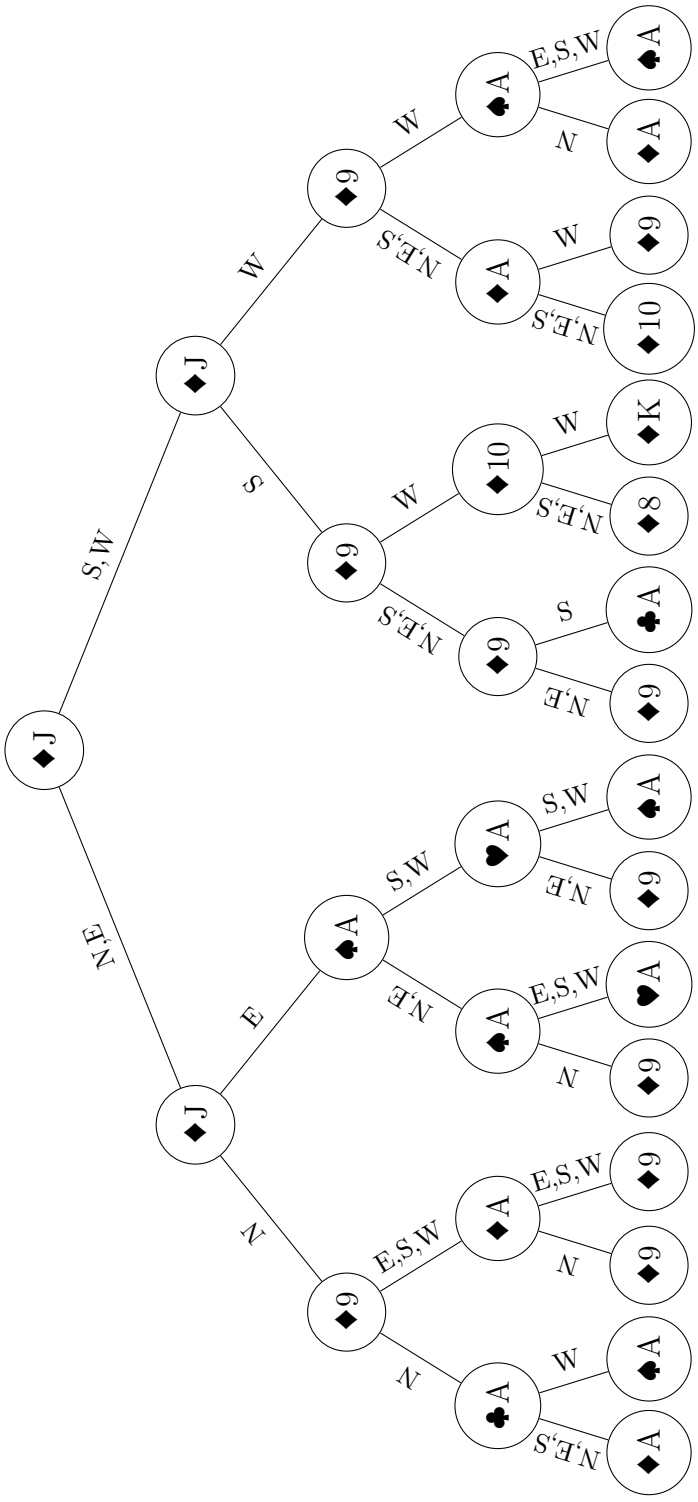

Figure 9: Decision tree of iteration 10 run 1.

on time limits for how long it can keep looking for a better model and the model we try to train is rather large and the time provided was limited.

7 Conclusions and Further Research

In this thesis, we have explored ways to classify Klaverjas by win or loss. More specifically, we improved upon the results from [vRTV18] by employing an active learning model, that actively determines which instances of Klaverjas are hard to solve and adding these to the training set.
In order for this strategy to be possible it is necessary that there is an other system in place to evaluate performance, in our case the alpha-beta algorithm. This makes this method not always useful since the use of this other model could be preferred over this method unless some other circumstances keep it from being useful.
In our case alpha-beta has very high accuracy (perfect by definition) however, computation time of this algorithm is so long that training a faster model may be a convenient solution. As the initial training of the model will take a long time seeing it uses the 'slow' algorithm but once the model is created, slightly less accurate predictions can be made much faster.
Using active learning we can achieve a measurable improvement over random data generation however, with the experiments done until now not enough for a meaningful difference. An interesting experiment to run would be finding the plateau of performance for both random and active learning and hopefully observe that the plateau for active learning would be higher. Unfortunately, the reason this was not included in this thesis is the CPU run-time constraints.
An other experiment that would avoid having to tackle the issue of CPU time constraints would be starting with a relatively small set and start active learning, since random is still much more efficient it would be interesting to see if active learning can provide even better efficiency at this dataset size. Applying this method to other games or other prediction problems would be interesting in order to further evaluate the efficacy of this method in general.
Finally, building upon the performance on Klaverjas and aiming for better performance for the game of Klaverjas, this can be done more concretely by:

- Experimenting with neural networks or other algorithms instead of the current classification algorithms, this should be fairly easy to implement since the algorithm is only used as a black box for predicting results in the current system.
- More in-depth/manual hyperparameter optimizations, for example applying auto scikit-learn with more CPU time available.
- Eliminating the open-card klaverjas constraint
- Converting from classification to regression in order to make multi-round decisions.
- Applying the model to different Klaverjas rule sets.
- Creating a real-time player.

The active learning method described in this paper seems worthwhile to further explore in many different ways, with the possibility to improve performance of efficiency of all sorts of optimization problems, already showing usefulness in this application in Klaverjas.

References

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[KM75] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Par08] David Parlett. The penguin book of card games. Penguin UK, 2008.

[Pea82] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, 1982.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine Learning research*, 12:2825–2830, 2011.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[VDHUVR02] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.

[vRTV16] Jan N van Rijn, Frank W Takes, and Jonathan K Vis. The complexity of Rummikub problems. *arXiv preprint arXiv:1604.07553*, 2016.

[vRTV18] Jan N van Rijn, Frank W Takes, and Jonathan K Vis. Computing and predicting winning hands in the trick-taking game of Klaverjas. In *Benelux Conference on Artificial Intelligence*, pages 106–120. Springer, 2018.