Mastering the game of Schnapsen with ensemble voting

Abstract

This research paper presents and analyzes the performance of several intelligent agents in playing the Austrian card-game Schnapsen, including a rule-based agent, several classic machine learning agents and a voting-based ensemble agent. It is shown that the ensemble agent consistently and reliably outperforms its opponents, and indeed comes close to the theoretical benchmark of optimal performance. This paper further seeks to develop an understanding of these impressing results by engaging background information and theoretical considerations.

Keywords: machine learning, supervised learning, intelligent agents, ensemble methods, Schnapsen

Ⅰ. Introduction

Games of imperfect information, in which (part of) the information position is private to the individual players, have long been an area of interest in the field of artificial intelligence. From a programming perspective, the main difference between games of perfect and games of imperfect information is that for the latter no optimal value function exists to determine the outcome of the game. (Silver et al. 2016, 484)

Approaches to games of imperfect information therefore rely on suboptimal value functions, in the form of heuristics. Such heuristics can be derived in different ways, such as simple rule-based heuristics, Perfect Information Monte Carlo sampling (PIMS; in which the perfect information is based on randomized states of the game), reinforcement learning (in which the heuristic is derived by a 'trial and error' approach), and the application of supervised learning (in which the heuristic is derived from a set of features applied to some training data).

Different methods have different strengths and weaknesses, making it challenging to achieve a near-optimal value function. A promising approach to tackle this challenge is the ensemble approach. Kim & Cho (2008) describe the ensemble approach as "a method to combine multiple decision models expecting synergism to increase the performance of systems" (p. 212). Using an ensemble enables the integration of different approaches, with different value functions, by combining their output in some sort of voting system or fusion method.

In this paper we apply an ensemble approach to the game of Schnapsen. Schnapsen is a variant of the card game SixtySix, which uses only a total of 20 playing cards (five cards from each suit). The game is played over two phases, the first of which has imperfect information, and the second has perfect information. We test the effectiveness of this approach by letting an ensemble compete against each of its constituting members. The ensemble itself consists of four different machine learning bots (with the same parameters and feature set, but different training data sets consisting of 20,000 games each), a PIMS bot, and a simple rule-based bot. To complement these subjective performance tests, the ensemble will also be evaluated by comparing its performance to a randomly playing opponent (as an objective lower performance bound) and a theoretically optimal opponent (as an objective upper performance bound).

Our analysis shows that the ensemble bot we used in our research significantly outperforms its constituent bots, providing evidence for the potential of ensemble approaches to gaming. Furthermore, even though the six bots that make up the ensemble individually are relatively simple approaches, when benchmarked against a theoretically optimal player, demonstrably achieves a near-optimal performance.

The rest of this paper is organized as follows. Section II provides background on games as a research area for artificial intelligence, the gaming environment of Schnapsen, and some relevant philosophical notions. Section III provides an overview of our methodology to measure and evaluate the performance of the ensemble and its constituent

models. Section IV details the design choices regarding the ensemble and relevant theoretical considerations. Section V describes the results and analysis. Section VI outlines our findings. Finally, the conclusion and suggestions for further research are given in Section VII.

Ⅱ. Background Information

2.1 Why Games?

Within the AI research community and beyond, games are widely viewed as a means to benchmark the performance and overall progress of AI technologies. This is mainly due to the fact that while games tend to resemble an abstraction of the real world (with clear rules and possible actions), they are also highly complex and engaging.

Many games are NP-hard, which necessitates the engineering of systems that are "truly" intelligent, in that they cannot just simply search the state-space in a mechanistic way. (Yannakakis and Togelius 2018, 15-16) Unlike regular search problems, games are generally adversarial. Thus, systems designed to play and win games cannot solely rely on good heuristics but need to devise and utilize strategies which surpass those of the opponent. Another constraint on games in contrast mechanistic search methods is the presence of time limits, which force the intelligent agent to deal with approximate solutions and act under conditions of uncertainty.

As games allow the researcher to manipulate the task environment, for instance by changing it from static to dynamic. Games thus offer targeted ways to test the capability of agents to learn and adapt, whis is a much more complicated undertaking in the chaos of the real world. Perhaps most importantly, games are fun and entertaining. They easily capture the fascination of a broader audience, as can be witnessed by the attention that has been given to Deep Blue and AlphaGo, who have notoriously beaten the human masters in chess and Go.

2.2 Understanding the environment

In order to develop an understanding of what kind of agent can perform well in the game of Schnapsen, it is useful to comprehend the environment first. This will allow the researcher to know what kind of circumstances the intelligent agent will need to adapt itself to, and thus design it accordingly. Schnapsen provides a particularly interesting environment for developing and testing intelligent agents. For one, the game is split into two different phases which require different strategic consideration. The agent faces an environment of imperfect information in phase 1, given that both the opponent's deck and the stash are unknown to the agent. In phase 2 the game shifts to a perfect information environment.

Finding an optimal solution in phase 2 is relatively straightforward but designing and engineering an intelligent agent who performs well in phase 1 requires an extensive effort on behalf of the AI researcher. As there is no element of chance involved after the shuffling

of the cards, the Schnapsen environment can be categorized as deterministic. This is favorable to a stochastic environment, which would introduce an additional layer of uncertainty. Schnapsen is also static, rather than dynamic, as the environment retains in general form after the cards are dealt and does not change in any unpredictable ways. This fact also generally makes life easier for the intelligent agent. Schnapsen is also a discrete environment, as it only deals with distinct card values and scores. Furthermore, Schnapsen is obviously a multi-agent (specifically: two agent), adversarial environment. Finally, the computational complexity of Schnapsen is quite high. With a state space of around 10^{20}, it would take around 10 exabytes to store all possible state representations in computer memory. (Wisser 2015, 125) As this is well beyond the capabilities of even the finest computational machines in 2020, it becomes clear that the game of Schnapsen is not be won by merely a mechanistic search approach, but that it indeed requires the development of genuinely intelligent methods.

Given all these factors, the realization arises that the greatest challenge in the design of an intelligent system for winning Schnapsen is to overcome the uncertainty (and the complexity that arises out of it) in phase 1. If the intelligent agent can successfully master this challenge, taking the game home in phase 2 should only be a matter of routine (given that proven algorithms for such environments already exist, e.g. minimax). The focus of this research paper will thus lie on developing and understanding an agent that can perform successfully in phase 1 of Schnapsen.

2.3 Philosophical notions

The prospects of Artificial Intelligence, and the influence the developments in this field may have on human civilization, are among the most heated philosophical discussions in contemporary times. This comes as no surprise, as the rise of AI challenges many of the capabilities that have for millennia been viewed as a unique feature of humankind. Measuring our ability to think strategically and outwit one another in games like chess, Go or Schnapsen is certainly one of them. Indeed, the very name homo sapiens, man the wise, gives away the strong identification we as human beings have with our degree of intelligence. (Russell and Norvig 2016, 1) Every AI researcher should therefore carefully consider the deeper implications of his work and realize that what may for one merely be a research objective might for another come nothing short of playing god.

Given that the larger objective of this research project is to engineer and understand an artificial intelligent system, it is useful to begin with a definition of intelligence that is agreeable with the ethical framework of the researcher. In order to avoid any theological implications, this definition should be broad enough and applicable to non-human agents. It should also be pragmatic, in the sense that it allows the researcher to categorize a system as intelligent (or not). A suitable definition was found in the work of MIT-professor Max Tegmark, who defines intelligence simply as “the ability to achieve complex goals”. (Tegmark 2018, 80) In the case of this research project, the ability to play Schnapsen can be viewed as such a complex goal.

Ⅲ. Methodology

The philosophical notions mentioned above lead the research to an important question. *How can one measure the degree of intelligence of any given system?* In the case of an intelligent agent playing Schnapsen, it is highly useful to take a pragmatic approach to this question. In other words, if it can be concluded that the agent plays Schnapsen reasonably well, it should also be concluded that the agent possesses at least some degree of intelligence.

Adopting this stance, and the narrow definition of intelligence it implies, allows the researcher to use statistical methods in order to measure the performance of the intelligent system, which is an important step in grounding the insights gained in the process of designing the agents.

3.1 Measuring performance

In the case of an adversarial environment like Schnapsen, the research process appears to be relatively obvious. In order to measure performance, the agent will need to play many games against other agents (for this research project we decided on 1,000 games between each two bots). For the sake of this research paper, the focus is not on time, but solely on how well one agent does against another in terms of its win-ratios. These games (face-offs) are essentially experiments that allow the researcher to access the performance of different agents.

The large number of games is chosen to optimize replicability and minimize the effects of randomness on the outcome. After the given amount of games is executed, the win/loss proportion, as a ratio of the points scored by one agent divided by the total points scored by both players, is recorded and analyzed in accordance with standard statistical methodology. Given the nature of the data, the statistical method chosen for the purpose of this analysis is hypothesis testing of population proportions (a series of right-tailed tests). The purpose of these tests is to statistically infer whether or not the results of this research can be regarded as significant. In other words, the experiments are viewed as samples, and the purpose of the statistical analysis is to project the validity of the findings out into the hypothetical population.

Conducting this analysis will add a layer of statistical strength to the claims made in this research paper. In this setup, there are three possible outcomes to our experiment: (i) the ensemble bot performance significantly worse than its individual constituent bots individually, (ii) the ensemble bot's performance is not significantly worse or better than that of its constituent bots (perhaps underperforming against some, and outperforming others), or (iii) the ensemble bot significantly outperforms all its constituents bots.
The exact details of the statistical testing procedure, including all the relevant code (in R), can be found in Appendix 3. These are the basic test statistics:

$$H_0: p = 0.5$$
$$H_a: p > 0.5$$
$$\alpha = 5\%$$
$$Z = \frac{\hat{p}_{n-p_0}}{\sqrt{\frac{p_0(1-p_0)}{n}}}$$

3.2 Benchmarking performance

In our analysis we adopt both subjective and objective benchmarks. As the performance of a Schnapsen player, in terms of its win-ratio, is dependent on the performance level of its opponent, the win-ratios between the ensemble and its constituent bots are inherently subjective measures.

There are, however, also two objective benchmarks that we adopt in our analysis. The first is the win-ratio of a randomly playing opponent (signifying the worst possible player) playing a theoretically optimal playing opponent. The second is the inverse, namely the win-ratio of the theoretically optimal player against a randomly playing opponent. These benchmarks are objective in that they provide hard lower and upper bounds to the win-ratio of any imaginable player. Since, given the uncertainty of imperfect information, an optimal value function is impossible during phase 1 of the game, we created a theoretically optimal player by changing the game engine to allow perfect information in phase 1 and thereby enabling the use of an optimal value function during the entire game.

We implemented the optimal value function with a Perfect Information MiniMax (PIMM) bot, which has optimal performance under the now perfect information conditions. PIMM serves as the theoretical benchmark of optimal performance. With the randomly playing 'rand bot' and PIMM as lower and upper bounds, there exists a wide spectrum of performance against which the bots proposed in this research paper can be measured against.

Ⅳ. Agent Design

The concept of voting in machine learning is perhaps most notoriously represented by the K-Nearest Neighbor-Algorithm (KNN). KNN is a supervised learning method that is commonly used for classification and regression problems. The concept is extremely simple: KNN measures the distance from the unlabeled datapoint to its n (labeled) neighbors, takes a majority vote from this neighborhood and applies the voted label onto the unlabeled datapoint. As this algorithm does not actually involve any real learning (it only copies pre-existing labels), it is also referred to as a *lazy algorithm*. (Rebala et al. 2019, 72) While ensemble bot works on the same fundamental "voting" principle as the KNN algorithm, there are some crucial differences between the two systems. In the case of KNN, the voting entities are merely labeled data points, who do not possess any degree of intelligence. On the other hand, the voting entities for the Ensemble bot are intelligent agents that have arrived at their vote through the application of a variety of strategic methods. Specifically,

the vote is cast between four different machine-learning bots, rdeep (a blend of monte carlo sampling and minimax) and a rule-based mybot implementation. Thus, while KNN might arguably be best described as the shouting of fools, Ensemble bot is better characterized as a *council of the wise*.

3.1 Ensemble bot implementation

We implemented the voting procedure for the ensemble bot as follows. Every time the ensemble bot needs to make a move, it asks the constituent bots for their best move, resulting in six move suggestions. If a certain move is suggested more often than any other, the ensemble will pick that move. If two or more moves get an equal number of votes, the ensemble bot breaks the tie by randomly choosing one of these options. Because there are, at maximum, only five legal moves (not counting the special tricks such as calling a marriage) and given that the ensemble consists of six voting bots, there will always be at least one move that receives a minimum of two votes.

As our focus is on the performance of our ensemble bot in an imperfect information context (thus during phase 1 of the game), the ensemble bot and all its constituent bots use the minimax method for phase 2. This way, any difference in win-ratios will be solely due to a difference in performance in phase 1 of the game. The only exception to this rule is the randomly playing 'rand bot', as it is intended to serve as an absolute lower performance bound, it plays randomly the entire game. The exact code for all the bots can be found in Appendix 2 of this research paper.

3.2 Theoretical considerations

Ensemble methods have gained increasing popularity within the AI research community in recent years, specifically machine learning and subfields like reinforcement learning. They have frequently been shown to outperform methods that rely on a single classifier and can be applied to both classification and regression problems. (Wiering and van Hasselt 2008, 2) There are two broad classes of ensemble methods, which are called *averaging methods* and *boosting methods*. The former includes methods such as random forest, bagging and voting classifier. (Zhang 2019) The method underlying Ensemble bot is best classified as a variation of the voting classifier with hard voting (meaning that a majority vote is taken).

According to Thomas Dietterich("Ensemble Learning"), ensemble methods tend outperform single hypothesis methods (i.e. classical machine learning methods), because they improve upon three problems posed by the latter methods. First, the statistical problem. The search space might be so large that the classical machine learning algorithm cannot come up with a final hypothesis, leading to uncertainty about which path to choose. This problem is addressed by introducing a voting-based system, which can direct the decision-making process in a meaningful way. Indeed, Schnapsen is with its search space complexity of10^{20}a case in which the statistical problem applies.

The second problem is the computational problem, which refers to the issue that the cost function of machine learning algorithms can get stuck in local minima, and thus fails to

find the global minima which would be the optimal choice. This problem is linked to the nature of the hill climbing algorithm, which is underlying many heuristic methods of machine learning methods (e.g. gradient descend). Again, the ensemble voting systems helps to overcome this challenge, as it increases the likelihood that the best decision will be taken, given a case in which a multitude of local minima are present. The third problem is called representational problem, and it arises when the agent fails to properly approximate the true (optimal) function *f*. There may be many reasons for such a case, but ensemble methods that take a weighted average of different voter functions tend to resolve this issue reasonably well. (Dietterich 2002, 3-4)

V. Results

Playing 1000 games between each pair of bots resulted in the following win-ratio distribution:

	ensemble	ml_mix	ml_rdeep	ml_mybot2	ml_rand	rdeep	mybot2	rand	avg win-rate
ensemble		0,59	0,57	0,58	0,54	0,59	0,69	0,91	0,64
ml_mix	0,41		0,51	0,52	0,50	0,55	0,62	0,88	0,57
ml_rdeep	0,43	0,49		0,52	0,51	0,50	0,62	0,88	0,56
ml_mybot2	0,42	0,48	0,48		0,47	0,52	0,64	0,86	0,55
ml_rand	0,46	0,50	0,49	0,53		0,53	0,59	0,86	0,57
rdeep	0,41	0,45	0,50	0,48	0,47		0,50	0,87	0,53
mybot2	0,31	0,38	0,38	0,36	0,41	0,50		0,84	0,45
rand	0,09	0,12	0,12	0,14	0,14	0,13	0,16		0,13

***Figure 1**: Win-ratio per bot combination based on the scored points as above (e.g. ensemble scored a 1001/(1001+703) = 0.59 ratio against ml_mix, and ml_mix a 703/(1001+703) = 0.41 ratio against ensemble)*

The results in Figure 1 are already insightful by themselves, but in order to be able to claim that one bot outperforms another, it needs to be verified which of these win-ratios are statistically significant, given the sample sizes and our test statistic. After running the statistical analysis (Appendix 3), the following results are obtained:

OP = Outperforms
TIE = Fails not outperform, is not outperformed
FOP = Fails to outperform, is outperformed

	ensemble	ml_mix	ml_rdeep	ml_mybot2	ml_rand	rdeep	mybot2	rand
ensemble		OP	OP	OP	OP	OP	OP	OP
ml_mix	FOP		TIE	TIE	TIE	OP	OP	OP
ml_rdeep	FOP	TIE		OP	TIE	TIE	OP	OP
ml_mybot2	FOP	TIE	FOP		FOP	TIE	OP	OP
ml_rand	FOP	TIE	TIE	OP		OP	OP	OP
rdeep	FOP	FOP	TIE	TIE	FOP		TIE	OP
mybot2	FOP	FOP	FOP	FOP	FOP	TIE		OP
rand	FOP	FOP	FOP	FOP	FOP	FOP	FOP	

***Figure 2**: Result matrix after running the statistical analysis*

Figure 2 shows clearly that ensemble outperforms all the other bots, and it can be concluded that its performance is statistically significant. Interestingly, the results between the different classical ML bots are rather different, with some surprising results (e.g. ml_rand outperforming ml_mybot2). In order to get a deeper understanding on the behavior of the ensemble agent, the ratio of agreement between the different voting member is recorded in an agreement matrix (Figure 3).

	ml_mix	**ml_rdeep**	**ml_mybot2**	**ml_rand**	**rdeep**	**mybot2**
ml_mix		0,38	0,36	0,35	0,29	0,16
ml_rdeep	0,38		0,36	0,32	0,28	0,14
ml_mybot2	0,36	0,36		0,33	0,29	0,19
ml_rand	0,35	0,32	0,33		0,27	0,24
rdeep	0,29	0,28	0,29	0,27		0,21
mybot2	0,16	0,14	0,19	0,24	0,21	

***Figure 3**: Agreement matrix*

As mentioned above, the theoretical benchmark of optimal performance was measured by changing the game engine, in order to essentially convert Schnapsen into a perfect information environment. This theoretical maximum bot was then played against ml_mix, and the win-ratio of this face-off represents the optimal performance benchmark. Figure 4 contrasts this optimal win-ratio with that of the other bots. As one can observe, ensemble approaches this theoretical benchmark.

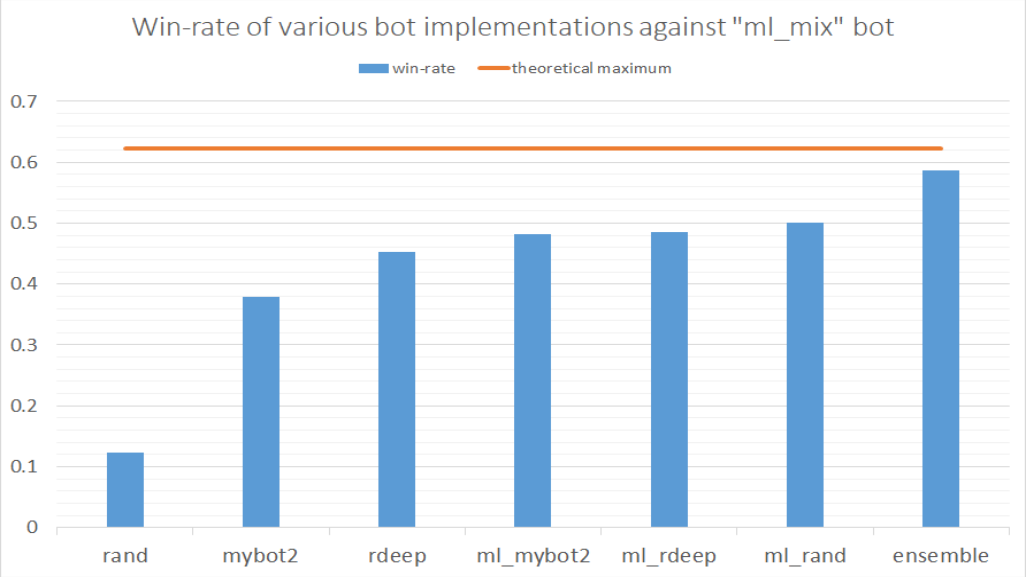


Figure 4: Measuring bots against theoretical optimal performance boundary

Ⅵ. Findings

The results show that those bots that tend to perform best also have the highest agreement ratio. Given that the ml-based bots performance almost equally well in terms of win-ratios, however, it is interesting to see that they agree with each other only 38% of the time at best (Figure 4).

The combination of the relatively poor performance of both rdeep and mybot2, and their seemingly limited contribution in the above agreement matrix, could suggest that their inclusion in the ensemble potentially undermines the ensemble's performance. Surprisingly, perhaps, it turns out that our ensemble consistently outperforms more exclusive variants of the ensemble. Against a more limited ensemble consisting only of the ml-based bots, our ensemble managed a 879-748 result (54% win-ratio; p = 0.0006). Playing against an ensemble consisting of the ml-based bots and rdeep (so without mybot2), the result was closer, but still in favor of our complete ensemble (860-784 result, or 52% win-ratio; p = 0.032). This result is in line with the observation by Kim & Cho (2008, 212) that diversity in the ensemble is important and that "each member should be good, although it is not necessary to be the best".

Ⅶ. Conclusion

This paper showed how the game of Schnapsen can be mastered using ensemble voting methods. To achieve the aim of designing an intelligent agent that would be up for the challenge, it was essential to properly understand the environment the agent would need to deal with. The most peculiar aspects of Schnapsen are the uncertainty (imperfect information) in phase 1 and the relatively large state space complexity of 10^{20}. *The focus of*

this research paper was thus on optimizing the performance of Ensemble in phase 1, given that phase 2 could be won by any agent applying the minimax algorithm. The methodology that was developed to assess the performance of the agents consists of playing a large amount of games (the exact number is 1000) between two bots, measuring the win-ratio, and running a statistical analysis on the results. This process was designed in order to ensure the robustness reliability and consistency of our features set convincing that changing environment tend to influence the performance of the agents, significantly of our results. We also set the theoretical benchmark for the higher (optimal performance) and lower (random) bound. We then described how the ensemble bot was implemented. Theoretical knowledge was consulted to explain why ensemble methods tend to outperform classic machine learning algorithms, addressing issues such as the statistical, computational and representational problem. Then the results of this research paper were presented, demonstrating empirical proof for the claim that ensemble outperforms a variety of ML agents, as well as the monte-carlo sampling agent rdeep. It was also shown that ensemble approximates the theoretical benchmark of optimal performance. Finally, we reflected on the meaning of the different agreement ratios within the voting system, and also showed empirically that ensemble bot benefits from including the weaker performing bots mybot2 and rdeep in the vote. *This idea can be backed up by different findings (e.g. Dietterich, Kim & Cho), which suggest that a diverse set of training data improves the intelligent behavior of the agent.*

Suggestions for further research

This paper showed exciting possibilities for the application of ensemble methods to optimize strategic performance in adversarial environments like Schnapsen. However, there is much more to be explored in this regard. One topic that has not been addressed in depth by this paper is time, for instance time restrictions and complexity measurements. Computational time complexity is an interesting field and understanding those metrics on different variations of ensemble agents would certainly provide deeper insights on their inner workings. To achieve this, the experimental setting would need to be altered to pay specific attention to time complexity and these results can then be contrasted with what is known about other methods in this regard.

Another branch of knowledge which needs further exploration is the exact constitution of voting bots. Interestingly, during the development of this research project, it was found that the ensemble bot indeed performs better if a poor-performing agent (in this case mybot2) was given a vote. This phenomenon deserves further investigation, and studies that could expand upon the differences in performance based on varying constitutions of the voting agent council would greatly enhance the understanding of such methods.

Bibliography

Dietterich, Thomas. “Ensemble Learning.” In *The Handbook of Brain Theory and Neural Networks*, 2nd ed. Cambridge: MIT Press, 2002.

Kim, K. J., & Cho, S. B. (2008). “Ensemble approaches in evolutionary game strategies: A case study in Othello”. *In 2008 IEEE Symposium On Computational Intelligence and Games* (pp. 212-219). IEEE.

Rebala, Gopinath, Ajay Ravi, and Sanjay Churiwala. *An Introduction to Machine Learning*. Cham: Springer, 2019.

Russell, Stuart, and Peter Norvig. *Artificial Intelligence - A Modern Approach*. 3rd ed. Harlow: Pearson, 2016.

Silver, David, Aja Huang, Chris Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” *Nature* 529 (January 27, 2016): 484–89.

Tegmark, Max. *Life 3.0: Being Human in the Age of Artificial Intelligence.* London: Penguin Books, 2018.

Wiering, Marco, and Hado van Hasselt. “Ensemble Algorithms in Reinforcement Learning.” *ResearchGate*, 2008, 1–6. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.535.5956&rep=rep1&type=pdf.

Wisser, Florian. “An Expert-Level Card Playing Agent Based on a Variant of Perfect Information Monte Carlo Sampling.” *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, 125–31.

Yannakakis, Georgios, and Julian Togelius. *Artificial Intelligence and Games*. Cham: Springer, 2018.

Zhang, Alina. “Ensemble Strategy in Machine Learning to Surpass Any Single Model.” Medium Data Driven Investor, July 10, 2019. https://medium.com/datadriveninvestor/ensemble-strategy-in-machine-learning-to-surpass-any-single-model-a75a92c3f1f0.

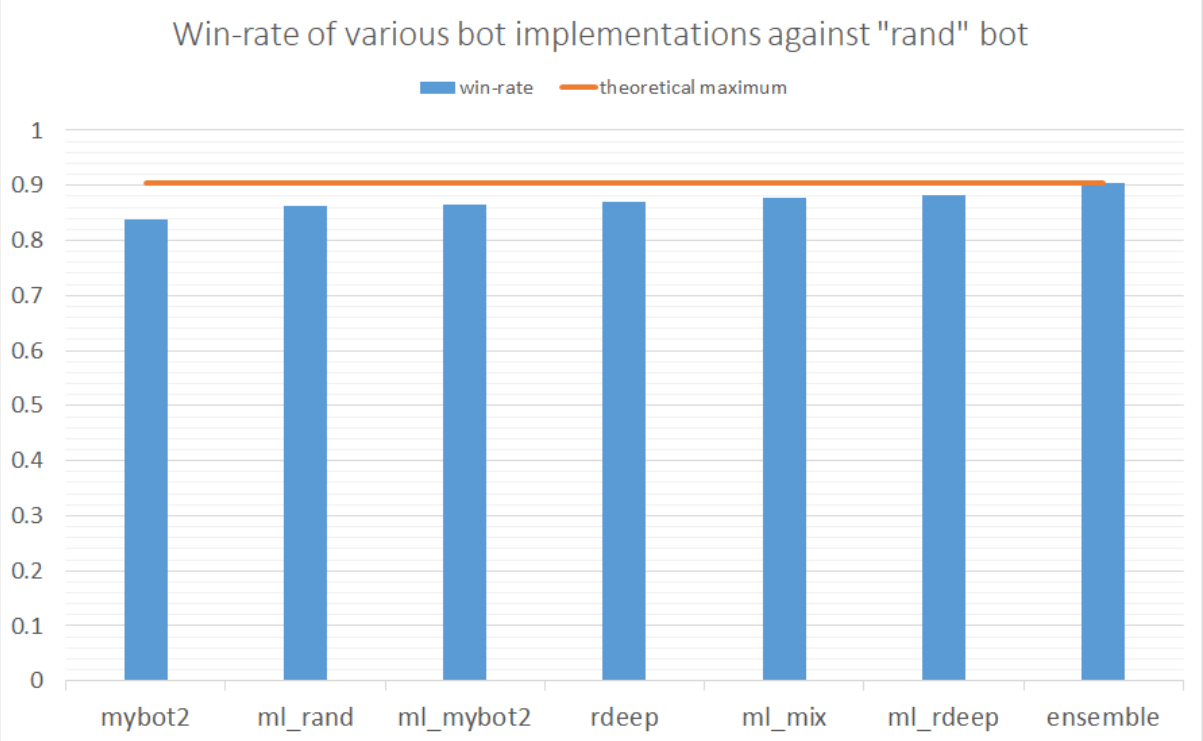
Appendix 1 – Further Results

Point results based on 1,000 games per bot combination

(eg. ensemble scored 1001 points against ml_mix, and ml_mix 703 against ensemble)

	ensemble	ml_mix	ml_rdeep	ml_mybot2	ml_rand	rdeep	mybot2	rand
ensemble		1001	974	960	876	916	1033	1792
ml_mix	703		903	877	840	857	897	1659
ml_rdeep	723	854		885	861	811	915	1690
ml_mybot2	696	816	810		769	804	920	1618
ml_rand	747	841	834	870		849	839	1642
rdeep	637	711	814	744	766		738	1666
mybot2	465	546	570	518	594	724		1416
rand	188	234	226	255	264	249	275	

Appendix 1.1 – Point results



Appendix 1.2 – Performance measure against theoretical lower bound (rand)

Appendix 2.1 - ensemble_bot

```
"""
Ensemble bot - this bot decides on the best move based on the most often suggested
move from six voting bots (four ml-based bots, a perfect information monte carlo
bot, and a rule-based bot).

The bot uses minimax for phase 2
"""

# Import the API objects
from api import State, Deck
import random
import numpy
import importlib
from collections import Counter

from ..ml_mix_20k import ml_mix_20k
from ..ml_rdeep_20k import ml_rdeep_20k
from ..ml_rand_20k import ml_rand_20k
from ..ml_mybot2_20k import ml_mybot2_20k
from ..rdeep import rdeep
from ..mybot2 import mybot2
from ..minimax import minimax

# initialize voting bots
bots = [ml_mix_20k, ml_rdeep_20k, ml_rand_20k, ml_mybot2_20k, rdeep, mybot2]

class Bot:

    def __init__(self):
        pass

    def get_move(self, state):
        # type: (State) -> tuple[int, int]

        suggested_moves = []

        # get votes from the voting bots
        if state.get_phase() == 1:
            for bot in bots:
                suggested_moves.append(bot.Bot().get_move(state))

            # select most often suggested move (and pick amongst the highest
scoring in case of a tie)
            occurence_count = Counter(suggested_moves)
            if occurence_count.most_common(1)[0][1] > 1 and
occurence_count.most_common(1)[0][1] > occurence_count.most_common(2)[0][1]:
                chosen_move = occurence_count.most_common(1)[0][0]
            elif occurence_count.most_common(1)[0][1] > 1 and
occurence_count.most_common(1)[0][1] == occurence_count.most_common(2)[0][1] and
occurence_count.most_common(1)[0][1] == occurence_count.most_common(3)[0][1]:
                i = random.randint(1,3)
                chosen_move = occurence_count.most_common(i)[0][0]
            elif occurence_count.most_common(1)[0][1] > 1 and
occurence_count.most_common(1)[0][1] == occurence_count.most_common(2)[0][1]:
                i = random.randint(1, 2)
```

```
            chosen_move = occurence_count.most_common(i)[0][0]
        else:
            chosen_move = random.choice(suggested_moves)


    # use minimax for phase 2
    else:
        chosen_move = minimax.Bot().get_move(state)

    return chosen_move
```

Appendix 2.2 - ml bots

The code for all four ml-based bots (ml_mix, ml_rand, ml_rdeep and ml_mybot2) is as below. The only difference between the bots is the dataset on which they are trained:

- ml_mix: trained on a dataset generated by randomly switching between rand, rdeep and mybot2
- ml_rand: trained on a dataset generated by playing rand
- ml_rdeep: trained on a dataset generated by playing rdeep
- ml_mybot2: trained on a dataset generated by playing mybot2

```
"""
A basic adaptive bot, based on the following parameters:
games = 20000
hidden_layer_sizes = (64, 32)
learning_rate = 0.0001
regularization_strength = 0.0001

The bot uses minimax for phase 2

"""

from api import State, util, Deck
import random, os
from itertools import chain
from ..minimax import minimax

import joblib

# Path of the model we will use. If you make a model
# with a different name, point this line to its path.
DEFAULT_MODEL = os.path.dirname(os.path.realpath(__file__)) + '/model.pkl'

class Bot:

    __randomize = True

    __model = None

    def __init__(self, randomize=True, model_file=DEFAULT_MODEL):

        #print(model_file)
        self.__randomize = randomize

        # Load the model
        self.__model = joblib.load(model_file)

    def get_move(self, state):

        if state.get_phase() == 1:
            val, move = self.value(state)

        # use minimax for phase 2
        else:
            move = minimax.Bot().get_move(state)

        return move
```

```
    def value(self, state):
        """
        Return the value of this state and the associated move
        :param state:
        :return: val, move: the value of the state, and the best move.
        """

        best_value = float('-inf') if maximizing(state) else float('inf')
        best_move = None

        moves = state.moves()

        if self.__randomize:
            random.shuffle(moves)

        for move in moves:

            next_state = state.next(move)

            value = self.heuristic(next_state)

            if maximizing(state):
                if value > best_value:
                    best_value = value
                    best_move = move
            else:
                if value < best_value:
                    best_value = value
                    best_move = move

        return best_value, best_move

    def heuristic(self, state):

        # Convert the state to a feature vector
        feature_vector = [features(state)]

        # These are the classes: ('won', 'lost')
        classes = list(self.__model.classes_)

        # Ask the model for a prediction
        # This returns a probability for each class
        prob = self.__model.predict_proba(feature_vector)[0]

        # Weigh the win/loss outcomes (-1 and 1) by their probabilities
        res = -1.0 * prob[classes.index('lost')] + 1.0 *
prob[classes.index('won')]

        return res

def maximizing(state):
    """
    Whether we're the maximizing player (1) or the minimizing player (2).
    :param state:
    :return:
    """
    return state.whose_turn() == 1
```

```
def features(state):
    # type: (State) -> tuple[float, ...]
    """
    Extract features from this state. Remember that every feature vector returned
should have the same length.

    :param state: A state to be converted to a feature vector
    :return: A tuple of floats: a feature vector representing this state.
    """

    feature_set = []

    # Add player 1's points to feature set
    p1_points = state.get_points(1)

    # Add player 2's points to feature set
    p2_points = state.get_points(2)

    # add number of trump-suited cards in hand to feature set
    #trump_suit_cards = [card for card in state.hand() if Deck.get_suit(card) ==
state.get_trump_suit()]
    #trump_suit_hand = len(trump_suit_cards)

    # value of cards in hand
    #hand_value = 0.0
    #score = [11, 10, 4, 3, 2]
    #for card in state.hand():
    #    hand_value += float(score[(card % 5)])
    #normalized_hand_value = (hand_value - 11 / 43)

    # Add player 1's pending points to feature set
    p1_pending_points = state.get_pending_points(1)

    # Add plauer 2's pending points to feature set
    p2_pending_points = state.get_pending_points(2)

    # Get trump suit
    trump_suit = state.get_trump_suit()

    # Add phase to feature set
    phase = state.get_phase()

    # Add stock size to feature set
    stock_size = state.get_stock_size()

    # Add leader to feature set
    leader = state.leader()

    # Add whose turn it is to feature set
    whose_turn = state.whose_turn()

    # Add opponent's played card to feature set
    opponents_played_card = state.get_opponents_played_card()


    ################## You do not need to do anything below this line
########################

    perspective = state.get_perspective()
```

```
    # Perform one-hot encoding on the perspective.
    # Learn more about one-hot here: https://machinelearningmastery.com/how-to-
one-hot-encode-sequence-data-in-python/
    perspective = [card if card != 'U'   else [1, 0, 0, 0, 0, 0] for card in
perspective]
    perspective = [card if card != 'S'   else [0, 1, 0, 0, 0, 0] for card in
perspective]
    perspective = [card if card != 'P1H' else [0, 0, 1, 0, 0, 0] for card in
perspective]
    perspective = [card if card != 'P2H' else [0, 0, 0, 1, 0, 0] for card in
perspective]
    perspective = [card if card != 'P1W' else [0, 0, 0, 0, 1, 0] for card in
perspective]
    perspective = [card if card != 'P2W' else [0, 0, 0, 0, 0, 1] for card in
perspective]

    # Append one-hot encoded perspective to feature_set
    feature_set += list(chain(*perspective))

    # Append normalized points to feature_set
    total_points = p1_points + p2_points
    feature_set.append(p1_points/total_points if total_points > 0 else 0.)
    feature_set.append(p2_points/total_points if total_points > 0 else 0.)

    # Append normalized pending points to feature_set
    total_pending_points = p1_pending_points + p2_pending_points
    feature_set.append(p1_pending_points/total_pending_points if
total_pending_points > 0 else 0.)
    feature_set.append(p2_pending_points/total_pending_points if
total_pending_points > 0 else 0.)

    # Convert trump suit to id and add to feature set
    # You don't need to add anything to this part
    suits = ["C", "D", "H", "S"]
    trump_suit_onehot = [0, 0, 0, 0]
    trump_suit_onehot[suits.index(trump_suit)] = 1
    feature_set += trump_suit_onehot

    # Append one-hot encoded phase to feature set
    feature_set += [1, 0] if phase == 1 else [0, 1]

    # Append normalized stock size to feature set
    feature_set.append(stock_size/10)

    # Append one-hot encoded leader to feature set
    feature_set += [1, 0] if leader == 1 else [0, 1]

    # Append one-hot encoded whose_turn to feature set
    feature_set += [1, 0] if whose_turn == 1 else [0, 1]

    # Append one-hot encoded opponent's card to feature set
    opponents_played_card_onehot = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0]
    opponents_played_card_onehot[opponents_played_card if opponents_played_card is
not None else 20] = 1
    feature_set += opponents_played_card_onehot
```

```
    # Return feature set
    return feature_set
```

Appendix 2.3 - rdeep

```
"""
RdeepBot - This bot looks ahead by following a random path down the game tree.
That is, it assumes that all players have the same strategy as rand.py, and
samples N random games following from a given move. It then ranks the moves by
averaging the heuristics  of the resulting states.
"""

# Import the API objects
from api import State, util
import random


class Bot:

   # How many samples to take per move
   __num_samples = -1
   # How deep to sample
   __depth = -1

   def __init__(self, num_samples=4, depth=8):
      self.__num_samples = num_samples
      self.__depth = depth

   def get_move(self, state):

      # See if we're player 1 or 2
      player = state.whose_turn()

      # Get a list of all legal moves
      moves = state.moves()

      # Sometimes many moves have the same, highest score, and we'd like the bot
to pick a random one.
      # Shuffling the list of moves ensures that.
      random.shuffle(moves)

      best_score = float("-inf")
      best_move = None

      scores = [0.0] * len(moves)

      for move in moves:
         for s in range(self.__num_samples):

            # If we are in an imperfect information state, make an assumption.

            sample_state = state.make_assumption() if state.get_phase() == 1 else
state


            score = self.evaluate(sample_state.next(move), player)

            if score > best_score:
               best_score = score
               best_move = move

      return best_move # Return the best scoring move
```

```
   def evaluate(self,
                state,     # type: State
                player     # type: int
            ):
      # type: () -> float
      """
      Evaluates the value of the given state for the given player
      :param state: The state to evaluate
      :param player: The player for whom to evaluate this state (1 or 2)
      :return: A float representing the value of this state for the given player.
The higher the value, the better the
         state is for the player.
      """

      score = 0.0

      for _ in range(self.__num_samples):

         st = state.clone()

         # Do some random moves
         for i in range(self.__depth):
            if st.finished():
               break

            st = st.next(random.choice(st.moves()))

         score += self.heuristic(st, player)

      return score/float(self.__num_samples)

   def heuristic(self, state, player):
      return util.ratio_points(state, player)
```

Appendix 2.4 - mybot2

```
"""
Mybot2 – a rule-based bot with a simple strategy that can best be described as
“save the best for last and win easy points if possible”

"""

# Import the API objects
from api import State
from api import Deck
import random
from ..minimax import minimax

class Bot:

   def __init__(self):
      pass

   def get_move(self, state):
      # type: (State) -> tuple[int, int]

      # All legal moves
      moves = state.moves()
      chosen_move = moves[0]

      moves_trump_suit = []
      moves_non_trump_suit = []
      moves_same_suit = []

      if state.get_phase() == 1:

         # If the opponent has played a card
         if state.get_opponents_played_card() is not None:

            # when opponent plays high card (A or 10) from a non-trump suit try to
win the trick with a low trump-suit card
            if Deck.get_suit(state.get_opponents_played_card()) !=
state.get_trump_suit() and state.get_opponents_played_card() % 5 <= 1:

               # Get all trump suit moves available
               for index, move in enumerate(moves):

                  if move[0] is not None and Deck.get_suit(move[0]) ==
state.get_trump_suit():
                     moves_trump_suit.append(move)

               if len(moves_trump_suit) > 0:
                  chosen_move = moves_trump_suit[-1]

            # when opponent plays low card (J, Q, or K) from a non-trump suit try
to win the trick with a higher card of the same suit
            elif Deck.get_suit(state.get_opponents_played_card()) !=
state.get_trump_suit():

               # Get all moves of the same suit as the opponent's played card
               for index, move in enumerate(moves):
                  if move[0] is not None and Deck.get_suit(move[0]) ==
```

```
Deck.get_suit(
                        state.get_opponents_played_card()) and move[0] % 5 <
state.get_opponents_played_card() % 5:
                    moves_same_suit.append(move)

            if len(moves_same_suit) > 0:
                chosen_move = moves_same_suit[-1]

        else:

            # get all non-trump suit moves available, and pick lowest rank
            for index, move in enumerate(moves):

                if move[0] is not None and Deck.get_suit(move[0]) !=
state.get_trump_suit():
                    moves_non_trump_suit.append(move)

            if len(moves_non_trump_suit) > 0:
                random.shuffle(moves_non_trump_suit)
                chosen_move = moves_non_trump_suit[0]
                for index, move in enumerate(moves_non_trump_suit):
                    if move[0] is not None and move[0] % 5 > chosen_move[0] % 5:
                        chosen_move = move

      # use minimax for phase 2
      else:
          chosen_move = minimax.Bot().get_move(state)

      return chosen_move
```

Appendix 2.5 - minimax

```
"""
Minimax – uses the minimax algorithm to decide on the best move in phase 2
Using an adjusted version of the game engine, this bot was also used to play as a
perfect information minimax (PIMM) bot starting from phase 1. In this setting PIMM
can provide theoretically optimal benchmarks.

"""

from api import State, util, Deck
import random

class Bot:

    __max_depth = -1
    __randomize = True

    def __init__(self, randomize=True, depth=6):
        """
        :param randomize: Whether to select randomly from moves of equal value (or
to select the first always)
        :param depth:
        """
        self.__randomize = randomize
        self.__max_depth = depth

    def get_move(self, state):
        # type: (State) -> tuple[int, int]

        val, move = self.value(state)

        return move

    def value(self, state, depth = 0):
        # type: (State, int) -> tuple[float, tuple[int, int]]
        """
        Return the value of this state and the associated move
        :param state:
        :param depth:
        :return: A tuple containing the value of this state, and the best move for
the player currently to move
        """

        if state.finished():
            winner, points = state.winner()

            return (points, None) if winner == 1 else (-points, None)

        if depth == self.__max_depth:
            return heuristic(state)

        moves = state.moves()

        if self.__randomize:
            random.shuffle(moves)

        best_value = float('-inf') if maximizing(state) else float('inf')
```

```
        best_move = None

        for move in moves:

            next_state = state.next(move)

            value, m = self.value(next_state, depth+1)

            if maximizing(state):
                if value > best_value:
                    best_value = value
                    best_move = move
            else:
                if value < best_value:
                    best_value = value
                    best_move = move

        return best_value, best_move

def maximizing(state):
    # type: (State) -> bool
    """
    Whether we're the maximizing player (1) or the minimizing player (2).

    :param state:
    :return:
    """
    return state.whose_turn() == 1

def heuristic(state):
    # type: (State) -> float
    """
    Estimate the value of this state: -1.0 is a certain win for player 2, 1.0 is a
certain win for player 1

    :param state:
    :return: A heuristic evaluation for the given state (between -1.0 and 1.0)
    """
    return util.ratio_points(state, 1) * 2.0 - 1.0, None
```

Appendix 3 - Statistical analyses

Claim (H_a): One bot performs better than the other.
Population parameter: **p** (population proportion)
Sample values: $\hat{p}_n$ (sample proportion, different for each series of games), i.e. win ratio
n (sample size, total points distributed between two players at 1000 games)
X (number of successes, relevant for the R code)

Test statistic: $Z = \frac{\hat{p}_n - p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}}$

The test statistic has a standard normal distribution.

Objective: The goal is to make a statistical inference about the population parameter, given the results from our experiment (i.e. the sample).

Procedure: Each of these right-tailed tests will be conducted using the P-value method, utilizing the statistical package R. For each comparison between two bots, if the measured P-value is smaller than α (given below), it can be concluded that one bot outperforms the other. If the measured P-value is greater than α, the null hypothesis cannot be rejected (i.e. there is not sufficient evidence to conclude that one bot performs better than the other). In this case, it will be checked whether the win-ratio of the first bot was larger than that of the second bot. If it was, it follows logically that the second bot does not outperform the first bot, so another test is not needed. If the win ratio of the other bot was in fact higher, the test will be reversed in order to find out if the second bot outperformed the first.

$$H_0: p = 0.5$$
$$H_a: p > 0.5$$
$$\alpha = 5\%$$

Test results:

ensemble | ml_mix

$\hat{p}_1 = 0{,}59$

X = 1001

n = 1704

```
> prop.test(1001, 1704, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1001 out of 1704, null probability 0.5
X-squared = 51.766, df = 1, p-value = 3.127e-13
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5674064 1.0000000
sample estimates:
        p
0.5874413
```

Conclusion: Since 3.127e-13 < 0.05, it can be concluded that ensemble outperforms ml_mix. Since the win-ratio of ensemble is greater than that of ml_mix, this implies that ml_mix does not outperform ensemble.

ensemble | ml_rdeep

$\widehat{p}_2 = 0{,}57$

X = 974

n = 1697

```
> prop.test(x=974, n=1697, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  974 out of 1697, null probability 0.5
X-squared = 36.83, df = 1, p-value = 6.446e-10
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.553811 1.000000
sample estimates:
        p
0.573954
```

Conclusion: Since 6.446e-10 < 0.05, it can be concluded that ensemble outperforms ml_rdeep. Since the win-ratio of ensemble is greater than that of ml_rdeep, this implies that ml_rdeep does not outperform ensemble.

ensemble | ml_mybot2

$\widehat{p}_3 = 0{,}58$

X = 960

n = 1656

```
> prop.test(x=960, n=1656, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  960 out of 1656, null probability 0.5
X-squared = 41.769, df = 1, p-value = 5.137e-11
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.559341 1.000000
sample estimates:
        p
0.5797101
```

Conclusion: Since 5.137e-11 < 0.05, it can be concluded that ensemble outperforms ml_mybot2. Since the win-ratio of ensemble is greater than that of ml_mybot2, this implies that ml_mybot2 does not outperform ensemble.

ensemble | ml_rand

$\widehat{p}_4 = 0{,}54$

X = 876

n = 1623

```
> prop.test(x=876, n=1623, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  876 out of 1623, null probability 0.5
X-squared = 10.095, df = 1, p-value = 0.0007434
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5190335 1.0000000
sample estimates:
        p
0.5397412
```

Conclusion: Since 0.0007434 < 0.05, it can be concluded that *ensemble outperforms* ml_rand. Since the win-ratio of ensemble is greater than that of ml_rand, this implies that ml_rand does not outperform ensemble.

ensemble | rdeep

$\widehat{p}_5 = 0{,}59$

X = 916

n = 1553

```
> prop.test(x=916, n=1553, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  916 out of 1553, null probability 0.5
X-squared = 49.764, df = 1, p-value = 8.668e-13
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5688334 1.0000000
sample estimates:
        p
0.5898261
```

Conclusion: Since 8.668e-13 < 0.05, it can be concluded that *ensemble outperforms* rdeep. Since the win-ratio of ensemble is greater than that of rdeep, this implies that rdeep does not outperform ensemble.

ensemble | mybot2

$\widehat{p}_6 = 0{,}69$

X = 1033

n = 1498

```
> prop.test(x=1033, n=1498, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1033 out of 1498, null probability 0.5
X-squared = 214.61, df = 1, p-value < 2.2e-16
```

```
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.6692577 1.0000000
sample estimates:
        p
0.6895861
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that *ensemble outperforms* mybot_2. Since the win-ratio of ensemble is greater than that of mybot2, this implies that mybot2 does not outperform ensemble.

ensemble | rand

$\widehat{p}_7 = 0{,}91$

X = 1792

n = 1980

```
> prop.test(x=1792, n=1980, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1792 out of 1980, null probability 0.5
X-squared = 1297.8, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8933899 1.0000000
sample estimates:
        p
0.9050505
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that *ensemble outperforms* rand. Since the win-ratio of ensemble is greater than that of rand, this implies that rand does not outperform ensemble.

ml_mix | ml_rdeep

$\widehat{p}_8 = 0{,}51$

X = 903

n = 1757

```
> prop.test(x=903, n=1757, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  903 out of 1757, null probability 0.5
X-squared = 1.3113, df = 1, p-value = 0.1261
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4940405 1.0000000
sample estimates:
        p
0.5139442
```

Conclusion: Since 0.1261 > 0.05, it cannot be concluded that *ml_mix outperforms* ml_rdeep. Since the win-ratio of ml_mix is greater than that of ml_rdeep, it is also clear that ml_rdeep does not outperform ml_mix.

ml_mix | ml_mybot2

$\hat{p}_9 = 0{,}52$

X = 877

n = 1693

```
> prop.test(x=877, n=1693, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  877 out of 1693, null probability 0.5
X-squared = 2.1264, df = 1, p-value = 0.07239
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4977323 1.0000000
sample estimates:
        p
0.5180154
```

Conclusion: Since 0.07239 > 0.05, it cannot be concluded that *ml_mix outperforms* ml_mybot2. Since the win-ratio of ml_mix is greater than that of ml_mybot2, it is also clear that ml_mybot2 does not outperform ml_mix.

ml_mix | ml_rand

$\hat{p}_{10} = 0{,}50$

X = 840

n = 1681

```
> prop.test(x=840, n=1681, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  840 out of 1681, null probability 0.5
X-squared = 0, df = 1, p-value = 0.5
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.479363 1.000000
sample estimates:
        p
0.4997026
```

Conclusion: Since 0.5 > 0.05, it cannot be concluded that *ml_mix outperforms* ml_rand. Since the win-ratio of ml_mix is the same as that of ml_mybot2, it is also clear that ml_rand does not outperform ml_mix.

ml_mix | rdeep

$\hat{p}_{11} = 0{,}55$

X = 857

n = 1568

```
> prop.test(x=857, n=1568, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction
```

```
data:  857 out of 1568, null probability 0.5
X-squared = 13.409, df = 1, p-value = 0.0001252
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5254948 1.0000000
sample estimates:
        p
0.5465561
```

Conclusion: Since 0.0001252 < 0.05, it can be concluded that ml_mix outperforms rdeep. Since the win-ratio of ml_mix is greater than that of rdeep, this implies that rdeep does not outperform ml_mix.

ml_mix | mybot_2

$\hat{p}_{12} = 0{,}62$

X = 897

n = 1443

```
> prop.test(x=897, n=1443, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  897 out of 1443, null probability 0.5
X-squared = 84.893, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.6000628 1.0000000
sample estimates:
        p
0.6216216
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_mix outperforms mybot2. Since the win-ratio of ml_mix is greater than that of rdeep, this implies that mybot2 does not outperform ml_mix.

ml_mix | rand

$\hat{p}_{13} = 0{,}88$

X = 1659

n = 1893

```
> prop.test(x=1659, n=1893, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1659 out of 1893, null probability 0.5
X-squared = 1071.2, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8631285 1.0000000
sample estimates:
        p
0.8763867
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_mix outperforms rand. Since the win-ratio of ml_mix is greater than that of rdeep, this implies that rand does not outperform ml_mix.

ml_rdeep | ml_mybot2

$\widehat{p}_{14} = 0{,}52$

X = 885

n = 1695

```
> prop.test(x=885, n=1695, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  885 out of 1695, null probability 0.5
X-squared = 3.2307, df = 1, p-value = 0.03614
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5018529 1.0000000
sample estimates:
        p
0.5221239
```

Conclusion: Since 0.03614 < 0.05, it can be concluded that ml_rdeep outperforms ml_mybot2. Since the win-ratio of ml_rdeep is greater than that of ml_mybot2, this implies that ml_mybot2 does not outperform ml_rdeep.

ml_rdeep | ml_rand

$\widehat{p}_{15} = 0{,}51$

X = 861

n = 1695

```
> prop.test(x=861, n=1695, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  861 out of 1695, null probability 0.5
X-squared = 0.39882, df = 1, p-value = 0.2638
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4876995 1.0000000
sample estimates:
        p
0.5079646
```

Conclusion: Since 0.2638 > 0.05, it cannot be concluded that *ml_rdeep outperforms* ml_rand. Since the win-ratio of ml_rdeep is larger than that of ml_rand, it is also clear that ml_rand does not outperform ml_rdeep.

ml_rdeep | rdeep

$\widehat{p}_{16} = 0{,}50$

X = 811

n = 1625

```
> prop.test(x=811, n=1625, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  811 out of 1625, null probability 0.5
X-squared = 0.0024615, df = 1, p-value = 0.5198
alternative hypothesis: true p is greater than 0.5
```

```
95 percent confidence interval:
 0.4783864 1.0000000
sample estimates:
        p
0.4990769
```

Conclusion: Since 0.5198 > 0.05, it cannot be concluded that *ml_rdeep outperforms* rdeep. Since the win-ratio of ml_rdeep is larger than that of rdeep, it is also clear that rdeep does not outperform ml_rdeep.

ml_rdeep | mybot2

$\hat{p}_{17} = 0{,}62$

X = 915

n = 1485

```
> prop.test(x=915, n=1485, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  915 out of 1485, null probability 0.5
X-squared = 79.688, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5948707 1.0000000
sample estimates:
        p
0.6161616
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_rdeep outperforms mybot2. Since the win-ratio of ml_rdeep is greater than that of mybot2, this implies that mybot2 does not outperform ml_rdeep.

ml_rdeep | rand

$\hat{p}_{18} = 0{,}88$

X = 1690

n = 1916

```
> prop.test(x=1690, n=1916, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1690 out of 1916, null probability 0.5
X-squared = 1117.1, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8691108 1.0000000
sample estimates:
        p
0.8820459
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_rdeep outperforms rand. Since the win-ratio of ml_rdeep is greater than that of rand, this implies that rand does not outperform ml_rdeep.

ml_rdeep | rand

$\hat{p}_{19} = 0{,}88$

X = 1690

n = 1916

```
> prop.test(x=1690, n=1916, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1690 out of 1916, null probability 0.5
X-squared = 1117.1, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8691108 1.0000000
sample estimates:
        p
0.8820459
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_rdeep outperforms rand. Since the win-ratio of ml_rdeep is greater than that of rand, this implies that rand does not outperform ml_rdeep.

ml_mybot2 | ml_rand

$\hat{p}_{20} = 0{,}47$

X = 769

n = 1639

```
> prop.test(x=769, n=1639, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  769 out of 1639, null probability 0.5
X-squared = 6.1013, df = 1, p-value = 0.9932
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4486762 1.0000000
sample estimates:
        p
0.4691885
```

Conclusion: Since 0.9932 > 0.05, it cannot be concluded that *ml_mybot2 outperforms* ml_rand. Since the win-ratio of ml_rdeep is lower than that of ml_rand, the test needs to be reversed to find out whether or not ml_rand outperforms ml_mybot2.

ml_rand | ml_mybot2

$\hat{p}_{21} = 0{,}53$

X = 870

n = 1639

```
> prop.test(x=870, n=1639, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
```

```
        correction

data:  870 out of 1639, null probability 0.5
X-squared = 6.1013, df = 1, p-value = 0.006754
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5101961 1.0000000
sample estimates:
        p
0.5308115
```

Conclusion: Since 0.006754 < 0.05, it can be concluded that ml_rand outperforms ml_mybot2

ml_mybot2 | rdeep

$\hat{p}_{22} = 0{,}52$

X = 804

n = 1548

```
> prop.test(x=804, n=1548, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  804 out of 1548, null probability 0.5
X-squared = 2.2487, df = 1, p-value = 0.06686
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4981538 1.0000000
sample estimates:
        p
0.5193798
```

Conclusion: Since 0.06686 > 0.05, it cannot be concluded that *ml_mybot2 outperforms* rdeep. Since the win-ratio of ml_mybot2 is larger than that of rdeep, it is also clear that rdeep does not outperform ml_mybot2.

ml_mybot2 | mybot2

$\hat{p}_{23} = 0{,}64$

X = 920

n = 1438

```
> prop.test(x=920, n=1438, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  920 out of 1438, null probability 0.5
X-squared = 111.82, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.6183582 1.0000000
sample estimates:
        p
0.6397775
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_mybot2 outperforms mybot2. Since the win-ratio of ml_mybot2 is greater than that of mybot2, this implies that mybot2 does not outperform ml_mybot2.

ml_mybot2 | rand

$\hat{p}_{24} = 0{,}86$

X = 1618

n = 1873

```
> prop.test(x=1618, n=1873, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1618 out of 1873, null probability 0.5
X-squared = 990.41, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8500174 1.0000000
sample estimates:
        p
0.8638548
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_mybot2 outperforms rand. Since the win-ratio of ml_mybot2 is greater than that of rand, this implies that rand does not outperform ml_mybot2.

ml_rand | rdeep

$\hat{p}_{25} = 0{,}53$

X = 849

n = 1615

```
> prop.test(x=849, n=1615, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  849 out of 1615, null probability 0.5
X-squared = 4.1635, df = 1, p-value = 0.02065
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.504923 1.000000
sample estimates:
        p
0.5256966
```

Conclusion: Since 0.02065 < 0.05, it can be concluded that ml_rand outperforms rdeep. Since the win-ratio of ml_rand is greater than that of rdeep, this implies that rdeep does not outperform ml_rand.

ml_rand | mybot2

$\hat{p}_{26} = 0{,}59$

X = 839

n = 1433

```
> prop.test(x=839, n=1433, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  839 out of 1433, null probability 0.5
X-squared = 41.546, df = 1, p-value = 5.755e-11
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5635868 1.0000000
sample estimates:
        p
0.585485
```

Conclusion: Since 5.755e-11 < 0.05, it can be concluded that ml_rand outperforms mybot2. Since the win-ratio of ml_rand is greater than that of mybot2, this implies that mybot2 does not outperform ml_rand.

ml_rand | rand

$\hat{p}_{27} = 0{,}86$

X = 1642

n = 1906

```
> prop.test(x=1642, n=1906, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1642 out of 1906, null probability 0.5
X-squared = 994.82, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8476899 1.0000000
sample estimates:
       p
0.86149
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that ml_rand outperforms rand. Since the win-ratio of ml_rand is greater than that of rand, this implies that rand does not outperform ml_rand.

rdeep | mybot2

$\hat{p}_{28} = 0{,}50$

X = 738

n = 1462

```
> prop.test(x=738, n=1462, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  738 out of 1462, null probability 0.5
X-squared = 0.1156, df = 1, p-value = 0.3669
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4829493 1.0000000
sample estimates:
        p
0.504788
```

Conclusion: Since 0.3669 > 0.05, it cannot be concluded that *rdeep outperforms* mybot2. Since the win-ratio of rdeep is the same as that of mybot2, it is also clear that mybot2 does not outperform rdeep.

rdeep | rand

$\hat{p}_{29} = 0{,}87$

X = 1666

n = 1915

```
> prop.test(x=1666, n=1915, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1666 out of 1915, null probability 0.5
X-squared = 1047, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8565367 1.0000000
sample estimates:
        p
0.8699739
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that rdeep outperforms rand. Since the win-ratio of rdeep is greater than that of rand, this implies that rand does not outperform rdeep.

mybot2 | rand

$\hat{p}_{30} = 0{,}84$

X = 1416

n = 1691

```
> prop.test(x=1416, n=1691, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  1416 out of 1691, null probability 0.5
X-squared = 768.54, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.8217706 1.0000000
sample estimates:
        p
0.8373743
```

Conclusion: Since 2.2e-16 < 0.05, it can be concluded that mybot2 outperforms rand. Since the win-ratio of mybot2 is greater than that of rand, this implies that rand does not outperform mybot2.

Final Results

The statistical analyses thus yield the following final results:

OP = Outperforms
FOP = Fails to Outperform *

	ensemble	ml_mix	ml_rdeep	ml_mybot2	ml_rand	rdeep	mybot2	rand
ensemble		OP	OP	OP	OP	OP	OP	OP
ml_mix	FOP		FOP	FOP	FOP	OP	OP	OP
ml_rdeep	FOP	FOP		OP	FOP	FOP	OP	OP
ml_mybot2	FOP	FOP	FOP		FOP	FOP	OP	OP
ml_rand	FOP	FOP	FOP	OP		OP	OP	OP
rdeep	FOP	FOP	FOP	FOP	FOP		FOP	OP
mybot2	FOP	FOP	FOP	FOP	FOP	FOP		OP
rand	FOP	FOP	FOP	FOP	FOP	FOP	FOP	

* Note: This does not imply that the bot underperforms against its opponent

OP = Outperforms
TIE = Fails not outperform, is not outperformed
FOP = Fails to outperform, is outperformed

	ensemble	ml_mix	ml_rdeep	ml_mybot2	ml_rand	rdeep	mybot2	rand
ensemble		OP	OP	OP	OP	OP	OP	OP
ml_mix	FOP		TIE	TIE	TIE	OP	OP	OP
ml_rdeep	FOP	TIE		OP	TIE	TIE	OP	OP
ml_mybot2	FOP	TIE	FOP		FOP	TIE	OP	OP
ml_rand	FOP	TIE	TIE	OP		OP	OP	OP
rdeep	FOP	FOP	TIE	TIE	FOP		TIE	OP
mybot2	FOP	FOP	FOP	FOP	FOP	TIE		OP
rand	FOP	FOP	FOP	FOP	FOP	FOP	FOP	

Additional tests

ensemble | ensemble_limited1(excluding rdeep and mybot2)

$\hat{p}_{31} = 0{,}54$

X = 879

n = 1627

```
> prop.test(x=879, n=1627, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  879 out of 1627, null probability 0.5
X-squared = 10.387, df = 1, p-value = 0.0006345
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5195771 1.0000000
```

```
sample estimates:
        p
0.5402581
```

Conclusion: Since 0.0006345 < 0.05, it can be concluded that ensemble outperforms ensemble_limited1.

ensemble | ensemble_limited2(excluding mybot2)

$\widehat{\boldsymbol{p}}_{\boldsymbol{32}} = 0{,}52$

X = 860

n = 1644

```
> prop.test(x=860, n=1644, p = 0.5, alternative = "greater")

        1-sample proportions test with continuity
        correction

data:  860 out of 1644, null probability 0.5
X-squared = 3.4215, df = 1, p-value = 0.03218
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.5025268 1.0000000
sample estimates:
        p
0.5231144
```

Conclusion: Since 0.03218 < 0.05, it can be concluded that ensemble outperforms ensemble_limited2.