# Optimizing Strategies for Dataset Construction in Training Intelligent Bots: A Comparative Analysis

Anonymous submission

January 2024

**Abstract**

The creation of effective datasets is at the crux of developing intelligent game-playing agents. This research paper investigates how a specific game strategy can be represented by a dataset which can then subsequently be used to train a machine learning (ML) agent. By developing Schnapsen bots with distinct strategies and generating diverse sets of training data, the study aims to discern which strategy produces the most efficient dataset for training an ML bot. By performing analysis on the win ratio (found by performing calculations on the amount of points won by each bot) we were able to quantify this relationship and attempted to establish which one of our ML bots (trained on this variety of datasets) performed the best and why. Each of our bots competed against the same intelligent agent which was provided for us (RDeep) and each other when training. We have found that a passive strategy slightly outperforms the rest. We have provided speculation for why however further research is needed to consolidate our hypothesis.

## 1 Introduction

In this rapidly evolving landscape of artificial intelligence (AI), the construction of an effective dataset is a cornerstone in the development of intelligent bots. As AI applications continue to permeate various domains, the need for robust and well-curated datasets becomes increasingly apparent. This research paper seeks to delve into the pivotal question: Is there an optimal strategy for creating a dataset that maximizes the performance and adaptability of a game-playing intelligent bot? The significance of a meticulously crafted dataset cannot be overstated, serving as the bedrock upon which intelligent bots build their understanding of the world. The choices made during the dataset construction process, including data selection, pre-processing methods, and the incorporation of diverse and representative samples, profoundly influence the body's ability to generalize and make informed decisions in real-world scenarios. As the AI community witnesses a surge in innovative approaches to data-driven learning, it becomes imperative to critically evaluate and compare various strategies employed in dataset construction. This research aims to address this question by rigorously analyzing the impact of different dataset construction methodologies on the overall performance and adaptability of an intelligent bot. Through a comparative analysis, this paper will explore the effectiveness of different game strategies, the role of domain-specific knowledge, and the implications of dataset diversity. By shedding light on the strengths and limitations of different strategies, our research endeavours to provide valuable insights into the nuanced process of constructing datasets for training intelligent bots, ultimately contributing to the advancement of AI capabilities in an increasingly data-centric landscape.

Through the development of bots with different Schnapsen strategies, we can create different sets of training data which can then be used to train our ML agent. This will allow us to analyze the efficiency of each dataset concerning the performance of our ML agent against the RDeep Schnapsen bot provided for us. The performance of our ML agent is directly influenced by the input or dataset it has used when being modelled. This research paper aims to distinguish which Schnapsen strategy (bot with specific playstyle) will produce the most efficient dataset to train an ML bot on and will hopefully help us understand what specific nuances of each dataset contributed to the performance of a bot specifically trained to play the game of Schnapsen.

## 2 Background Information

### 2.1 The Game: Schnapsen

Schnapsen is a two-player, turn-based, trick-taking card game. The primary objective is to get to 66 points before your opponent. Points are earned by winning tricks or declaring marriages. The cards are valued as such: Ace = 11, 10 = 10, King = 4, Queen = 3, Jack = 2. A trump suit is designated, and each player starts with 5 cards, with one player leading the initial card. The remaining cards constitute the stock, progressing through two phases: an open stock phase and a closed/empty stock phase. During their turn, a player can choose to change their jack trump card for the one which was designated at the start. This is called a trump exchange. The leading player also has the ability to declare a marriage if they hold both the king and queen of the same suit. They can win 40 points for a trump marriage and 20 for a regular one. During a trick, the highest-value card that is led wins if no trump card is played; otherwise, the highest trump card wins the trick. If your opponent won no points you get 3 points, if they won less than 33 you get 2 points, and if they got more than 33 points or you won the last trick, you get 1 point. [3]

### 2.2 The Opponents: Prewritten bots

We have selected two of the prewritten bots from the library to use as opponents for the bots we will create. We think that the two chosen opponents will work great because they function in two very different ways. Let's now have a brief look at the selected bots and their inner workings.

#### 2.2.1 RandBot

This bot's logic is as simple as it seems. This bot selects a random move from the available legal moves. Moves that are considered legal, are those that are possible to make in the current game state, taking into account the cards in our hand, the possibility of a marriage or trump exchange.

#### 2.2.2 RdeepBot

The Rdeep bot uses the PIMC(Perfect Information Monte Carlo sampling) method which works a lot like Alpha-Beta pruning or the Minimax algorithm but it does not require perfect information. It uses this algorithm because, in the first phase of the game, the bots have to rely on imperfect information, contrary to the second phase, where the Alpha-Beta pruning or Minimax can be used. PIMC works by selecting random samples and playing them further, reducing the imperfect information into perfect information scenarios. We have used 12 samples and a search depth of 6 for our experiment making our Rdeep bot rather strong. This algorithm is very heavy in the computing power department, so the tournaments took quite a long time to run.

### 2.3 The Tech: What we used

#### 2.3.1 Coding

For all the coding we have used Python 3.10 mixed with some Batch to trivialize running the experiments and training the models. We have also utilized the SKLearn Python library[1], built on NumPy, matplotlib and SciPy, to help us create our machine-learning models and process the datasets. We have used the MLPClassifier class to make our machine-learning models and used the library to help with the fitting. We have used the ReLu activation function as [1] showed that this was way more effective in these types of scenarios than the sigmoid activation function. The hyper-parameters for the MLPClassifier are the following: we have used 2 hidden layers the first having 64 and the second one having 32 nodes, these are rather small compared to what some other researchers out there, like [1], however, we stuck with these numbers as we had to finish training the models in a reasonable time frame.

#### 2.3.2 Schnapsen Library

We have also utilized the Schnapsen library[2] provided for us. We had to modify it slightly to accommodate our needs, however it is mostly the same. However, need to point out that the stock closing mechanic is not implemented in this

[1]https://scikit-learn.org/stable/
[2]https://github.com/intelligent-systems-course/schnapsen

library, but it is a game mechanic in Schnapsen, therefore we won't talk about this mechanic anymore, however, please pay attention that our bots are not equipped to play with this element of the game.

## 3 Research Question

In this research paper, we aim to answer the question: What strategy produces the best dataset to train an ML Schnapsen bot on? Our strategies are discussed below. By playing these bots against each other and the random bot we aim to see if a more passive or aggressive (or a combination of the two) strategy is best for training an ML bot. This will help us understand which strategy works specifically for Schnapsen, and will hopefully shed light on why a certain dataset is more efficient than another.

We theorise that p2a2(PassBot2 playing against AgrBot2, which will be detailed later) would be the best because they represent two complex strategies from each play style. Using sophisticated strategies for their respective style (passive or aggressive). We reason that a combination of these two will grant the bot the most holistic understanding of the game and therefore be the best-performing bot.

## 4 Experimental Setup

### 4.1 Methodology

To test the strength of each of the strategies the initial test was to run each of the bots against RDeep. Each of the bots played one thousand games against the RDeep bot. This will allow us to gauge the strength of each bot and to see which strategy performs best in its raw state. We will then set each of the bots against themselves and create a dataset by letting them play fifty thousand games. This essentially creates datasets which are representative of each of our five strategies. These datasets are then used to train our ML bot. Each dataset is used to train the ML bot. By the end of this training phase, we are left with five bots powered by datasets which represent each of the strategies. These five ML-powered bots are then each set against RDeep for another round of a thousand games. By analyzing the result of this tournament we can compare the amount of points won by the raw bot, and then the ML bot and see if there was a performance improvement. These two tests essentially allow us to quantify the performance of each strategy against the performance of RDeep. The third test will have each bot play against each other fifty thousand times. The datasets produced by this will then be used to train ten more bots which represent the ten different combinations of strategies. This will allow us to see if combining strategies can help the performance of an ML bot and which specific strategy is most compatible with another strategy. This helps us understand how the dataset influences the performance of each bot and what strategy or combination of strategies has the most influence on performance.

Our independent variable for each of these experiments is essentially the strategy. At first, it is represented through the algorithm written for each bot. Each bot has a specific strategy which we are interested in testing the efficiency of. As we move through the tests however our independent variable changes to the dataset produced by running the bot against itself and then the other bots. Essentially, the dataset is a binary representation of the strategy which will be the factor which changes each time the ML bot is trained. Our dependent variable is the amount of points won, and the difference between the points won by RDeep and our bot. As we only want to investigate the influence of the different datasets we must ensure that certain variables are controlled. The algorithm to train the bot and the hyper-parameters of the neural network must always stay the same. Changing this algorithm would mean that the training is different each time which would result in bias. It is also important that we keep the feature selection the same so that the only influence on the playstyle is the strategy we implemented. We also made sure that in each of the training phases the dataset collected was always from fifty thousand games and each time a bot played against RDeep it also only played one thousand times. The configuration of RDeep was also always kept the same and for example, the depth of sampling, and number of samples taken, remained uniform for each of the games and bots played against.

We will conduct five experiments. In all of which all the created agents will play 1000 games against the bot RDeep and Random bot, unless otherwise specified.

## 4.2 Recording Points/Number of Wins

As winning in Schnapsen is represented by wins over a tournament we have decided that to accurately record which bot is the strongest instead of recording wins we would record the points won compared to the opponent. This way we can better quantify the strength of the bots and understand by what margin they have won. It allows us to check how strong or weak a bot is more accurately. A bot could lose games but still be a strong bot because we would not know if it lost by a small or big margin. Analyzing the points or the difference between the winners' points and losers' points allows for a more nuanced analysis of strength than the win/loss ratios would have provided us. We had to modify the base library to achieve our goal of recording the points for training our ML algorithm.

## 4.3 The Playing Agents

### 4.3.1 PassBot1

This bot is the simplest out of the four algorithms we wrote. It uses a very basic, passive strategy to play. This bot always chooses the smallest value non-trump card from its hand and plays it. It does not care either about exchanging trumps or about playing its marriages. This bot aims to simulate a player who tries to establish trump control and goes for long-term gains, instead of short-term points. The logic of this was sampled from [5].

### 4.3.2 PassBot2

A part of this strategy is based on a passage from a Schnapsen play guide book: Martin Tompa: Winning Schnapsen. If the bot is in the lead for the current trick it uses the Uncle Tibor strategy from the book. "Whenever Tibor was on lead, he would lead out a nontrump ace or ten, a play unthinkable in my family. I would trump, gleefully collecting all those points in my tricks, and return a jack. The result was that Tibor nearly always had trump control when the stock was exhausted (or he closed the stock) because I had used most or all of my trumps to capture his aces and tens. Because of this trump control, Tibor won nearly every deal."[4] So if the bot is on lead it tries to play non-trump tens or aces - baiting the opponent into taking it with a trump to establish trump control later in the game. Whereas when it is following it lists all the moves that would beat the leader's card, excluding trump card moves of course because that would defeat the purpose of the strategy's essence, and the lowest value cards it has, then it chooses from those moves randomly.

### 4.3.3 AgrBot1

This bot implements a simple aggressive strategy. If this bot leads the current trick it plays the highest card. Highest in this case means highest rank and if it has any trump card, that takes precedence. However, if this bot is following a trick, it will try to win the trick, if it can't it will play the lowest card in its hand. If in the lead it will pay attention to trump exchange chances and marriages. The strategy used for this simple aggressive bot was sampled from [4,5]

### 4.3.4 AgrBot2

This bot is an advanced version of AgrBot1, as it usually acts very similar to it, but as you will see it implements more advanced strategic moves that professional Schnapsen players use. Firstly it tries to avoid playing Queens or Kings for which there is still a possibility to get the other one for a marriage. Marriages are a huge bump in points so it is worth it to optimize for them, especially for a royal one. The next advanced strategy this bot implements is that it tries to avoid winning the last trick in the first phase to get the trump card (usually Jack) at the bottom of the talon - it does this only if the opponent is not close to 66 points to avoid giving them a free win. Besides these implemented strategies it acts the same way as AgrBot1 would act. The strategies chosen for this bot were implemented from the description in [2,3]

### 4.3.5 RandBot

This bot was discussed earlier in the Background Information section.

### 4.3.6 MLPlayingBot

The setup and hyper-parameters for the training of this bot were discussed earlier in the Background Information section above. This bot works by passing the current state into the generated model, getting back six percentage chances for each of the possible outcomes of the game (lose by 3, 2, 1 points or win by 1, 2, 3). From these percentages available for each possible move we have to decide which one to choose. Most examples we have seen [5,6,7] only took into account the win-chances to maximize the chance of winning the game, however, we think that for our approach we should also take into account the loss-chances as well to minimize the chance of possibly losing as well as maximize are chances to win the game. We settled on this formula to decide which options would be best:

$$p(WinBy1) * 0.8 + p(WinBy2) + p(WinBy3) * 1.2 -$$
$$- p(LoseBy3) * 0.5 - p(LoseBy2) * 0.4 - p(LoseBy1) * 0.3$$

This formula is completely arbitrary and based on intuition, and requires further testing to find the best values for this formula, however, we stuck with these numbers throughout our experiments.

## 4.4 First experiment

In the first experiment, we will be running the bots written by us against RDeep to have a baseline to see how the ML bots compare later. It also helps us understand what strategies work the best, in a game of Schnapsen. We will create 4 additional bots 2 of which are going to adopt a more passive playstyle while the other 2 will have a more aggressive playstyle.

## 4.5 Second experiment

In the second experiment, we will be running our bots against themselves 50000 times, gathering data from that which will serve as a dataset for our ML algorithm for training. After the training, we will put these bots against the RDeep bot as well to compare how much they have improved when encapsulating their core strategies into a dataset and feeding it into an ML algorithm. This will also show how important it is which kind of algorithm's dataset an ML is trained on because it basically will still use the same algorithm, but it will do better than just the algorithms themselves. We will test them by making them play against RDeep. In this experiment, we will create five datasets from our five playing agents playing against themselves, and train 5 ML models on them.

## 4.6 Third experiment

In the third experiment, we will do a very similar training element to the second experiment, however, in this experiment, the dataset generated for the ML is going to be created by putting every bot against every other bot in 50000 rounds of Schnapsen. In this way, we hope to get a mix of different strategies incorporated into a model accelerating the performance of the bots, compared to the previous experiment. We will also see which mix of strategies works the best. We hypothesize that the bot that will do the best is the mix of the two strongest algorithms from opposite sides of the spectrum, a mix of PassBot2 and AgrBot2. We will test them by making them play against RDeep. In this experiment, we will create 10 more datasets from the remaining combination of bots playing against each other, generating a dataset for our ML algorithm to train on.

### 4.7 Fourth experiment

In this experiment, we will try to chase the numbers and make our agents stronger by changing their second-phase behaviour. We will replace all of our trained models' second phase to play with the Alpha-Beta pruning technique. We will test them by making them play against RDeep. In this experiment, we will modify all of our previous ML-powered agents.

### 4.8 Fifth experiment

In this experiment, we will make all our previously created agents (35) fight against RandBot so that we can see just how good they are. This experiment is interesting because in this way we can compare all of our previously created agents to each other and draw conclusions from that.

## 5 Results and Conclusions

### 5.1 First Experiment

The first graph allows us to see which bot in its natural state is the strongest. According to this graph, the aggressive strategies are stronger when played against RDeep for 1000 games. However, all of these bots, even if the stronger strategy is clear, are incredibly weak compared to RDeep. This is because they are restricted in their choices and will always follow the same strategy blindly. Therefore, when competing with RDeep which has access to a more informed choice they will always lose. The aggressive bots being better is a surprise and contradicts both our hypothesis and the findings presented later. Considering the findings of the other experiments we are still unsure of why this is the case.

### 5.2 Second Experiment

The second experiment saw the ML bots be trained on the dataset where each of the strategy bots was trained against itself. Here the passive bots were stronger, but not by a substantial amount. This is interesting to note as after the training phase the passive strategy is more optimal. We speculated this is because throughout the training phase, the aggressive bot is more prone to taking risks which resulted in more unfavourable game states. Therefore,

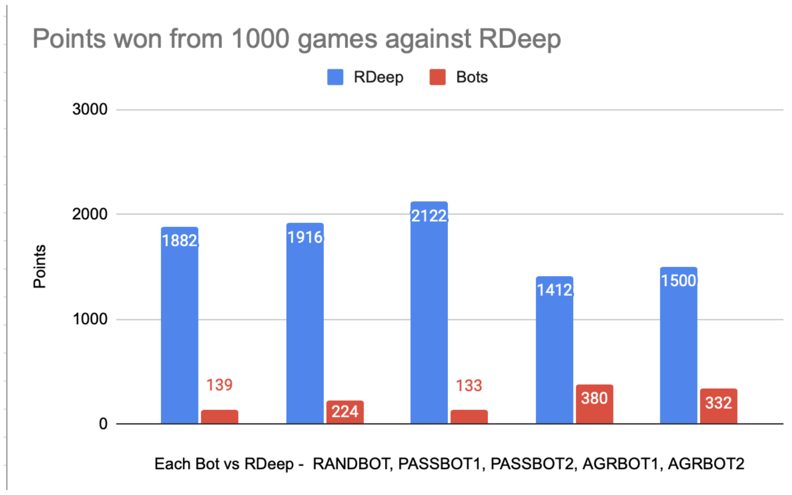

Figure 1: The results of making the base bots fight against RDeep in a 1000-game tournament

when training the ML bot on this dataset there were more instances where the possibility of losing or being in a losing position was higher and the correct choice to be passive or less aggressive was not available. It is interesting how close the points are (except for the random bot) between all of the bots and therefore, strengthens our hypothesis that a combination of opposite strategies might be the most efficient and optimal dataset for a Schnapsen bot. This is because a bot which combines the two has access to the greatest diversity of game states that it has 'experienced'. This versatility in strategy would logically be beneficial.

## 5.3 Third Experiment

As we can see from Figure 3. the two bots that did considerably better are A2R and P1P2. We can see this by counting their win differences. This is a very unexpected but interesting finding. P1P2 did well because the passive bots are not that prone to make risky moves and the mix of them should do way better than other bots, this finding was expected by us and confirms the findings in [5] that a relatively balanced playstyle skewed towards passive playing does the best in the game of Schnapsen. However A2R doing so well was not at all expected, we have to look for reasons in the bots, firstly it has a Randbot in it which makes every training better because of the large array of moves it explores this is confirmed by Experiment 2. However, it has an A2 in it as well, how did this happen? Well A2 implements the most advanced techniques used by pros in tournaments. So this has to be the reason why it did so well, it made the ML teach itself these more advanced concepts while not being hindered by another strategy on the R's part. It seems like the optimal model would come

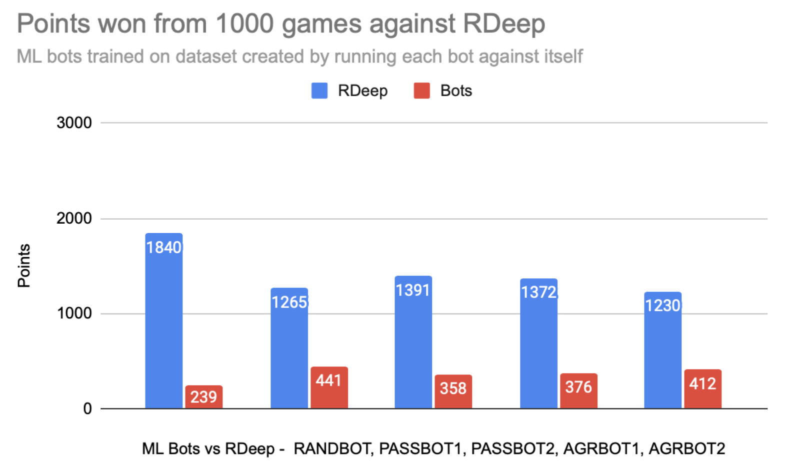

Figure 2: The results of making the ML bots trained on the matches of base bots against themselves fight against RDeep in a 1000-game tournament

from a bot that implements most of the advanced strategic moves used by pros fighting against an R so that it has room to explore a lot of states as well.

## 5.4 Fourth Experiment

In this experiment, we can see how adding Alpha-Beta pruning to the second phase of the ML models makes everything pop. It made every single bot way better. It is interesting to see how it upgraded the A1A2, and other bots using any A bots inside them more than the ones having only P bots inside them. This could be because A bots usually run out of great cards right around the start of the second phase and a much more tactical play provided by Alpha-Beta pruning is necessary to get through and not lose instantly, while the P bots usually keep their most valuable cards till the endgame and with valuable cards you do not need perfect knowledge to win. This, however, did not stop the ML-bot trained on P1P2 from absolutely devastating everyone, this is expected and is because of reasons mentioned in the Third experiment. Close runners-up are P2P2, A2R, RR. The RR is explained by earlier chapters and is an expected outcome of this experiment, we have already explained why A2R did so well in the Third Experiment and this experiment just makes it way better. It is however interesting to see P2P2 here, it has not done so well in the Second experiment, but as we can see Alpha-Beta pruning fixed this bot's late-game shortcomings and made this bot be very good.

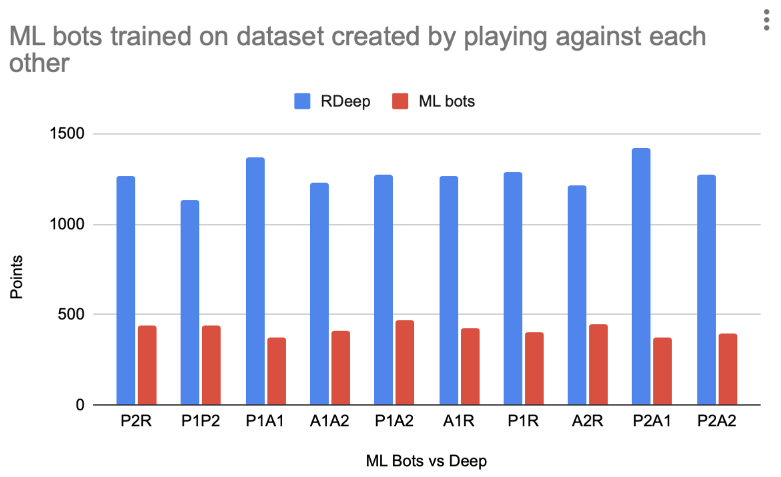

Figure 3: The results of making the ML bots trained on the matches of base bots against every other base bot fight against RDeep in a 1000-game tournament

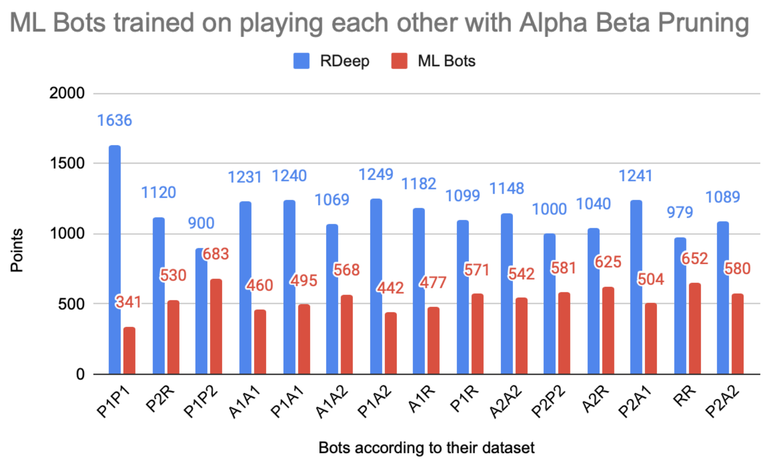

Figure 4: The results of making all the ML bots - second phase replaced by Alpha-Beta Pruning fight against RDeep in a 1000-game tournament

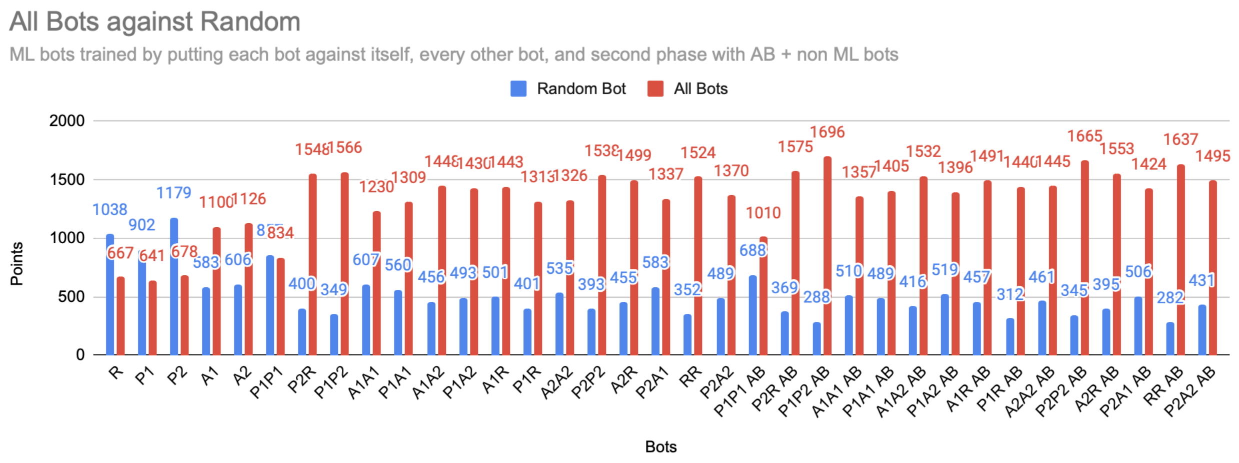

Figure 5: The results of making all the bots fight against RandBot in a 1000-game tournament

## 5.5 Fifth Experiment

We decided to run all of the bots against the random bot as we thought it would be representative of a neutral strength metric. We really wanted to see if alpha-beta pruning would significantly alter the performance (as it would dominate in phase 2 of the game where the information is perfect). Albeit it had a noticeable effect and did increase the performance, this did not occur with an overwhelming effect. The strongest bots were still the ones with the passive-focused strategies. This is interesting as it shows that in Schnapsen the strategy is slightly skewed towards waiting and playing tactically, rather than attempting to attack and control the game from the start. Due to the structure of the game, and the limited number of cards, we did theorize that if you made it to the middle of the game and have played it passively your chances of winning are much higher than if you dominated the beginning and will now lose due to marriages and trump exchanges (because you played your best cards already). It takes a lot of luck to dominate so substantially from the beginning that you win in the first third of the game, therefore we concluded that playing passively is more efficient in the long term. As you can see the P1P2 AB, and P2P2 AB are the leaders in points.

## 6 Suggestions For Further Research

In this last section, we will discuss a lot of already mentioned ideas, that would be very interesting to research in the future. First off, there are many more strategies to use for dataset creation in the future, we have implemented a lot, but because of the time constraints, we did not have the capacity to do even more. Our second idea is that, of course, we could have also experimented with the fine-tuning of the hyper-parameters of our ML training algorithm, and that would have improved the strength of our bots tremendously, even just using bigger and more hidden layers would have helped, however, that would have taken so much time to train that we couldn't have finished the paper on time. This also applies to the number of games played for both the dataset generation as well as the tournaments. Maybe the most interesting formula to tweak is the formula used for deciding which legal move to play based on the chances given by the model for the MLPlayingBot, as this could have a huge effect on how our bots play. The paper focused on how the dataset affects the effectiveness of ML-powered agents, and we only explored one side of this problem: the strategies used to generate the dataset, however in the future as a lot of others did before us, we would like to delve into feature-selection ourselves as well, combining it with the exploration of different strategies could result in a very comprehensive understanding of the Schnapsen card game Also would like to create the optimal dataset and model mentioned in the Results and Conclusions section under Third Experiment, by making a bot that implements the most advanced strategies used by pros and combining it with a RandBot.

## References

1. Tobias S.:Reinforcement Learning for Games with Imperfect Information, `https://web.archive.org/web/20230203115840id\_/https://repositum.tuwien.at/bitstream/20.500.12708/148083/1/Salzer\%20Tobias\%20-\%202023\%20-\%20Reinforcement\%20Learning\%20for\%20Games\%20with\%20Imperfect...pdf`
2. Schnapsen Strategy Guide, `http://schnapsenstrategy.blogspot.com/2010/.`
3. Tompa, M.: Winning Strategy for Schnapsen or Sixty-Six, `http://psellos.com/schnapsen/strategy.html.`
4. Tompa, M.: Importance of Trump Control in Schnapsen, `https://psellos.com/schnapsen/strategy.html#trumpcontrol`
5. Noortje ten Wolde, Gulizar Taskopru and Idries Nasim. IS 2020.
6. Seamus Mol, Gal Cohen and Yingdi Xie. IS 2020
7. Ioana Teaca, Vincent van de Langenberg and Jacob Gebreegziabeher. IS 2018

# Appendices

## A Code for the four bots: PassBot1, PassBot2, AgrBot1, AgrBot2

```python
from schnapsen.game import Bot, PlayerPerspective, SchnapsenDeckGenerator, Move, Trick, GamePhase, RegularMove, Card
from typing import Optional, cast, Literal, List, Tuple
from schnapsen.deck import Suit, Rank

import pathlib
import time

import random

class PassBot1(Bot):
    '''Really basic passive strategy
    '''
    def __init__(self, rand : random.Random, name : Optional[str] = None):
        self.rand = rand
        super().__init__(name)

    def get_move(self, perspective: PlayerPerspective, leader_move: Move | None) -> Move:
        valid_moves = perspective.valid_moves()
        trump_suit = perspective.get_trump_suit()

        chosen_move : Move = self.rand.choice(valid_moves)

        for move in valid_moves:
            #This bot does not care about making moves that arent reguler like trump exchange or marriages
            if not move.is_regular_move():
                continue
            else:
                #tries to find the smallest possible nontrump card available and play it
                if  rankIsSmaller(move.cards[0].rank, chosen_move.cards[0].rank):
                    if move.cards[0].suit == trump_suit:
                        continue
                    elif move.cards[0].suit != trump_suit:
                        chosen_move = move

        #print(valid_moves, chosen_move, trump_suit)

        return chosen_move

class PassBot2(Bot):
    '''Uncle Tibor strategy - Martin Tompa: Winning Schanpsen
    https://psellos.com/schnapsen/strategy.html#trumpcontrol
    If not leading acts similar AgrBot1
    '''
    def __init__(self, rand : random.Random, name : Optional[str] = None):
        self.rand = rand
        super().__init__(name)

    def get_move(self, perspective: PlayerPerspective, leader_move: Move | None) -> Move:
        valid_moves = perspective.valid_moves()
        trump_suit = perspective.get_trump_suit()
        is_lead = perspective.am_i_leader()

        leading_m : Move
        if leader_move:
            leading_m = leader_move

        chosen_move = self.rand.choice(valid_moves)

        #If leading try to play trump exchange or marriage
        if is_lead:
            for move in valid_moves:
                if move.is_trump_exchange():
                    chosen_move = move
                    break
                if move.is_marriage():
                    chosen_move = move
                    break
                #If no special moves play nontrump tens or aces to bait out trumps - else play jack nontrump
                if move.is_regular_move():
                    if move.cards[0].suit is not trump_suit:
                        if move.cards[0].rank == Rank.TEN or move.cards[0].rank ==  Rank.ACE:
                            chosen_move = move
                            break
                        elif move.cards[0].rank == Rank.JACK:
                            chosen_move = move
                    #if no good option we go with something randomly
                    else:
                        chosen_move = move
        else:
            not_dumb_moves : List[Move] = []

            for move in valid_moves:
                #gather all possible moves that arent dumb regarding our strategy
                if move.is_regular_move():
                        #a move in this case is not dumb is it can beat the enemy and its not trump
                        if (move.cards[0].suit == leading_m.cards[0].suit or \
                            (move.cards[0].suit != trump_suit and leading_m.cards[0].suit != trump_suit)) \
                                and rankIsSmaller(leading_m.cards[0].rank, move.cards[0].rank):
                            not_dumb_moves.append(move)
                        #else we just want to play any non trump
    card of our own
                        else:
                            if move.cards[0].suit == trump_suit:
                                continue
                            elif move.cards[0].suit != trump_suit:
                                not_dumb_moves.append(move)

            #randomly selecting from best seeming moves
            if len(not_dumb_moves) != 0:
                chosen_move = self.rand.choice(not_dumb_moves)

        return chosen_move

class AgrBot1(Bot):
    '''Really basic agressive strategy
    If leading play highest card - if following and cant win play
    lowest
    '''
    def __init__(self, rand : random.Random, name : Optional[str] =
     None):
        self.rand = rand
        super().__init__(name)

    def get_move(self, perspective: PlayerPerspective, leader_move:
     Move | None) -> Move:
        valid_moves = perspective.valid_moves()
        trump_suit = perspective.get_trump_suit()
        is_lead = perspective.am_i_leader()

        chosen_move = self.rand.choice(valid_moves)

        leading_m : Move
        if leader_move:
            leading_m = leader_move

        for move in valid_moves:
            #if leading checks for special moves if none found...
            if is_lead:
                if move.is_trump_exchange():
                    chosen_move = move
                    break
                if move.is_marriage():
                    chosen_move = move
                    break
                #tries to play the highest card it has regarding
    rank and trumpness
                if move.is_regular_move():
                    if move.cards[0].suit is trump_suit:
                        if  rankIsSmaller(chosen_move.cards[0].rank
    , move.cards[0].rank):
                            chosen_move = move
                    elif chosen_move.cards[0].suit is not
    trump_suit:
                        if  rankIsSmaller(chosen_move.cards[0].rank
    , move.cards[0].rank):
                            chosen_move = move
```

```python
            else:
                #if following see if it can win the trick if yes play that if not plays smallest card
                if move.is_regular_move():
                    if (move.cards[0].suit == leading_m.cards[0].suit or \
                        (move.cards[0].suit == trump_suit and leading_m.cards[0].suit != trump_suit)) \
                            and rankIsSmaller(leading_m.cards[0].rank, move.cards[0].rank):
                        chosen_move = move
                        break
                    else:
                        if rankIsSmaller(move.cards[0].rank, chosen_move.cards[0].rank):
                            if move.cards[0].suit == trump_suit:
                                continue
                            elif move.cards[0].suit != trump_suit:
                                chosen_move = move

        return chosen_move

class AgrBot2(Bot):
    '''Tries not to play kings and queens if theres still a chance for a marriage, tries to lose last
    trick of first phase for the extra trump - otherwise plays like agrbot1
    '''
    def __init__(self, rand : random.Random, name : Optional[str] = None):
        self.rng = rand
        super().__init__(name)

    def get_move(self, perspective: PlayerPerspective, leader_move: Move | None) -> Move:
        valid_moves = perspective.valid_moves()
        trump_suit = perspective.get_trump_suit()
        is_lead = perspective.am_i_leader()

        chosen_move = self.rng.choice(valid_moves)

        leading_m : Move
        if leader_move:
            leading_m = leader_move

        for move in valid_moves:
            #if on last trick of phase one tries to lose the trick to get the extra trump
            if perspective.get_talon_size() == 2:
                if is_lead:
                    if move.is_regular_move():
                        if rankIsSmaller(move.cards[0].rank, chosen_move.cards[0].rank):
                            chosen_move == move
                else:
                    if move.is_regular_move():
```

```python
                    if rankIsSmaller(move.cards[0].rank, leading_m.cards[0].rank) and \
                          rankIsSmaller(move.cards[0].rank, chosen_move.cards[0].rank) and \
                            ((move.cards[0].suit == trump_suit and leading_m.cards[0].suit == trump_suit) or \
                            move.cards[0].suit != trump_suit and leading_m.cards[0].suit != trump_suit):
                        chosen_move == move

        #if leading tries to play special moves
        elif is_lead:
            if move.is_trump_exchange():
                chosen_move = move
                break
            if move.is_marriage():
                chosen_move = move
                break
            #and tries to find the biggest trump card to play while avoiding playing parts of possible marriages
            if move.is_regular_move() and rankIsSmaller(chosen_move.cards[0].rank, move.cards[0].rank):
                if move.cards[0].suit is trump_suit:
                    if move.cards[0].rank == Rank.QUEEN and \
                        Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                        Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                        continue
                    if move.cards[0].rank == Rank.KING and \
                        Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                        Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                        continue
                    chosen_move = move
                elif chosen_move.cards[0].suit is not trump_suit:
                    if move.cards[0].rank == Rank.QUEEN and \
                        Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                        Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                        continue
                    if move.cards[0].rank == Rank.KING and \
                        Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                        Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                        continue
                    chosen_move = move
        else:
            #if following tries to beat enemy if cant it plays the lowest card, while trying to avoid playing parts of
            #still possible marriages
```

```python
                if move.is_regular_move():
                    if (move.cards[0].suit == leading_m.cards[0].suit or \
                        (move.cards[0].suit == trump_suit and leading_m.cards[0].suit != trump_suit)) \
                            and rankIsSmaller(leading_m.cards[0].rank, move.cards[0].rank):
                        if move.cards[0].rank == Rank.QUEEN and \
                            Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                            Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                            continue
                        if move.cards[0].rank == Rank.KING and \
                            Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                            Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                            continue
                        chosen_move = move
                    else:
                        if rankIsSmaller(move.cards[0].rank, chosen_move.cards[0].rank):
                            if move.cards[0].rank == Rank.QUEEN and \
                                Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                                Card.get_card(Rank.KING, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                                continue
                            if move.cards[0].rank == Rank.KING and \
                                Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.get_hand().get_cards() and \
                                Card.get_card(Rank.QUEEN, move.cards[0].suit) not in perspective.seen_cards(leader_move).get_cards():
                                continue
                            if move.cards[0].suit == trump_suit:
                                continue
                            elif move.cards[0].suit != trump_suit:
                                chosen_move = move

        return chosen_move

#This function determines if rank1 is smaller in rank than rank2
def rankIsSmaller(rank1 : Rank, rank2 : Rank) -> bool:
    if  (rank1 == Rank.JACK and (rank2 == Rank.QUEEN or rank2 == Rank.KING or rank2 == Rank.TEN or rank2 == Rank.ACE)) or \
        (rank1 == Rank.QUEEN and (rank2 == Rank.KING or rank2 == Rank.TEN or rank2 == Rank.ACE)) or \
        (rank1 == Rank.KING and (rank2 == Rank.TEN or rank2 == Rank.ACE)) or \
        (rank1 == Rank.TEN and (rank2 == Rank.ACE)):
        return True
    return False
```

## B Code for generating the datasets and training the ML model

```python
import pickle
import os.path
from argparse import ArgumentParser
import time
import sys
# This package contains various machine learning algorithms
import sklearn
import sklearn.linear_model
from sklearn.neural_network import MLPClassifier
import joblib

from schnapsen.game import PlayerPerspective, SchnapsenGamePlayEngine, GameState
from schnapsen.game import *

from schnapsen.bots.rand import RandBot
from schnapsen.bots.plusbots import PassBot1, PassBot2, AgrBot1, AgrBot2
from schnapsen.bots.rdeep import RdeepBot
from schnapsen.bots.ml_bot import create_state_and_actions_vector_representation, get_state_feature_vector

rand = Random()

def create_dataset(path, player=RandBot(rand), player2=RandBot(rand), games=2000):
    data = []
    target = []

    # For progress bar
    bar_length = 30
    start = time.time()

    for g in range(games-1):
        # For progress bar - easily track prgress!!! Very important to have
        if g % 10 == 0:
            percent = 100.0*g/games
            sys.stdout.write('\r')
            sys.stdout.write("Generating dataset: [{:{}}] {:>3}%".format('='*int(percent/(100.0/bar_length)),bar_length,
            int(percent)))
            sys.stdout.flush()

        # Randomly generate a state object starting in specified phase.
        engine = SchnapsenGamePlayEngine()

        #Play a game of schnapsen
        winner_persp, score, loser_persp, winBot = engine.play_game(player, player2, rand)

        #extract all explored states of the game
        winner_persp = winner_persp.get_game_history()[:-1]
        loser_persp = loser_persp.get_game_history()[:-1]

        #MLBOT
        #Loop through both perspectives because its free data so that we functionally played 100000 games not 50000
        for p in (winner_persp, loser_persp):
            #loop through all perspectives of them gam and one hot encode them then append them to the data/target lists
            for round_player_perspective, round_trick in p:

                if round_trick.is_trump_exchange():
                    leader_move = round_trick.exchange
                    follower_move = None
                else:
                    leader_move = round_trick.leader_move
                    follower_move = round_trick.follower_move

                # we do not want this representation to include actions that followed. So if this agent was the leader, we ignore the followers move
                if round_player_perspective.am_i_leader():
                    follower_move = None

                #generating state vectors
                state_actions_representation = create_state_and_actions_vector_representation(
                    perspective=round_player_perspective, leader_move=leader_move, follower_move=follower_move)

                data.append(state_actions_representation)
                target.append(score if p == winner_persp else -score)#'win' if player == winner_persp else 'loss')
        #MLBOT

    #dumping into file
    with open(os.getcwd() + path, 'wb') as output:
        pickle.dump((data, target), output, pickle.HIGHEST_PROTOCOL)

    # For printing newline after progress bar
    print("\nDone. Time to generate dataset: {:.2f} seconds".format(time.time() - start))
    return data, target

## Parse the command line options
parser = ArgumentParser()
#These options can be used to call the script through
parser.add_argument("-d", "--dset-path",
                    dest="dset_path",
                    help="Optional dataset path",
                    default="dataset.pkl")
parser.add_argument("-m", "--model-path",
                    dest="model_path",
                    help="Optional model path. Note that this path starts in bots/ml/ instead of the base folder, like dset_path
```

```python
above.",
                    default="model.pkl")
parser.add_argument("-o", "--overwrite",
                    dest="overwrite",
                    action="store_true",
                    help="Whether to create a new dataset regardless of whether one already exists at the specified path.")
parser.add_argument("--no-train",
                    dest="train",
                    action="store_false",
                    help="Don't train a model after generating dataset.")
parser.add_argument("-p1", "--player1",
                    dest="player1",
                    help="Specify the bot1 used to train the model (rand, pass1, pass2, agr1, agr2)",
                    choices=['rand', 'pass1', 'pass2', 'agr1', 'agr2', 'rdeep'],
                    default="rand")
parser.add_argument("-p2", "--player2",
                    dest="player2",
                    help="Specify the bot2 used to train the model (rand, pass1, pass2, agr1, agr2)",
                    choices=['rand', 'pass1', 'pass2', 'agr1', 'agr2', 'rdeep'],
                    default="rand")
options = parser.parse_args()

#setting up the two bots that play againts each other based on the info recieved through cmd
if options.overwrite or not os.path.isfile(options.dset_path):
    random = Random()

    p1 : Bot
    p2 : Bot

    if options.player1 == 'pass1':
        p1 = PassBot1(random)
    elif options.player1 == 'pass2':
        p1 = PassBot2(random)
    elif options.player1 == 'agr1':
        p1 = AgrBot1(random)
    elif options.player1 == 'agr2':
        p1 = AgrBot2(random)
    elif options.player1 == 'rand':
        p1 = RandBot(random)
    elif options.player1 == 'rdeep':
        p1 = RdeepBot(12,6,random)

    if options.player2 == 'pass1':
        p2 = PassBot1(random)
    elif options.player2 == 'pass2':
        p2 = PassBot2(random)
    elif options.player2 == 'agr1':
        p2 = AgrBot1(random)
    elif options.player2 == 'agr2':
        p2 = AgrBot2(random)
    elif options.player2 == 'rand':
        p2 = RandBot(random)
    elif options.player2 == 'rdeep':
        p2 = RdeepBot(12,6,random)

    #calling the data generation function with the given variables and making them play 50000 games agants each other
    create_dataset(options.dset_path, player=p1, player2=p2, games=50000) #15000

if options.train:
    # Play around with the model parameters below
    # HINT: Use tournament fast mode (-f flag) to quickly test your different models.
    # The following tuple specifies the number of hidden layers in the neural
    # network, as well as the number of layers, implicitly through its length.
    # You can set any number of hidden layers, even just one. Experiment and see what works.
    hidden_layer_sizes = (64, 32)
    # The learning rate determines how fast we move towards the optimal solution.
    # A low learning rate will converge slowly, but a large one might overshoot.
    learning_rate = 0.0001
    # The regularization term aims to prevent overfitting, and we can tweak its strength here.
    regularization_strength = 0.0001

    #############################################

    start = time.time()

    print("Starting training phase...")

    with open(os.getcwd() + options.dset_path, 'rb') as output:
        data, target = pickle.load(output)

    # Train a neural network
    learner = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes, learning_rate_init=learning_rate,
    alpha=regularization_strength, verbose=True, early_stopping=True, n_iter_no_change=6)

    # learner = sklearn.linear_model.LogisticRegression()
    model = learner.fit(data, target)

    # Check for class imbalance
    count = {}

    for t in target:
        if t not in count:
            count[t] = 0
        count[t] += 1

    print('instances per class: {}'.format(count))

    # Store the model in the ml directory
    joblib.dump(model, "./models/" + options.model_path)

    end = time.time()

    print('Done. Time to train:', (end-start)/60, 'minutes.')
```