The benefits of an extended feature space in a machine learning bot for the game Schnapsen

Lucas Faijdherbe, Daan Schrage and Ruben van der Ham

Intelligent Systems
Vrije Universiteit Amsterdam

January 25, 2018

Abstract

This paper examines the effect of extended and narrowed feature spaces on logistic regression machine learning bot. To illustrate this, 5 different machine learning bots were created: two bots with narrowed feature spaces, three bots with extended feature spaces. These bots were trained multiple times, and played multiple tournaments against multiple other bots. The results are..

1 Introduction

Artificial Intelligence is becoming a huge subject, and we see it appearing more and more in daily life: voice assistants on smartphones, self-driving Tesla's, AlphaGo defeating Go masters, and so on. within the field, we can make a distinction between strong and weak AI [1]: *weak AI* is focused on mastering problems in one limited area, whereas *strong AI* is considered to be Artificial Intelligence with *actual* intelligence, and even self-awareness. For this paper, we will concentrate on weak AI in the game of Schnapsen. The weak-AI in this game are machine-learning bots. In this paper, we will specifically focus on the feature spaces of these machine-learning bots, and how the differences in features affect performance of the bots. In section 2 we will describe Schnapsen and the Schnapsen framework in detail, alongside the non-AI bots that have already been implemented. The research question will be described in section 3. In section 4, we will talk about the machine-learning (AI) bots, and the research setup. Consequently in section 5 we will describe the results of the experiment, and in section 6 we will elaborate on our findings. Finally, in section 7 we will draw our conclusion.

2 Background Information

2.1 Schnapsen

Schnapsen is an Austrian card game, played with 20 cards; Only the Jacks, Queens, Kings, 10s and Aces are in the deck. The goal of the game is to win tricks, and to be the first player to score 66 points. Each game has a trump suit, which is determined randomly. A card from the trump suit lets you automatically win the trick, provided that there is no higher trump card in the game.

The game has a multi-agent (we have an opponent), deterministic (the environment is determined by the current state and executed action), static (the environment does not change when an agent is choosing an action) and discrete (the game is turn-based) environment. The game consists of two phases: In phase one, when there are still cards in stock, we have an imperfect information game. In phase two, when the stock is depleted, we have a perfect information game, since we know exactly what the cards of our opponent are. The point count in the game is as follows:

J	**Q**	**K**	**10**	**A**
2	3	4	10	11

A lot of points can also be won by marriages. If the player is leading the trick and has a king or a queen of the same suit, it can play one card, and show the other, so the opponent knows that there is a marriage.

Normal marriage (non-trump suit)	**Royal marriage (trump suit)**
20	40

The game starts in phase one, an imperfect-information game. Each agent has 5 cards. Player 1 plays a card, thereby determining the suit that has to be played. To win the trick, player 2 has to play a higher-ranking card from the same suit, or a trump card. Player 2 does not have to follow suit in phase 1, but he/she will lose the trick in that case. The winner of the trick now becomes player one and gets the cards, with the amount of points the two cards represent. This goes on until the stock is depleted; then we enter phase 2, a perfect-information game. Player 2 now has to follow suit. If it cannot do that, the player has to trump, and otherwise play any card. If a player reaches 66 points or more, the game is over.

2.2 Intelligent Systems Schnapsen Framework

An API in Python has been provided, together with a framework for the Schnapsen game. Furthermore, we have two scripts: one where bots can play against each other in a single game where all moves in the game can be seen, and a script where bots play a tournament against each other - with a specifiable number of games to be played - and where the number of won games per robot can be seen. There are already multiple bots present in the framework to play against:

Minimax / Alphabeta: This bot only works in a perfect-information game (phase 2 of Schnapsen). It recursively looks through all the states, and decides what the best move to play will be, provided that the opponent plays optimally, and will try to minimize the outcome of the game for the player.

Bully: Plays by a defined set of rules. If it has a trump card, it will play that. Otherwise, it will play a card of the same suit player 1 played. If it cannot do either of those things, it will play the highest ranking card of any suit.

Rand: Chooses a random move out of all of the legal moves, and plays that.

Rdeep: Functions quite like a Monte-Carlo sampling bot. It looks through multiple states, and averages the heuristics of a state, thereby ranking the moves. It then picks one of the highest ranking moves to play.

KBbot: Logical reasoning bot. The strategy can be defined in propositional logic. The current strategy of the bot is to always play cheap cards (i.e kings, queens, jacks) first.

2.3 Machine Learning Bots

For this paper, we will use machine learning bots. These bots use a logistic regression algorithm to create the model. The model estimates to probability of a certain response based on on or more features. [2] The model can be retrieved by training the bot against another player. The bot will use the model when playing the actual games. For more information on our implementation of the machine learning bots, refer to section 4.

3 Research Question

In our research, we want to find *what feature set will improve the machine learning bot's performance the most in the game of Schnapsen.* With improve the most, we mean as seen from the baseline. There is already a machine learning bot present, which will act as this base line. This bot has, what we will call the *default feature space*- provided by the framework.

The *default feature space* contains the following features:

- Perspective information
- Player 1's point count
- Player 2's point count
- Player 1's pending points
- Player 2's pending points
- The trump suit
- The phase
- The stock size
- The leader
- Whose turn it is currently
- The played card of the opponent

Our goal is to build multiple bots machine learning bots, both with extended feature spaces and narrower feature spaces, and see which bot will improve most in relation to the default machine-learning bot. Our hypothesis is that the bots with an extended feature space will have a better performance the bots with a narrower feature space, and that the extended feature space bots will perform better than the default bot, where the narrow feature space will perform worse. This is because the narrowing of the feature space can cause underfitting, where the model 'cannot capture the underlying structure of the data'[3] increasing the chance of making a wrong prediction. Consequently, extending the feature space can cause overfitting, where the model has more features than the data needs, which can also lead to making the wrong predictions.

4 Experimental setup

For the purpose of our research, we created 5 machine-learning bots accompanying the original machine-learning bot, each competing in multiple 120 game Schnapsen tournaments against each individual non-learning bot. All bots were trained and tested in the same way. Below, we will give an overview of the bots and the changes made with respect to the default machine-learning bot. For the whole overview, refer to appendix A. After that, we will describe the testing procedure.

4.1 The bots

The non- learning opponents were Rand, Bully and Rdeep. The learning bots we created were:

ml_minimal: This bot has the smallest feature space of all: only the perspective and the point count of both player 1 and player 2 as features. We wanted to see if a bot can still perform well, even if it just has minimal amount of information about the game state.

ml_stripped: ml_stripped has almost all the features of the *default feature space* except for both players pending points and trumpsuit id.

ml: This ml bot is provided by the framework and consists of the *default feature space*

ml_enriched: This bot has a slightly extended feature space in comparison to the original ml-bot. The following features were added to the feature space:

- Both player's points squared
- Both player's pending points squared
- Both player's points cubed
- Both player's pending points cubed

These features were added to go more to a design-matrix like feature space. Since the amount of points a player has are important for the game, we chose to focus solely on this.

ml_advanced: This bot has a more extended feature space in addition to the *default feature space*. Added are:

- Difference between points
- Difference between pending points
- Trump card ratio in hand
- High card ratio in hand (10/A's)
- Number of trump cards in hand in phase 2
- Number of cards with same suit as opponents played suit

- Number of points in hand (squared)

This is the bot where we chose to add more of our own thought of features. Point difference was added, since this is a good indicator if a player is leading the game by a high margin. Furthermore, we chose to focus on both trump and high-ranking cards in the hand. The more you have, the better it is. The number of trump cards in phase 2 is also important: If you still got all your trump cards in phase 2 of the game, you have a high chance of beating your opponent. Furthermore the number of cards with same suit as opponents played suit feature was added, alongside the number of points in the player hand squared feature.

ml_combined: This bot has the most extended feature space. It combines the features of ml_enriched and ml_advanced.

4.2 The testing procedure

Each learning bot played a total of 9 tournaments with 3 different models against each non-learning bot. The testing procedure was as follows:

- Training
- 120 Game tournament
- 120 Game tournament
- 120 Game tournament

This procedure was repeated twice to reach the total of 9 tournaments respectively. The testing only occurred between learning and non-learning bots. The learning bots did not compete against each other. In case of Rand and Bully, each training process consisted of 100,000 observations of the (non-learning) opponent. Rdeep was only observed 10,000 times during training, since training this bot takes a substantial amount of time.

5 Results

The tournament results are presented in Table 1.

	rand		**bully**		**rdeep**		**Total**	
	wins	losses	wins	losses	wins	losses	wins	losses
ml_minimal	843	237	837	243	348	732	2028	1212
ml_stripped	736	344	843	237	283	797	1862	1378
ml	866	214	876	204	257	823	1999	1241
ml_enriched	881	119	797	283	284	798	1962	1278
ml_advanced	856	224	808	272	292	788	1956	1284
ml_combined	949	131	759	321	274	806	1982	1258

Table 1: Overview of tournament wins and losses per ml-bot.

	rand	**bully**	**rdeep**	**Total**
ml_minimal	-2,65%	-4,45%	+35,4%	**+1,45%**
ml_stripped	-15%	-3,76%	+10,11%	**-6,85%**
ml_enriched	+1,73%	-9,01%	+10,5%	**-1,85%**
ml_advanced	-1,15%	-7,76%	+13,6%	**-2,15%**
ml_combined	+9,58%	-13,35%	+6,61%	**-0,85%**
Average improvement	**-1,5%**	**-7,67%**	**+15,24%**	**-2,05%**

Table 2: Improvement over ml.

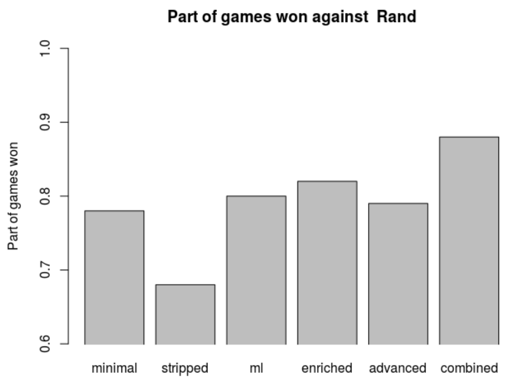

Figure 1: Part of games won against Rand

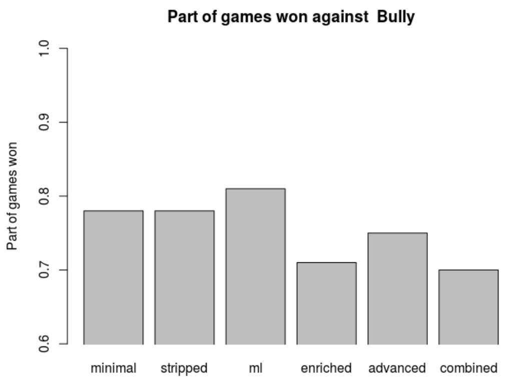

Figure 2: Part of games won against Bully

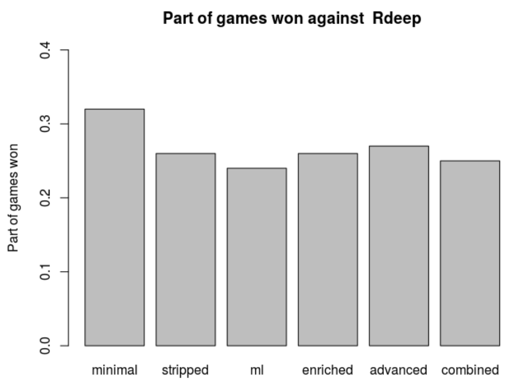

Figure 3: Part of games won against Rdeep (notice the Y-axis)

6 Findings

From the collected data we have x conclusions.

Against Rand

Against Bully

Against Rdeep

First of all, the learning bots performed worse than rdeep. In addition to that, the ml bot performed the worst of all learning bots. The performance of each bot

7 Conclusion

8 References

1. http://humanparagon.com/strong-and-weak-ai/ 2. https://towardsdatascience.com/machine-learning-part-3-logistics-regression-9d890928680f 3. https://en.wikipedia.org/wiki/Overfitting

Appendix

9 Overview of machine-learning bots' feature spaces

9.1 ml_minimal

- Perspective information
- Player 1's point count
- Player 2's point count

9.2 ml_stripped

- Perspective information
- Player 1's point count
- Player 2's point count
- The phase
- The stock size
- The leader
- Whose turn it is currently
- The played card of the opponent

9.3 ml_enriched

- Perspective information
- Player 1's point count
- Player 1's point count squared
- Player 1's point count cubed
- Player 2's point count
- Player 2's point count squared
- Player 2's point count cubed
- Player 1's pending points
- Player 1's pending points squared
- Player 1's pending points cubed
- Player 2's pending points
- Player 2's pending points squared
- Player 2's pending points cubed
- The trump suit
- The phase
- The stock size

- The leader
- Whose turn it is currently
- The played card of the opponent

9.4 ml_advanced

- Perspective information
- Player 1's point count
- Player 2's point count
- Player 1's pending points
- Player 2's pending points
- Difference between player 1 and player 2's point count
- Difference between player 1 and player 2's pending point count
- The trump suit
- Trump card ratio (Number of trump cards in hand / total number of cards in hand * 100)
- High-ranking card ratio (Number of 10's and A's in hand / total number of cards in hand * 100)
- The phase
- Number of trump cards in hand in phase 2
- The stock size
- The leader
- Whose turn it is currently
- The played card of the opponent
- Number of cards in same suit as opponent's played suit in hand.
- Number of points in hand squared ($card1^2$ + $card2^2$ ...)

9.5 ml_combined

- Perspective information
- Player 1's point count
- Player 1's point count squared
- Player 1's point count cubed
- Player 2's point count
- Player 2's point count squared
- Player 2's point count cubed
- Player 1's pending points
- Player 1's pending points squared
- Player 1's pending points cubed
- Player 2's pending points
- Player 2's pending points squared
- Player 2's pending points cubed
- Difference between player 1 and player 2's point count
- Difference between player 1 and player 2's pending point count
- The trump suit
- Trump card ratio (Number of trump cards in hand / total number of cards in hand * 100)
- High-ranking card ratio (Number of 10's and A's in hand / total number of cards in hand * 100)
- The phase
- Number of trump cards in hand in phase 2
- The stock size
- The leader
- Whose turn it is currently
- The played card of the opponent
- Number of cards in same suit as opponent's played suit in hand.
- Number of points in hand squared ($card1^2$ + $card2^2$ ...)

10 Worksheet I

1. The *rdeep* bot performs best, winning 20 out of 30 games.
   Output:

```
Results:
bot <bots.rand.rand.Bot instance at 0x101a70fc8>: 7 wins
bot <bots.bully.bully.Bot instance at 0x101a71098>: 3 wins
bot <bots.rdeep.rdeep.Bot instance at 0x101a71170>: 20 wins
```

2. The bully bot operates as follows:
   - If it has a trump card, it will play that.
   - If it does not have a trump card, it will play a card from the same suit
   - If it cannot do either of those things, it will choose the highest ranking card of any suit.

3. Using this strategy means that you will try to get as many points as possible each move. However, this might also mean that early in the game, the bot will play valuable cards in order to get a high point ratio, but later in the game it has no good cards left.

4. Have the bots complete a large number of tournaments (preferably n ¿ 30), see if the rdeep bot still outperforms the rand bot, and how big the difference in wins is. If there is a significant difference, we can assume that rdeep is better than rand.

5. MyBot implementation: It plays any available trump card first. If none is available it picks a random card.

```
def get_move(self, state):
        moves = state.moves()
        return get_trump_card(moves, state)
def get_trump_card(moves, state):
    randomChoice = random.choice(moves)
    counter = 0
    if randomChoice == None:
        print "NONE FOUND"
    if moves == None:
        print "None available"
        return None
    while True:
        randomChoice = random.choice(moves)
        counter +=1
        if randomChoice == None:
            print "NONE returned"
            return randomChoice
        try:
            if util.get_suit(randomChoice[0])
            == State.get_trump_suit(state):
                #print "Playing trump card"
                return randomChoice
        except:
            #print "KEY ERROR"
            pass
        if counter > 40 and not randomChoice == None:
            #print "Playing normal card"
            return randomChoice
    return randomChoice
```

When no legal moves are to be made except the best trump card you have no legal moves when you remove the trump card from the list of legal moves. In that case the python program expects a move but it gets a return type "None". That's why it raises an error.

Output:

```
Results:
    bot <bots.rand.rand.Bot instance at 0x7fe73586fab8>: 25 wins
    bot <bots.mybot.mybot.Bot instance at 0x7fe73587a440>: 45 wins
```

6. The heat map: - Insert pdf here - This heatmap means: If player 2 has a high non-move probability, this is better for player 1 if player 1 has a low-non move probability. Both players seem to have a good value if the non-move probability is around 0.4 - 0.6.

7. The added line:

   ```
   value, move = self.value(next_state, depth + 1)
   ```

   Output:

   ```
   python tournament.py -p rand,minimax -s 2 -r 10
   Playing 10 games:
   Played 1 out of 10 games (10%): [0, 1]
   Played 2 out of 10 games (20%): [0, 2]
   Played 3 out of 10 games (30%): [0, 3]
   Played 4 out of 10 games (40%): [0, 4]
   Played 5 out of 10 games (50%): [1, 4]
   Played 6 out of 10 games (60%): [2, 4]
   Played 7 out of 10 games (70%): [2, 5]
   Played 8 out of 10 games (80%): [3, 5]
   Played 9 out of 10 games (90%): [4, 5]
   Played 10 out of 10 games (100%): [4, 6]
   ```

```
Results:
bot <bots.rand.rand.Bot instance at 0x102a70f80>: 4 wins
bot <bots.minimax.minimax.Bot instance at 0x102a731b8>: 6 wins
```

8. The added code:

   It creates one instance for each bot, minimax and alphabeta. Then in generates a state, and it does a few random moves. It measures the time to get the next move from both the minimax and alphabeta, then it compares the outcomes. If they are equal it will print "Agreed." else it will display the answers of both bots. This process is repeated a few times. Even though the code is right, the checking program does not behave correctly. That's why we do not have a valid output.

9. The heuristic is a value between -1.0 and 1.0, derived from the point ratio of the current player. If the player has a 1.0 value, the player has a definite win. If the value is -1.0, the opponent has a definite win.

   We came up with, the ratio of trump cards in hand, it turned out to be a horrible heuristic. Holding on to your trumps seems to be a bad tactic. Therefore we tried ditching trumps as soon as possible, this lead to somewhat better results, but random was still a better bot in the end.

11 Worksheet II

1. The added clause that makes the knowledge base unsatisfiable:

   ```
   kb.add_clause(~B, ~C)
   ```

2. The three clauses converted to CNF:

   - A V B
   - -B V A
   - -A V C
   - -A V D

   The code that creates the variables and the knowledge base:

   ```
   # Define our symbols
   A = Boolean('A')
   B = Boolean('B')
   C = Boolean('C')
   D = Boolean('D')

   # Create a new knowledge base
   kb = KB()
   ```

```
# Add clauses
kb.add_clause(A, B)
kb.add_clause(~B, A)
kb.add_clause(~A, C)
kb.add_clause(~A, D)
```

Because the knowledge base entails A & C & D, all three have to be true. For now, the model has two solutions:

```
{A: True, C: True, B: False, D: True}
{A: True, C: True, B: True, D: True}
```

B can be whatever it wants, but A, B and D have to be true.

3. Three constraints have to be met:

   I $X = Y$

   II $X + Y > 2$

   III $X + Y < 5$

   The model is satisfiable, as long as: $1 < x < 2.5$ and $1 < y < 2.5$

4. We now have three models:

```
(a) [x = y] = True, [x + y > 2] = True, [x + y < 5] = True
(b) [x = y] = True, [x + y > 2] = True, [x + y < 5] = False
(c) [x = y] = True, [x + y > 2] = False, [x + y < 5] = True
```

   The three constraints per model:

   (a) This model already has three constraints.
   (b) Only the last clause has to change, namely to $[x + y >= 5]$= True
   (c) Only the second clause has to change, namely to$[x + y <= 2]$ = True.

   Now we end up with this:

```
(a) [x = y] = True, [x + y > 2] = True, [x + y < 5] = True
(b) [x = y] = True, [x + y > 2] = True, [x + y >= 5] = True
(c) [x = y] = True, [x + y <= 2] = True, [x + y < 5] = True
```

5. What we want, is to find points on the line $x = y$, where the points either lie between $x + y > -5$ and $x + y < -2$, or where the points lie between $x + y > 2$ and $x + y < 5$.

   The clauses we need to add are:

   - $x == y$

- $x + y > 2Vx + y < -2$
- $x + y > -5Vx + y < 5$
- $x + y > -5Vx + y > 2$
- $x + y < -2Vx + y < 5$

The code:

```
# Define our integer symbols
x = Integer('x')
y = Integer('y')

q = x == y
a = x + y > 2
b = x + y < 5
c = x + y < -2
d = x + y > -5

kb = KB()

kb.add_clause(q)
kb.add_clause(a, c)
kb.add_clause(a, d)
kb.add_clause(b, c)
kb.add_clause(b, d)
```

6. 6)The playing ace strategy work:

```
kb.add_clause(A0)
kb.add_clause(A5)
kb.add_clause(A10)
kb.add_clause(A15)


#PLAY ACE STRATEGY
kb.add_clause(~A0, PA0)
kb.add_clause(~A5, PA5)
kb.add_clause(~A10, PA10)
kb.add_clause(~A15, PA15)
kb.add_clause(~PA0, A0)
kb.add_clause(~PA5, A5)
kb.add_clause(~PA10, A10)
kb.add_clause(~PA15, A15)
```

When adding:

```
kb.add_clause(~PA0)
```

The statisfiability with the KB fails, therefore we know PA0 is entailed by the KB

7. We implemented the PlayCheap strategy:
8. Game output
9.

12 Worksheet III

1. The added code:

   ```
   value = self.heuristic(next_state)

   p1_points = state.get_points(1)

   p2_points = state.get_points(2)

   p1_pending_points = state.get_pending_points(1)

   p2_pending_points = state.get_pending_points(2)

   trump_suit = state.get_trump_suit()

   phase = state.get_phase()

   stock_size = state.get_stock_size()

   leader = state.leader()

   whose_turn = state.whose_turn()

   opponents_played_card = state.get_opponents_played_card()
   ```

   The result of the played tournament against rand and bully:

   ```
   bot <bots.ml.ml.Bot instance at 0x102271290>: 16 wins
   bot <bots.rand.rand.Bot instance at 0x1022711b8>: 4 wins
   bot <bots.bully.bully.Bot instance at 0x1059b71b8>: 10 wins
   ```

2. We let the mlbot learn by playing against rdeep. But since the games took way longer than the games of mlbot vs rand, we set the number of games to 1000 (Which still took quite some time).

   We get a lower output now:

```
bot <bots.ml.ml.Bot instance at 0x102a73248>: 15 wins
bot <bots.rand.rand.Bot instance at 0x102a73170>: 7 wins
bot <bots.bully.bully.Bot instance at 0x10599df80>: 8 wins
```

ML now wins 15 times, where it won 16 times on the first tournament. This might have to do with the training set being smaller, and it was trained against rdeep, which does not compete in the tournament. However, the difference is not comparably worse.

We decided to train the mlbot again against rdeep, but now with 5000 games. Even after this long wait, the robot does not play significantly better or worse.

```
bot <bots.ml.ml.Bot instance at 0x102a71248>: 14 wins
bot <bots.rand.rand.Bot instance at 0x102a71170>: 5 wins
bot <bots.rdeep.rdeep.Bot instance at 0x105aa2cf8>: 11 wins
```

3. We created three ml bots:

   - The first one was trained on rdeep
   - The second one was trained on rand
   - The third one was trained on the second bot.

   We played a tournament with 300 games in total. The results:

   ```
       bot <bots.ml.ml.Bot instance at 0x10ad81ef0>: 92 wins
       bot <bots.ml.ml.Bot instance at 0x10ad81f80>: 99 wins
       bot <bots.ml.ml.Bot instance at 0x10b80cb00>: 109 wins
   ```

   As you can see, bot 3 only performed a bit better than the other two, but there is no really significant difference between the bots.

4. For added features, read this research paper. It's all about added features.