Comparing Generative and Discriminative Learning Methods using an Imperfect Information Game

Polina Boneva | Hardit Singh | Anca Coman

{pba590, hsh310, acn840}@student.vu.nl

Vrije Universiteit Amsterdam

February 1st, 2018

Abstract

In this paper we use two different learning methods, namely generative and discriminative, to train an automated bot to play the game of Schnapsen (an Austrian version of the game Sixty-six). We seek to evaluate the two methods against each other to conclude which one leads to the best performing bot. The machine is learning from 2 pre-defined automated bots, each of which has an entirely distinct approach to the game. While one of them uses a knowledge base and a set of predefined rules to make a decision, the other one judges a random set of future possibilities and chooses one amongst them. This created 4 new intelligent agents: 2 of them have learned to play Schnapsen by creating a probabilistic model of the tactics used; and 2 of them - by supervising and estimating the boundary of the decisions taken by the prior 2 pre-programmed bots. We obtained the results by placing the bots in a sufficient number of tournaments. For our conclusion we compare the number of games won vs. lost and the points which were produced as outcome [16].

1. Introduction

In games of imperfect information the player does not have a full image of the situation. On the contrary, they know their own abilities and what has previously happened in the game, but not where the opponent stays and what is coming from them. Thus, it is impossible to traverse the full game tree and check which choice will lead to the best outcome in the far or near future, even if we had the computational power to do it. In fact, there is not such a complete tree to be searched at all [7].

A common tactic one relies on in a situation of imperfect information game is having their own strategy. This strategy can be viewed as a set of preferred actions, regardless of the circumstances. The method is to check one's options in the current environment and compare each with the given general plan of action. If one has an option that fits the strategy, they play it. If this is not the case, the next step will be a free choice of the human playing. Automated agents which are playing a game of imperfect information use a collection of rules as a base [13]. This, conveniently, is called the knowledge base. We have created a bot, called Ultra Bot, which uses such a base as a standard to which it evaluates its possible moves in the current situation. Our KB is based on our logical reasoning as to what a good strategy to play Schnapsen is. It consists of the following rules: If a card is a jack and all higher cards of the suit of this jack have been played, play this jack; if a card is a queen and all higher cards of the suit of this queen have been played, play this queen; if a card is a king and all higher cards of the suit of this king have been played, play this king; if a card is a ten and all higher cards of the suit of this ten have been played, play this ten. Creating a backbone for an automated bot is significant in games in which the state is partially hidden. It provides an assurance for the robot that the set of rules in its integrated KB will always help discover the best current move. The engineer can add clauses to this background knowledge, so as to increase the resilience of the bot in present time [13].

2. Background Information

Some imperfect information games include Poker and Bridge Belote. The adversary's actions continuously add

information to the game when one is playing with an imperfect information. However, there is a difference in the degree in which the opponent plays a role. In order to win a Poker deal, the player does not need to give his opponent's cards and actions as much attention, as they would need to if they were to win a Bridge Belote game [3]. While in Poker it is needed to be able to distinguish the enemy's reactions in order to try to evaluate one's own chance of winning, their hand will not be revealed before the end of the game [10]. On the other hand, both in Bridge Belote and Schnapsen every move reveals a card of the opponent, which adds information and updates the current state of the player, specifically how much their points weight. Due to the immense significance of the toughness of the opposition in Schnapsen, we have included 2 automated bots, in addition to the 2 we have chosen to be used as ground for learning, to participate in the tournaments for the final decision. A variety of challenges emphasizes the weaknesses and strengths of our machine learning bots in comparison to each other and to the two core ones.

The implementation of the automated bot Ultra includes: a knowledge base, an approach to the knowledge base, and a technique to handle a hand of cards not entailed by the knowledge base. The clauses entailed by the KB were described semantically in the previous section. The syntax of a knowledge base is done using first-order logic, which is later translated to propositional logic in order to be implemented easily [3]. The definition of our knowledge base focuses on what is known already from past experience. Due to the limited information in Schnapsen, creating strategic moves which adhere to the past states of the game is crucial, as it determines a move which fits the policy of the bot and takes this decision in relation to what has already happened in the particular game [3]. The technique used to check the KB (now including a negation of the card that is being examined for the current move) for satisfiability is done with the use of the Intelligent System's predefined program, which was provided to us by the Vrije Universiteit Amsterdam [6]. It is making use of a solver of satisfiability modulo theories (SMT) to determine if the new KB has a solution [4].

Two widely used methods for learning are generative and discriminative. Each of them has more than one way to be implemented. In this paper to test out the generative and discriminative methods we apply Naive Bayes and Linear Regression methods, respectively, to the machine learning bots. Their common ground is found in the fact that both of them are used as classification techniques: they categorize each event space as positive or negative and divide them accordingly. The manner in which they approach the classification problem is what distinguishes one from the other.

2.1 Learning Methods

A generative learning method builds two models: one of positives and one of negatives. It takes into account the full data that is presented with. Even though Naive Bayes is an algorithm which performs better with a limited amount of datasets, this will not show in our experiment as it consists of 2000 games per tournament. It does not proceed immediately to solve which card should be played at the moment, but uses an intermediate stage in which it models the probability that the origin bot, from which it learned in the first place, would play it [2].

In a discriminative learning method, such as Linear Regression, a transitional step is not needed. Instead of two models, only one is created: it describes the boundary between the positives and the negatives of the dataset. Thus, there is considerable focus on the data close to this boundary and almost none on the edge cases. It even ignores most of the cases if the data space is enormous. On the other hand, once the training examples are many, it performs powerfully as it doesn't waste time to walk through all of them [1].

3. Intelligent Agents

The intelligent bots we have used for the experiment use contrasting strategies. Ultra Bot is based entirely on a pre-defined strategy. There is no point at which it tries to predict the turn of the game. The only information it has access to is its own and the ones already taken in one of the player's wins. RDeep, on the other hand, takes in no consideration the past events of the game. On the contrary, it performs a number of random plays to determine possible outcomes for the future.

The bot created specifically for this experiment is discussed below. The second bot we used as ground for learning can be found in *bots/rdeep* the Intelligent Systems framework we were equipped with in the beginning of January, 2018 [6]. It is also discussed below. The third and fourth bots were also given in the beginning of the course.

3.1 Ultra Bot

We have divided the procedure that is the game plan of our bot, Ultra, in two scenarios: 1. Bot is leader; and 2. Opponent is leader. Let's discuss the first scenario now. Ultra bot begins by checking for the possibility of a trump exchange or a marriage, in this order. We have given priority to the preceding because it happens rarely. Follows a check whether its own score has reached 45 points and if it has, it begins an aggressive play. This advancing game gives a preference to a trump when a move is chosen. They are preferable due to their power: the player is close to winning the game and at this point the standard of the knowledge base matters less than the amount of points. In the case where none of the above is true, the bot is directed to use the KB with each of its cards. In the case in which more than one card is allowed, we take advantage of the smallest one. If none of its cards are entailed by the knowledge base, it plays a non-Ace random card in phase 1 and absolutely any random card in phase 2.

Let's turn our attention to the second scenario: It is Ultra's turn to answer the opponent's card, meaning the opponent is the leader in this situation. The scene is divided in two cases. First case, the adversary's card is a trump. Our automated agent will answer with the lowest higher trump if it has one or with the lowest non-trump card it possesses. Provided that neither condition is satisfied, it will use any lowest card. Second case, the enemy's card is not a trump. Ultra will begin by looking for a stronger card of the same suit and proceed to check for a suitable trump, is such is not found. A suitable trump is the lowest possible, if the card on the table is a King, a Queen, or a Jack, and a highest possible in the case of a Ten or an Ace. Supposing that the current hand does not have either, we assume the hand is lost and play any lowest card.

3.2 RDeep Bot

As stated in the beginning of this paper, RDeep does not have access to a knowledge base, has no notion of what the opponent has played, and does not take into account previously won points. It goes through a number of possible results calculated by considering different plays at random. Assessment through heuristics is applied to determine the leading outcome, which is afterwards chosen. This is essentially Perfect Information Monte-Carlo (PIMC) sampling method and it is repeated at every move.

This paper proceeds to give valuable information on the strategies incorporated in the two bots used exclusively for testing purposes. This is needed because, as explained in section 2, the opponent in an adversary game with imperfect information flashes new light on the performance of its enemy. We are able to evaluate our learning bots with sufficient amount of results after the additional default bots, provided by the Intelligent Systems course.

3.3 Rand Bot

Rand is one more default bot provided at the beginning of the Intelligent Systems course. For every turn, it does nothing more than making a random choice out of all the cards in hand, without taking any other variable in consideration.

3.4 Bully Bot

Another default bot is Bully. As its name suggests, Bully is an aggressive bot always playing the highest ranking card, considering its options. The strategy of the Bully bot can also be divided into two scenarios. In the first scenario Bully is the leader and play its highest card available, trump or non-trump. In the second scenario Bully is the opponent. If it has a card of the same suit as the opponent and also higher ranking, it will play that card but if not

When it plays as a leader it gives the highest ranking trump or non-trump possible. If the opponent is the leader and a card has been played, Bully is checking for same suit, higher ranking card than the one played by the opponent; if such card doesn't exist it will follow up with the highest ranking card available.

4. Related Work

The discriminative and generative learning methods are vastly popular in the machine learning field. A large percent of the time they are used separately, as we also did in this experiment. It has been proven that discriminatively trained classifiers perform better in the case of vast amount of labelled training data being available, although they can be outperformed by generative classifiers in limited training data. In a paper by Andrew.Y.NG the aim is to empirically and theoretically find the extent to which the belief that the discriminative classifiers outperform the degenerative classifiers holds [8]. Their approach is to study the behaviour of both generative and discriminative learning methods as the number of training examples is increased. From that, one would expect Naive Bayes to initially perform well but as the training set increases linear regression would catch up and eventually overtake Naive Bayes.

A similar comparison can also be seen in B. Efron's paper *The Efficiency of Logistic Regression Compared to Normal Discriminant Analysis.* The asymptotic relative efficiency of the two procedures is computed. Typically, logistic regression is shown to be between one half and two

thirds as effective as normal discrimination for statistically interesting values of the parameters [5].

To get the best of both worlds, *Classification with Hybrid Generative/Discriminative Models* studies a hybrid generative/discriminative model where a large part of the model's parameters are "trained to maximize the generative, joint probability of the inputs and outputs of the supervised learning task and another, much smaller, subset of the parameters are discriminatively trained to maximize the conditional probability of the outputs given the inputs" [11].

5. Research Question

The aim of this paper is to compare the generative and discriminative learning methods using an imperfect information game, namely Schnapsen. The centre of interest is to determine how does a bot (an agent that plays Schnapsen autonomously) which uses the generative classifier Naïve Bayes perform in comparison to a bot which relies on a discriminative learning method, in particular Linear Regression. Discriminative learning models are more often than not preferred over generative ones, as Vapnik puts it "one should solve the [classification] problem directly and never solve a more general problem as an intermediate step [such as modelling P(x|y)]" [15]. We have used a variety of bots to put our recently taught intelligent agents to the test. One might think that the generative model will work best, as it takes into account the full dataset. On the other hand, the discriminative model might make the best use of the limited data it uses, as it is fully concentrated on finding the boundary. This research has the ambition to recognize one of the methods as preferable.

6. Experimental Setup

In order to run the simulation, we train a total of four bots. Two are trained following Ultra bot's decision making and the other two are trained using RDeep. Of each of those two, one uses Naïve Bayes method and the other - a Linear Regression method. These four bots will henceforth be referred to as nb-ultra, lr-ultra, nb-rdeep and lr-rdeep, respectively.

To ensure that we get accurate results, we do not alter the code of Ultra or RDeep in any way in the duration of the experiment.

In the simulation the six bots Ultra, RDeep, nb-ultra, lr-ultra, nb-rdeep and lr-rdeep are benchmarked against four different bots mentioned in Section 3. In this experiment, we create a competitive environment by running a tournament of 2000 matches between each pair of bots. Our main six bots against the four test bots can help in the following way:

- The Rand bot provides a base for estimation how well the player does against a random decision
- The Bully and Ultra serve as examples of the use of a strict set of rules
- The RDeep is equipped with Perfect Information Monte-Carlo Sampling, which adds another layer of perfect information game-like strategy.

To be able to draw conclusions from the simulations, we compare the performance of the bots on three different parameters, described below.

Win-Percentage: For each bot, we calculate the percentage of matches that a bot wins from the total matches it plays in the duration of the simulation. This is an indicator of the average performance of

the bot against the variety of bots it plays with.

Average Points Taken per Match (PT/M): When an agent wins, it gets either 1, 2 or 3 points for that particular match depending on the score of the loser. We calculate the average points an intelligent agent gains winning so that we can estimate the strength of the bot against its opponent. The higher the average, the stronger the bot.

Average Points Given per Match (PG/M): When a bot loses, the opponent gains 1, 2 or 3 points depending on the score it managed to reach during the game. We calculate the average points in a similar way as PT/M. Note that in this case the lower the average, the stronger the bot is.

6.1 Feature Set

In the interest of making the test results uniform, we created a set of features to train all of the ml models. We keep these features constant during the experiment, which assists in getting an absolute overview of the results. A large amount of features can lead to overfitting of training data which would lead to inaccurate predictions during gameplay [12]. This does not happen in our experiment. The features used for training the machine learning model are present in the Table 1.

TABLE 1: FEATURE SET

Feature	Description
Player's Perspective	This is a list of the cards which are in the player's hand, have already been played or are unknown.
Player's Points	These are the total points scored by the player.
Opponent's Points	These are the total points scored by the opponent.
Player's Pending Points	These are the point pending due to a marriage towards the player.
Opponent's Pending Points	These are the point pending due to a marriage towards the opponent.
Trump Suit	The trump suit in a game
Game Phase	The current phase of the game (1 or 2).
Stock Size	The number of cards present in stock.
Leader	The contender leading the game.
Turn	The contender whose turn it is to play a card.
Opponent's Played Card	The card played by the opponent if it is leading.

7. Results and Findings

The results of the simulation are stated in the Table 2. The Win Percentage of all the bots is shown in Table 3.

TABLE 2: SIMULATION RESULTS

	Rand				Bully				RDeep				Ultra			
	Win	PT/M	Loss	PG/M	Win	PT/M	Loss	PG/M	Win	PT/M	Loss	PG/M	Win	PT/M	Loss	PG/M
ultra	1520	2.11	480	1.29	1291	1.57	709	1.82	860	1.62	1140	1.59				
MI-naivebayes(ultra)	1498	1.95	502	1.19	1637	1.52	363	1.66	774	1.43	1226	1.44	998	1.56	1002	1.57
MI-linear-reg(ultra)	1484	2.02	516	1.22	1611	1.53	389	1.56	822	1.53	1178	1.39	1052	1.54	948	1.45
rdeep	1661	2.09	339	1.22	1725	1.54	275	1.71					1140	1.59	860	1.62
MI-naivebayes(rdeep)	1465	2.03	535	1.20	1622	1.53	378	1.62	816	1.51	1184	1.45	956	1.49	1044	1.56
mi-linear-reg(rdeep)	1493	1.99	507	1.17	1617	1.53	383	1.54	793	1.52	1207	1.39	1011	1.50	989	1.47

TABLE 3: WIN PERCENTAGE OF BOTS

Bot	Win Percentage
ultra	61.18%
MI-naivebayes(ultra)	61.34%
MI-linear-reg(ultra)	62.11%
rdeep	75.43%
MI-naivebayes(rdeep)	60.74%
mi-linear-reg(rdeep)	61.43%

The Win-Percentage results of all the machine learning bots is very similar (around 65%), irrespective of which bot/training method they were formulated from. Therefore, the difference between the generative and discriminative learning methods is not apparent when judged on the basis of Win-Percentage.

Judging the average points Taken/Given per match, it is noticeable that the difference is significant. In the case of nb-ultra and lr-ultra playing against Bully, when a discriminative bot (nb-ultra) wins, it allows its opponent to have 6% less points in comparison to the generative bot (lr-ultra). A similar trend is visible with nb-rdeep and lr-rdeep playing against Bully. Hence, we can conclude that bots trained using discriminative model tend to minimize the points the adversary receives at the end of a match better than the intelligent agents based on the generative model.

The bots nb-ultra and lr-ultra perform significantly better than Ultra against Bully. Ultra is a bot which plays according to a set or pre-defined strategy, and thus is not particularly flexible during gameplay. The trained bots on the other hand tend to be significantly less rigid as the trained model equips them to choose the best possible solution for different situations. On the contrary, the nb-rdeep and lr-rdeep do not compete with RDeep well. This is an unexpected segment of our findings in which we conclude that machine learning bots adapt to static strategies easier and more efficiently than to changeable ones. The latter is referring to RDeep's use of PIMC which subjects the bot to adjust its play at every move based on randomness.

8. Conclusion

This paper examined two distinct learning models used in machine learning to train intelligent agents by observation. The classification techniques are similar in the sense that both of them can perform with supervised learning and labelled data. What they differentiate in is the way they distinguish positives from negatives, thus we were intrigued to find out

which would surpass the other if they observed the game strategies of contrasting intelligent agents - to allow for an even more definite conclusion. Surprisingly, the number of games won by each bot that has learned in any model did not produce a significantly outstanding result. Nevertheless, the points by which a learning agent won did make a difference. A bot that has used a discriminative classification to play has proven to be more uncompromising even when it is losing. It shows endurance in all stages of the game, unlike a bot trained with a generative model, which loses with greater difference in final score. It seems that using a determination of the probability that a certain move is correct is less bullet-proof than relying on the boundary between the classes of moves and deciding based on those central cases, instead of taking into account all edge nodes of the dataset.

9. Citation

1. Aiolli, Fabio, and Claudio E. Palazzi. *Enhancing Artificial Intelligence on a Real Mobile Game*. 2008, pp. 5–7, *Enhancing Artificial Intelligence on a Real Mobile Game*.
2. B. Myerson, Roger. *Comments on "Games with Incomplete Information Played by 'Bayesian' Players, I–III."* Management Science, pp. 3–5, *Comments on "Games with Incomplete Information Played by 'Bayesian' Players, I–III."*
3. "Belote." *Pagat. Rules of Card Games*, www.pagat.com/jass/belote.html.
4. De Moura, Leonardo, et al. *A Tutorial on Satisfiability Modulo Theories (Invited Tutorial)*. Microsoft Research, pp. 1–3, *A Tutorial on Satisfiability Modulo Theories (Invited Tutorial)*, link.springer.com/content/pdf/10.1007/978-3-540-73368-3_5.pdf.
5. Efron, Bradley. *The Efficiency Of Logistic Regression Compared to Normal Discriminant Analysis* . pp. 2–3, *The Efficiency Of Logistic Regression Compared to Normal Discriminant Analysis* .
6. intelligent-systems-course. "Intelligent-Systems-Course/Schnapsen." *GitHub*, 9 Jan. 2018, github.com/intelligent-systems-course/schnapsen.
7. Gilpin, A. and Sandholm, T. 2007. *Lossless abstraction of imperfect information games.* J. ACM 54, 5, Article 25 (October 2007), 30 pages.
8. McGraw, Hill, and Tom Mitchell. *GENERATIVE AND DISCRIMINATIVE CLASSIFIERS: NAIVE BAYES AND LOGISTIC REGRESSION*. pp. 1–7, *GENERATIVE AND DISCRIMINATIVE CLASSIFIERS: NAIVE BAYES AND LOGISTIC REGRESSION*.
9. Ng, Andrew, and Michael Jordan. *On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes*. pp. 1–8, *On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes*.
10. "Poker Hand Rankings." *Poker Star School*, www.pokerstarsschool.com/article/Poker-Hand-Rankings.
11. Raina, Rajat, et al. *Classification with Hybrid Generative/Discriminative Models*. pp. 1–4, *Classification with Hybrid Generative/Discriminative Models*.
12. "Regularization: Solving ML's Overfitting (Underfitting) Problem." *Moreintelligent*, 13 Apr. 2014, moreintelligent.wordpress.com/2014/04/13/regularization-solving-mls-overfitting-underfitting-problem/.

13. RUSSELL, STUART NORVIG PETER. *ARTIFICIAL INTELLIGENCE: a Modern Approach*. 3rd ed., PEARSON, 2018.
14. "Satisfiability modulo Theories." *Wikipedia*, Wikimedia Foundation, 23 Jan. 2018, en.wikipedia.org/wiki/Satisfiability_modulo_theories.
15. V. N. Vapnik. Statistical Learning Theory. John Wiley & Sons, 1998.
16. "Winning Strategy for Schnapsen or Sixty-Six." *Psellos*, 29 Sept. 2013, psellos.com/schnapsen/strategy.html.

10. Appendix

Worksheet 1 Part A

1. Which of the three default bots does the best? Add the output to your report.

```
Results:
    bot <bots.rand.rand.Bot instance at 0x00000000053D0E88>: 5
wins
    bot <bots.bully.bully.Bot instance at 0x00000000053D0F08>:
9 wins
    bot <bots.rdeep.rdeep.Bot instance at 0x00000000053D0FC8>:
16 wins
```

2. Have a look at the code: what strategy does the bully bot use?

   The bully bot first chooses as a move the first card in hand. Then it checks to see if it has a trump move and if yes then it plays it and if not, it plays the card chosen above. If the opponent already played a card , the bully bot checks to see if it has a card of the same suit and if it does it plays that card otherwise it plays the highest card in hand.

3. For our game, this strategy would be no good. Why not?

   Hill climbing does not necessarily find the global maximum, instead it is concerned with finding the local maximum. In the case of game Schnapsen we are interested in finding the global maximum that is the combination of moves that leads to winning the game and therefore hill-climbing is not the best solution since it will throw in the biggest card in hand and might lose the trick.

4. If you wanted to provide scientific evidence that rdeep is better than rand, how would you go about it?

   Basically what Perfect Information Monte Carlo Sampling does is for each card in our deck it evaluates the quality of playing that card and it does that by running a number of random games using the belief states (possible moves the opponent could make) and the one card that comes up with the highest average is the one to be suggested.

5. Add your implementation of get_move() and the result of a tournament against rand to your report.

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

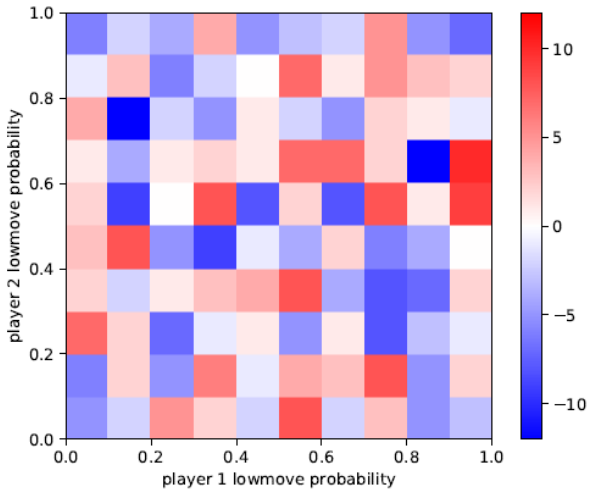
```
    """
    Function that gets called every turn. This is where to implement the strategies.
    Be sure to make a legal move. Illegal moves, like giving an index of a card you
    do not own or proposing an illegal mariage, will lose you the game.
    :param State state: An object representing the gamestate. This includes a link to
        the states of all the cards, the trick and the points.
    :return: A tuple of integers or a tuple of an integer and None,
        indicating a move; the first indicates the card played in the trick, the second a
        potential spouse.
    """

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

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

        # If mybot has cards of the same suit as opponent's card
        if len(moves_same_suit) > 0:
            # Take out the first higher card, if there is one
            higher_ranked_same_suit = False
            for index, move in enumerate(moves_same_suit):
                if move[0] is not None and moves_same_suit[0][0] % 5 <= state.get_opponents_played_card() % 5:
                    chosen_move = move
                    higher_ranked_same_suit = True
            # Take out the first card with the same suit, if none are bigger than opponent's
```

```
            if not higher_ranked_same_suit:
                chosen_move = moves_same_suit[0]
            return chosen_move
        # If mybot does not have cards of the same suit
        else:
            moves_trump_suit = []
            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)
            # Choose any trump, if mybot has any
            if len(moves_trump_suit) > 0:
                chosen_move = moves_trump_suit[0]
                return chosen_move

        # If mybot does not have the same suit and has not trumps
        # Get move with lowest rank available, of any suit
        for index, move in enumerate(moves):
            if move[0] is not None and move[0] % 5 >= chosen_move[0] % 5:
                chosen_move = move

    # If opponent has not played a card
    else:
        # Get move with highest rank available, of any suit
        for index, move in enumerate(moves):
            if move[0] is not None and move[0] % 5 <= chosen_move[0] % 5:
                chosen_move = move

    # Return the choice
    return chosen_move
```

Worksheet 1 Part B

6. Two simple bits of code are missing (indicated with lines starting with #IMPLEMENT). Read the whole script carefully and finish the implementation of the Bot. Run the experiment. It should output a file experiment.pdf containing a heatmap. Add this heatmap to your report, and discuss briefly what it means.

   This heatmap pictures the probability of both players playing a low move.



7. All you need to do to finish the minimax bot is to add one line of code on line 58. Take your time to really understand the minimax algorithm, recursion, and the rest of the code. Finish the bot, and let it play against rand (use flag to start in phase 2). Add the line you wrote and the results of the tournament to the appendix of your report.

   value, mov = self.value(next_state)

8. Once again, crucial parts of the implementation are missing. Finish the implementation of the alphabeta bot. The script check_minimax.py lets you see if you implemented alphabeta and minimax correctly. What does it do? Run it and add the output to the appendix of your report.

   The script check_minimax.py check to see if minimax and alphabeta bots agreee on the card to be played. If they agree the output is:

```
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Done. time Minimax: 0.28066666921, time Alphabeta: 0.256666580836.
Alphabeta speedup: 1.09350686909
```

9. What heuristic do these implementations use? Try to come up with a better one. Implement it and see how it does.

Worksheet 2

1. Add a clause to the knowledge base to that it becomes unsatisfiable. Report the line of code you added.

   kb.add_clause(~B,~C)

2. Exercise 8 of this week's work session on logical agents contained the following knowledge base:

   - A -> B

   B -> A

   A -> (C ^ D)

   Convert it to clause normal form, and write a script that creates this knowledge base. Print out its models and report them. As seen in the exercise, the knowledge base entails A ^ C ^ D. What does that say about the possible models for the knowledge base?

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

   True = satisfiable

   If the KB entails A^C^D it becomes unsatisfiable.

3. What does this mean for the values of x and y? Are there any integer values for x and y that make these three constraints true?

   If we want all three constraints to be true then x and y must equal 2.

4. If we know that [x + y < 5] must be false, as in the second model, we know that x + y >= 5 must be true. Write each of these three models as three constraints that must be true.

   [x = y] = True, [x + y > 2] = True, [x + y < 5] = True

   [x = y] = True, [x + y > 2] = True, [x + y >= 5] = False

   [x = y] = True, [x + y <= 2] = False, [x + y < 5] = True

5. Write this down in propositional logic first (two sentences). Then convert this to clause normal form. Now create a knowledge base with the required clause and report which models are returned. The script test3.py does almost all the work for you. All you need to do is fill in the blanks.

kb.add_clause(~a,~c)

kb.add_clause(a,c)

kb.add_clause(b)

kb.add_clause(d)

{[y + x > 2]: True, [y + x <= 2]: False, [y + x < 5]: True, [y + x >= -2]: True, [(-y) + x == 0]: True, [y + x < -2]: False, [y + x > -5]: True}

{[y + x > 2]: False, [y + x <= 2]: True, [y + x < 5]: True, [y + x >= -2]: False, [(-y) + x == 0]: True, [y + x < -2]: True, [y + x > -5]: True}

6. Look at the code in test4.py. (The long list in the beginning is just the variable instantiation, the real modelling starts at line 50.) Extend the document with the knowledge for a strategy PlayAs, always playing an As first. Check whether you can do reasoning to check whether a card is entailed by the knowledge base or not.

   If we add the negated clause ~PA5 then the result after running the test is False but if we add the negated clause ~PA4 the result after running the test is :

   {pa0: True, a15: False, pa4: False, pa5: True, a19: True, a0: True, pa10: True, a5: True, pa15: False, a10: True}

   {pa0: True, a15: True, pa4: False, pa5: True, a19: True, a0: True, pa10: True, a5: True, pa15: True, a10: True}

   True

7. Build a more complex logical strategies. For examples, you can define the notion of a cheap card, as being either a jack, king or queen, and devise a strategy that plays cheap card first. Test whether you can use logical reasoning to check whether the correctness of a move w.r.t. this strategy is entailed by your knowledge base.

   We try to build a strategy where a card of a particular suit is played only if all the cards of the same suit higher than that card have already been played. Upon testing using test5.py, we see that it works as expected.

8. Replace the knowledge and strategy you modelled in test5.py into load.py. You might want to add some print statements to check whether kbbot now really follows your strategy. Provide an example game where you show that the strategy works.

   Start state: The game is in phase: 1

   Player 1's points: 0, pending: 0

Player 2's points: 0, pending: 0

The trump suit is: H

Player 1's hand: KC QC 10D KH JH

Player 2's hand: 10C KD QH KS QS

There are 10 cards in the stock

player1: <bots.kbbot.kbbot.Bot instance at 0x7fe885299f80>

player2: <bots.bully.bully.Bot instance at 0x7fe86896f680>

* *Player 2 plays: QH*

The game is in phase: 1

Player 1's points: 0, pending: 0

Player 2's points: 0, pending: 0

The trump suit is: H

Player 1's hand: KC QC 10D KH JH

Player 2's hand: 10C KD QH KS QS

There are 10 cards in the stock

Player 2 has played card: Q of H

* *Player 1 plays: KH*

The game is in phase: 1

Player 1's points: 7, pending: 0

Player 2's points: 0, pending: 0

The trump suit is: H

Player 1's hand: KC QC JC 10D JH

Player 2's hand: 10C KD 10S KS QS

There are 8 cards in the stock

* *Player 1 performs a trump jack exchange*

The game is in phase: 1

Player 1's points: 7, pending: 0

Player 2's points: 0, pending: 0

The trump suit is: H

Player 1's hand: KC QC JC 10D 10H

Player 2's hand: 10C KD 10S KS QS

There are 8 cards in the stock

* *Player 1 plays: KC*

* *Player 1 melds a marriage between KC and QC*

The game is in phase: 1

Player 1's points: 7, pending: 20

Player 2's points: 0, pending: 0

The trump suit is: H

Player 1's hand: KC QC JC 10D 10H

Player 2's hand: 10C KD 10S KS QS

There are 8 cards in the stock

Player 1 has played card: K of C

* *Player 2 plays: 10C*

The game is in phase: 1

Player 1's points: 7, pending: 20

Player 2's points: 14, pending: 0

The trump suit is: H

Player 1's hand: QC JC 10D 10H AS

Player 2's hand: AD KD 10S KS QS

There are 6 cards in the stock

* *Player 2 plays: AD*

The game is in phase: 1

Player 1's points: 7, pending: 20

Player 2's points: 14, pending: 0

The trump suit is: H

Player 1's hand: QC JC 10D 10H AS

Player 2's hand: AD KD 10S KS QS

There are 6 cards in the stock

Player 2 has played card: A of D

* *Player 1 plays: 10H*

The game is in phase: 1

Player 1's points: 48, pending: 0

Player 2's points: 14, pending: 0

The trump suit is: H

Player 1's hand: AC QC JC 10D AS

Player 2's hand: KD JD 10S KS QS

There are 4 cards in the stock

Strategy Applied

* *Player 1 plays: AC*

The game is in phase: 1

Player 1's points: 48, pending: 0

Player 2's points: 14, pending: 0

The trump suit is: H

Player 1's hand: AC QC JC 10D AS

Player 2's hand: KD JD 10S KS QS

There are 4 cards in the stock

Player 1 has played card: A of C

* *Player 2 plays: 10S*

The game is in phase: 1

Player 1's points: 69, pending: 0

Player 2's points: 14, pending: 0

The trump suit is: H

Player 1's hand: QC JC 10D AH AS

Player 2's hand: KD JD KS QS JS

There are 2 cards in the stock

Game finished. Player 1 has won, receiving 2 points.

9. Produce a new knowledge based bot that allows you to play knowledge based strategies consequently. Compare the performance w.r.t. simpler knowledge based bot (with the individual strategies) and other bots.

We modeled a kbbot with two strategies, it either plays the strategy discussed in the above questions or plays a Jack. This is the result of this kbbot playing against the one from the previous question:

Playing 10 games:

Strategy Applied

Strategy Applied

Played 1 out of 10 games (10%): [1, 0]

Strategy Applied

Strategy Applied

Played 2 out of 10 games (20%): [1, 1]

Played 3 out of 10 games (30%): [2, 1]

Strategy Applied

Strategy Applied

Played 4 out of 10 games (40%): [3, 1]

Strategy Applied

Played 5 out of 10 games (50%): [4, 1]

Strategy Applied

Strategy Applied

Played 6 out of 10 games (60%): [5, 1]

Strategy Applied

Played 7 out of 10 games (70%): [6, 1]

Strategy Applied

Played 8 out of 10 games (80%): [7, 1]

Strategy Applied

Played 9 out of 10 games (90%): [7, 2]

Played 10 out of 10 games (100%): [8, 2]

Results:

bot <bots.newkbbot.newkbbot.Bot instance at 0x7f77a92b1ef0>: 8 wins

bot <bots.kbbot.kbbot.Bot instance at 0x7f778c986710>: 2 wins

Worksheet 3

1. Fill in the missing code (all the '???' lines) and run the training script. Run a tournament between rand, bully and ml. Show the code you wrote, and the result of the tournament.

```
DEFAULT_MODEL = os.path.dirname(os.path.realpath(__file__)) + '/rand.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):
```

```
    val, move = self.value(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)

        # IMPLEMENT: Add a function call so that 'value' will
        # contain the predicted value of 'next_state'
        # NOTE: This is different from the line in the minimax/alphabeta bot

        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 = []
```

```
    perspective = state.get_perspective()

    # Convert the card state array containing strings, to an array of integers.
    # The integers here just represent card state IDs. In a way they can be
    # thought of as arbitrary, as long as they are different from each other.
    perspective = [card if card != 'U' else (-1) for card in perspective]
    perspective = [card if card != 'S' else 0 for card in perspective]
    perspective = [card if card != 'P1H' else 1 for card in perspective]
    perspective = [card if card != 'P2H' else 2 for card in perspective]
    perspective = [card if card != 'P1W' else 3 for card in perspective]
    perspective = [card if card != 'P2W' else 4 for card in perspective]

    feature_set += perspective

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

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

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

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

    # Get trump suit
    trump_suit = state.get_trump_suit()

    # 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_id = suits.index(trump_suit)
    feature_set.append(trump_suit_id)
```

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

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

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

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

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

    # You don't need to add anything to this part
    opponents_played_card = opponents_played_card if opponents_played_card is not None else -1
    feature_set.append(opponents_played_card)

    # Return feature set
    return feature_set
```

Tournament Results:

Playing 30 games:

Played 1 out of 30 games (3%): [0, 1, 0]

Played 2 out of 30 games (7%): [1, 1, 0]

Played 3 out of 30 games (10%): [2, 1, 0]

Played 4 out of 30 games (13%): [2, 2, 0]

Played 5 out of 30 games (17%): [3, 2, 0]

Played 6 out of 30 games (20%): [3, 3, 0]

Played 7 out of 30 games (23%): [3, 4, 0]

Played 8 out of 30 games (27%): [3, 5, 0]

Played 9 out of 30 games (30%): [3, 6, 0]

Played 10 out of 30 games (33%): [3, 7, 0]

Played 11 out of 30 games (37%): [3, 7, 1]

Played 12 out of 30 games (40%): [3, 7, 2]

Played 13 out of 30 games (43%): [3, 7, 3]

Played 14 out of 30 games (47%): [3, 7, 4]

Played 15 out of 30 games (50%): [4, 7, 4]

Played 16 out of 30 games (53%): [5, 7, 4]

Played 17 out of 30 games (57%): [6, 7, 4]

Played 18 out of 30 games (60%): [7, 7, 4]

Played 19 out of 30 games (63%): [7, 7, 5]

Played 20 out of 30 games (67%): [8, 7, 5]

Played 21 out of 30 games (70%): [8, 7, 6]

Played 22 out of 30 games (73%): [8, 8, 6]

Played 23 out of 30 games (77%): [8, 8, 7]

Played 24 out of 30 games (80%): [8, 8, 8]

Played 25 out of 30 games (83%): [8, 8, 9]

Played 26 out of 30 games (87%): [8, 8, 10]

Played 27 out of 30 games (90%): [8, 8, 11]

Played 28 out of 30 games (93%): [8, 8, 12]

Played 29 out of 30 games (97%): [8, 8, 13]

Played 30 out of 30 games (100%): [8, 8, 14]

Results:

bot <bots.rand.rand.Bot instance at 0x7efc5fd7a9e0>: 8 wins, 10 points

bot <bots.bully.bully.Bot instance at 0x7efc5fd7ab48>: 8 wins, 16 points

bot <bots.ml.ml.Bot instance at 0x7efc5f220290>: 14 wins, 19 points

2. Re-run the tournament. Does the machine learning bot do better? Show the output, and mention which bot was used for training.

In this instance, we trained the bot using rdeep bot provided, the outcome was better than that in the previous question

Tournament Results:

Playing 30 games:

Played 1 out of 30 games (3%): [1, 0, 0]

Played 2 out of 30 games (7%): [1, 1, 0]

Played 3 out of 30 games (10%): [2, 1, 0]

Played 4 out of 30 games (13%): [2, 2, 0]

Played 5 out of 30 games (17%): [2, 3, 0]

Played 6 out of 30 games (20%): [2, 4, 0]

Played 7 out of 30 games (23%): [2, 5, 0]

Played 8 out of 30 games (27%): [3, 5, 0]

Played 9 out of 30 games (30%): [3, 6, 0]

Played 10 out of 30 games (33%): [3, 7, 0]

Played 11 out of 30 games (37%): [3, 7, 1]

Played 12 out of 30 games (40%): [3, 7, 2]

Played 13 out of 30 games (43%): [3, 7, 3]

Played 14 out of 30 games (47%): [3, 7, 4]

Played 15 out of 30 games (50%): [3, 7, 5]

Played 16 out of 30 games (53%): [3, 7, 6]

Played 17 out of 30 games (57%): [4, 7, 6]

Played 18 out of 30 games (60%): [5, 7, 6]

Played 19 out of 30 games (63%): [6, 7, 6]

Played 20 out of 30 games (67%): [6, 7, 7]

Played 21 out of 30 games (70%): [6, 7, 8]

Played 22 out of 30 games (73%): [6, 7, 9]

Played 23 out of 30 games (77%): [6, 7, 10]

Played 24 out of 30 games (80%): [6, 7, 11]

Played 25 out of 30 games (83%): [6, 7, 12]

Played 26 out of 30 games (87%): [6, 8, 12]

Played 27 out of 30 games (90%): [6, 8, 13]

Played 28 out of 30 games (93%): [6, 8, 14]

Played 29 out of 30 games (97%): [6, 8, 15]

Played 30 out of 30 games (100%): [6, 8, 16]

Results:

bot <bots.rand.rand.Bot instance at 0x7f80db7829e0>: 6 wins, 8 points

bot <bots.bully.bully.Bot instance at 0x7f80db782b48>: 8 wins, 19 points

bot <bots.ml.ml.Bot instance at 0x7f80dac21638>: 16 wins, 26 points

3. Make three models: one by observing rand players, one by observing rdeep players, and one by observing one of the ml players you made earlier. Show the results in your appendix.

Tournament Results:

Playing 30 games:

Played 1 out of 30 games (3%): [0, 1, 0]

Played 2 out of 30 games (7%): [0, 2, 0]

Played 3 out of 30 games (10%): [1, 2, 0]

Played 4 out of 30 games (13%): [1, 3, 0]

Played 5 out of 30 games (17%): [1, 4, 0]

Played 6 out of 30 games (20%): [1, 5, 0]

Played 7 out of 30 games (23%): [2, 5, 0]

Played 8 out of 30 games (27%): [3, 5, 0]

Played 9 out of 30 games (30%): [4, 5, 0]

Played 10 out of 30 games (33%): [4, 6, 0]

Played 11 out of 30 games (37%): [4, 6, 1]

Played 12 out of 30 games (40%): [4, 6, 2]

Played 13 out of 30 games (43%): [5, 6, 2]

Played 14 out of 30 games (47%): [5, 6, 3]

Played 15 out of 30 games (50%): [6, 6, 3]

Played 16 out of 30 games (53%): [6, 6, 4]

Played 17 out of 30 games (57%): [6, 6, 5]

Played 18 out of 30 games (60%): [6, 6, 6]

Played 19 out of 30 games (63%): [6, 6, 7]

Played 20 out of 30 games (67%): [6, 6, 8]

Played 21 out of 30 games (70%): [6, 7, 8]

Played 22 out of 30 games (73%): [6, 7, 9]

Played 23 out of 30 games (77%): [6, 7, 10]

Played 24 out of 30 games (80%): [6, 7, 11]

Played 25 out of 30 games (83%): [6, 7, 12]

Played 26 out of 30 games (87%): [6, 8, 12]

Played 27 out of 30 games (90%): [6, 9, 12]

Played 28 out of 30 games (93%): [6, 9, 13]

Played 29 out of 30 games (97%): [6, 9, 14]

Played 30 out of 30 games (100%): [6, 10, 14]

Results:

bot <bots.ml.ml.Bot instance at 0x7ff234d00518>: 6 wins, 8 points

bot <bots.ml2.ml2.Bot instance at 0x7ff235402ab8>: 10 wins, 11 points

bot <bots.ml3.ml3.Bot instance at 0x7ff212f254d0>: 14 wins, 19 points

4. Add some some simple features and show that the player improves.

We added a feature 'remaining points to win' for both player 1 and player 2. If we rerun the tournament in the first question, we see that there is a little improvement.

Tournament Result:

Playing 30 games:

Played 1 out of 30 games (3%): [0, 1, 0]

Played 2 out of 30 games (7%): [0, 2, 0]

Played 3 out of 30 games (10%): [0, 3, 0]

Played 4 out of 30 games (13%): [0, 4, 0]

Played 5 out of 30 games (17%): [1, 4, 0]

Played 6 out of 30 games (20%): [1, 5, 0]

Played 7 out of 30 games (23%): [2, 5, 0]

Played 8 out of 30 games (27%): [3, 5, 0]

Played 9 out of 30 games (30%): [3, 6, 0]

Played 10 out of 30 games (33%): [3, 7, 0]

Played 11 out of 30 games (37%): [3, 7, 1]

Played 12 out of 30 games (40%): [4, 7, 1]

Played 13 out of 30 games (43%): [4, 7, 2]

Played 14 out of 30 games (47%): [4, 7, 3]

Played 15 out of 30 games (50%): [4, 7, 4]

Played 16 out of 30 games (53%): [4, 7, 5]

Played 17 out of 30 games (57%): [4, 7, 6]

Played 18 out of 30 games (60%): [4, 7, 7]

Played 19 out of 30 games (63%): [4, 7, 8]

Played 20 out of 30 games (67%): [4, 7, 9]

Played 21 out of 30 games (70%): [4, 8, 9]

Played 22 out of 30 games (73%): [4, 9, 9]

Played 23 out of 30 games (77%): [4, 9, 10]

Played 24 out of 30 games (80%): [4, 9, 11]

Played 25 out of 30 games (83%): [4, 10, 11]

Played 26 out of 30 games (87%): [4, 10, 12]

Played 27 out of 30 games (90%): [4, 10, 13]

Played 28 out of 30 games (93%): [4, 10, 14]

Played 29 out of 30 games (97%): [4, 10, 15]

Played 30 out of 30 games (100%): [4, 11, 15]

Results:

bot <bots.rand.rand.Bot instance at 0x7f1ecdabe9e0>: 4 wins, 4 points

bot <bots.bully.bully.Bot instance at 0x7f1ecdabeb48>: 11 wins, 22 points

bot <bots.ml.ml.Bot instance at 0x7f1eccf64290>: 15 wins, 29 points