Investigating an optimal set of features for a machine learning bot playing Schnapsen and how these features influence performance and training time

Julius Fechner and Robin Herlan

Abstract:

This paper investigates how different features implemented in a machine learning bot using neural networks impacts the performance and training time of said bot. This is achieved the in the card playing game schnapsen. For this experiment, a framework was provided with which to test the bots in playing schnapsen.

20 different features were implemented in a bot and then ranked according to the RFE method using a linear regression algorithm. In accordance with this, 4 bots were created to test the hypothesis that the 11 best ranked features implemented in a bot will outperform any other bot and achieve at least a 50% win rate against the control bot. The bots are; allFeatures bot which has all the features that were ranked with RFE implement, 11MostRelevant bot which has the 11 highest ranked(rank closest to 1) features, Mixed bot which contains the six highest ranked features as well as the five features ranked directly below the eleven highest ranked features, and the 11LeastRelevant bot which has the 11 lowest ranked(ranks farthest away from 1) features. The hypothesis that “the training time of a bot is affected both by the total amount of features in the feature set as well as the existence of irrelevant features” was also tested.

To determine which bot performed best they played 1000 games against the control bot each. In contrast to the aforementioned hypothesis, the 11MostRelevant bot did not perform the best. The allFeatures bot had the highest number of wins and the 11LeastRelevant had the least amount of wins as was expected. The lowest training time was achieved by the Mixed bot, which was 4 times faster than the second fastest 11MostRelevant bot. This may be because when certain features are hot-encoded, they are represented in a binary vector, which can have many values in it. The Mixed bot does not have a feature, which requires that much space and hence was faster.

Table of Contents:

1 Introduction:

Artificial Intelligence is becoming a more prevalent topic in modern computing on a daily basis. It can be defined as a program that is able to perform actions that previously could only be performed by humans.[3] Humans have the inherent ability to recognize and process patterns and adapt, whereas machines have to be programmed to exhibit similar behaviour. While not being able to fully mimic the human brain yet, Artificial Intelligence has progressed to the extent of being able to recognize and process patterns in a better and faster manner than humans. Examples of this can be seen in many games where AI can outperform even the best human players; some examples of this are in chess, poker and go.

For this research project we endeavour to find an optimal set of features and find out how they affect the machine learning agent playing schnapsen. Schnapsen is a trick and draw turn based card playing game consisting of two phases. An imperfect information phase and a perfect information stage.

Developing a machine learning agent for Schnapsen is a challenge not only because it contains an imperfect information stage, but it's associated search tree is also extremely large.

Multiple different bots that would play Schnapsen were provided, ranging from a random agent to a knowledge based one to a machine learning one, which used either regression or neural networks. The agent using a neural network included a set of eleven features. This bot will be used as the control bot against which the other bots will be evaluated to determine their performance. Four bots, each containing a different set of features, will be evaluated to see which set of features can be deemed "optimal". In order to rank each feature, Recursive Feature Elimination with a Logistic Regression Algorithm is used. This feature elimination technique recursively removes features based on a model's attributes until an arbitrarily specified number of features is reached. In the process, a ranking of the features is created, which will be used to select features for the feature sets.

2 Background Information:

2.1 Schnapsen

Schnapsen is a turn based point-trick card game. The goal of the game is to capture valuable cards through trick and draw. Schnapsen is played with a 20-hand deck consisting of tens, jacks, queens, kings and aces of all four suits. These cards all hold different point values, with the ace giving the most points at 11, then the ten with 10 points, followed by 4, 3 and 2 points for each king, queen and jack respectively. In addition, each hand has a suit that is specified the "trump suit".

A hand is won if a player has captured 66 points or more. Points are accumulated by winning tricks. A higher valued card of the same suit wins the the trick, but a trump card of any value will beat a card of another suit regardless of its value. The points won are the total of the two cards played. There are also special moves such as a marriage that garner extra points, which can be announced when a player has the matching king and the queen of a suit.[4]

Additionally, Schnapsen evolves from an imperfect information game to a perfect information game. This means that initially, the player does not have all the information pertaining to the opponents cards. However, once the game enters its perfect information stage, all the possible cards are known.

2.2 Artificial Neural Network

The Artificial Neural Network(ANN) gets its name from its structure, which roughly looks like a neuron in the human brain. An integral part of an ANN is something called a perceptron.

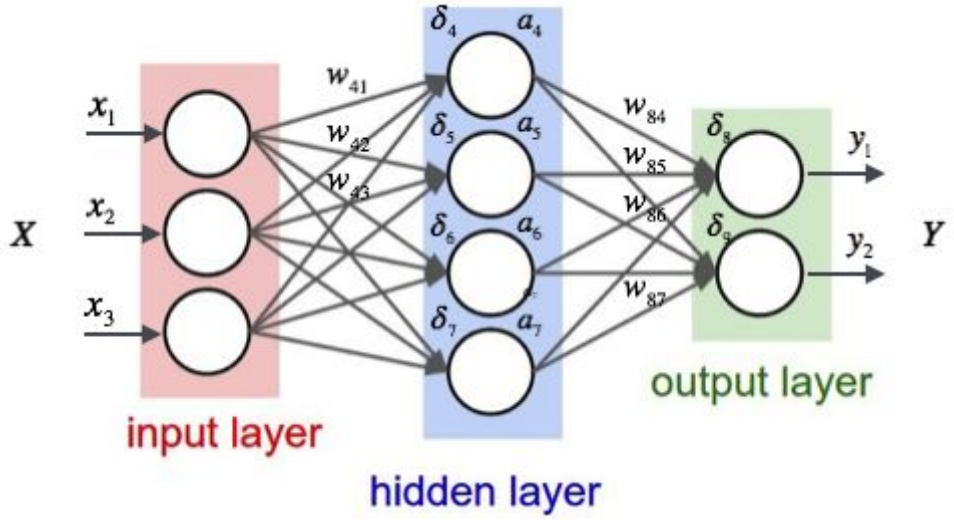


Figure 1.1 - Diagram for a layered neural network. The input layer is represented by x where the input data enters. The data is propagated through the weights w, which connects the nodes to the hidden layer and further to the output layer y. [7]

A perceptron essentially takes multiple numbers as input like x1,x2,x3 and produces a single binary output. Whether the output is 0 or 1 is determined by a threshold value and whether the sum of wj and xj are higher or lower than that value. An equation for this is provided below.[5]

$$output = \begin{cases} 0 \ if \sum_j w_j x_j < threshold \\ 1 \ if \sum_j w_j x_j > threshold \end{cases}$$

For this experiment, the features that will be implemented in the bot contain categorical data, which the neural network can't use. Because of this, it had to be one-hot encoded. One-hot encoding is a way of representing this categorical data as binary vectors[1], which the neural network can process and use to learn from.

2.3 Provided bots

Two bots that were provided with the framework play an important role in the evaluation of the hypothesis. The first is a simple implementation of the Perfect Information Monte-Carlo Sampling strategy where the bot plays a game randomly an x amount of times for a certain depth of y. This bot is important as it will be used to train all the machine learning bots that will be evaluated, including the control bot. The number of samples for this bot, called "rdeep", is 4 and the depth it searches to is 8.

The other important bot is the control bot, a simple intelligent agent that utilizes neural networks using sklearn. This bot came with 11 built in features, namely:

1. Perspective
2. Normalized points for player 1
3. Normalized points for player 2
4. Normalized pending points for player 1
5. Normalized pending points for player 2
6. Trump suit
7. Phase
8. Stock size
9. Leader
10. Whose turn
11. Opponent's last played card

3 Research Question:

Feature selection and optimization is an integral part of machine learning. It has been found and proven that irrelevant or redundant features may slow down the learning process and produce overfitting. However, the problem of finding an "optimal" set of features is NP-hard, and there is no perfect way of classifying whether a feature is relevant or not. One suggested method is ranking features in three separate categories: strongly relevant, weakly relevant and irrelevant. However, there exists no specific threshold that separates these categories. For example, the weakly relevant category is defined as features who may be relevant under certain circumstances, [7] which could be a potentially large amount of features, some relevant under most circumstances whilst others are only rarely relevant. As such, the features evaluated in this paper will be ranked from most relevant to least, and based on that ranking different features will be implemented in different bots.

To find an optimal set of features for an intelligent agent playing Schnapsen, different combinations of features, dependent on their ranking, will be examined. Four bots, each

containing different feature sets, will be evaluated against the provided control bot to examine the effectiveness of the used feature set. The four bots that will be evaluated are:

1. AllFeatureBot: This bot will contain all features that were initially ranked using RFE.
2. 11MostRelevantBot: This bot will contain the eleven features deemed most relevant by the ranking.
3. 11LeastRelevantBot: This bot will contain the eleven features deemed least relevant.
4. MixedBot: The mixed bot will contain the six highest ranked features as well as the five features ranked directly below the eleven highest ranked features.

The number of features, eleven, was selected as the bot initially provided had eleven features.

Utilizing the previously found research, the following hypothesis were developed:

Hypothesis 1: As irrelevant features may cause overfitting, the bot containing the eleven most relevant features will perform the best against the control bot, and will achieve at least an even win rate (50%) against it.

Hypothesis 2: The training time of a bot is affected both by the total amount of features in the feature set as well as the existence of irrelevant features. As such, the bot containing the eleven most relevant features should have the lowest training time, while the bot containing the complete set of features should display the longest training time.

4 Experimental Setup:

The RFE algorithm that was implemented in the training program outputs a number between 1 and n(number of features) in the dataset in order to determine the best rank, where 1 is the best and n is the worst. When trying to determine the rank for each feature, it was observed that ranks may change between training sessions, likely due to the random nature of Schnapsen causing certain properties to be more or less relevant during different games. Due to this, the average of 10 rankings were taken.

Feature	Average ranking according to RFE after 10 training sessions
Number of trump cards in hand	66.4
Number of king's, queen's or jack's in hand	129
Number of 10's or ace's in hand	121.6
If there is a marriage in hand	145.6
Proximity to 33 for player 1	40.4
Proximity to 66 for player 1	1
Proximity to 33 for player 2	35.4
Proximity to 66 for player 2	1
If a trump exchange is possible	126.5
Normalized points for player 1	26.6
Normalized points for player 2	25.8
Normalized pending points for player 1	13.8
Normalized pending points for player 2	10.4
Trump suit	73.3
Phase	122.5
Stock Size	84.2
Leader	52.1
Whose turn	109.3
Opponent's last played card	91.8
Perspective	69.9

Table 2.1 - Table showing the average RFE ranking of each feature. Values closest to 1 are rated as most important.

Table 2.1 was used to determine which feature set each bot would employ (see appendix A for the feature sets for each bot).

Each bot was trained with 2000 games while observing the rdeep bot, which was provided from the Intelligent Systems Framework. To measure the performance of the bots, each bot will play 1000 games against the control bot.

5 Results and Findings:

In Table 2.1 the results of the tournaments of each bot against the control bot can be seen. There is an immediate trend that can be seen from this table. From all the 4 bots, the bot with allFeatures performed the best by winning 529 out of 1000 games. That is a win percentage of 52.9%, which is not very high. Compared to this, the 11MostRelevant bot is only a few wins behind with 517 total wins. Considering that this bot had the best ranked features implemented, that is not a very good performance and was unexpected. The Mixed and 11LeastRelevant bot performed as was expected. They both lost more than half of the games to the control bot. The Mixed bot with 527 losses and the 11LeastRelevant bot with 561 losses.

	allFeatures Bot	11MostRelevant Bot	Mixed Bot	11LeastRelevant Bot
Win	529	517	473	439
Loss	471	483	527	561

Table 2.2 - Table showing the wins/losses of each bot when playing a tournament of 1000 games against the control bot.

Hypothesis 1 stated, as irrelevant features may cause overfitting, the bot containing the eleven most relevant features will perform the best against the control bot, and will achieve at least an even win rate (50%) against it. According to the experiment results seen above, the 11MostRelevant bot has achieved a minimally higher win rate against the control bot of 51.7%. However, the bot did not perform the best against the control bot. The allFeatures bot with 52.9% outperformed the 11MostRelevant bot and hence has the highest win rate against the control bot. This was unexpected since having irrelevant or weakly relevant features in a model can cause misleading data reducing the overall accuracy of said model.[6] This can be seen in the 11LeastRelevant bot, which lost more than any other bot. Implementing the irrelevant features in this bot has caused it to underperform and not be as accurate as other bots.

In Table 2.3 the results of the average training time of each bot can be seen. The lowest average training time was by the Mixed bot with a time of 0.063 minutes, which is 3.78 seconds. This is considerably lower than all the other bots. The second fastest training time was achieved by the 11MostRelevant bot with a time of 0.207 minutes, which is 12.42 seconds. This means that the Mixed bot is approximatly 4 times faster than the second fastest time. The longest training time was by the allFeatures bot with 0.444 minutes, which is 26.64 seconds.

Feature set	The average training time of 10 training sessions with 2000 games(minutes)
Control Feature set	0.25544
All Features set	0.44406
11 Most Relevant Features set	0.206802
11 Least Relevant Features set	0.316561
11 Mixed Features set	0.062863

Table 2.3 - Table showing the average training time of each bot when trained with 2000 games while observing rdeep

Hypothesis 2 stated that the training time of a bot is affected both by the total amount of features in the feature set as well as the existence of irrelevant features. As such, the bot containing the eleven most relevant features should have the lowest training time, while the bot containing the complete set of features should display the longest training time.

According to the averages shown in Table 2.3, the allFeatures bot, which contained the most features, did take the longest to train by a large margin. This highlights how the amount of features can negatively impact training time. However, according to the hypothesis, the 11MostRelevant bot was supposed to have the fastest training time as it had the least amount of irrelevant features and also less features in total compared to the maximum number of features. The fact that the Mixed bot had the fastest training time was unexpected. According to the hypothesis, since the bot contained some irrelevant features it should not have been this fast in training. It being faster may be due to the fact that one hot encoding a certain feature adds a lot more data than encoding another feature. For example, encoding the perspective feature(Appendix B) adds 120 total binary numbers to the dataset and the model. This feature was not included in the Mixed bot as it was not in the rank range. This may explain why the bot was trained a lot faster. Perhaps the training time does not completely depend on the amount of features, but rather the amount of variables and data present in the dataset and model.

6 Conclusion:

This paper attempted to find an optimal set of features for a machine learning agent that plays schnapsen. Through the use of four different bots, evaluated against a control bot, it was found that there exists a combination of features that improves both the learning time of the agent as well as its performance. However, due to the scope of the problem - there exists not only a large amount of possible features for a learning agent playing Schnapsen, but an infinite number of combinations between them - it can not be determined whether or not an actually “optimal” set has been found. Within the scope of this paper, however, the optimal feature set was not the 11MostRelevant one as hypothesised, but the AllFeaturesBot instead.

This revelation hints at several properties of features and their effective selection. Firstly, ranking features in one constant feature set may not fully explain the relevancy of individual features. For this paper, all included features were ranked together, however different combinations of features, as well as the inclusion of other not-thought-of features, may change the ranking. This is proven by the performance of the AllFeaturesBot, which performed better than the 11MostRelevantBot. Another property of feature selection is that it is difficult to properly categorize relevant features. In this paper the features were selected based on ranking, but other papers suggest categorizing features based on strong to weak relevancy. However, this categorization would be relatively arbitrary for this paper as there is no set connection between a features rank as found by RFE and what category it would belong to. In addition, the rankings of all but two features (points to 66 for both players) changed throughout training. This would imply that those two are the only “strongly relevant” features present, as the definition for this category is that a feature must always be relevant to be considered “strongly relevant”. Therefore, more so than finding the optimal set of features for a specific game/situation, or for a certain list of features, it may be more applicable to research proper categorization techniques of features, as that is extendable past the scope of any one paper or learning agent etc.

There were a handful of issues encountered during the creation and evaluation of the bots explored in this paper. One important aspect of the bots that could have impacted results was the usage of one-hot encoding to add categorical data to the feature set. The problem this created was that each element in a binary one-hot encoded array, that would together describe a single categorical feature, was ranked individually by the RFE algorithm. To alleviate this problem, the individual rankings for each element of an array corresponding to one one-hot encoded categorical feature was averaged to obtain a general ranking for each individual feature. This was the best compromise that could be found for this paper. This also resulted in the ranking not being from 1 to 20, as would be suggested by our total number of features, but from 1 to 168 instead. An obvious improvement would be to explore other ranking algorithms that work better with one-hot encoding, or utilizing a different encoding scheme that would work better with the RFE algorithm. Another issue with the use of one-hot encoding was that it potentially skewed the training time of the learning agents, as can be seen by the MixedBot having the lowest training time while also being the only one that did not include the “perspective” feature, which due to the use of one-hot encoding contributed for a large amount of data (an array of 120 elements).

Lastly, some possible future research opportunities that have been discovered during the writing of this paper include:

1. As discussed before, an interesting alternative research question would be to explore proper categorization techniques for features that could be more applicable to other scenarios, as well as exploring proper boundaries between feature relevance categories.
2. Another research opportunity would be to create a more exhaustive set of features for a learning agent playing Schnapsen and attempting to hone in on a more definitive set of features that could be called optimal.
3. Lastly, this paper did not delve into properly representing the effects feature sets have on overfitting. Therefore, a more numerically focused paper that explores this relationship may present interesting findings.

7 References:

[1] Brownlee, J. (2017). *How to One Hot Encode Sequence Data in Python*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/ [Accessed 31 Jan. 2019].

[2] Brownlee, J. (2016). *Feature Selection For Machine Learning in Python*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/feature-selection-machine-learning-python/ [Accessed 31 Jan. 2019].

[3] Computersciencedegreehub.com. (2019). *Is Artificial Intelligence a Growing Field?*. [online] Available at: https://www.computersciencedegreehub.com/faq/artificial-intelligence-growing-field/ [Accessed 31 Jan. 2019].

[4] McLeod, J. (2016). *Rules of Card Games: Schnapsen*. [online] Pagat.com. Available at: https://www.pagat.com/marriage/schnaps.html [Accessed 31 Jan. 2019].

[5] Nielsen, M. (2018). *Neural Networks and Deep Learning*. [online] Neuralnetworksanddeeplearning.com. Available at: http://neuralnetworksanddeeplearning.com/chap1.html [Accessed 31 Jan. 2019].

[6] Tang, J., Alelyani, S. and Liu, H. (2017). *Feature Selection for Classification: A Review*. [online] pp.4,5. Available at: https://pdfs.semanticscholar.org/310e/a531640728702fce6c743c1dd680a23d2ef4.pdf [Accessed 31 Jan. 2019].

[7] Yu, L., Liu, H.: Efficient feature selection via analysis of relevance and redundancy. Journal of Machine Learning Research 5, 12051224 (2004)[Accessed 31 Jan. 2019].

Appendix A: Feature Sets for Each Bot

11MostRelevantBot:

The 11MostRelevantBot includes the following features, in order of highest ranked to lowest ranked:

1. Proximity to 66 for player 2
2. Proximity to 66 for player 1
3. Normalized pending points for player 2
4. Normalized pending points for player 1
5. Normalized points for player 2
6. Normalized points for player 1
7. Proximity to 33 for player 2
8. Proximity to 33 for player 1
9. Leader
10. Number of trump cards in hand
11. Perspective

11LeastRelevantBot:

The 11LeastRelevantBot includes the following features, in order of lowest ranked to highest ranked:

1. If there is a marriage in hand
2. Number of king's, queen's or jack's in hand
3. If a trump exchange is possible
4. Phase
5. Number of 10's and ace's in hand
6. Whose turn
7. Opponent's last played card
8. Stock size
9. Trump suit
10. Perspective
11. Number of trump cards in hand

MixedBot:

The MixedBot has the following feature set, in order of highest ranked to lowest ranked:

1. Proximity to 66 for player 2
2. Proximity to 66 for player 1
3. Normalized pending points for player 2
4. Normalized pending points for player 1
5. Normalized points for player 2
6. Normalized points for player 1
7. Trump suit
8. Stock size
9. Opponent's last played card
10. Whose turn
11. Number of 10's and ace's in hand

Appendix B: Implementation of Features and RFE

Implementation of features:

```
feature_set = []
# 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()

# 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()

hand = state.hand()

# Append ratio of number of trump cards in hand
counter = 0
for card in hand:
    i = util.get_suit(card)
    if (i == trump_suit):
        counter += 1
if (counter == 0):
    feature_set.append(0)
else:
    ratio = counter/len(hand)
    feature_set.append(ratio)

# Append ratio of number of cheap cards in hand
counter = 0
for card in hand:
    if (card % 5 == 2) or (card % 5 == 3) or (card % 5 == 4):
        counter += 1
if (counter == 0):
    feature_set.append(0)
else:
    ratio = counter/len(hand)
    feature_set.append(ratio)
```

```
# Append ratio of number of expensive cards in hand
counter = 0
for card in hand:
    if (card % 5 == 0) or (card % 5 == 1):
        counter += 1
if (counter == 0):
    feature_set.append(0)
else:
    ratio = counter/len(hand)
    feature_set.append(ratio)

# Append number of marriages in hand
marriages = 0
for card in hand:
    x = util.get_suit(card)
    if (card % 5 == 2):
        for card1 in hand:
            y = util.get_suit(card1)
            if (card % 5 == 3) and x == y:
                marriages += 1
    if (card % 5 == 3):
        for card1 in hand:
            y = util.get_suit(card1)
            if (card % 5 == 2) and x == y:
                marriages += 1
feature_set += [1, 0] if marriages == 0 else [0, 1]

# Append how close to 33 points p1 is
feature_set.append(1 if p1_points > 33 else p1_points / 33)

# Append how close to 66 points p1 is
feature_set.append(1 if p1_points > 66 else p1_points / 66)

# Append how close to 33 points p1 is
```

```
    feature_set.append(1 if p2_points > 33 else p2_points / 33)

    # Append how close to 66 points p1 is
    feature_set.append(1 if p2_points > 66 else p2_points / 66)

    # Append if you have trump jack in hand
    has_trump_jack = 0
    for card in hand:
        i = util.get_suit(card)
        if (i == trump_suit) and (card % 5 == 4):
            has_trump_jack = 1
    feature_set += [1, 0] if has_trump_jack == 0 else [0, 1]



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

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

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

    # Append one-hot encoded phase to feature set
```

```
    feature_set += [1, 0] if phase == 1 else [0, 1]

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

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

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

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

    perspective = state.get_perspective()

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

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

Implementation of RFE (only the relevant code that was added to train-ml-bot.py is added):

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE

model1 = LogisticRegression()
rfe = RFE(model1, 12)
fit = rfe.fit(data, target)
print("Num Features: %d") % fit.n_features_
print("Selected Features: %s") % fit.support_
print("Feature Ranking: %s") % fit.ranking_
```

Appendix C: Worksheets

Worksheet 1

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

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%): [1, 2, 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%): [4, 4, 0]
Played 9 out of 30 games (30%): [5, 4, 0]
Played 10 out of 30 games (33%): [6, 4, 0]
Played 11 out of 30 games (37%): [6, 4, 1]
Played 12 out of 30 games (40%): [6, 4, 2]
Played 13 out of 30 games (43%): [6, 4, 3]
Played 14 out of 30 games (47%): [7, 4, 3]
Played 15 out of 30 games (50%): [8, 4, 3]
Played 16 out of 30 games (53%): [9, 4, 3]
Played 17 out of 30 games (57%): [9, 4, 4]
Played 18 out of 30 games (60%): [9, 4, 5]
Played 19 out of 30 games (63%): [9, 4, 6]
Played 20 out of 30 games (67%): [9, 4, 7]
Played 21 out of 30 games (70%): [9, 4, 8]
Played 22 out of 30 games (73%): [9, 4, 9]
Played 23 out of 30 games (77%): [9, 5, 9]
Played 24 out of 30 games (80%): [9, 5, 10]
Played 25 out of 30 games (83%): [9, 5, 11]

Played 26 out of 30 games (87%): [9, 5, 12]

Played 27 out of 30 games (90%): [9, 5, 13]

Played 28 out of 30 games (93%): [9, 5, 14]

Played 29 out of 30 games (97%): [9, 5, 15]

Played 30 out of 30 games (100%): [9, 5, 16]

Results:

bot <bots.rand.rand.Bot instance at 0x7f077a2b4ab8>: 9 wins

bot <bots.bully.bully.Bot instance at 0x7f077a2b4a28>: 5 wins

bot <bots.rdeep.rdeep.Bot instance at 0x7f077a2bc950>: 16 wins

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

If the bot has a trump card it plays that card. If the opponent played a card first then the bot tries to play the highest card of that suit. If the bot can't follow suit it will play the highest card. It employs a greedy playing style.

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

The bot uses a greedy approach as a strategy. This would not work well for us because the bot determines the best move at the time of its turn and does not take into account the best long term solution. A game like Schnapsen needs to be played with future moves and cards taken into account and not by just playing the current optimal move.

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

Have rdeep and rand play against each other to gather as much statistical information as possible. Once sufficient information is available a significance test can determine which of the bots performs better and hence provide evidence in that regard.

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

Playing 100 games:

Played 1 out of 100 games (1%): [1, 0]

Played 2 out of 100 games (2%): [2, 0]

Played 3 out of 100 games (3%): [3, 0]

Played 4 out of 100 games (4%): [4, 0]

Played 5 out of 100 games (5%): [4, 1]

Played 6 out of 100 games (6%): [4, 2]

Played 7 out of 100 games (7%): [4, 3]

Played 8 out of 100 games (8%): [4, 4]

Played 9 out of 100 games (9%): [4, 5]

```
...
Played 98 out of 100 games (98%): [52, 46]
Played 99 out of 100 games (99%): [53, 46]
Played 100 out of 100 games (100%): [54, 46]
Results:
    bot <bots.mybot.mybot.Bot instance at 0x7f2e5d694560>: 54 wins
    bot <bots.rand.rand.Bot instance at 0x7f2e5d694b00>: 46 wins
```

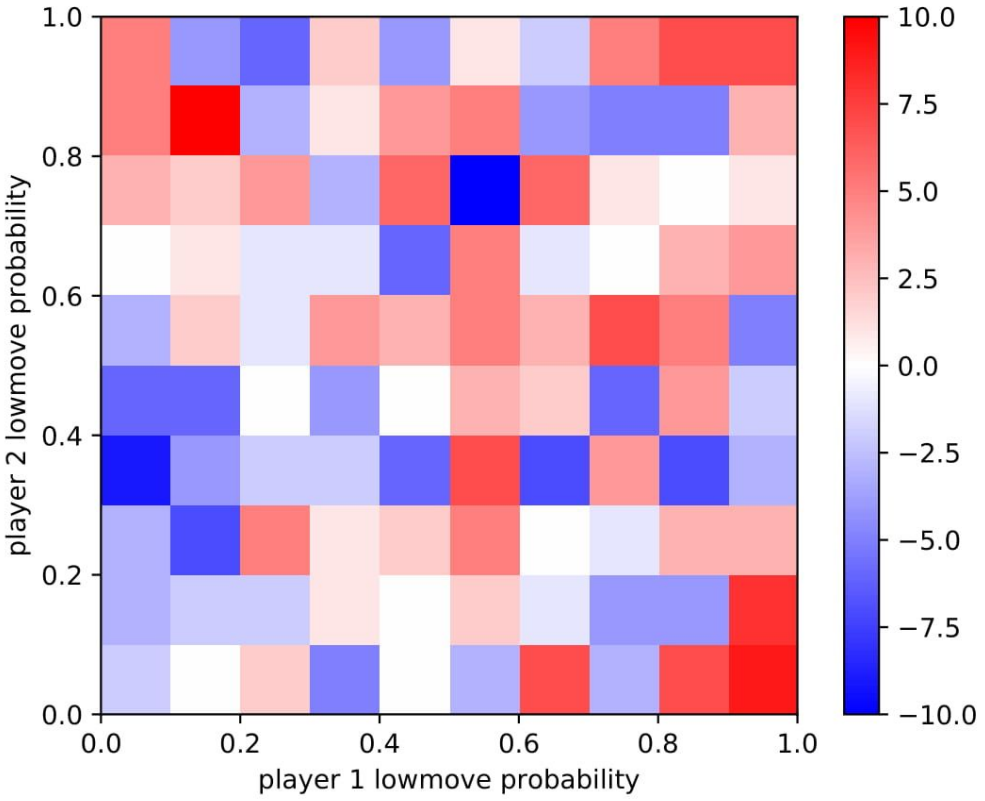
We implemented mybot with a simple set of rules. If the opponent leads the bot will try to follow suit with the lowest card. If it can't follow suit it will play a trump card. When the player has the turn it will play the highest card available. Above you can see the typical result. We made them play 100 games and as you can see the results were quite close.

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

#if opponent plays first try to follow suit with lowest card. Otherwise play lowest card
if state.get_opponents_played_card() is not None:
   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()):
         if move[0] % 5 > chosen_move[0] % 5:
            chosen_move = move
      else:
         if Deck.get_suit(move[0]) == state.get_trump_suit():
            chosen_move = move
return chosen_move

# Play highest card
for index, move in enumerate(moves):
   if move[0] is not None and move[0] % 5 < chosen_move[0] % 5:
      chosen_move = move
return chosen_move
```

Question 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.



From the heatmap depicted above we can see that the lowmove probability of any player does not seem to influence winning probability.

Question 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, _ = self.value(next_state, (depth + 1))

Playing 10 games:
Played 1 out of 10 games (10%): [1, 0]
Played 2 out of 10 games (20%): [2, 0]
Played 3 out of 10 games (30%): [3, 0]
Played 4 out of 10 games (40%): [4, 0]
Played 5 out of 10 games (50%): [5, 0]
Played 6 out of 10 games (60%): [6, 0]
Played 7 out of 10 games (70%): [7, 0]
Played 8 out of 10 games (80%): [8, 0]
Played 9 out of 10 games (90%): [9, 0]
Played 10 out of 10 games (100%): [9, 1]
Results:
bot <bots.minimax.minimax.Bot instance at 0x7fccf2e44518>: 9 wins
bot <bots.rand.rand.Bot instance at 0x7fccf2e44ab8>: 1 wins

Question 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.

Difference of opinion! Minimax said: (5, None), alphabeta said: (3, None). State: The game is in phase: 2
Player 1's points: 35, pending: 0
Player 2's points: 35, pending: 0
The trump suit is: S
Player 1's hand: QC JC AD QD AS
Player 2's hand: JD 10H KH JH JS

There are 0 cards in the stock

Difference of opinion! Minimax said: (9, None), alphabeta said: (8, None). State: The game is in phase: 2

Player 1's points: 29, pending: 0

Player 2's points: 29, pending: 0

The trump suit is: H

Player 1's hand: AC KC 10D KD AH

Player 2's hand: QD JD 10H KH QH

There are 0 cards in the stock

Difference of opinion! Minimax said: (4, None), alphabeta said: (3, None). State: The game is in phase: 2

Player 1's points: 40, pending: 0

Player 2's points: 40, pending: 0

The trump suit is: H

Player 1's hand: QC JC QD 10S JS

Player 2's hand: KC 10D 10H KH AS

There are 0 cards in the stock

Agreed.

Agreed.

Difference of opinion! Minimax said: (4, None), alphabeta said: (0, None). State: The game is in phase: 2

Player 1's points: 26, pending: 0

Player 2's points: 26, pending: 0

The trump suit is: S

Player 1's hand: AD KD AH AS JS

Player 2's hand: AC JC QD 10S QS

There are 0 cards in the stock

Agreed.

Agreed.

Difference of opinion! Minimax said: (10, None), alphabeta said: (3, None). State: The game is in phase: 2

Player 1's points: 41, pending: 0

Player 2's points: 41, pending: 0

The trump suit is: S

Player 1's hand: QC AH 10H QH KS

Player 2's hand: JC AD 10D JH JS

There are 0 cards in the stock

Done. time Minimax: 0.00570885340373, time Alphabeta: 0.000454664230347.

Alphabeta speedup: 12.5561964691

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

These bots use a very simple heuristic based on the ratio of points the player has compared to the total points earned so far. One way to potentially improve this heuristic is to add the value of a person's current hand to the heuristic, as more expensive cards would suggest a stronger position:

```
def heuristic(state):
        counter = 0
        hand = state.hand()
                for card in hand:
                        if(card % 5 == 0):
                                counter += 11
                        if(card % 5 == 1):
                                counter += 10
                        if(card % 5 == 2):
                                counter += 4
                        if(card % 5 == 3):
                                counter += 3
                        if(card % 5 == 4):
                                counter += 2
        ratio = counter / 100
        ratio = (ratio + util.ratio_points(state, 1)) / 2
        return ratio * 2.0 - 1.0, None
```

This gives the following results:

Results:

bot <bots.alphabeta.alphabeta.Bot object at 0x7f7746929b70>: 59 points

bot <bots.alphabeta2.alphabeta2.Bot object at 0x7f7746939358>: 62 points

Results:

bot <bots.rand.rand.Bot object at 0x7fc5932b4cf8>: 33 points

bot <bots.alphabeta2.alphabeta2.Bot object at 0x7fc5932b43c8>: 100 points

Results:

bot <bots.rand.rand.Bot object at 0x7fdd535ff048>: 44 points

bot <bots.alphabeta.alphabeta.Bot object at 0x7fdd535f9748>: 85 points

Worksheet 2

Question 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)

Question 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?

CNF

A V B

-B V A

-A V C

-A V D

A: True, C: True, B: False, D: True

A: True, C: True, B: True, D: True

True

Question 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 only integer value can be used, then the only value both x and y can have that would make these constraints true is 2.

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

Question 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.

(x + y > 2 ^ x + y < 5) V (x + y > -5 ^ x + y < -2)

Question 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.

kb.add_clause(A4)

kb.add_clause(A9)

kb.add_clause(A14)

kb.add_clause(A19)

kb.add_clause(~A4, PA4)

kb.add_clause(~A9, PA9)

kb.add_clause(~A14, PA14)

kb.add_clause(~A19, PA19)

kb.add_clause(~PA4, A4)

kb.add_clause(~PA9, A9)

kb.add_clause(~PA14, A14)

kb.add_clause(~PA19, A19)

Question 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.

```
# GENERAL INFORMATION ABOUT THE CARDS
# This adds information which cards are expensive
kb.add_clause(E0)
kb.add_clause(E1)
kb.add_clause(E10)
kb.add_clause(E11)
kb.add_clause(E20)
kb.add_clause(E21)
kb.add_clause(E30)
kb.add_clause(E31)

# Add here whatever is needed for your strategy.
kb.add_clause(~E0, PE0)
kb.add_clause(~E1, PE1)
kb.add_clause(~E10, PE10)
kb.add_clause(~E11, PE11)
kb.add_clause(~E20, PE20)
kb.add_clause(~E21, PE21)
kb.add_clause(~E30, PE30)
kb.add_clause(~E31, PE31)
kb.add_clause(~PE0, E0)
kb.add_clause(~PE1, E1)
kb.add_clause(~PE10, E10)
kb.add_clause(~PE11, E11)
kb.add_clause(~PE20, E20)
kb.add_clause(~PE21, E21)
kb.add_clause(~PE30, E30)
kb.add_clause(~PE31, E31)
```

Question 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.

Results:

bot <bots.kbbot.kbbot.Bot instance at 0x7fd63c55f0e0>: 5 wins

bot <bots.modifiedkbbot.modifiedkbbot.Bot instance at 0x7fd61c4915f0>: 15 wins

Question 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.

Results:

bot <bots.consqkbbot.consqkbbot.Bot instance at 0x7f68219ef0e0>: 9 wins

bot <bots.modifiedkbbot.modifiedkbbot.Bot instance at 0x7f68019215f0>: 11 wins

As can be seen above, our bot did not perform better than the simpler bot. Playing multiple tournaments showed that this is a typical result. Our bot only loses by a small margin. This leads us to think that perhaps the implementation has been done in the wrong way.

Worksheet 3

Question 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.

```
value = self.heuristic(next_state)
# 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()

# 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()
```

Results:

bot <bots.rand.rand.Bot instance at 0x7f75380d5cf8>: 15 points

bot <bots.bully.bully.Bot instance at 0x7f75380d5c20>: 9 points

bot <bots.ml.ml.Bot instance at 0x7f75380ff4d0>: 25 points

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

We used rdeep to train the bot and let it play 2000 games.

Results:

bot <bots.rand.rand.Bot instance at 0x7f287f735cf8>: 9 points

bot <bots.bully.bully.Bot instance at 0x7f287f735c20>: 15 points

bot <bots.ml.ml.Bot instance at 0x7f287f75f4d0>: 29 points

The bot trained on rdeep only did minimally better than the bot trained on rand. Perhaps with more games the rdeep bot will do better.

Question 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.

Results:

bot <bots.mlrand.mlrand.Bot instance at 0x7f226835f0e0>: 11 points

bot <bots.mlrdeep.mlrdeep.Bot instance at 0x7f2248756fc8>: 11 points

bot <bots.mlrdeeprand.mlrdeeprand.Bot instance at 0x7f2248286908>: 22 points

The bot trained on ml did substantially better than the other two bots trained on rand and rdeep. However, when we ran a second attempt we received this.

Results:

bot <bots.mlrand.mlrand.Bot instance at 0x7ff5ea66f0e0>: 16 points

bot <bots.mlrdeep.mlrdeep.Bot instance at 0x7ff5c9026fc8>: 17 points

bot <bots.mlrdeeprand.mlrdeeprand.Bot instance at 0x7ff5c8b56908>: 19 points

This shows that the games are very close and that the bots are getting similar points.

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

We added a simple feature from rdeep concerning the game when in phase 1.
makeAssumption = state.make_assumption() if state.get_phase() == 1 else state
The following is the outcome of the added feature.

Results:

bot <bots.mlrdeep.mlrdeep.Bot instance at 0x7fb85a2df0e0>: 14 points

bot <bots.mlrand.mlrand.Bot instance at 0x7fb83879a560>: 7 points

bot <bots.mlfeatures.mlfeatures.Bot instance at 0x7fb83879a830>: 19 points

Most games that we run with this features showed that the bot does outperform the other two bots and only loses on rare occasion.