CE811

'The Resistance' Bot

Assignment 1 -The Resistance

Name: Fred Knight
Registration Number: 1804162

Contents

Summary

This report is on my submitted bot for the game ‘The Resistance’; a game which consists of two teams: the resistance who are trying to complete 3 missions, and the spies who are trying to sabotage three missions Research that I did about this game before working on my bot was to look at similar games that I had either played myself or knew about. From this, I could get ideas on the behaviours my bot should have.

I had decided to use the neural network bot from the third lab as the base of my bot, which I have then worked on to try and improve the neural network model created by the classifier by altering hyperparameters and adding some functionality to optimise the returns of my model. I also added behaviour to try and improve its performance in-game as previously the bot didn’t do much other than predicting the probability of someone being a spy.

Introduction

The problem was to create a bot that would be able to play the game ‘The Resistance’, and that would be able to successfully compete against other bots within the ‘The Resistance’ Python framework. To do this, an algorithm that was discussed in the lectures and/or labs would have to be used and adapted to fit the behaviour of the bot we desired.

This paper will be explaining what the game ‘The Resistance’ is, as well as similar games that I have either played myself or know about, that would aid in providing ideas for behaviours that may increase the bot’s chance of success. I will then explain my chosen algorithm used for the bot; describing how the bot learns information about other players to try to increase its chance of success.

Lastly, I will be explaining the process in which I added this functionality to the bot and describe what I had done to try and improve it at each step and how effective this had been.

The Resistance

The resistance is a board game consisting of five turns, and where players are members of one of two teams: resistance and spies. Resistance team members need to select and vote for teams to go on missions that will increase their chance of success, whilst spies need to remain undetected whilst trying to sabotage a mission. The less suspicious the spy seems, the more likely it is to be sent on a mission. The objective of the resistance team members is to complete three missions successfully, whilst the objective of the spies is to successfully sabotage three missions; so, this game is essentially a ‘best-three-of-five’ turn game.

Each turn in a game consists of three stages: team selection, voting, and mission sabotaging. The team selection stage is where a team leader is allowed to select members for their team up to the limit for that round. If the leader is a resistance member, they will want to choose members that are likely to pass the mission, and spies would want to choose teams that have the possibility of failing. The voting stage is where every player (including the team leader) can vote on whether the selected team goes on a mission. If most team members vote for the team, the team goes on a mission. If they vote against, then the team is discarded, and the next player then becomes the team leader and can select a new team. If five teams are voted down in a row, then the resistance loses the game. Finally, the sabotaging phase is when any spies present on the team can choose to sabotage the mission or not. If no sabotage is made, then the mission is successful, and the resistance gains a point. If the mission is sabotaged, then the mission fails, and the spies gain a point. A mission can only be sabotaged if a spy is present on the team at this stage.

Various other games share the style of this game. The closest one that I have had experience with is a game titled 'Secret Hitler'. This game is played in a very similar fashion to 'The Resistance', however, there is an additional special role of 'Hitler'. If the player designated this role is killed, then the 'Liberal' team wins. If the player designated this role is elected as 'Chancellor' after the 'Fascist' team has gained 3 points, then they win [1].

There are two other games that are nearly identical to each other that I have had experience with: 'Mafia' [2] and 'Werewolves' [3]; both of which have a very similar style of playing as this game. In both games, roles are randomly assigned to players. There are two main roles, however, within these, there are also special roles that provide the game with a twist in each phase. Roles could include having members be able to reveal their identity to them, which they then need to convince the rest of their team is that player's role. Other roles could consist of either protecting members of your team or killing a player that they think could be on the enemy team. The objective of both games is to remove every member of the opposite team, or in the case of the enemy team (the 'Mafia' of the 'Werewolves'), they need to ensure that they are at least equal in number to the other team.

Background

The algorithm that the submitted agent's technique is based upon is a feed-forward neural network, as described in the third lab classifier notebook. A training bot is used to collect data that will be used to train the neural network. This data is based on a feature vector taken from each bot in the current game, such as their voting habits, perceived suspicion, and how many missions they had failed. The network then trains a model using most of this data, using a portion of it as input data and a portion of it as the expected output of the model. The part of the data that was not used is then tested against the model as unseen validation data, where it acts as external data against the model's prediction.

Using the functions, the neural network bot has inherited from the training bot, the neural network bot will collect data from each player, and at each phase, it will put them through the model to make a prediction.

Before deciding which algorithm to use for my agent, I did some research for agents that had been created or proposed for either this game, or any of the other games I have talked about in the previous section.

What I have found is that most proposals for social deduction games such as this base their ideas around the game ‘Werewolves’. A paper discussing strategies that an AI agent may have for the game [4], mentions multiple times how strong werewolf agents are able to manipulate the other team without drawing too much attention to itself; and that misleading the other team can often aid in this deception. This is something I would like to implement in my agent, as after calculating the probabilities of each player being a spy (including itself), it can use this against the other players by purposely making the safer players seem less trustworthy, such as purposely putting safe players in a position where the mission will fail and allowing team members that are more suspicious to successfully complete some missions. As the suspicion of the other players goes up, this agent’s suspicion in turn goes down, reducing the disparity between the chance of each player being a spy.

Another paper supports this [5], suggesting that basing behaviour off the other players’ perspective can greatly increase a bot’s ability to persuade the other players. This could be done as has been described above, as well as providing the other players information that they could want, such as using the ***say(self, message)*** or ***announce(self)*** functions. These functions allows the agent to communicate with the other players (assuming they’re able to understand or have the functionality to receive these messages); providing them with both information (be it truthful or fabricated), as well as getting clues on what other bots may be thinking. For this, some sort of natural language engineering could be implemented into the bot; allowing it to pick out key details from the messages that are provided based on their context (if the agents are discussing who the spy could be, if a sentence contains ‘not’ and ‘spy’, it is likely they are not suspecting the player they are talking about). However, for the bots within the *intermediates.py* file, this functionality is too advanced and will likely hinder the performance of this agent. If this functionality cannot be utilized properly.

Techniques Implemented

The submitted agent consists of five parts:

- ***fredbot.py:***

  This is a bot designed for the collection of training data for the neural network. It’s a simple ‘observer’ bot that monitors the behaviour of each other player in the game, such as voting habits, failed missions, missions been on, and its apparent suspicion to the bot. This data is then written to a log file *FredBot.log*, where it can be used by the neural network.

- ***neural_fredbot.py:***

  This is the actual agent that will be competing against the other bots. Using the functions inherited from the *fredbot.py* bot, it will create feature vectors of each other bot in the game as the game goes on. At every selection and voting phase, it will put these vectors into the model created by the classifier. The state of each player is then calculated as a probability; allowing the bot to identify what other bots are the most likely to be spies.

- ***classifier_fredbot.ipynb:***

  This is the neural network classifier that is used to create a model to calculate the possibility of a player being a spy, given a feature vector for each player. The neural network is trained using a set of feature vectors provided within the *FredBot.log* file. It takes these features and disregards the first four columns as these are simply the identification of the bot and the current game turn, which does not affect the probability of the bot being a spy. It then separates the last column from the remaining set, as this is the column we are wanting to predict. The model is then trained using a portion of the training data and tested using the remaining portion of the data.

- ***FredBot.log:***

  This is the log file that contains the training data (feature vectors of each bot) to train the neural network model.

- ***fredbot_classifier***

  This is the model created by the neural network, that is used to calculate the probability of each player being a spy at a point in the game.

The bot located within *neural_fredbot.py* has its effectiveness judged on how well it performs against the bots located within the *intermediates.py* file.

Experimental Study

Describe the experimental study performed to obtain the final submitted agent (e.g., parameter tuning, effect on the results of adding different functionality to the agent, etc.).

I first went about trying to improve the model from the original classifier by tuning its hyperparameters and adding extra layers to the neural network. I was trying to make the validation accuracy be as close to the shape of the training accuracy as possible and eventually ended up with the following model.

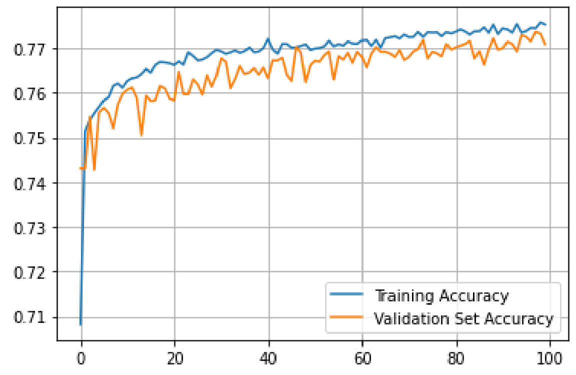


Figure 1: Screenshot of the model created by the original classifier

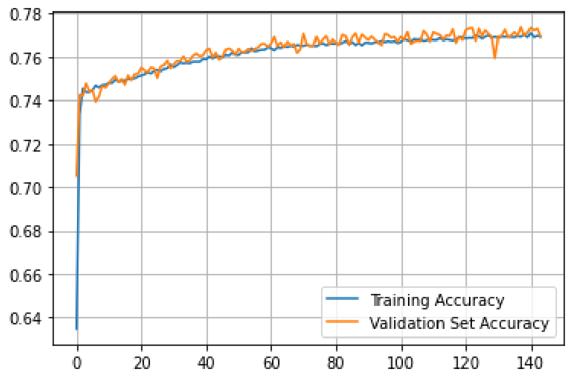
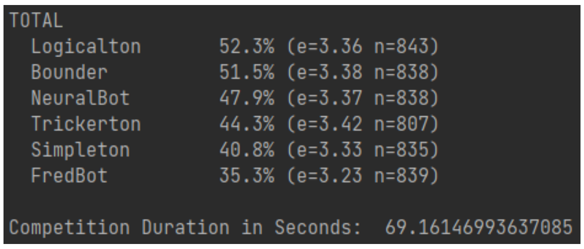


Figure 2: Screenshot of the model created by my classifier with tuned hyperparameters and additional layers

As can be seen, the validation set accuracy for the first model follows the trend of the training set, however, is constantly below the training set accuracy. This could indicate some overfitting, however, the model on the right uses the same proportion of its data for training as the model on the right. After experimenting with different values (0.7, 0.65, 0.6), it was clear that this was not the main factor as to the occurrence of this. I then decided to increase the number of layers from three to five and increase the number of neurons in each layer from ten to twenty. This helped in improving the validation accuracy in comparison to the test accuracy; however, the accuracy at each epoch still had a lot of variation. I then decided to reduce the batch size, increase the number of epochs, and decrease the learning rate of the neural network by a factor of 10; as currently, it seemed too high. As can be seen, these changes produced a much more consistent model.

I then went about trying to improve the efficiency of the bot, as it would be taking ages to run each time. In a game where a bot may be disqualified if it takes too long to make decisions, this is important to keep in mind. The original function to load the feature vector into the model would do this in a loop for each player; meaning that each time the function is called the neural network would be run five times. A more efficient way of doing this was to store the feature vector of each player in a NumPy array, which could then be evaluated using the model to perform all player calculations at the same time.

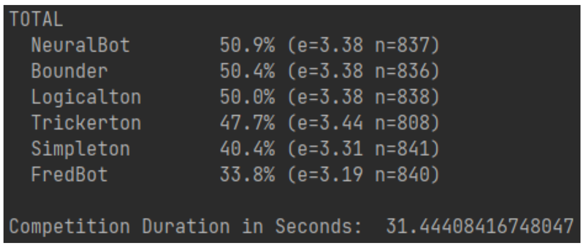
To monitor the exact difference this took, I measured the time taken to complete the competition within the *competition.py* file. This edit was purely to get a reading of exactly how more efficient this made the function, and the output can be seen at the bottom of the screenshots below. As can be seen, the new code results in the competition taking less than half the time to complete than the original code.



```
TOTAL
  Logicalton       52.3% (e=3.36 n=843)
  Bounder          51.5% (e=3.38 n=838)
  NeuralBot        47.9% (e=3.37 n=838)
  Trickerton       44.3% (e=3.42 n=807)
  Simpleton        40.8% (e=3.33 n=835)
  FredBot          35.3% (e=3.23 n=839)

Competition Duration in Seconds:  69.16146993637085
```

Figure 3: Screenshot of competition duration after 1000 runs with original function code



```
TOTAL
  NeuralBot        50.9% (e=3.38 n=837)
  Bounder          50.4% (e=3.38 n=836)
  Logicalton       50.0% (e=3.38 n=838)
  Trickerton       47.7% (e=3.44 n=808)
  Simpleton        40.4% (e=3.31 n=841)
  FredBot          33.8% (e=3.19 n=840)

Competition Duration in Seconds:  31.44408416748047
```

Figure 4: Screenshot of competition duration after 1000 runs with original function code

Now that I could run the competition at a much faster rate, I began to add better functionality to the bots. Currently, the bots from the third lab only perform basic behaviours: *loggerbot.py* tries to not affect the game too much by returning basic fixed values, and *neuralbot.py* makes decisions based on the calculated probabilities of other players being spies regardless of if the bot is a spy or not. I believed that by adding more conditions to the behaviour of the bots, I would be able to make better use of the calculated probabilities.

I first added to the ***select(self, players, count)*** function a conditional statement that changes the selection behaviour of the bot depending on if it is a spy or not. If not, it will keep the original bot's behaviour; as this behaviour tries to make the safest team based on its calculations. As it knows it is a resistance member it will always choose itself to go on a mission, before selecting the safest players in the current game for the rest of the team. If the bot is a spy, it will randomly select a spy to be on the team, before filling the rest of the team with the safest players. This is done to ensure that a spy is always on a team, but a single spy does not have too many failed missions. It also ensures that the possibility of players being a spy remains as close as possible, as exceptionally safe players will likely always be chosen for missions. As the win rate of these bots is somewhat by chance, I will run both the original and the new selection functions three times and take an average of the scores.

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate (%)
Run 1	76.7	31.0	49.9
Run 2	78.9	31.4	50.5
Run 3	76.8	32.7	49.8
Average	77.5	31.7	50.1

Table 1: Table of run scores of 1000 games with the original selection function

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate(%)
Run 1	81.9	31.9	52.2
Run 2	81.4	28.9	50.0
Run 3	82.4	32.3	52.5
Average	81.9	31.0	51.6

Table 2: Table of run scores or 1000 games with the new selection function

As can be seen, the average resistance win rate remains similar, as we did not change the behaviour for this during selection. However, the average spy win rate has increased drastically after adding the extra conditional statement.

I then went about altering the existing behaviour for the ***vote(self, team)*** method. The existing method checks to see if the bot currently has the spy role. If not, then it will check to see if any player within the proposed team is in the top two of players that are most likely to be spies. If so, it votes the team down, otherwise it votes for the team. Like the original selection function, the current voting function only contains a simple return, that is mostly in the resistance team's favour; however, leaves out some important game rules. If five teams have been rejected in a row, the spies automatically win. For this reason, the bot should always vote up the team if it is a resistance member, and the past four teams have all been rejected; and the bot should vote down the team if it is a spy for the same reason.

If the bot is a spy, it should check to ensure that all spies are not currently present on the team, by seeing if the set of spies is a subset of the team. If so, the bot should not vote up the team, as doing so could increase the risk for both spies as the other spy could sabotage and make all the spies seem suspicious. The scores after implementing this behaviour can be seen below.

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate(%)
Run 1	81.9	31.9	52.5
Run 2	82.0	32.4	52.5
Run 3	79.2	33.8	51.9
Average	81.0	32.7	52.1

Table 3: Table of run scores or 1000 games with the new voting function

As can be seen, this doesn't seem to have affected the win rate of the bot as a spy. However, this has slightly improved the win rate of the bot when on the resistance team, which could be due to the bot now trying to avoid an automatic loss when the team has already been rejected four times.

Lastly, I wanted to add behaviour to the ***sabotage(self)*** function. Currently the bot always sabotages the mission, which wouldn't be bad against very basic bots. However, most players will try to keep some sort of record of what players are on missions when they fail, which they then use to judge if a player is suspicious or not. The first check that should be made is to see if a sabotage will result in an instant win. This is possible if the spies have already sabotaged two missions, or if the current game turn is turn five. If this condition is met, the bot should always sabotage.

If this condition isn't met, the bot should look at who is in the team. If the team contains the full set of spies, it should opt to not sabotage; as to avoid increasing the risk for all spies in a single mission (although, the other spy could still sabotage). It should then check to see if the team contains one of the two most trustworthy characters. If so, then the bot should always sabotage to increase that player's risk. If none of these conditions are met, the bot will not sabotage.

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate(%)
Run 1	83.2	34.4	55.6
Run 2	87.8	31.9	52.6
Run 3	90.1	33.0	55.7
Average	87.0	33.1	54.6

Table 4: Table of run scores or 1000 games with the new sabotaging function

As can be seen from the scores, this had a drastic effect on the scores for when the bot is a spy, increasing the average win rate by over 5%.

Now that the main behaviour of the bot has been completed, I wanted to try and have my bot use the ***announce(self)*** function to broadcast the dictionary of players and their probabilities of being a spy to other players, as this is something I mentioned briefly in the 'Background' section of this report. Below is the bot's scores after implementing this function.

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate(%)
Run 1	89.0	32.8	56.2
Run 2	86.4	30.9	53.2
Run 3	87.9	34.7	56.2
Average	87.8	32.8	55.2

Table 5: Table of run scores or 1000 games with the announce function

As can be seen, the bots scores did not change all that much from implementing the previous function.

I decided to go back and test some different activation functions for my neural network, as I have currently only been using the tanh function that was set in the base classifier code. I decided to see what would happen if I altered nothing except from the activation value; leaving the other hyperparameters set before as they were. Below are the models that were created using these new activation functions.

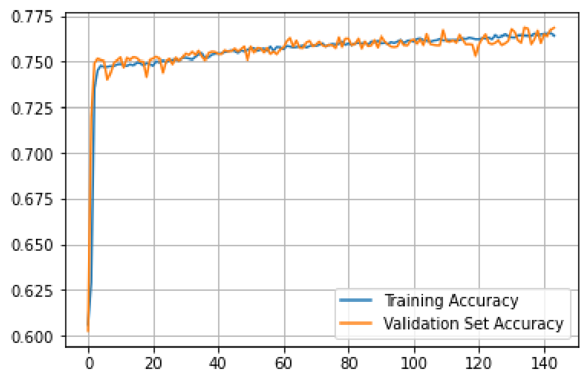


Figure 5: Screenshot of the model created by the classifier using a sigmoid activation function instead of tanh

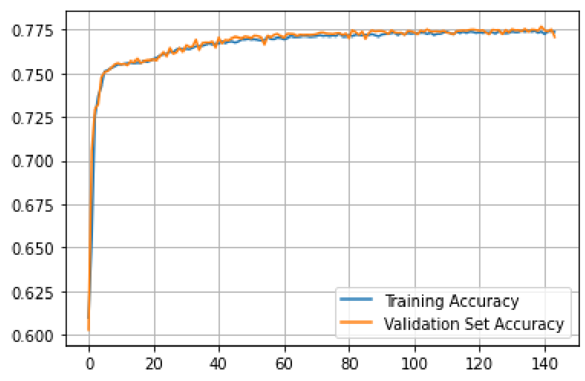


Figure 6: Screenshot of the model created by the classifier using a relu activation function instead of tanh

As can be seen from this, the relu activation function produced a slightly better model than that of my model that used the tanh activation function. The scores for the bot after using the new model can be seen below.

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate(%)
Run 1	85.6	35.5	53.7
Run 2	85.9	35.5	56.5
Run 3	85.9	33.0	53.9
Average	85.8	33.7	54.7

Table 6: Table of run scores or 1000 games with the model using a relu activation function

As can be seen in these results, although the scores are slightly lower than that of the previous model used, they remain much more consistent between runs. For this reason, I will continue using this activation function for my agent, as a consistent score shows that the model is consistently evaluating the feature vector of each player correctly.

Currently, the neural network is only training the model using the feature vectors relating to the bot '*Bounder*' within the *intermediates.py* file. Whilst this should be a good estimate of a competent player, this still means that data relating to the other players is not being learnt by the neural network. For this reason, I want to try and have the module use the data for the remaining bots in the file too. This involves replacing the line of code:

df=df0.query("PlayerName=='Bounder'") , with:

df=df0.query("PlayerName in ['Simpleton', 'Bounder', 'Logicalton', 'Trickerton']")

Below is the model created with this, as well as the scores that the bot gets from using this model.

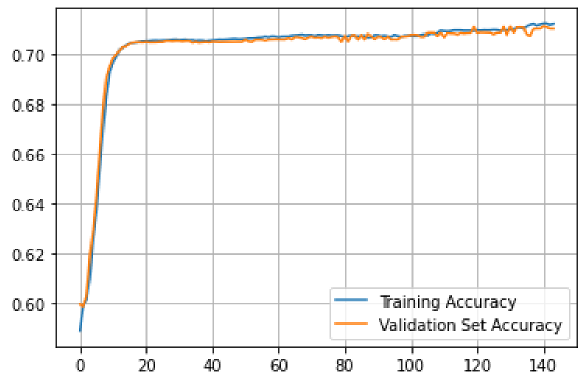


Figure 7: Screenshot of the model created by the classifier using data from more bots than just 'Bounder'

	Spy Win Rate (%)	Resistance Win Rate (%)	Total Win Rate(%)
Run 1	86.5	34.6	56.0
Run 2	87.6	34.5	56.1
Run 3	88.0	33.8	57.6
Average	87.4	34.3	56.6

Table 7: Table of run scores or 1000 games with the bot using the new model

As can be seen from the model and the scores, although the model seems to look worse than the previous model, the results are actually slightly better whilst remaining consistent. For this reason, this is the classifier and model that I will be submitting with my final agent.

Analysis

I tried to perform the development process in a way that made sense. I first wanted to improve the model that we had from the third lab, so that I would be able to get a better sense of how well the functions within my agent would perform. Based on this, I can say that this decision worked well, as I was able to get a reliable model that showed expected differences between each addition to the functions I made. The additions to the selection, voting, and sabotage functions produced results matching that of what I described the behaviour should be able to accomplish.

One function that I did not see a change of results in though was the ***announce(self)*** function. This didn't work as none of the bots within the *intermediates.py* file made use of the ***onAnnouncement(self, source, announcement)*** function; which takes the data from the announcement. If I were playing a bot that had this function implemented, I could supply them with the probabilities from the neural network model when the bot was a resistance member but supply them with an incorrect list of probabilities when a spy to throw off the other players.

Revisiting the different types of activation functions I could use for the model was to test to see what would allow me to optimise the functionality within my bot now that it had been successfully implemented and tested. I tried the other two activation functions we had looked at so far in the module, and from my models decided that the relu function produced a better model. I then compared the results of this with the previous set of results, and found them to be slightly worse, but much more consistent. As I wanted to have the bot play as consistently well as possible, I did not rely on 'lucky games', and chose the model that provided more consistency.

A surprising occurrence however was when I created a model using all bots within the *intermediates.py* file instead of just a single bot. Although the model seemed worse than the one before it, the results from my testing proved otherwise; with the results being both highly consistent and higher in score. However, this did take much longer to train than the previous model, and required some slight adjustment to the hyperparameters ad more data was being used to train and test the model.

Overall Conclusions

One of the surprising things that were made immediately clear to me was that my bot's behaviour was not able to be used to the fullest against the bots located within the *beginners.py* and *intermediates.py* Python files. This is because although my bot tries to manipulate how other players may be viewing them to increase its chance of being selected for a mission, the bots located in these files are still basic in their operation; and as such are not prone to having their behaviour exploited. A more complicated bot will take in more data about the current game, so manipulating it is just a case of providing the bot data that would aid in keeping the spy from remaining hidden. However, many of the bots within these python files have fixed '***return True***' or '***return False***' returns in their methods. This is not able to be exploited, as the bot does not base its decision on any outside information.

Because of this, I had to try to simplify the behaviour expressed by my bot; as to begin with I was treating it as if I was playing a human player (or a higher-skilled bot), rather than a bot with most of its behaviour being hardcoded and fixed.

Furthermore, I found that although the results from a run of games can be higher for one model, if the model is not consistent it cannot be relied upon. A good example of this is between the original model I used with the tanh activation function, and the newer model that used the relu activation function. Although the original occasionally produced results that were higher, they were more spread out and so weren't as consistent than those of the newer function. I also learnt that it was important to test the models before judging whether they were bad. This can be seen when I created my last model, where although the training and validation accuracy had the lowest of any model that was made, it produced the best results consistently. This was likely due to a wider variety of information being present increasing the loss within the model, however ultimately providing the model with a clearer understanding of the behaviour of all present bots.

References

[1] M. Boxleitner, T. Maranges and M. Schubert, “Secret Hitler - Rules,” in *https://www.secrethitler.com/assets/Secret_Hitler_Rules.pdf*, Goat, Wolf, & Cabbage.

[2] “Mafia - Rules,” in *https://images-cdn.fantasyflightgames.com/filer_public/83/7a/837a5b73-ea6d-4aaa-b723-cbc1a201ce21/va95_mafia_rules_eng_v5compressed.pdf*, Fantasy Flight Games.

[3] P. d. Pallières and H. Marly, “Werewolves - Rules,” in *https://cdn.1j1ju.com/medias/af/08/f4-the-werewolves-of-millers-hollow-rulebook.pdf*.

[4] S. Nagayama, J. Abe, K. Oya, K. Sakamoto, H. Shibuki, T. Mori and N. Kando, “Strategies for an Autonomous Agent Playing the “Werewolf game””.

[5] N. Nakamura, M. Inaba, K. Takahashi, F. Toriumi, H. Osawa, D. Katagami and K. Shinoda, “Constructing a Human-like agent for the Werewold Game using a psychological model based multiple perspectives”.