BayesWatch: applying Heuristic and Neural Network-based Bayesian classifiers to *The Resistance*

Rachel Lowe

This paper discusses an implementation of an agent which plays *The Resistance*, using a Naïve Bayes Classifier to try identifying which two players are the spies. A hand-coded spy probability function, as well as a spy probability function implemented via a neural network, have been created, and the performance of the two is discussed.

Introduction:

This paper attempts to solve the game of *The Resistance* via applying classification techniques to the problem of identifying which players are spies as a resistance member, and to the problem of working out how suspicious one's actions may appear to other players when playing as the spy in *The Resistance*. Ultimately, due to many terrible decisions being made on the way to achieving this goal, nothing of any real importance has been discovered, and no real conclusions can be made.

The Resistance:

The Resistance [1] is a social deception game for 5-10 players; for purposes of this paper, the 5-player variant will be played. At the start of the game, each player is given a role; 3 players are resistance members, 2 are spies. The spies know what role every player has, unlike the resistance members. Gameplay takes place over 5 'missions'; the Resistance members win by successfully completing 3 missions, whilst the spies win by successfully sabotaging 3 missions. Players take turns (in a fixed order) as the 'leader', who must nominate a 'team' of 2 or 3 players (depending on which mission it is) to participate in a mission; a team only does the mission (giving spies on the team the option to sabotage it) after a majority of players vote in favour of that team; however, if 5 nominations fail, the mission is considered failed. Regardless of the outcome of the vote or the mission, the leadership role will then pass to the next player (and the process repeats until one team wins) [2]. Despite being at an initial knowledge disadvantage, resistance members are still the majority, so, should they all work out who the spies are soon enough, they will be able to win: therefore, the spies need to spread distrust among the other players, making them think that the other resistance members are untrustworthy, to prevent this from happening. For humans, this is usually a relatively simple task involving some lying; but, for a computer, this social aspect can be a lot less trivial. Additionally, *The Resistance* does have a particularly large gamestate space (which, from Serrino *et al*'s calculation of a minimum bound for *The Resistance: Avalon*, calculated before factoring in player roles or games shorter than the longest possible game) consists of at least 10^{56} distinct states [3].

Some research on general deception-based games has been done already. Abramson's discussion on deceptive learning machines proposes that a learner can be manipulated via 'machine probing' (finding the learner's blind spots, for prediction manipulation, as well as probing it to 'divulge information about its predictability for evasion purposes'), followed by 'machine teaching' (attempting to 'poison a learner to make inaccurate predictions in as few attempts as possible'). Additionally, Abramson proposed a countermeasure to these deceptive approaches, involving a 'meta-learner as a wrapper to a learner', which adapts the actions of the learner in response to the adversary, via Monte-Carlo Tree Search to try working out what the end goal of the adversary is, attempting to deceive the adversary back, and using other Regret Minimization techniques [4]. Regret Minimization, in context of games with incomplete information (such as *The Resistance*) is discussed at length by Zinkevich *et al*, proposing the concept of 'Counterfactual Regret'. 'Regret' is defined as the difference between the utility of the strategy used by an agent, and the utility of the 'best' strategy; 'Counterfactual Regret' is an extension of this, where the regret and utility are calculated based on 'an individual information set' (the information known by the agent at the given point at the game). By creating a poker-playing agent that uses Counterfactual Regret in decision-making instead of pure Regret, Zinkevich *et al* demonstrated that using this approach not only effectively can handle very large sample spaces of gamestates, applying it to Poker has produced AI agents capable of beating prior Poker-playing agents by a significant margin. [5]

Additionally, Serrino *et al* have produced an AI for playing *The Resistance: Avalon* (a version of *The Resistance* with a different dressing, and additional player roles, adding an extra level of complexity to the gameplay [6]), called *DeepRole*. *DeepRole* consists of a planning algorithm, using Counterfactual Regret Minimization and Bayesian logic-based deductive reasoning, combined with 'neural value networks that are used to reduce the size of the game tree'. Serrino *et al* observed that the size of the search space comes mostly from the number of players, not from the available actions, therefore, they split the game into 45 neural networks (one for each potential proposal given the history of successful/failed missions), and, given the current belief probabilities of each player having each role (from the deductive reasoning component), uses the neural network to work out the chances of the resistance winning with each combination of player roles, and then, weighting these win chances by the beliefs of each combination of role assignments, uses that to predict which team composition is most likely to be correct and also succeed [3].

Background:

At first, a rules-based approach for this task was considered, based on several key assumptions about the game and strategies of other players (such as 'any resistance member would vote against any 3-person team that does not include them (due to it definitely containing a spy) except when it's the 5th nomination (as rejecting that means the spies win)', 'if the team consists of 2 people, and 2 sabotages happen, both team members are spies', and 'when multiple spies are in a team, if the leader was a spy, only that spy should sabotage, otherwise only the 'first' spy in the team should sabotage'), however, that approach was promptly shelved, due to it relying on those aforementioned assumptions about the techniques of other agents (assumptions that could be completely wrong), and this sort of fixed strategy could be learned (and exploited) by another agent, rendering it ineffective. Additionally, during the labs for the CE811 Game AI module (which this agent is being developed for), a basic neural network-based agent was created to play *The Resistance*, meaning that the opponents which this agent would need to play against could potentially be able to learn and adapt their strategies whilst playing; meaning that a fixed, rules-based approach would eventually fail (and, as there are 37 students registered for the CE811 course, one can assume that this agent would need to compete against at least 36 other agents, all of whom could be capable of learning the actions of a hypothetical rules-based agent).

Recalling the literature discussed earlier, an approach loosely based on Serrino *et al*'s *DeepRole* AI [3] was chosen for this particular task. An exact replication of that approach is not going to be achieved, due to the very high likelihood that the author of this paper has misinterpreted an aspect of *DeepRole*'s approach to the problem (and the necessity of adapting that approach to work for the mildly less complex game of *The Resistance*), however, at least in theory, it should work well enough for this problem. When playing as a spy, the agent will be in possession of all the information about the roles of the players, therefore, the agent will not need to worry about working out what role each player has; however, as a resistance member, there will be 6 possible teams of spies, and the agent must work out which team of spies they are having to outsmart. This, in turn, means that whilst *DeepRole* needed 60 potential belief states to keep track of [3], this agent could potentially get away with only needing 10 (or, as 4 of those are states where the agent is a spy meaning it would then know with 100% certainty what role every player has, it effectively only needs to care about 6).

However, two anticipated problems with this approach are the problems of training and storage. There is a risk that insufficient examples of each possible game state may be provided to the agent, meaning that the agent's decisions are likely to suffer from this incomplete information. Additionally, loading up to 45 neural networks into memory at the same time could be somewhat expensive in terms of memory (especially if up to 36 other agents which may be using similar techniques could potentially be running at the same time), and that's before factoring in any Bayesian belief networks, which could also be an issue.

However, if this technique is not successfully implementable, the implemented technique will be discussed in the following section instead.

Techniques Implemented:

Ultimately, a somewhat different approach to was used in my bot. To summarize, my bot uses a rules-based approach, but uses a naïve bayes classifier to identify the spies (via how likely each pair of players is to be the team of spies) by first calculating the probability of each individual player being a spy (from observations of the actions taken by each player) and then uses those probabilities to work out what to do from there; and the probabilities themselves are calculated via a neural network. This is accompanied by a rather rudimentary pickle-based counter of how many wins/losses have been reached from a certain gamestate (it's not a proper monte carlo search tree), and an equally rudimentary pickle-based counter of how likely spies are to sabotage at a given gamestate. There is also a mechanism to allow the agent to make a more general estimate of how likely a player is to be a spy via some observations of what actions they have taken, which was used as the sole method of estimating the probabilities of each player being a spy before the neural network-based probability calculator was implemented.

It works on a rules-based system. When selecting a team, it will pick teams by choosing itself and the player/pair who it currently deems to be the least suspicious, but, on turn 1, it will pick the player next in line to be leader, so turn 2 can start with immediately useful information (additionally, as a resistance member on turn 3, if both teams have 1 point, there is a chance that it might pick a team consisting only of the people it suspects to be spies, in an attempt to bait those players into both sabotaging and revealing that they're spies, or at very least confuse everyone involved. Similarly, it will always shuffle the team order before returning it, to potentially confuse any agents who may have logic that is based on team order). When voting, as a resistance member, it will always vote yes during hammer[1] or to its own teams, otherwise, votes no to any teams containing either of the two players it thinks are most likely to be the spies, and votes no to any 3-person teams that it is not on (as that would mean that there's at least one spy on the team instead). As a spy, it will always vote against hammer it's the final hammer vote, or if the resistance are one mission from victory whilst there are no spies on the team. Otherwise, it will first work out the likelihood of a resistance overall win happening depending on the outcome of this nomination round (whether it's a pass, failure, or rejected) how it would have voted if it wasn't a spy, and uses the heuristic-based spy probability method, using a predicted outcome to this round (assuming all team members vote yes and all spies sabotage), to work out how suspicious every player would look after this mission if it happens. If there are no spies, it votes against it if that mission would guarantee a resistance win, or if voting against it would cause both it and the other spy to appear more suspicious than the resistance members, or if a resistance win is more likely if it is rejected. If there's only one spy, it will vote for it if it would have voted for it anyway as a resistance member, if it can vote for it without appearing too suspicious in that lookahead, or if it can win if that mission gets sabotaged. If it's on the team with another spy, it will vote for it if it will lead to a win, or if the other spy is the leader, or if it is more likely to lead to a win whilst not incriminating both spies. Finally, when sabotaging, if it's the only spy, it will always sabotage, unless it's the leader on turn 1. If the other spy is there, it won't sabotage if the other spy is the leader, will sabotage if this mission will lead to one team winning, otherwise it uses the sabotage chance lookup table to see, on average, how many sabotages, on average, have happened whilst the other spy has been on a two-spy team lead by someone else, and will refrain from sabotaging if, on average, more than one sabotage has been recorded in this situation for this spy.

The naive bayes classifier itself is rather simple. There are 5 players, 2 of whom are spies, this means that there are 10 pairs of spies; so, to work out if a pair of players are spies, we combine the probability of each of those players being spies with the probability of the other players not being spies, as follows:

$p(AB\ spies) = p(A\ spy) * P(B\ spy) * 1 - P(C\ spy) * 1 - P(D\ spy) * \ 1 - P(E\ spy)$

In practice, when the agent is playing on the resistance team, it only needs to factor in the six combinations of the other four players, as it knows that it is not a spy (therefore the four combinations which include it as being a spy are impossible). However, when playing as a spy, the agent will need to factor in how suspicious it appears to the other players, meaning that it does properly calculate the full belief table in that case.

As mentioned earlier, the individual probabilities rely on observations of the actions taken by others; these being their votes, the outcomes of missions they have lead, and the outcomes of missions they have been on. This agent records these observations via 4 lists of gamestate indices for each player (for the missions that player lead, was on, voted in favour of, and voted against), with a central game record object holding information about the missions themselves (such as the outcomes, how many players were on the team, the sum of how many sabotages have happened on missions the participants of a given mission have been on). This allows the data to be stored with minimal overhead from redundant duplication of data, however, it does mean that, to get information such as the average number of sabotages for missions that the player was on, this data

[1] The 5th nomination attempt within a round, which resistance members are obliged to vote for.

will need to be aggregated on the fly. One thing of note is that the gamestate indices used for this do not refer to the nth turn since the game began; instead, they indicate which one of the 45 distinct nomination states this happened at, with the intent being to keep track of what the current situation is, as the actions of other agents may differ depending on the situation, allowing some situational actions (such as how likely spies are to sabotage at a given gamestate, or how likely the resistance are to win after a given gamestate is reached) to be recorded as well.

The neural network spy identifier is given 18 inputs per player per turn (in a single batch of all 5 players, running in parallel), and returns two, softmaxed, outputs per player, these indicating the likelihood of the player being a spy or not. The inputs consist of the average number of sabotages on teams the player lead/was on/voted for/voted against/voted for whilst not on, how many times the player voted no to a hammer vote, and how many times the player voted for/against teams with 'suspect counts[2]' of 0, 1, 2, 3, 4, or >=5. These inputs are all normalized to values between 0 and 1 (based on the maximum possible values for each), in an attempt to prevent the neural network from being swamped by disproportionately large inputs. The network consists of alternating dense tanh layers and ReLU layers (all of which output 36 values, with the ReLU layers being used to effectively kill negative outputs), ending with a dense softmax layer which produces the two softmaxed spy/not spy probability outputs. The performance of this shall be discussed in the following section.

The other, heuristic, spy identifier, is much simpler. If there is no data for the agent, it gives up and returns 0.5. If the agent has voted no to any hammer vote, it will return a probability of 1, as no rational resistance member would guarantee a loss by voting against it. Otherwise, it will return the average number of sabotages that have happened on missions which this agent has been on (and 0.5 if the agent hasn't been on any). I initially made it return an aggregate of the average number of sabotages on missions the agent lead, was on, and voted for (with the 'lead' and 'voted for' being weighted to only have a quarter of the importance of the 'been on'); however, as I observed that it tended to cause innocent resistance members to get a disproportionately large suspicion value if they made the mistake of picking a spy to be in the team, and the games tended to end before the spies made enough errors to re-incriminate themselves, those are no longer considered .

Finally, the agent will complain in the chat (only up to 5 times per turn, to avoid any infinite dialogue loops with another auto-reply bot) whenever either of the players it suspects as being on the spy team speak (basically telling them to shut up), and, after a mission ends, it will also say, in chat, which two players it thinks are the spies (and how likely it thinks each of them are to be spies), as well as sharing an important thought and a 'gg' at the start and end of the game, for the benefit of nobody. Whilst this does cause the rl18730.log file to be filled with junk, this agent still logs the json-format data used to train the neural network in a dedicated GameRecord.log file within the /bots/rl18730/ folder, causing nothing of value to be lost.

Experimental Study:

Regrettably, the initial experimental study leading to the initial configuration of the agent, which took place over a span of several weeks, was not properly documented, meaning that not much evidence of it can actually be shown in this report. However, the approach used to evaluating this agent can still be discussed, at some length.

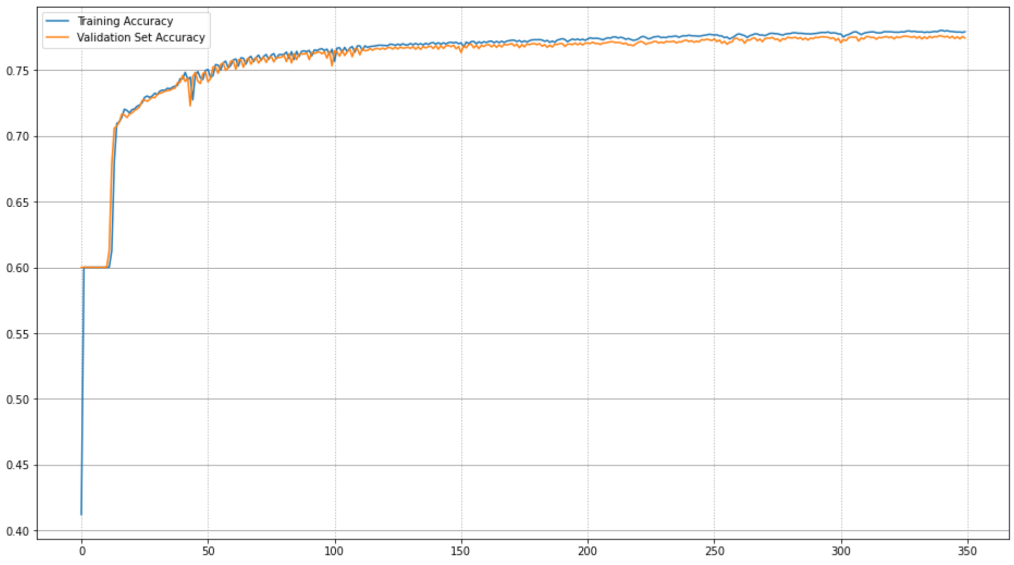
[2] Defined as the sum of the number of sabotages have happened on teams that each player on the team has been on

When the initial functionality of the agent was being assessed (such as ensuring that it was recording data correctly, looking in depth at how the spy probability estimates changed over the course of the game, and trying to debug certain functions via print abuse), this agent would be run in a competition of length 1 against the four agents in the provided 'intermediates.py' file, with the rationale being that, as that file conveniently contains four agents, I could just run a single game with only those five agents in it, and not have to worry about having to deal with printouts from multiple copies of my agent at once, or the chance of not being able to look at the problem due to RNG not letting the agent be run in the first place. This method was only used for immediate-term, 'is this working' assessment of the agent, not for any proper assessment of the agent in a 'real-world' scenario.

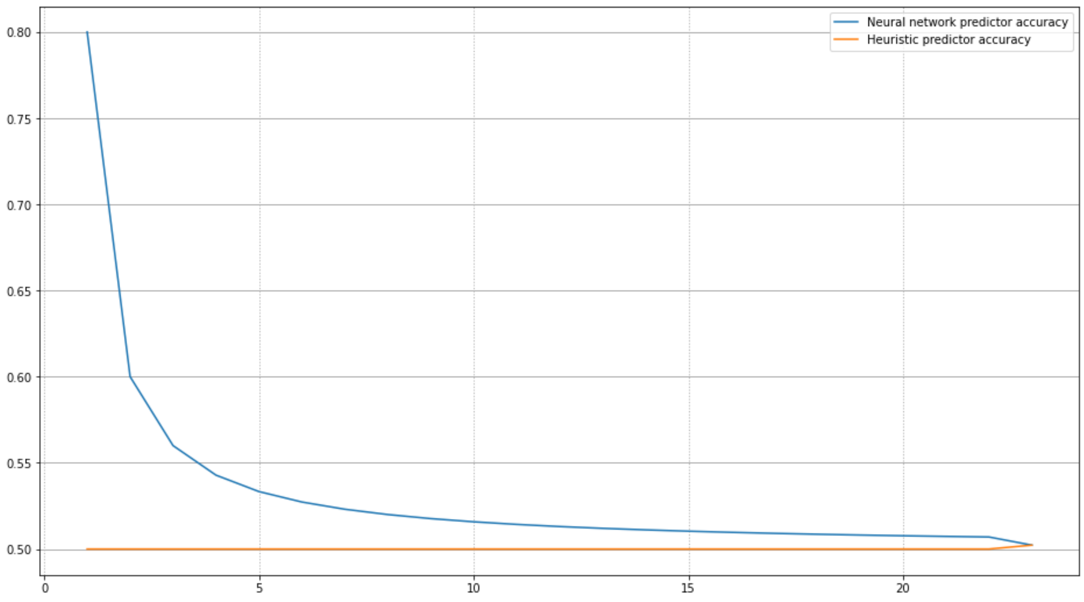
I did not want to run the risk of producing an agent which was only capable of competing against the agents in 'intermediates.py', so, before developing beyond the initial basic functionality stage, I opted to increase the variety of opponents the bot would encounter. I opted to not bother with the beginners (because whilst that would add some quantity, the self-defeating nature of some of them did seem a bit counterintuitive when trying to find good opponents for my bot, and adding only some bots from a file to a competition is a bit awkward to do). However, the bots/0/ and bots/1/ subfolders were full of a rather large quantity of agents, of some presumable quality, as they were also made for a competition, similar to the one that this agent was being built for. Despite many of those agents being in an initially unusable state (due to most of them being written in python 2, some of them attempting to use some comparison functions which don't appear to exist, and other such minor annoyances), after applying some 2to3 to them, updating a few method signatures to properly match those of the framework being used, and deciphering a few cryptic error messages, I was able to get all 17 of them working again, and I added them to the competitions (along with my agent and the intermediates), to get a rather decent competition of 22 agents working. I wanted to try adding a few more agents to these competitions when testing it, however, due to the competition.py script not appearing to work as soon as I added a 23rd agent, I opted to leave it at 22. I increased the length of these competitions to 20000 games (which was sped up significantly by uncommenting the line in competition.py which enables multiprocessing).

Against all of these opponents, my agent ultimately performed rather unimpressively. I have saved a couple of text files which contain data about the bot's performance in these competitions (using a competition of 20000 games), and my bot's win rate ultimately settled at a rather uninspiring ~40% (with a spy win rate around ~70-75%, and a resistance win rate at around ~20-30%). Even after adding the neural network component to the agent, the win rate remained relatively constant. However, whilst the version with the simple heuristic-based probability calculator ended up only voting in favour of resistance teams ~50% of the time as a resistance member, the version with the neural network-based probability classifier properly voted for resistance teams ~72% of the time instead; I will admit that I am not sure if that is a truly statistically significant difference, and I am not entirely sure if it's possible to try properly analysing it without needing to significantly rewrite the competition.py file to dump even more stats to decipher. However, it does suggest that the neural network has done something beneficial for the bot, no matter how small.

On the subject of the neural network; after working out what data I would be putting into the neural network, and how I could store that data in a format I could then read later, I started work on the neural network within the game_record_nn_trainer.ipynb notebook (which is included in the faser upload). To prepare data for the neural network, I would run a competition of ~60000 games, and copy the logs from the ~12000 or so of those games which my bot participated in to a dedicated 'heuristic bayes training data.log' file (a copy of which is not included in the faser submission for this assignment, as it is well beyond the 50MB limit for faser file uploads. However, it is present within the assignment branch of my CE811 repository on cseegit). Within the notebook, the data is deserialized from JSON, and, before it can be accessed, it is immediately split into three disjoint testing, training, and validation sets, to ensure that the outcomes of this process aren't completely invalid. After putting these, now distinct, sets of data into an object that makes it a bit easier to get the important data from them, the model is constructed, and is then trained using the training data, and specifying that it must be validated on the test data. A copy of the graph showing how accurately the model performed on the training and validation data can be seen below.



As you can see, it reached an overall accuracy of ~77%, which is reasonably good, and doesn't appear to suffer from much overfitting, at least when looking at the overall data. After this, I then compared the performance of the neural network over the course of a full game to that of the heuristic classifier (after turning the outputs of the heuristic estimator into the same sort of one-hot encoding format which the labels used for the examples are in), using the validation set. The results of that, which can be seen below, are somewhat awkward, however, there is an explanation for it.



Due to how I structured the data (simply recording it nomination attempt by nomination attempt), there's no specific information about turn numbers, just the overall quantity of nomination attempts. This means that it's not possible for me to filter the data to turns 1, 2, 3, 4, and 5, without a substantial refactor of the logging data format, as well as the code which reads that data, which is not entirely feasible (and the reason why this problem happened in the first place is because I did not anticipate that the lack of a turn number label would be an issue). For the dataset in the above graph, whilst there were 4511 games in that dataset, only 3907 went past three nomination attempts, 2 lasted for 23 nomination attempts, and 0 lasted for 24 or 25 attempts. I will admit that I'm not sure if the accuracy of the neural network is rapidly deteriorating over time, or if the accuracy metric itself is being affected by the rapidly deteriorating size of the dataset (as there are not many examples of games which lasted a particularly long time, meaning that the dataset might have somewhat generalized for the many early game examples, whilst not really knowing what to do for the very few late game examples). However, regardless of what the specific cause of that rather concerning shape was, it still shows that the neural network classifier was more accurate than the heuristic classifier on the final, previously completely unseen, validation set. In other words, it shows that this neural network does better, albeit by a very slight amount, than the heuristic estimator does at classifying whether or not a player is a spy.

Analysis:

The development process of this agent was somewhat problematic overall. I frequently found myself getting distracted from the task at hand (implementing a bot to play The Resistance) via the creation of a particularly overengineered, overly-extensive (and mostly unused) blackboard, which, whilst it does allow the agent to have relatively easy access to a lot of observations of the game, ultimately has not allowed the agent to effectively compete against other, much less overengineered, agents. I suspect that the particularly inflexible variety of rules-based approach I employed to create this agent is the main culprit for this poor performance, as these rules are written such that the agent only truly distrusts the two players which the naïve bayes classifier declares as being most likely to be the spies, and implicitly trusts the other two players. In theory, this could expose this agent to exploitation from another, more advanced, agent which properly learns the actions of other agents; with this exploitability potentially rendering this agent completely beatable. Furthermore, whilst the playstyle of this bot follows very simple rules as a resistance member, as it only follows additional rules (which have a chance of overriding the normal resistance member strategy this bot would otherwise be using), a particularly perceptive agent could notice the subtle differences in how this agent is acting, and use that to call out this agent as being a spy much faster, and much more effectively, than any sort of neural network-based probability estimator ever could.

However, in spite of those things that very clearly did not work, some aspects of the development process have gone rather well. Whilst the blackboard which this agent has access to may not be utilized to its fullest potential, it's still a blackboard, which has quite a bit of data about the state of the game, and there are several methods within the various classes which this blackboard consists of which allow some filtered data to be obtained with minimal hassle, meaning that, in theory, one could use it to create an even better agent, without needing to worry too much about how to start storing the data collected as a game progresses. I am particularly pleased yet frustrated with the GamestateTree data structure present within the code for my agent; it offers a rather elegant solution to giving gamestates a unique identifier (offering a means of decoding them to get the information about the current wins/losses/nomination attempt for the current gamestate, and encoding that data into a single int, whilst also having the gamestates numbered in an order which means that they, themselves, could be used as a heuristic estimate of the utility of a given gamestate, with those which have lower numbers being generally more useful to the spies, and those with a higher number being generally more useful to the resistance), but, again, I didn't use it anywhere near to its fullest potential, as the only things that the uniquely indexed gamestates help with (the pickle with data about how many times the resistance have won after reaching a given gamestate via ID, and the pickle with data about how likely each identified player is to sabotage at each given gamestate in each sort of team leadership and spy quantity situation) are ultimately footnotes, both of which only being used once in particularly niche situations within the bot's logic, meaning that there's no real point in discussing the particulars about how they work, as most of their functionality goes unused, despite still storing easily accessible blackboard data, like most of the other parts of the blackboard used by my agent. The Bayesian classifier within this bot, similarly, could be used as a component of a more advanced agent (along the lines of the *DeepRole* agent [3], which I initially tried to emulate, before failing spectacularly), despite, by itself, not being able to get my bot to be anything better than mediocre. On a more positive note, whilst this probably was not the point of the assignment, I have been able to develop my Python abilities somewhat whilst working on this, and I now know how to, among other things, abuse properties, generators, and maths in order to minimize the number of redundant variables which may need to be held within instances of classes or within functions.

Additionally, I am surprised by how well the agents within the /0/ and /1/ subfolders work, especially the Bounder2 bot (in bots/others0/myplayers.py), as I have noticed that it was able to consistently perform well in the competitions which I ran during the development process, despite the readme about the provided agents in the framework folders mentioning that it, along with the other bots in bots/0/, were only produced in a few hours (whilst my agent, despite taking a particularly long time to produce (which is rather evident by my commit history to the assignment branch of my CE811 repository on the university gitlab instance) is, to put it lightly, not that good).

Overall conclusions:

Whilst a functional AI agent, able to play *The Resistance*, has been implemented, this project has, overall, been a series of failures leading to an overengineered, sub-par agent which fails to compete against other, much simpler, agents. Whilst some background research was performed, in hindsight, it's clear that nowhere near enough was done, as the key points of some of those papers had been completely forgotten about by when it was time to start implementing the bots, and, instead of taking time to properly research the aspects which were clearly not understood properly (such as counterfactual regret, adversarial learning, or the specifics of the *DeepRole* implementation, imitation of which was the basis of the choice to use a naïve bayes classifier, despite that only being one aspect of that implementation [3]), time was wasted flailing around in the implementation, and instead of spending the time productively, such as addressing these gaps in knowledge, or working on the remainder of the report (beyond the 'research' and 'background' sections), it was squandered by implementing parts of an overengineered blackboard and obsessing over the minute details of the barely functional implementation which was produced.

Furthermore, despite the various competitions run with the agent showing some mediocre results overall, at no point were any truly substantial changes to the agent's logic made after they were first implemented, under a misguided belief that, as soon as the neural network was implemented (with the final choice of implementing it in the form of using that to work out the individual spy probabilities being made too late into development to truly matter), the results would become less mediocre. Instead, this warning was completely ignored, and the poor performance of the agent is a natural consequence of this sheer apathy.

Despite those many flaws with the implementation, there is, at very least, a rather extensive blackboard, with data about almost all aspects of the observable actions in the game logged (and accessible) in there. Furthermore, the GamestateTree data structure implemented during the creation of this agent is an elegant solution to the problem of identifying individual gamestates within games of *The Resistance*, and, for any data in the blackboard which is especially in need of being stored in such a tree structure (for example, any per-gamestate neural networks or other such decision-making item which could need to make a different decision depending on the state of the game, and instead of having to use a single model generalized for all states, using up to 45 more specialized models, any regret minimization implementation, or anything that benefits from a general idea of the utility of a certain gamestate) would become much easier to store and find later on with this. Unfortunately, as this blackboard was attached to a substandard implementation, the full benefit of it may never end up realized.

Finally, looking at the bigger picture, there is one final flaw with the outcome of this: nothing of any real value was created. Yes, there might be a neural network which, when given some normalized inputs describing some aspects of the given game state for The Resistance is able to correctly identify whether the data describes a spy or a resistance member correctly some of the time, but that's about it. Whilst good practices regarding the segregation of a full dataset into disjoint training, testing, and validation sets were used to train and then assess the performance of the model, no stratification was performed when splitting the data sets (making overfitting/underfitting more likely by not having training subsets which properly represented the population), and the barriers posed to proper analysis by the badly-structured data (both in terms of the lack of any real metadata stored with them making it impossible to do things such as grouping states by mission number for ease of graphing, and in terms of the imbalance of game lengths represented by the samples potentially causing the neural network to overfit for short games and underfit for long games). Ultimately, I genuinely don't know anything about how good the model is, nor do I have any proof of any tangible benefits it provides, not even a simple comparison of training two neural network models on the data, showing both of them next to each other, and evaluating which one is best. Additionally, whilst there still are the pickled sabotages and win probabilities objects, holding some hopefully easier to digest statistics about the game, as the choice was made to store them as pickles instead of in an easier to digest form (such as JSON), that poses a significant barrier to the ability of anyone to review how well the agent is performing, therefore making it even harder to properly come to a conclusion about the performance of the agent. The only real knowledge gained throughout this was the experience of doing it, some knowledge of some of Python's quirks, and first-hand-knowledge of what the consequences are of charging in to work on an assignment without a clear plan.

On a related note, the main reason for this work being finished so close to the deadline is also because of this lack of a clear plan; I failed to research ways to serialize/deserialize data in Python, and certain important aspects about the expected format of inputs and outputs for keras, causing progress on the work to be delayed for several massive refactors to address the problems caused by my lack of care towards the task at hand.

To conclude, this project has been a near-complete failure, and it shows. Whilst the agent does have access to many sources of information, with functionality to get filtered versions of that data with unimportant data removed/important data added, this agent does not properly utilize it, and will not deviate from a hardcoded, predicable, sequence of rules, despite it probably having more than enough data on hand to adequately satisfy most varieties of machine learners, or, at very least, having most of the initial setup done to make it easier to calculate or implement some form of observer for anything else it might need.

References

[1] D. Eskridge, *The Resistance,* 2010.

[2] UltraBoardGames, "How to play The Resistance | Official Rules | UltraBoardGames," [Online]. Available: https://www.ultraboardgames.com/the-resistance/game-rules.php. [Accessed 28 October 2021].

[3] J. Serrino, M. Kleiman-Weiner, D. C. Parkes and J. Tenenbaum, "Finding Friend and Foe in Multi-Agent Games," in *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, Vancouver, 2019.

[4] M. Abramson, "Toward Adversarial Online Learning and the Science of Deceptive Machines," in *Papers from the 2015 AAAI Fall Symposium Series*, Arlington, 2015.

[5] M. J. M. B. C. P. Martin Zinkevich, "Regret Minimization in Games with Incomplete Information," in *Advances in Neural Information Processing Systems 20 (NIPS 2007)*, Vancouver, 2007.

[6] D. Eskridge, *The Resistance: Avalon,* 2012.

[7] M. Fairbank, *CE811 Game Artificial Intelligence,* Colchester: University of Essex, 2021.