Harvard University

CS 182

Artificial Intelligence

Mao: *Learning to Win Under Dynamic Uncertainty*

Authors:
John Lane
Matthew Sciamanna
Andrew Soldini

December 19, 2018

1 Introduction

Opening Note: Before reading this paper, it is best to run our system and experience the game for yourself. To play 3 rounds, just run this from the command line:

```
python tests.py on 3 human heuristic
```

1.1 Abstract

Mao is a round-based card game in the shedding family. If a player is the first to discard all their cards, they are dubbed "Mao". A set of rules defines when a card can be played; however, when a player becomes Mao, they get to change these rules.

We created a framework to play the game of Mao, complete with a total of 7 changeable rules and effects. We then created AI agents in order to learn to play the game. We split the challenge into two parts:

1. Deduce the underlying (and ever changing) rules of the game using a Hidden Markov Model.
2. Choose the best card to play, given the current belief in the rules. We employed expectimax, Q-learning, and a basic Heuristic to do so.

For the first part, we had great success in deducing the rules using an HMM, and found for the second part that the Heuristic-based method was the most effective tactic. We took these results to confirm the the power of rigorous, probabilistic models in many scenarios, and the need for human judgment in matters of higher-level strategy.

1.2 The Stakes of Mao

The game of Mao presents challenges to an AI solver that do not exist in most other card games. While in most card games, an agent simply has to determine the best action, in Mao, we instead have to determine the rules, and then the best action to take.

Such a setup is one that mimics the real world. In practice, the world is a changing place, with "rules" that are constantly in flux. By finding a solution to both the rule-recognition and then the action-taking problems in a toy problem like Mao, we hoped to learn techniques that we could apply to more complex, real life situations.

1.3 Rules and Simplifications

The initial rules of Mao are similar to those of UNO, where each player is dealt a hand of (generally) 5 cards, and they can only play a card that is either larger than, or of the same suit, as the card on the top of the discard pile.

However, as soon as the first player discards all of the cards in his/her hand becomes "Mao", they make a new rule for the next round – which they don't disclose to the rest of the players. Players who violate this rule in later rounds are penalized, usually by having to draw another card in addition to having to keep their own. It is up to the players to infer what exactly the new rule is. We will continue this discussion in part 3.

2 Background and Related Work

There have been many studies on the use of AI and other concepts from the course in the world of playing card games. Namely, the literature so far has been concerned with finding the ideal playing strategies for card games with imperfect information (as is the case in any game where you can't see your opponent's hand). For instance, Sturtevant and White [4] used approximate TDlearning with 60 features to create an agent to play the game of Hearts. Their idea of modelling hands as states was the inspiration for our solution to the second part of the problem.

The general strategy of these AI based card game "solvers", has been to train on a data source gathered from human players. They generally conclude that even "... training on data from average human players can result in expert-level playing strength" [1]. This, in combination with TD-Learning's ability to learn off-policy [3] makes the problem much more feasible than if optimal strategy data or expert level data were required for training.

But despite the similarities to other card game based AI projects, we were not able to find any AI based Mao solver. The web-comic XKCD jokes that Mao will never be solved by AI, which is potentially part of the reason that such a project seems never to have been tackled by a team in academia. Though our project is not necessarily an attempt at solving Mao so much as an attempt at solving a Mao-inspired card game, we hope to contribute some novel methods and a roadmap of our successes and failures to what little does exist in the literature on this topic.

3 Problem Specification and Framework

3.1 Framework

As a problem like this has not been solved, we had to build our own game playing infrastructure rather than using a prebuilt library. In building the infrastructure, we realized that the addition of arbitrary rules would be far too complex. Also, having multiple players would further reduce the effectiveness of our agents. So, we reduced the problem down to a 2-player game with 8 changeable constraints and effects.

Constraints: The constraints, when activated, determine which card from your hand can be played, given the current constraints of the game. They are as follows:

1. Basic Value: either high cards (ie Aces), or low cards (2's) take precedence.
2. Basic Suit: cards of the same suit can always be played.
3. Wild Value: A card with the given value can always be played (ie, a 7 is always playable)
4. Wild Suit: A card of a certain suit can always be played
5. Poison Distance: A card that is the "poison distance" away from the card on top of the deck is illegal. Ex) if a 7 is on top of the deck, and the poison distance is 2, then a card valued 5 or 9 is illegal.

Effects: Cards of a certain value, when played LEGALLY, invoke a certain effect. They are all triggered by a certain "activating value", which is subject to change. They are:

1. Skip Player: When activated, the next player is skipped.
2. Screw Opponent: When activated, the activating player chooses a card from their hand, and unloads it on their opponent.
3. Poison Card: When played, the player has to draw another card.

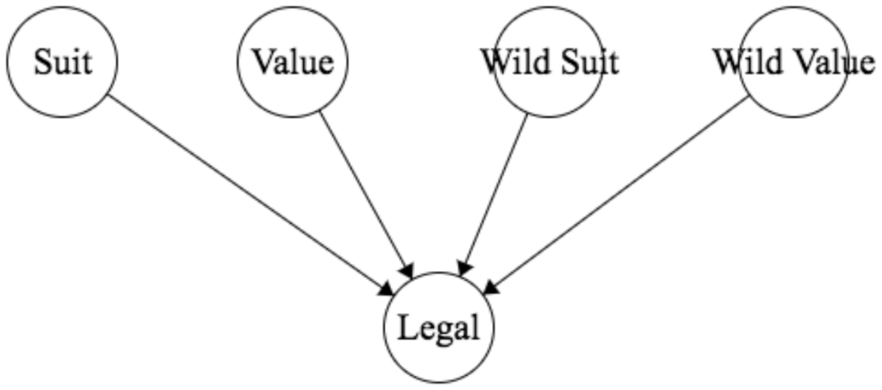


Figure 1: Components of Simplified Rule Space determine a card's legality.

3.2 Formalization of Problem

There were two problems to be solved: the question of the current rule state, and the question of which card to play, given the rule state.

3.2.1 Rule-state Space for rule discovery

In order to play a card effectively, an agent needs to have an idea of which cards in its hand are legal. This can be thought of as a "rule space", which is modelled by Figure 1. However, this is a challenging issue, because the size of the state space is enormous: if you multiply all the possible options for the values you can tweak , there are $2 * 4 * 5 * 14^4 = 1536640$ possible rule-states. This is an impossibly large number of rule states to search quickly with a normal computer.

However, we realized that all the effects can be inferred independently of the constraints (if a skip is activated, that gives us more information than merely swapping a card would), which allowed us to reduce the problem in constraint-rule space to $2 * 4 * 5 * 14 = 560$ constraint-states and 14 logically-distinct states per effect. We then made agents (the most effective of which was an HMM Agent) that attempted to reason through the card legalities and effects emitted by the game to deduce the true rule state.

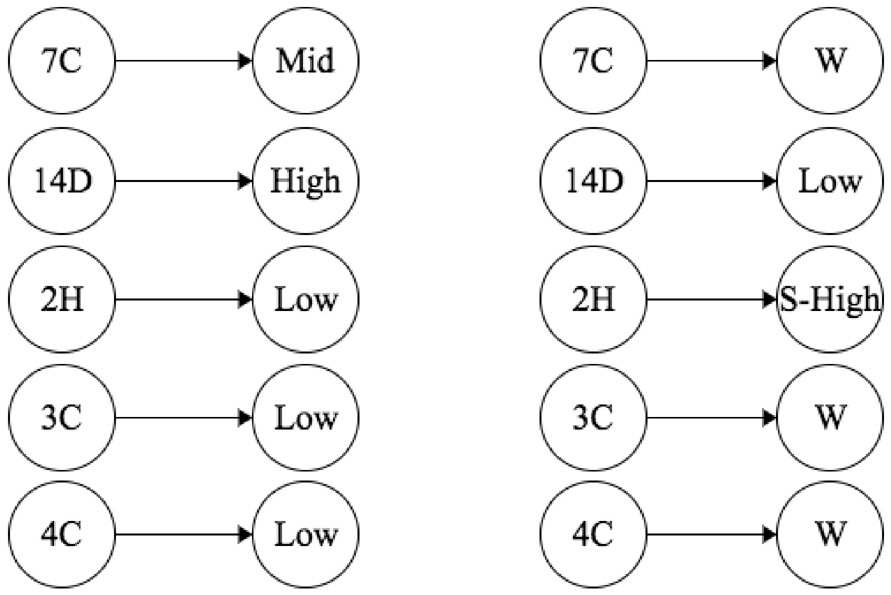


Figure 2: Demonstration of different mappings of hands to features given initial (right) and custom (left) rule states. W = wild card, S = skip player.

3.2.2 Feature-space for card choosing

After solving the problem of determining the current rule-state of the game (in the code, we defined this as the "combination state" – a combination of the constraint and effect states), we need to determine which was the best card in the hand we could play. Given that the same hand can have dramatically different values given different rule sets, we needed to formulate an abstract way to understand what a card means, given a rule set. These essentially are "feature-space", a way to understand the meaning of a card given the underlying rule state.

4 Approach

4.1 Basic Playing Agent

Our first attempt at creating a computerized Mao player was a non-rigorous learning agent. This agent learned its environment by constructing a set of beliefs over the possible sets of rules that are in play. Every time that a card is played, the agent compares the response (whether or not a card is legal) to what that response would be under every possible state of rules in the game. All game states had a running “score”. States that were not consistent had score subtracted, while those consistent were given a positive score. The maximum value were then used to determine the best-guess of the rule state at any given time. The agent then played a random card from the states which they believed to be legal, if any. This agent provided a huge improvement over a random agent and became our next baseline for performance which to test against.

4.2 HMM Agent

After building our first agent, we attempted to formalize it using a Hidden Markov Model.[2] The rules are states, and the legality of the cards serve as evidence. A diagram depicting the model can be found in Figure 3. There are several implementation specific details to our agent.

At every step the agent checks the result of putting down a specific card to be either Legal or Penalty. If the result is inconsistent with a currently believed possible set of rules, it sets its true belief to 0, and then renormalizes the belief values. This is because the probability that a given card will give a Legal emission when played is either 1 or 0 depending on the rules. This gives our variant of an HMM a deductive flavor, as it *systematically eliminated states.* Zeroing out states dramatically increased computational efficiency.

When a player has player all of their card and the end of a round is reached, one of two cases occur. If the player sets a rule, **the set of their beliefs move subjectively from the previous belief state to the new possible belief states, taking the new rule as given**. In the case of the agent losing, however, it performs an update to account for the opponent (potentially) changing the set of rules, expanding the set of possible states by adding weighted values of the child states.

In choosing a card, the HMM agent simply examines its hand for cards believed to be legal, and plays one of them.

4.3 Naive Heuristic Agent

To create an agent which outplays the HMM agent, we realized that we could simply build an agent that invokes some strategy on which cards in hand to

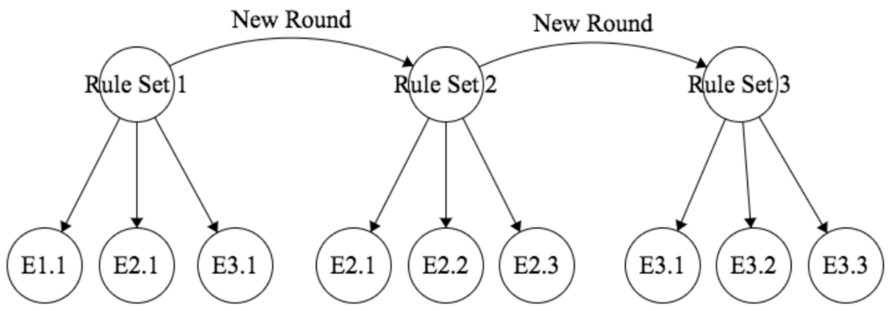


Figure 3: Expanded HMM model of Mao Rule Dynamics. Evidence consists of card legality or illegality.

play. As such, we created a heuristic for card valuation and employed it in an improved model of the HMM agent. This "Naive Heuristic" took some simple factors into consideration like which cards in its hands were subject to rules, and their suits and values. This heuristic simplified the search problem of finding the best card to play for our agent and improved our performance as measured by our win rate when playing against the heuristic-less hmm agent. The heuristic used can be modelled as

$$h(x) = \alpha x + \beta y + \gamma v + \tau z$$

Where x is the card's "screw opponent" status, y is the "skip opponent" status, v is the card value from 1-14, and z is the suit. After some tuning, we determined that the agent should place emphasis on screw opponent status, skip opponent status, and card value so we set τ equal to 0 in our final heuristic. $\alpha, \beta, \& \gamma$ were tested and set dynamically.

4.4 Expectimax Agent

Another attempt to improve upon the HMM agent was to create an agent that minimizes the expected number of moves that the opponent will be able to make. In order to perform this calculation, however, we needed to be able to predict which cards are in the opponent's hand. We carried this out using the probabilities of any given card being in the opponents' hand. This was determined in a similar way to the state beliefs for the HMM agents rules. Instead of adding up to 1, or the belief for the single state, the beliefs added up to the number of cards in the opponent's hand to account for there being multiple cards in any given hand. This belief of the opponent's hand was then used to calculate the expected value of the opponents hand, measured in number of legal moves. When choosing a card to play in this algorithm, the agent played the legal card in its hand that gives the opponent the fewest legal moves. To better account for the effects, this agent also placed a two times positive multiplier on both ScrewOpponent and and Skip Cards, with a negative coefficient on poison cards.

4.5 Approximate Q-Learning Agent

In trying to answer the question of which card to play given the belief state, we created a reinforcement learning based agent. This agent was based in part on the PS3 framework.[2]

This agent was provided a belief state on the current set of rules by an HMM based engine. Then, it used feature-based qLearning to transform its current hand into relevant rule-dependent features such as whether it was playing an illegal card, how many "valuable" cards were in its hand, and whether it was about to play a screwOpponent card. We created 11 features for use in our Q-learning model, which can be seen in features.py, and used 62,000 training rounds to find the weights for the features using the approximate Q-learning algorithm shown in class.

$$w_i \leftarrow w_i + \alpha[R(S, A, S') + \gamma \max_{a'} Q(s', a') - Q(s, a)]f_i(s, a)$$

For a reward function, we award a point for every card successfully removed between rounds, and punished with a point for gaining a card in the hand. If a round ended, we awarded 15 points for winning, and subtracted 15 for losing a round.

For our tuning parameters, we used a discount factor of 0.8, because that is what Sturvesant and White used [4]. We also used an exploration rate of 0.2, to mimic that of pset3, and a learning rate of 0.02, because if it was any larger, our floats tended towards infinite.

Player1	Wins Player 1	Player2	Wins Player 2	Winner	P-Value
HMM	3995	Random	1005	HMM	<0.00001
CardCounter	4080	Random	920	CardCounter	<0.00001
Heuristic	4121	Random	879	Heuristic	<0.00001
QLearning*	149	Random	351	Random	<0.00001
CardCounter	2590	HMM	2410	CardCounter	0.000317
Heuristic	2680	HMM	2320	Heuristic	<0.00001
Qlearning	77	HMM	423	HMM	<0.00001
Heuristic	2650	CardCounter	2350	Heuristic	<0.00001
Qlearning	110	CardCounter	390	CardCounter	<0.00001
Qlearning	69	Heuristic	431	Heuristic	<0.00001

Table 1: Wins Recorded Over Many Training Rounds

5 Experiments

5.1 Winning Results

Our primary results are summarized in Table 1, which show two different agents playing either 5000 rounds or 500 rounds (in the case of any game involving the qLearning agent) against each other, and includes the number of rounds won by each player as well as the P-value of that result.

There are a couple of notable takeaways.

Heuristic Agent was our best performing agent. To recap, Heuristic Agent was an expansion of the HMM agent, which deduced the legal rules of the game. However, it was extended *to take account of effects, and try playing cards like SkipPlayer and ScrewOpponent more.* As can be seen from the table, it beat every single other agent with statistical significance.

That our heuristic agent was the most successful agent is not surprising. It is an agent that takes advantage of the computer's ability to make accurate deductions about the current state of the rules, while it combines *human knowledge in order to advice higher level strategies.*

Also interesting is the result that **CardCounter agent is slightly better than HMM agent.** The distinction is not enormous as CardCounter agent only won 180 more games than HMM agent. However, this does show that an expectimax strategy given legal cards is slightly more effective than a random choice among legal cards.

Finally, there is one more very interesting finding: *all the agents are better than RandomAgent EXCEPT QLearning agent.* Indeed, RandomAgent won over QLearning agent at more than a 2:1 ratio. While admittedly this finding is not encouraging for the effectiveness of our qLearning agent, it has one important result: **qLearning is capable of influencing the card selection strategy.** In fact, that qLearner was able to lose so much more dramatically means that the agent did indeed learn how to play the game in a *deliberate* fashion, albeit one that was not effective. This suggests that a qLearning agent which was better tuned could conceivably learn how to play the game. We will discuss this more extensively shortly.

5.2 HMM Agent Analysis

All of our most effective agents are extensions of the HMM agent. As such, we believed that further inspection of the HMM agent would indeed be fruitful.

A significant difference between the HMM model we learned in class, and the implementation of our own agent, was that our agent had to incorporate knowledge of the changing game state when it made a rule – in short, **our HMM agent had to understand its own competitive advantage of rulemaking.** In order to test this, we pitted two HMM agents against each other, and observed their belief over the number of possible states in *the same game*. The results are found in Figure 4.

There are several important features to notes. The first is that the peaks on the graph, which represent a change in the rules, alternate. This shows that the ability of one agent (the rulemaker) is successful in maintaining knowledge across state. The second thing to notice is that the peaks come in bursts. This confirms the common-knowledge of the game that winning is an advantage, as knowing the rules means you can act upon them more successfully.

5.3 Ineffectiveness of qLearning

One of the main curiosities of our experiments was the ineffectiveness of the qLearning agent. The idea behind the agent was that it would use HMM Agent's belief state to get an idea of these rules, and then use these rules to transform its hands into feature space, and play the optimal card. However, **it turned out that the qLearning agent performed worse than the random agent**. Not only did it perform far worse, but it also took far more computational power to calculate the HMM and the corresponding Q-states of each action. As an added hassle, it was liable to get into infinite loops by playing the same card again and again while also triggering the skip effect.

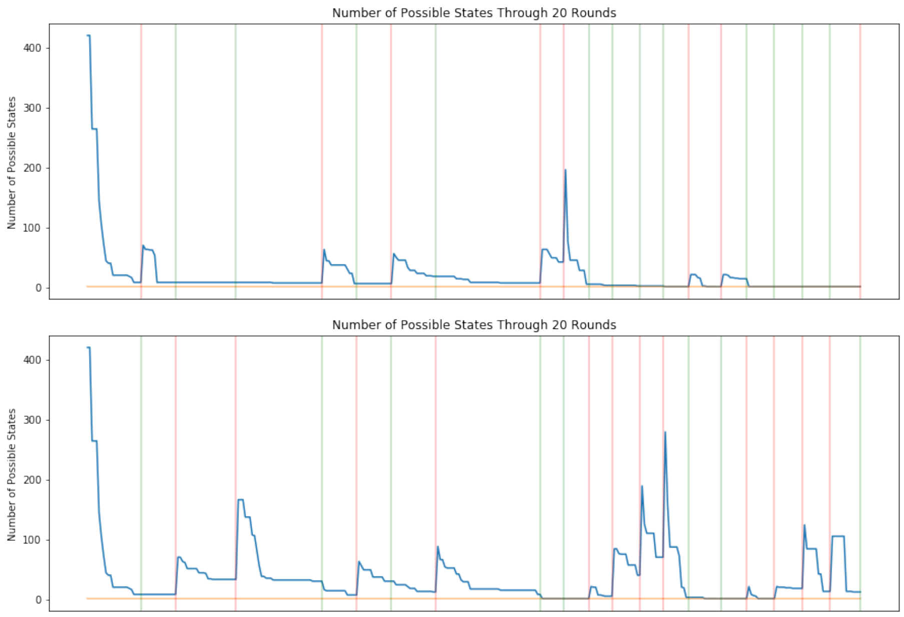


Figure 4: Evolution of belief distributions of two HMM agents playing one another. Vertical lines represent new games. Note how the winner (who makes the rule) remains certain of the rule state across rounds.

In investigating this quirk, we wondered whether we had not trained it sufficiently. In Figure 5, you can see the tuning of the weights throughout the training rounds. When initialized at -1, the weights quickly converged to somewhat stable values, and then stayed there throughout all of training.

As a reminder, our FeatureState (the one evaluated using qLearning) used the hand as a state, and the card the agent was about to play as an action. It then used the ruleState to map these cards to features, which were then optimized.

The odd thing about these weights is that they are **directionally correct.** Consider a few of the final values:

1. PoisonCount (number of poisonous cards in your hand): -1.43
2. Illegality (the card you are about to play is illegal): -4.42
3. SizeofHand (the hand value is negatively correlated to its size): -0.93
4. ScrewCount (more "screw opponent" cards in your hand): 2.95

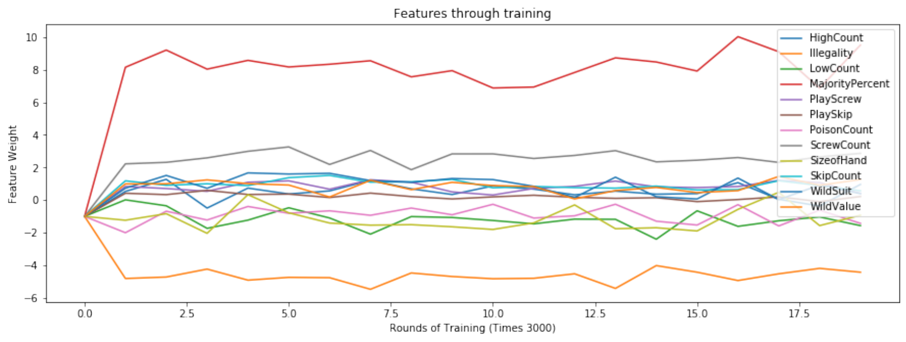
These values all make sense – in our experience, having a lot of poisonous cards in your hand is a recipe for disaster, a smaller hand is preferable to a large one, playing an illegal card is *very* bad, and having more ScrewCards is good (because these allow you to get rid of two cards at once).

However, when we pitted the qLearning agent against the HMM agent or even RandomAgent, the qLearning agent got pulverized, showing that simply playing a random card performed better than our agent.

6 Discussion & Conclusion

Our main question was why qLearning agent performed so poorly. At first, we wondered whether the uncertainty of the rulestate could cause qLearner's poor performance. However, we made a qCheating agent, which attempted to play the game using the REAL game state (which it always knew – hence its name, "qCheating") – but to our chagrin, that performed no better.

We are unsure why exactly this behavior occurred. The training weights' directional correctness seems to indicate that the qLearning agent was implemented correctly. It could be the case that behavior of playing an illegal card is not sufficiently punished. Or it could be that we overfit or underfit with only 11 features, but underfitting seems far more likely, as we had



	HighCount	Illegality	LowCount	MajorityPercent	PlayScrew	PlaySkip	PoisonCount	ScrewCount	SizeofHand	SkipCount	WildSuit	WildValue
0	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000
1	0.530623	-4.811997	0.014861	8.150398	0.834704	0.418000	-2.003282	2.224949	-1.236324	1.181439	0.727204	0.990835
2	1.273314	-4.721868	-0.350612	9.204354	0.706505	0.333480	-0.681897	2.315978	-0.833984	0.906805	1.512746	0.998189
3	-0.491243	-4.234045	-1.734676	8.032978	0.536193	0.593720	-1.223539	2.590521	-2.040269	0.999318	0.721314	1.239183
4	0.733242	-4.914067	-1.229804	8.572264	1.092772	0.341997	-0.392322	2.991917	0.332138	0.887450	1.669149	1.008935
5	0.371966	-4.741991	-0.472245	8.167386	1.180441	0.364972	-0.811659	3.261698	-0.781257	1.375750	1.599057	0.922134
6	0.560795	-4.762121	-1.098345	8.333947	0.667198	0.156173	-0.658283	2.186144	-1.408129	1.515448	1.643372	0.188448
7	1.213427	-5.474713	-2.090725	8.549038	1.245287	0.426268	-0.933204	3.044552	-1.538814	1.099605	1.194991	1.215209
8	0.689348	-4.471644	-1.012136	7.561893	1.098552	0.213359	-0.489539	1.864032	-1.508465	1.122280	1.087389	0.637426
9	0.341428	-4.689684	-1.067540	7.942760	0.500371	0.068605	-0.901246	2.833638	-1.639100	1.270773	1.322465	1.090079
10	0.866925	-4.827492	-1.251636	6.877524	0.310190	0.194377	-0.258797	2.834057	-1.802450	0.753295	1.256761	0.890781
11	0.671461	-4.799554	-1.457733	6.935917	0.711342	0.295793	-1.102502	2.551981	-1.399717	0.836914	0.845577	0.825816
12	0.322330	-4.522177	-1.167325	7.829840	0.842674	0.169892	-0.965476	2.739431	-0.296433	0.764814	0.158360	0.068543
13	0.549900	-5.420745	-1.180023	8.726537	1.166507	0.103999	-0.256612	3.032581	-1.758533	0.731144	1.405100	0.582338
14	0.359771	-4.019431	-2.402002	8.472446	0.808357	0.143714	-1.295341	2.345390	-1.697232	0.857480	0.203881	0.775248
15	0.396364	-4.424812	-0.657036	7.917893	0.768190	-0.095156	-1.531963	2.446017	-1.895080	0.619322	0.067741	0.458575
16	1.068539	-4.937595	-1.612247	10.027288	0.831074	0.031673	-0.281113	2.611042	-0.569844	0.641599	1.351720	0.577166
17	0.049418	-4.523812	-1.268074	9.101618	1.209518	0.194923	-1.586275	2.312287	0.474955	1.189307	0.027605	1.452766
18	-0.365914	-4.189798	-1.034431	6.867237	1.070474	-0.095076	-0.619913	2.648346	-1.565964	0.895981	0.917232	0.909933
19	0.960122	-4.425698	-1.567341	9.513368	0.411116	0.217404	-1.428052	2.867191	-0.929793	0.567358	0.392506	1.259565

Figure 5: Adjustment of Feature Weights over Training

similar results when we tested with only three features, and Sturtevant and White's Heart's bot had over 60 features [4]. If we were to continue work on this project, our next steps would be tweaking the qLearning agent's reward function and features to make sure it understands to play the basics of the game (play a legal card) before trying to understand the more nuanced details of the game such as effects and skips.

However, we take the directional-correctness of the learning agent to be a good sign that this is a potentially valid line of attack.

On a more encouraging note, we were pleased to see the success of our HMM-based agent and its various extension classes, CardCounter and Heuristic agent. The CardCounter agent was an interesting foray into the expectimax component of the game; however, in a dynamic game like Mao, oftentimes survival is more important than optimizing against the other player, which was shown in its relatively small effect.

Lastly, that Heuristic agent performed the best shows that computational methods like HMM's are great in a rigorous setting, but when higher level concepts like strategy come into play, human knowledge is often crucial for enhanced performance.

As a final note, we feel lucky that the scope of our project allowed us to apply or consider applying most of the algorithms we discussed in the course. It was a uniquely fun experience, and we felt privileged to learn in a course like this. Thank you!

A System Description

A.1 Files Included

Infrastructure.py - houses most of the types which are accessed by other files. Methods for use by other game playing agents.

game.py - The architecture that allows two or more agents to play against one another and records the results

constraints.py - contains definitions of the rules that can be set by agents playing the game.

effects.py - contains definitions of the effects which players can choose to enact.

agents.py - contains the methods and architecture for the game playing agents which are not based on qlearning and the associated super class.

tests.py - methods which makes testing our output simple for the teaching staff.

features.py - contains methods that obtain values of the features which we use in our approximate td learning.

player.py - defines the player class, where Agents inherit from.

qLearner.py - the agent which plays the game using a TD learned strategy.

qTraining.py - plays the q learner against another agent for many simulated games in order to determine the correct weights for our model

qPlaying.py - the qlearning equivalent of the game.py, uses weights from one of the training session to play the Q-learned agent against another agent.

qOutput.py - contains "pickled" text that encodes the weights we received from model training sessions.

A.2 Testing

One of the simplest ways to understand how the game works is to play a round against one of our agents. We tried to make this as straightforward as possible, simply run the following from the command line:

```
python tests.py on 3 human heuristic
```

And you can play 3 rounds of Mao against our hmm agent. Instructions for playing other variants of our Mao inspired game are below.

A.3 Reproducing Results

To replicate all data analysis, simply load

```
    data_analysis.ipynb
```

in a jupyter notebook, using python 2.7.

In order to make it simple for the course staff to test our many agents and reproduce the results which we reference above, we created a file called tests.py in the main repository.

Tests.py is meant to be run with command line input in order to maximize its ease of use. It takes up to six simultaneous command line arguments. Usage instructions can be accessed with the "-h" flag.
The first argument simply specifies the mode to run the file in. Its options are "off" and "on". Specifying off turns some of our output off and is best for watching the game run naturally. On runs the tests with full output and shows some of what's happening under the hood.

The second slot is an option for the number of rounds to play and it takes an int argument. For instance, providing 1 in this slot means that as soon as a player runs out of cards in their hand, the game is over.

The next four slots are all to specify combinations of agents. The options are "heuristic", "hmm", "cardcounter", "human", "simple", "qlearn". These refer to our heuristic agent, hidden markov agent, Expectimax agent, a human player, our basic agent, and our qlearned agent respectively. These can be played in any combination against each other. For instance,

```
python tests.py on 5 human human hmm
```

corresponds to 2 human players playing against an hmm agent. While,

```
python tests.py off 500 heuristic cardcounter hmm
```

Plays the heuristic agent against the expectimax agent and the hmm agent for 500 rounds of gameplay. One usage note is that because of time and processing power contraints, the qlearn agent can only play against up to one other agent.

B Group Makeup

Jack Lane - Wrote the infrastructure code, the game code, the qLearning agent, the qlearner's files, and contributed to the HMM Agent. Additionally contributed to the project proposal and update as well as the project poster, and report.

Andrew Soldini - Did most of the data visualization. Wrote the expectimax agent as well as the simple agent players. Contributed to the outline of game and infrastructure. Contributed significantly to hmm agent. Contributed to project proposal, update and poster, and report.

Matt Sciamanna - Built the naive-heuristic based agent. Contributed to Qlearning agent. Contributed to project proposal and project update as well as poster and report. Built many of the features used in features.py. Ran montecarlo simulated methods to determine card values. Created testing files.

References

[1] Michael Buro, Jeffrey Richard Long, Timothy Furtak, and Nathan R Sturtevant. Improving state evaluation, inference, and search in trick-based card games. In *IJCAI*, pages 1407–1413, 2009.

[2] Goran Radanovic and Haifeng Xu. Cs182 section notes on hidden markov models, Fall 2018.

[3] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[4] Nathan R Sturtevant and Adam M White. Feature construction for reinforcement learning in hearts. In *International Conference on Computers and Games*, pages 122–134. Springer, 2006.