Developing a poker AI agent

1 Introduction

Game playing is an important part of AI. The game of poker, especially Texas Hold'em, has become very popular in the last decades, and the number of players has been growing ever since. Poker is different from games like Chess or Backgammon, because it has incomplete information, namely the opponent's cards. This makes it challenging to come up with a policy for our moves, since they will depend on the opponent's play style. Therefore it is important to create a model for the opponent, which would allow us to predict his next move and "guess" his cards.

Creating a computer agent to play the game intelligently is a great challenge for this class, since it fits under the category of two-player zero sum game which we have previously encountered. To date, the game of poker is still unsolved and this gives us flexibility in choosing our model and plenty of room for improvement.

2 Problem Definition

The goal of our project is to create an agent capable of playing the game of poker in a manner that reflects intelligent behavior. A good player must be able to assess the game state based on his cards, the cards on the table, the size of the pot, and the opponent's actions. He should make decisions given the above factors and also random future events. Also, the agent should be competitive, in that it should do its best to maximize its gains.

For the purpose of this project, we will focus on the two-player (heads up) game with limited bets. Each round, a player can bet $5, $10 or $15. Throughout our project, we use both a smaller deck with 20 cards and 4 suits, and a full-sized deck of 52 cards. In our games, there is no concept of bankroll. However, the winnings and the losses of each player are tracked, such that we can see the reward at the end of the game. All these constraints help reduce the state-space of the game and also the importance of bluffing, since the players don't go all-in. Some of the features that we analyzed include opponent playing style, Q-learning feature vector, agent policies, size of the deck and size of the bets.

3 Rules of the game

Each player is dealt two cards face down, followed by the first round of betting. Then three cards are revealed on the table. This is the *flop* and another round of betting occurs. At the *turn*, a fourth card is placed on the table and players have the chance to bet again. Finally, the *river* reveals the last card and the final round of betting occurs before all the players reveal their cards (*showdown*). The best five card poker hand formed from the two initial cards and the five cards on the table wins the pot. If a tie occurs, the pot is split. At every point in the game, the player

has the options to *fold* (withdraw from the game), *call* (match the current outstanding bet) or *check* (if there is no outstanding bet), and *raise/bet* (add something more than the current bet).

4 Related Work

Poker has been studied by both mathematicians and computer scientists over the past decades. However, due to its incomplete information and randomness, the game remains unsolved. There are entire groups of scientists working on this topic, like the Poker Research Group at University of Alberta, who has been a leader in the field for the past 20 years. Some of their most famous bots include POKI (based on knowledge and simulation methods), VEXBOT (based on game tree search), and Polaris. As of today, the best human player is still better than an AI agent. However, the bots mentioned above are capable of playing against up to ten players at the level of an average human player, which is pretty impressive given the complexity of the game. Researchers use techniques such as reinforcement learning, epsilon-Nash equilibrium, multi-armed bandits, hidden Markov models etc. Unfortunately, most of the literature on this topic is based on fairly advanced concepts and requires knowledge far beyond our class. This is why we tried to limit ourselves to techniques covered in this class and modeled the game using first principles.

5 Challenges

Creating programs to operate with incomplete information in such a complicated manner is fundamentally challenging. The main obstacles are:

- Incomplete information, since a single player can't determine the full state of the game.
- Stochastic outcomes, since the cards are dealt and revealed randomly.
- Extremely large state space (~10^{18} with betting constraints).
- Multiple players. The agent has to take into account the actions of each player (not applicable in our case).
- Adapting to an opponent during a match.

6 Approach

Our first step was to create the game infrastructure from scratch. We designed all the classes and functions need to simulate poker games, which we later used in our experiments with the learning model. The game itself is turn-based and therefore can always be described by states. The opponents follow a fixed policy (at least in our training case) and there is also randomness involved. All these characteristics suggest modeling the game as a Markov Decision Process (MDP). Instead of using a data set, we use five hard-coded opponents as our training models. We implement the Q-learning algorithm with a poker feature extractor, and the epsilon-greedy algorithm in order to explore different states and learn an optimal policy. While learning this

policy, we run Monte Carlo simulations that create games against our fixed-policy opponents and update a feature vector. After learning is done, we run another set of games against our opponents, but now following the policy provided by Q-learning. Then we analyze the results, the weaknesses and strengths of different policies.

6.1 MDP

The distinctive feature of an MDP is its state. In our case, the state should be a collection of observables that would allow our agent to make an intelligent action. As human players, we usually make our decisions based on our cards, the cards on the table, the pot and the opponent's action. All these are part of our state, plus a number $i \in [0, 4]$ that serves as an indicator for current phase of the game.

In order to reproduce the interaction between players, we came up with a 4-step model (Fig. 1) that applies to all four game stages (pre-flop, flop, turn, river). The first stage ($i = 0$) corresponds to the opponent's first move. All the actions are available in this stage. The second stage ($i = 1$) is the agent's response to the opponent's move. The agent can bet an amount greater than or equal to the opponent's last bet, or fold. The agent never folds if the opponent just checks. The third stage ($i = 2$) is the opponent's response to the agent's move. Since we are doing only one round of bets per stage, the available actions are either call the agent's last bet or fold. The fourth stage ($i = 3$) exists only in order for the agent and opponent to be symmetric. In this stage, the agent "acknowledges" the opponent's last move and always checks. Whenever one of the players folds, or we reach showdown, set $i = 4$ as an indicator of end state. Note that the agent and opponent can swap places, such that the opponent doesn't always go first.

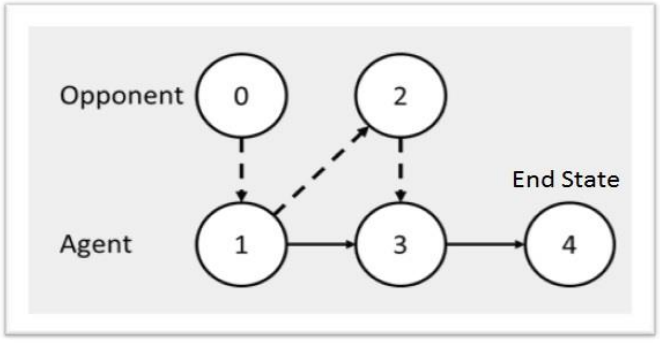

Figure 1 4-steps model of agent-opponent interaction

Similar to the majority of the games, poker has all the rewards at the end. Therefore we set the rewards to zero for intermediate stages, while the reward for end stage is either the amount of money won or lost in that game. The transition probability comes in play only when a new card is dealt. Since all the cards are distinct, the probability to transition to a certain state when revealing a new card is

$$T(s, a, s') = \frac{1}{\#\ of\ cards\ left\ in\ deck}$$

When 3 cards are drawn at flop, the probability is the product of three individual probabilities of drawing a card. The complete MDP model is formulated below:

- *State* = (cards in hand, cards on table, pot, opponent last action, indicator i)
- *Start state* = (cards in hand, [], 0, None, 0) when agent starts first
  (cards in hand, [], opp. first bet, opp. first action, 1) when agent starts second
- *Actions* = (Fold, 0), (Check, 0), (Bet, 5), (Bet, 10), (Bet, 15). Each action is a tuple of action name and amount of money added to pot. Depending on i, actions may be restricted as described above.
- *Transition probability:*

$$T(s, a, s') = \begin{cases} \frac{1}{d}, & if\ i = 3\ before\ turn\ and\ river \\ \frac{1}{d(d-1)(d-2)}, & if\ i = 3\ before\ flop \\ 1, & otherwise \end{cases}$$

- *Rewards:*

$$R(s, a, s') = \begin{cases} pot - total\ agent\ bet, & if\ s'[4] = 4\ and\ agent\ won \\ -total\ agent\ bet, & if\ s'[4] = 4\ and\ agent\ lost \\ 0, & otherwise \end{cases}$$

- *Is End :* $s[4] == 4$

6.2 Opponent modeling

Types of Players	
Loose-Passive	Less dangerous players; Emphasis on playing many hands and calling when possible. These players hardly ever raise or bet.
Loose-Aggressive	Hard to predict, but easily exploitable, they tend to raise even with weak hands.
Tight-Passive	They hardly ever raise, play few hands and call a lot.
Tight-Aggressive	These opponents play few hands, but when they have good cards, they tend to raise a lot. They are probably the most dangerous category.

The opponent's actions are crucial to determining the optimum actions that the agent will take, and thus the agent's utility (in this case, money earned). In order to take this into account, we created 5 opponent models for the agent to learn against. One was completely random in its actions (equal probability of all actions); the remaining four correspond to the four combinations

of loose and tight with passive and aggressive. Behaviors were hard-coded for each state of the game using a hand strength evaluation function based on Sklansky hand ratings for initial hands (bucketing by hand rank) [3]. Initially, the behavior of each opponent was very deterministic for each hand rank. This was changed to more of a probabilistic model, such that the hand strength was adjusted depending on the new revealed cards. The probabilities of the opponent taking an action given a hand rank can be found in the appendix.

6.3 Q-learning and Features

In order to find the optimal policy that the agent should follow in his games, we resort to reinforcement learning. More specifically, we applied Q-Learning as shown in class, and in the expression below.

On each state, action, reward and successor state (s, a, r, s'):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta[\widehat{\mathrm{Q}}_{\mathrm{opt}}(\mathrm{s}, \mathrm{a}; \mathbf{w}) - \left(\mathrm{r} + \gamma \widehat{\mathrm{V}}_{\mathrm{opt}}(\mathrm{s}')\right)]\phi(\mathrm{s}, \mathrm{a})$$

Where $\widehat{\mathrm{V}}_{\mathrm{opt}}(\mathrm{s}') = \max_{\mathrm{a}' \in \mathrm{Actions}(\mathrm{s}')} \widehat{\mathrm{Q}}_{\mathrm{opt}}(\mathrm{s}', \mathrm{a}')$. η is the step size and $\gamma = 1$ is the discount factor. For this project, we take $\eta = \frac{1}{\sqrt{\mathrm{k}}}$, where k is the iteration count.
Exploration probability in the epsilon-greedy algorithm was set to 0.2 during the learning phase. The main challenge was figuring out the feature extractor that would result in the best learning.

We initially started with a features extractor that kept track of the total pot, the table and hand cards, both together and separately, the winning cards, the winning cards value as well as the individual cards remaining in the deck, similar to the blackjack implementation. This resulted in gigantic weight vectors that had redundant information.

In our next iteration of the features extractor, we managed to reduce the size of the resulting weight vector by a factor of three. We simplified some features to only contain the card ranks (without the suits) to make them more generalizable. We also bucketed the assessed values for the hands to get rid of unnecessary features. This also allows for more generalization as, for example, a pair of 7s is treated similar to a pair of Kings when it comes to making a decision. As it currently stands, our feature vector contains the following indicator features:

1. Pot Value, Action tuple. eg. ('pot', 25, ('Bet', 5))
2. Sorted Hand Ranks, Action tuple. eg. if hand = (13, ‘Spades’), (3, ‘Clubs’), feature is ('h', (3, 13), ('Bet', 0))
3. Sorted Table Ranks, Action tuple. eg. if table = (10, ‘Spades’), (8, ‘Clubs’), (9, ‘Hearts’), feature is ('t', (8, 9, 10), ('Bet', 0))
4. Only if more than two cards of the same suit in table and hand cards combined: Suit, Number of suited cards, Action. eg. ('Spades', 4, ('Bet', 0))
5. Value of best 5 cards combination, Action. eg. ('value', 80, ('Bet', 0))
6. Opponent action, Indicator i, Action. eg. ('oppAct', ('Bet', 0), 4, ('Bet', 0))

The improved feature extractor did significantly reduce the size of the weight vector and slightly improved learning. However, through experimentation we realized that we were missing a very

important feature that links a suit multiplicity to the phase of the game. For instance, having three spades in a set of 5 cards (at the flop) has different implications than having those in a set of 7 cards (at the river). Our feature vector does not reflect that information and thus sometimes results in decisions that do not maximize rewards. One way to solve this is to include the full set of cards between hand and table as an additional feature.

7 Experiments and results

For our experiments, we ran four separate trials against the four separate opponents, resulting in 16 tests. Because Q-Learning incorporates feedback on only the states it sees, and the state space of the game is massive, the four trials guarded against any large variations. After running the initial sets of trials, we found that the weight vectors were too massive - saving them in files resulted in the creation of 1 MB to 5 MB files. As these were cumbersome, we reduced the number of features extracted to better fit the data. We ran more iterations of the 16 sets of trials, varying single parameters such as the inclusion or exclusion of a feature and the number of Q-Learning iterations run.

For the final set of data, we ran a total of 32 trials: 16 of which were four trials against the four opponent types using 5,000 iterations of Q-Learning, and another 16 used 10,000 iterations of Q-Learning. The following tables express the average results and winnings against each of the four opponents at the two levels of Q-Learning. We define a 'good fold' below as when the agent would have lost if he had remained in the game, and a 'bad fold' as when the agent would have won.

Percentage of Game Results Against Various Opponents

	LAG5k	LAG10k	TAG5k	TAG10k	LPA5k	LPA10k	TPA5k	TPA10k
Win	37.7%	36.8%	27.5%	25.6%	48.1%	48.1%	29.6%	28.2%
Opponent Left	1.5%	0.9%	27.8%	26.9%	0.8%	0.5%	27.1%	29.7%
Lose	27.9%	27.4%	21.2%	19.7%	44.8%	44.7%	25.8%	24.7%
Good Fold	10.1%	10.7%	6.6%	8.3%	1.8%	1.9%	5.0%	5.0%
Bad Fold	22.8%	24.1%	16.9%	19.4%	4.5%	4.7%	12.4%	12.3%

Histograms for the agent's winnings against the opponent's play style are included in the appendix. Each player is represented by an abbreviation of its play type plus the number of iterations it was ran on (eg. LAG5k means Loose-Aggressive opponent with 5000 iterations).

7.1 Analysis:

After performing the experiments, we collected the results in the histograms 1A-5A in the appendix. Notice that in order to better understand the behavior of our player, we distinguish

between wins by card advantage and wins due to opponent's fold. Also, when the agent folds, we keep simulating the game to see whether he would have won at showdown. This way we can distinguish unnecessary folds from the justified ones.
One of the distinctive features of the histograms is peak values around rewards of -15, 0 and +15. The peak at -15 can be explained by the tendency of our opponent to prevent huge losses. When it sees that it has nothing in hand and the opponent is still betting, our player decides it is safer to fold at an early stage, rather than loose at showdown. This is actually a desired behavior in a player. The symmetric peak at +15 is especially emphasized for passive players. Passive opponents hardly ever raise, and instead prefer to call. This is why the winning pot is so small against those players and states with bigger winning pot are not explored. The peak at 0 is only characteristic for tight players. This is because they play only when they have a strong combination in their hand. If this is the case, they bet a lot, and this behavior makes our agent fold early and save money instead of accepting their challenge, which again contributes to the peak at 0.
It is also important to look at the exploration of states with different rewards. In the case of aggressive players, the states with bigger pots are well explored, because this type of players makes high bets and our agent has the opportunity to learn on all these states. However, the agent finds it hard to learn the behavior of passive players at high stakes, because they simply don't reach those states.
The table below reveals that our agent performed best as a *loose player* type. Indeed loose players more willing to bet, and our reward per win is higher in that case. However, the win rate is significant enough that we can see the agent learning and taking advantage of the opponent's tendency to bet a lot.

Agent type	**Percentage of games won**	**Average reward per game($)**
Loose-Aggressive	37.7%	19.44
Tight-Aggressive	52.5%	9.52
Loose-Passive	48.6%	14.98
Tight-Passive	57.9%	8.54

8 Conclusion and future work

As of today, our AI Poker agent plays the game below the average level of a casual human player. In the long run, even against its own opponent type, the agent loses on average $0.5-1.5 per game. However, this is still a significant improvement from our baseline, where bets were not considered at all as the opponent was static.

One of the reasons why our player isn't very efficient is because during training it doesn't explore enough states and adopts a very conservative playstyle, where folding is preferred over high stakes risk. This behavior is quite different from the human players', because the agent doesn't have the notion of "luck" or "risk" that would make him more aggressive.

Adding personality to the player (features like bluffing) could improve its performance by adding variance in strategy.

In general, it is quite impressive how using AI techniques, the agent, without having any previous knowledge of the game, figured out the correlation between the strength of its hand and the actions it should adopt. Most importantly, it figured out that folding early is better than losing at showdown.

One way we could improve the results is by improving the feature vector and exploring more the state space. The feature vector implemented ultimately did not do enough to consider the full range of interactions between terms. Even with such a change though, it would still be difficult to run this agent in short periods of time due to the large weight vector generated from this. The extremely large state-space prevents efficient runtimes. Also, we could improve the hand strength evaluation function, by using some sort of clustering (K-means). Our current work includes training the agent against itself, instead of fixed opponents. This should create a more generic and broad play style.

In this project we restricted ourselves to techniques used in class. However, it would be interesting to see how other techniques, such as neural networks, genetic algorithms, hidden Markov models, and epsilon-Nash equilibrium, apply to the complex game of poker. For example, hidden Markov models would fit great in the context of an incomplete information game.

9 References

[1] AITopics. "Poker." AITopics, n.d.Web. 30 Nov. 2014. <http://aitopics.org/topic/poker>.

[2] PokerBots – Smart Poker Agents. Rep. University of Minnesota, 13 Dec. 2005

[3] A complete list of Sklansky-Chubukov rankings: http://www.liveschoolpoker.com/poker-school/table/numbers-of-sklanski-chubukov.html

[4] University of Alberta Computer Poker Research Group: http://poker.cs.ualberta.ca/

10 Appendix

Table A1: Decision Matrix for Tight-Aggressive Opponent (Initial)

Sklansky Hand Rank	Decision				
	(Fold, 0)	(Bet, 0)	(Bet, 5)	(Bet, 10)	(Bet, 15)

1	0	0	0	0.2	0.8
2	0	0	0.2	0.6	0.2
3	0.1	0.18	0.54	0.18	0
4	0.2	0.16	0.48	0.16	0
5	0.3	0.14	0.42	0.14	0
6	0.4	0.48	0.12	0	0
7	0.5	0.4	0.1	0	0
8	0.6	0.32	0.08	0	0

Table A2: Decision Matrix for Loose-Aggressive Opponent (Initial)

Sklansky Hand Rank	Decision				
	(Fold, 0)	(Bet, 0)	(Bet, 5)	(Bet, 10)	(Bet, 15)
1	0	0	0	0.2	0.8
2	0	0	0.2	0.6	0.2
3	0	0.2	0.6	0.2	0
4	0	0.2	0.6	0.2	0
5	0	0.2	0.6	0.2	0
6	0	0.8	0.2	0	0
7	0	0.8	0.2	0	0
8	0	0.8	0.2	0	0

Table A3: Decision Matrix for Tight-Passive Opponent (Initial)

Sklansky Hand Rank	Decision				
	(Fold, 0)	(Bet, 0)	(Bet, 5)	(Bet, 10)	(Bet, 15)
1	0	0	0.2	0.6	0.2
2	0	0.2	0.6	0.2	0
3	0.1	0.18	0.54	0.18	0
4	0.2	0.16	0.48	0.16	0
5	0.3	0.56	0.14	0	0
6	0.4	0.48	0.12	0	0

7	0.5	0.5	0	0	0
8	0.6	0.4	0	0	0

Table A4: Decision Matrix for Loose- Passive Opponent (Initial)

Sklansky Hand Rank	Decision				
	(Fold, 0)	(Bet, 0)	(Bet, 5)	(Bet, 10)	(Bet, 15)
1	0	0	0.2	0.6	0.2
2	0	0.2	0.6	0.2	0
3	0	0.2	0.6	0.2	0
4	0	0.2	0.6	0.2	0
5	0	0.8	0.2	0	0
6	0	0.8	0.2	0	0
7	0	1	0	0	0
8	0	1	0	0	0

Table 5A: Betting Probability Matrix for Aggressive Opponents (Post-Flop)

Hand Rank	Decision			
	(Bet, 0)	(Bet, 5)	(Bet, 10)	(Bet, 15)
Nothing	0.84	0.16	0	0
Pair	0.84	0.16	0	0
Two Pairs	0.84	0.16	0	0
Three-of-a-Kind	0.36	0.48	0.16	0
Straight	0.36	0.48	0.16	0
Flush	0	0	0.6	0.2
Full House	0	0	0.2	0.8
Four-of-a-Kind	0	0	0.2	0.8
Straight Flush	0	0	0.2	0.8
Royal Flush	0	0	0.2	0.8

Table 6A: Betting Probability Matrix for Passive Opponents (Post-Flop)

Hand Rank	Decision

	(Bet, 0)	(Bet, 5)	(Bet, 10)	(Bet, 15)
Nothing	**0.98**	**0.02**	**0**	**0**
Pair	**0.98**	**0.02**	**0**	**0**
Two Pairs	**0.98**	**0.02**	**0**	**0**
Three-of-a-Kind	**0.92**	**0.06**	**0.02**	**0**
Straight	**0.92**	**0.06**	**0.02**	**0**
Flush	**0.2**	**0.6**	**0.2**	**0**
Full House	**0.2**	**0.6**	**0.2**	**0**
Four-of-a-Kind	**0.2**	**0.6**	**0.2**	**0**
Straight Flush	**0**	**0.2**	**0.6**	**0.2**
Royal Flush	**0**	**0**	**0.2**	**0.8**

Table 7A: Folding Probability Matrix for Each Player (Post-Flop)

Hand Rank	Tight	Loose
Nothing	**0.2**	**0.05**
Pair	**0.1**	**0.01**
Two Pairs	**0.05**	**0**
Three-of-a-Kind	**0**	**0**
Straight	**0**	**0**
Flush	**0**	**0**
Full House	**0**	**0**
Four-of-a-Kind	**0**	**0**
Straight Flush	**0**	**0**
Royal Flush	**0**	**0**

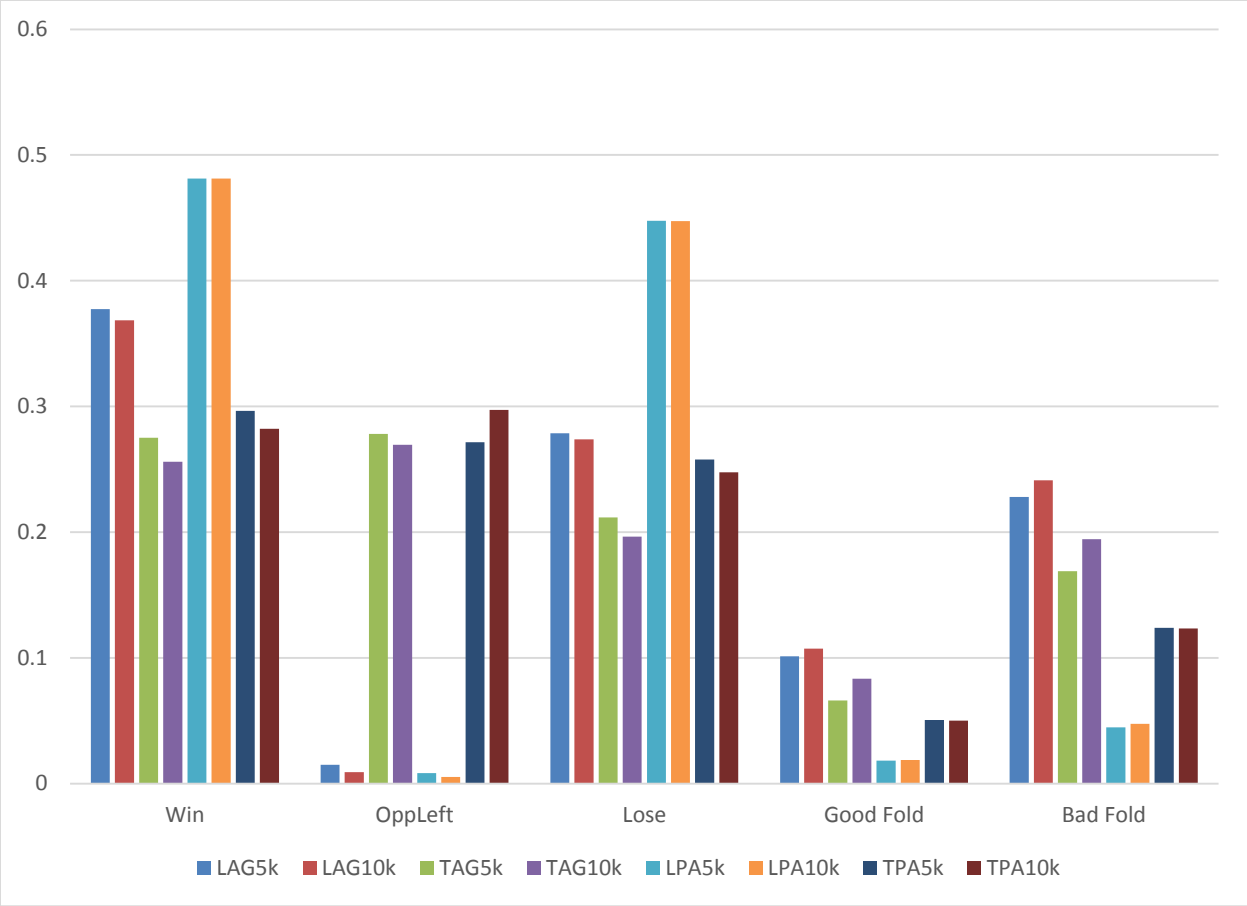

Figure 1A: Agent Result Histogram

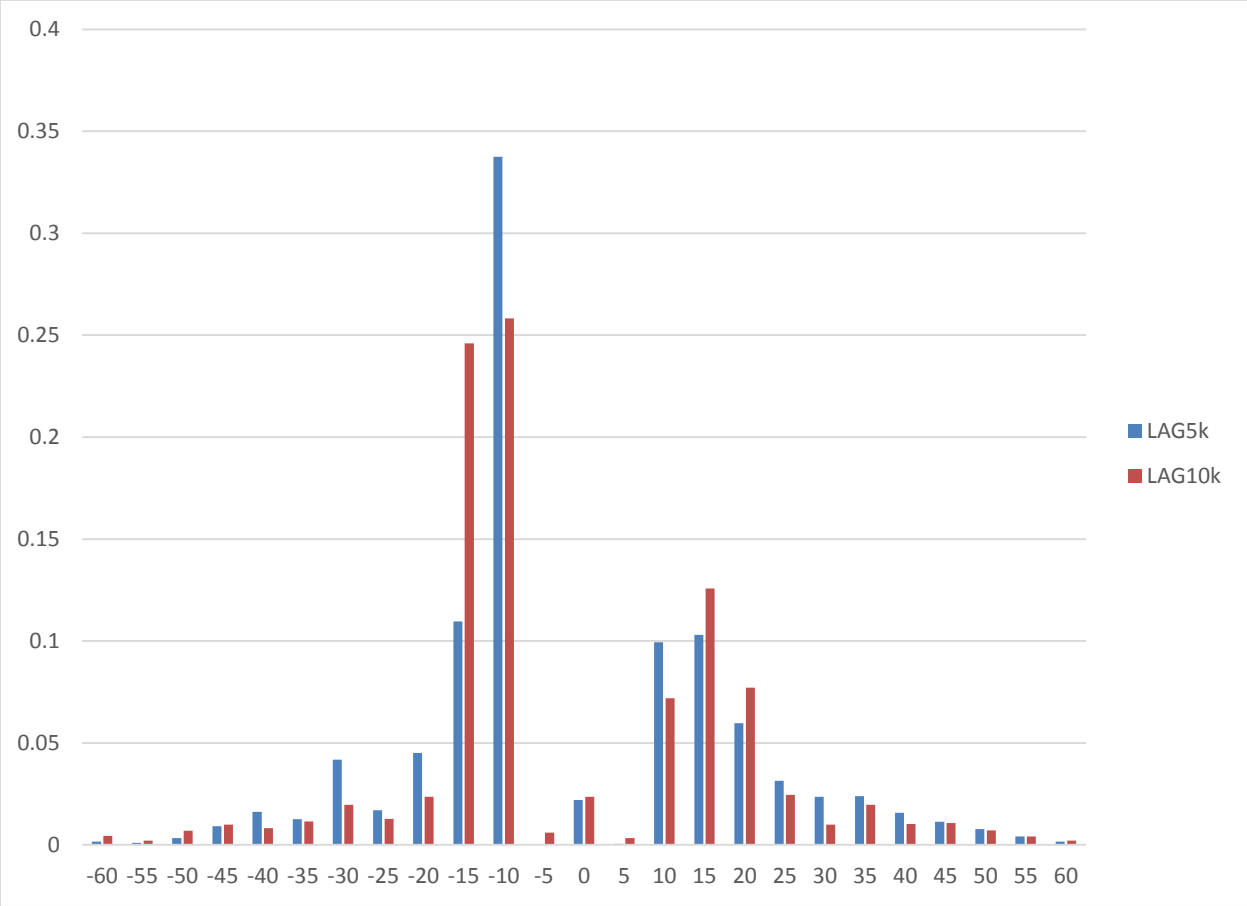

Figure 2A: Histogram of Earnings Against LAG Opponents

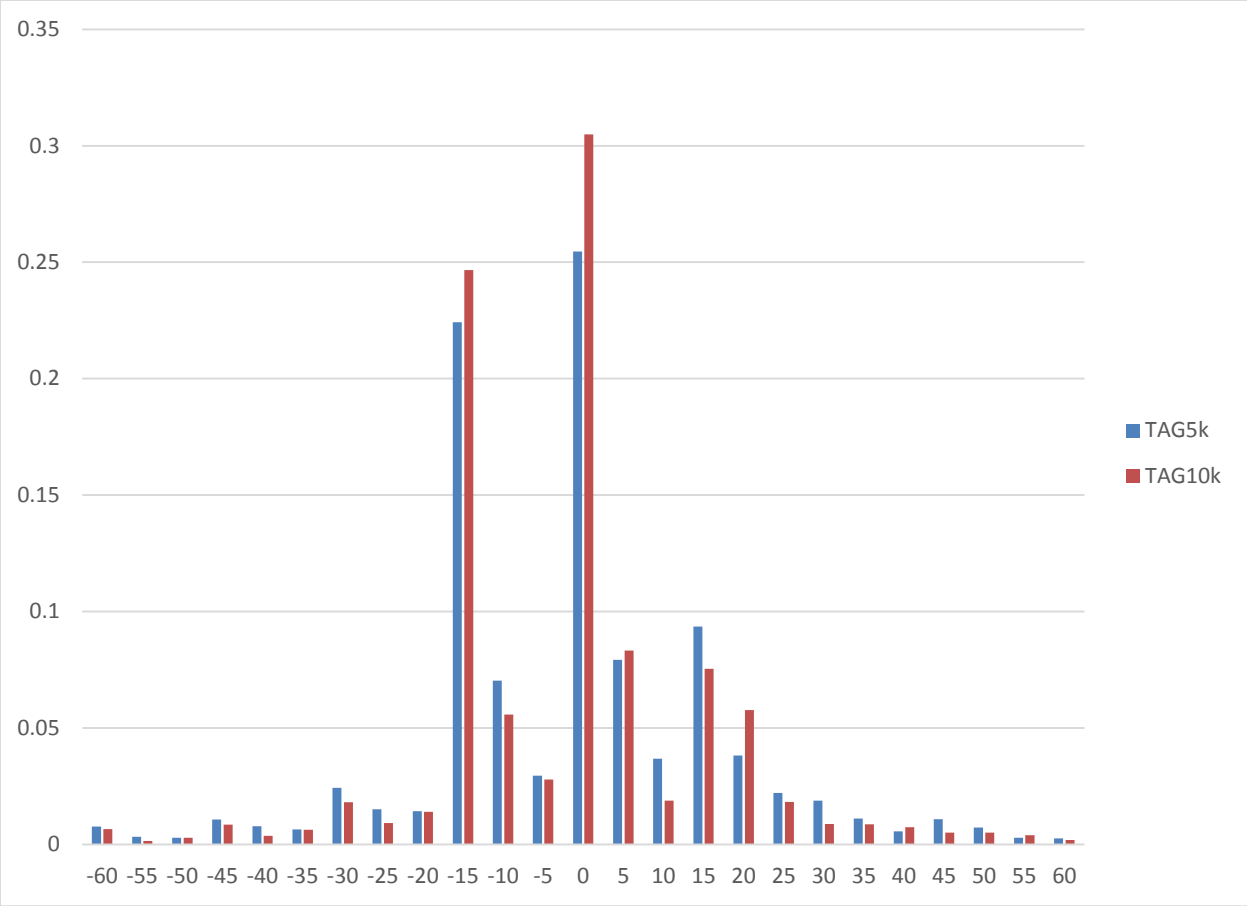

Figure 3A: Histogram of Earnings Against TAG Opponents

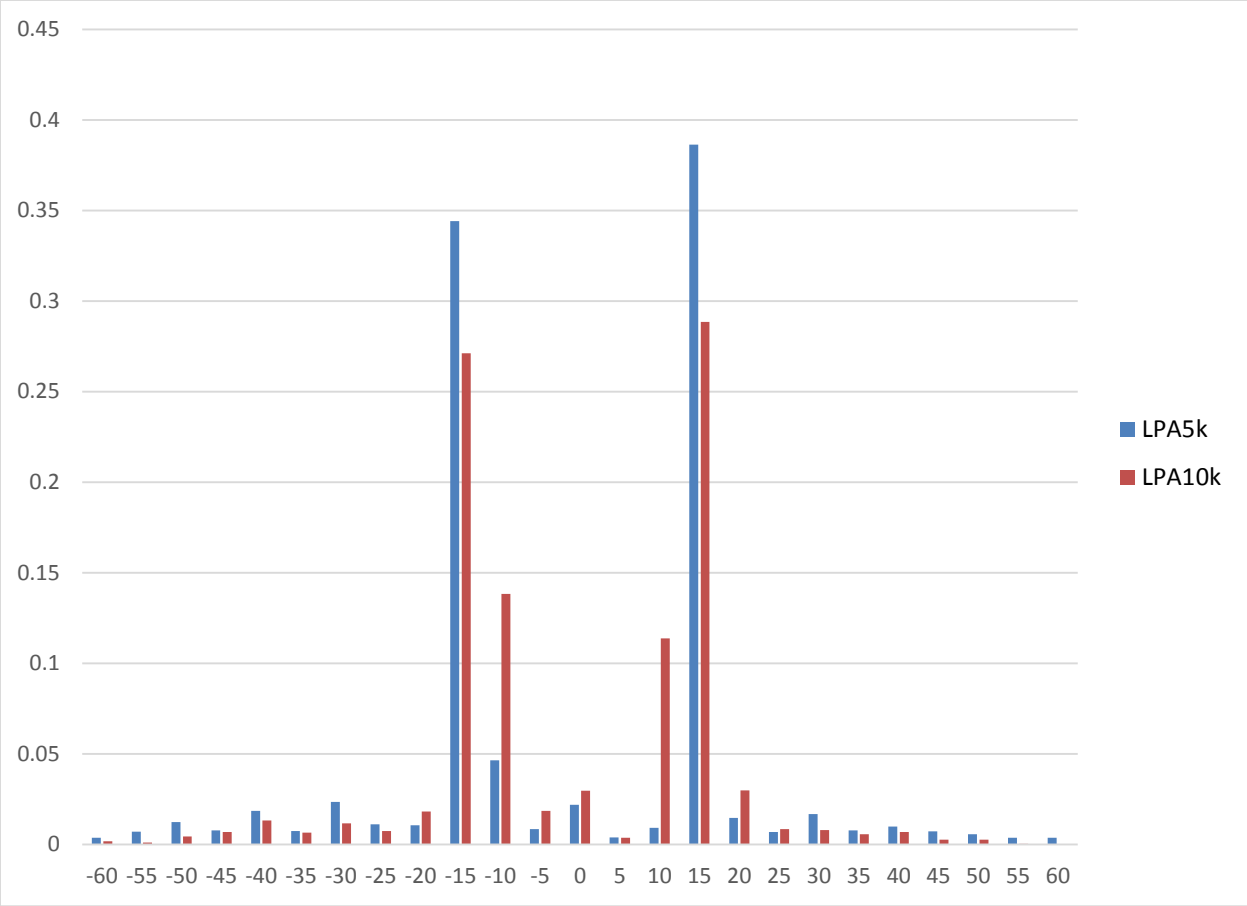

Figure 4A: Histogram of Earnings Against LPA Opponents

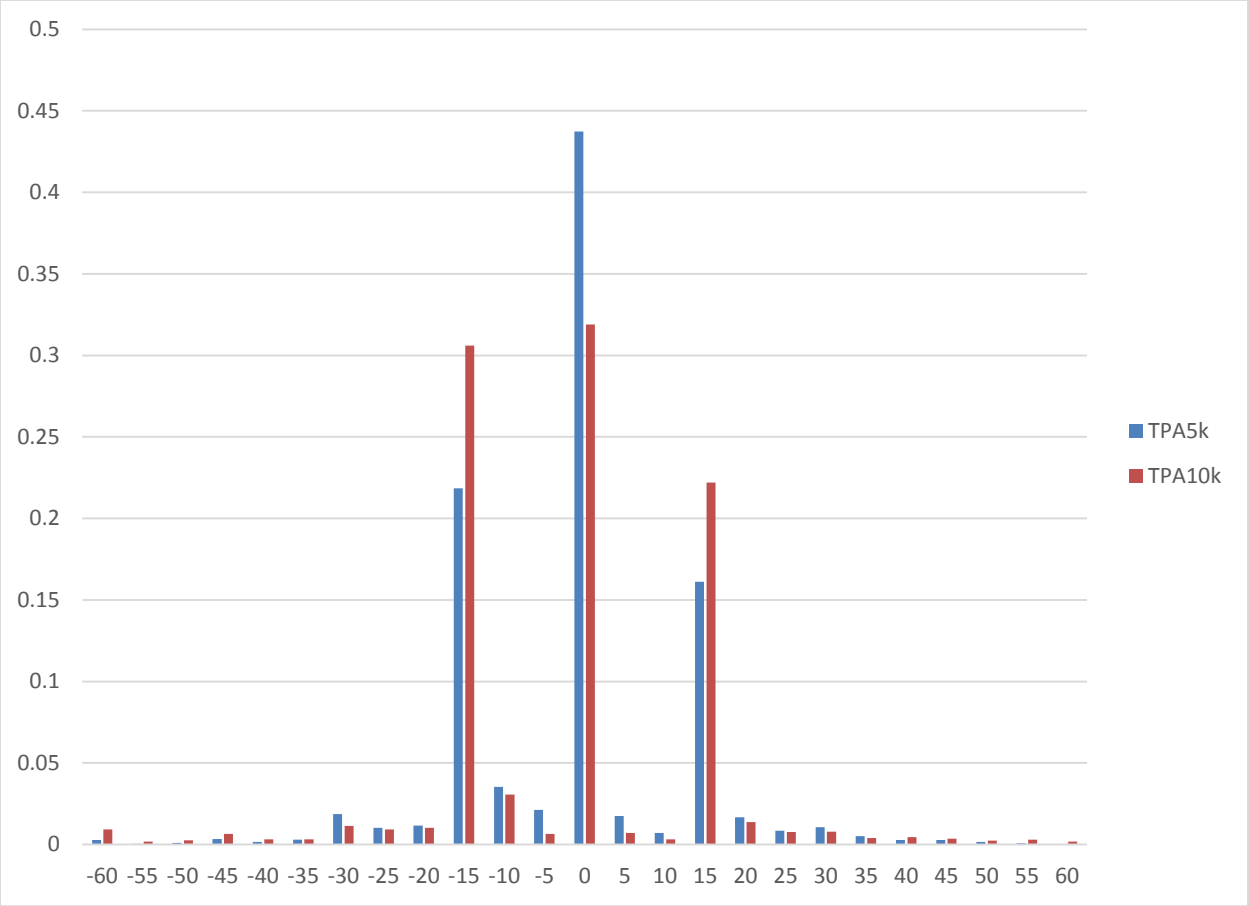

Figure 5A: Histogram of Earnings Against TPA Opponents