Texas Holdem Poker AI

Utilizing Machine Learning Techniques

By

Isaac Lewis

Advised by

Prof Brian Mak

Submitted in partial fulfillment of the

Requirements for COMP397 (Final Year Project)

In the

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

2010-2011

“Poker is the game closest to the western conception of life, where life and thought are recognized as intimately combined, where free will prevails over philosophies of fate or of chance, where men are considered moral agents and where - at least in the short run - the important thing is not what happens but what people think happens.”

John Luckacs

"The smarter you play, the luckier you'll be."

Mark Pilarski

Abstract

Poker is a stochastic and imperfect information domain, and therefore requires the use of different artificial intelligence techniques to those employed successfully in deterministic, perfect information domains such as chess. We implement an artificial poker player, or pokerbot, that is based on the miximax algorithm, a generalization of minimax.

This algorithm relies on a good model of an opponent's strategy. We experiment with the use of the supervised learning algorithms, namely naïve Bayes classifiers and neural networks, to construct such a model. We demonstrate the effectiveness of this approach, along with the slight superiority of neural networks over naïve Bayes classifiers. We also analyze some of the shortcomings of our implementation.

1. Introduction

1.1. Overview

Poker is an interesting area for artificial intelligence (AI) research, as, unlike games such as Chess, Go or Checkers, it shares many of the features of the real world; specifically, it *is stochastic* – chance plays a large role – and a game of *imperfect information* – players must make decisions without knowing the exact state of the world.

AI techniques that are useful in games like Chess – such as the minimax algorithm – are therefore ineffective in the context of poker [1]. Instead, different AI techniques must be developed to create a winning poker AI. Poker research does not only aim at winning poker games - it also aims to gain insights into AI that could be utilized in real world applications.

Another factor in poker AI research is that poker is a game that rewards players for misleading opponents – bluffing and trapping – and conversely, trying to second-guess opponents' behavior, calling their bluffs and avoiding their traps. Just as Go forces computers to compete with humans' innate pattern recognition ability, so poker requires them to compete with humans' capacity for misdirection and deceit. Again, such AI may find utility in the real world - such as in business or politics.

It should be noted there are several different forms of poker and several different game types, varying by the number of players and reward structure. This project was focused on Heads-Up Limit Texas Holdem (an explanation of the rules of poker and special terms can be found in appendix B). Subsequent references to 'poker' refer to this game specifically, unless otherwise indicated.

1.2. Objectives

An important element of creating a successful computer poker player (or pokerbot) is opponent modeling; that is, attempting to predict an opponent’s actions from past observations [2]. For this project, we wrote a pokerbot that implemented an algorithm known as miximax, and attempted to find a superior opponent modeling strategy by utilizing machine learning techniques. More information on the algorithms used can be found in sections 1.3.1, 1.3.2 and 2.4.

The first part of the project required creating a **poker server** that could referee a game of Texas Holdem poker between two opponents. We also created a **web-based frontend** for the server, to provide a more convenient interface for humans to play against computer players.

We then implemented a **basic pokerbot**; that is, one that implemented the miximax algorithm, but used a very basic form of opponent modeling. The basic pokerbot will from now on be referred to as the Terrific Exploitative Algorithmic bot, or TeaBot for short.

We then attempted to use the approach outlined above to create an **improved pokerbot**; this required experimenting with different machine learning techniques to discover which provided the best gameplay performance. The improved pokerbot will be referred to as the Correct Opponent Forecasting For Enhanced Effectiveness bot, or CoffeeBot.

Testing the performance of TeaBot and CoffeeBot involved created several **rule-based pokerbots**, to function as their opponents. We also had to develop an **experimental design** that could accurately determine the relative strengths of each player when faced with the high levels of variance inherent in the game of poker.

1.3. Literature Survey

With the rise in popularity of online poker in recent years, an increasing number of people are attempting to create computer poker players that can win real money, with varying degrees of success. Since online casinos usually frown upon this practice (known as *botting*), such people tend not to publicize their successful techniques. [3]

On the other hand, computer poker has also begun to receive increasing amounts of academic attention, with a large amount of research coming from the Computer Poker Research Group at the University of Alberta. As survey of common approaches to poker AI follows, along with examples of some notable pokerbots.

There are two main types of strategies followed by successful poker AIs, **equilibrium** and **exploitative** strategies.

Equilibrium strategies attempt to find a strategy that approximates a Nash equilibrium. The Nash equilibrium is a concept from game theory. It refers to a set of strategies for a particular game, where, if each player follows a strategy within the set, no player can benefit from switching strategies. Poker players that follow an equilibrium strategy are not exploitable by their opponents, and so are not likely to lose a large amount; however, they are also unable to exploit their opponent's weaknesses effectively, and so are also unlikely to win by a large amount. However, in practice, equilibrium players can do well, especially when faced with opponents that make mistakes. Another consideration is that computing the true Nash equilibrium for poker is currently infeasible, and so equilibrium players generally play an approximation to the equilibrium strategy. Such approximations tend to be slightly exploitable.

Exploitative strategies, on the other hand, attempt to adapt their play to take advantage of their opponent's weaknesses; this means they can potentially win much more, but also lose much more, especially against an opponent that changes its strategy over time. As an example, consider an opponent that follows an exploitable strategy X; an exploitative player may quickly switch to strategy Y, which takes full advantage of X. However, strategies which exploit other strategies are usually themselves exploitable (for game theoretic reasons mentioned above), which means that the opponent could then switch to a strategy Z which takes advantage of Y. This type of behavior occurs frequently between advanced human players, who are adept at adapting their strategy as the situation dictates. For example, a player might be playing too cautiously, leading to her opponent bluffing more frequently to take advantage of this over caution. Once the player realizes how often her opponent is bluffing, she too can change her strategy; for example, by catching her opponent bluffing she may be able to win a large amount.

Rule-based bots form another class; they follow explicitly defined behavior depending on their situation. They are generally ineffective, as there are too many possible situations in poker to define rules for all of them. However, since they are simple to create they are very common. In fact, software exists that allows such bots to be created by individuals with little or no programming knowledge [3]. **Loki** was one of the first bots created by the University of Alberta's Computer Poker Research Group, and followed a rule-based strategy.

Polaris utilizes **Counterfactual Regret Minimization**; an algorithm which constructs a Nash-equilibrium strategy for an abstract form of poker and then repeatedly plays the bot against itself to refine the strategy [1]. In short games, it is very strong. At the 2007 Man-Machine Poker Tournament it defeated a human poker champion, Phil Laak. However, given sufficient time, an exploitative opponent can find weaknesses in its play. These weaknesses stem from the difference between the real game of poker and the abstracted version Polaris uses to construct its strategy.

Vexbot is an exploitative bot that utilizes the **miximix** algorithm. This is a generalization of the minimax algorithm which takes into account uncertainty about an opponent's actions (a detailed explanation of the algorithm can be found below). A key element of this algorithm is opponent modeling, that is, calculating the probability an opponent will take a particular action in a given situation. Vexbot uses observations of an opponent's previous betting patterns as the basis for its opponent modeling strategy. As an exploitative bot, Vexbot is generally weak at the beginning of a game, but becomes stronger as it adapts to its opponent. Its main weaknesses are that it can take too long to learn an opponent's strategy, and that it cannot adapt to an opponent that changes its strategy over time.

1.3.1. Miximax and Miximix Algorithms

As the name suggests, the miximix and miximax algorithms are modified versions of the minimax algorithm. Minimax is a commonly used AI strategy for deterministic, zero-sum, perfect information games, and has seen success when applied to popular games such as TicTacToe, Chess, and Checkers [10].

The minimax algorithm constructs a game tree where every node represents a possible game state, the root node represents the current state of the game, and a node's children represent the set of possible successor states to the parent node's state. Leaf nodes represent endgame states, and are assigned a negative or positive expected value (EV) depending on whether they correspond to a win for the opponent or the player. Non-leaf nodes are of two types, player nodes and opponent nodes, representing which player must make a decision at that particular point.

A basic assumption of the minimax algorithm is that the maximizing player always tries to maximize their EV, and their opponent always tries to minimize it (hence the name minimax). This is achieved by recursively calculating the EV for each node in the tree. Player nodes are assigned the maximum EV of their child nodes (representing the player making the move that maximizes EV), and opponent nodes are assigned the minimum EV of their child nodes (representing the opponent making the move that minimizes EV). Once the EV of the root node is calculated, the player knows the optimum move to make in the current situation, and can act accordingly.

For simple games such as Tic-tac-toe, the minimax algorithm can be used as described, and finds an unbeatable strategy, if one exists. For more complex games such as Chess, the game tree is so large as to make running the minimax algorithm over the entire tree computationally infeasible. The algorithm must be adapted to prune uninteresting branches of the tree, and incorporate a heuristic evaluation function to estimate the EV for non-leaf nodes without having to recursively expand them. Nevertheless, the basic concept remains the same.

For stochastic games such as backgammon, a generalized version of the minimax algorithm called expectimax is used. Expectimax incorporates chance nodes in addition to player and opponent nodes; the EV of a chance node is defined as the sum of the expected values of its children, with each child node weighted by the probability of reaching that child node.

For imperfect information games such as poker, even if we knew the opponent was following a perfect strategy, it would be impossible to predict their actions perfectly, since some of the information relevant to their decision is hidden to us (in the case of poker, this hidden information is the opponent's two hole cards). We therefore assume the opponent is following a mixed strategy, where in a given situation the actions they take follow a fixed probabilistic distribution. For example, in the case of poker, we may determine that in a given situation our opponent will raise 23% of the time, call 72% of the time, and fold 5% of the time. With this information, we can then treat opponent nodes similarly to chance nodes; the EV of an opponent node is defined as the sum of the expected values of its children, with each child node weighted by the probability the opponent will take the corresponding action. Calculating the probabilities for an opponent's actions in a given situation is therefore an important component of this strategy, and is called opponent modeling.

Leaf nodes are also treated slightly differently, since we must consider how likely we are to win at the end of the hand, given a particular sequence of actions. The amount of chips we could potentially win at a particular leaf node is multiplied by our probability of winning to determine the true EV of that leaf node. In the case of poker, this requires determining a probability distribution of the opponent's hand strength given a particular sequence of actions; by comparing this to our own hand strength, we can generate an estimate of our probability of winning.

If we still treat player nodes as in the minimax algorithm, assuming the player always takes the action with the maximum EV, the resulting algorithm is known as **miximax**.

Always following the action with that maximizes EV may lead to play that is too predictable, and therefore we may ourselves wish to follow a mixed strategy. If this is the case, we will therefore treat player nodes in a similar way to opponent nodes, and the resulting algorithm is called **miximix**. Determining the mixed strategy to follow is therefore an additional issue to be dealt with when implementing the miximix algorithm.

Our final year project made use of the miximax algorithm, as we wished to focus our attention on the problem of opponent modeling. We attempted to find a solution to this problem through the use of machine learning techniques. The basic strategy was to treat opponent modeling as a **supervised learning** problem.

1.3.2. Supervised Learning

Supervised learning problems are a class of problems where a machine learning algorithm is given a set of training data, where each piece of data consists of a set of input features, and an output value. From that data, the algorithm attempts to find a function from the input features to the output. The function can then be used to make predictions on new data [4]. If the output value is discrete, the function is called a classifier; if the output is continuous, the function is known as a regression function.

For example, spam filters typically use a supervised learning algorithm known as a naïve Bayes classifier, and are trained on a set of emails previously classified as either spam or non-spam. In this case, the features may be the frequency of various words found in the emails, and the output will be a classification of the message as spam or legitimate. Other common supervised learning algorithms include decision trees, neural networks, and support vector machines.

When treating opponent modeling as a supervised learning problem, we obtained training data from observations of an opponent's past actions, and attempted to find a function that mapped known information about the game to a prediction of the opponent's action. Therefore, the problem was a classification problem, as we wished to classify a given state as leading to a raise, a call, or a fold.

Half of the problem was choosing the correct input variables to the function - in other words, determining which pieces of information were most relevant to an opponent's decision. Determining which supervised learning algorithm to use was the other half. During our project, we experimented with naïve Bayes classifiers and neural networks.

2. Design

2.1. Design Poker Server and Interface

We considered how the poker server was to be implemented, the communications channel used to send messages between the server and clients, and the protocol for the messages sent. We also designed a web interface to facilitate play between TeaBot, CoffeeBot and human players.

We decided to use an object-oriented design for the poker server, facilitating the use of the software engineering concept of separation of concerns. For example, the code for enforcing the rules of poker and the code for communicating with clients were encapsulated in different classes, allowing the implementation of one component to be changed without affecting other components. A basic overview of the object-oriented architecture follows.

The PokerServer class, when instantiated, initializes some configuration constants and creates two Player objects. The Player objects both represent the state of one of the poker players, and handle the network interface to the client programs. Since the Players can hold chips, they include the ChipStore module, which facilitates chip transactions. Another ChipStore object is created to represent the pot.

Whenever a hand is played between the two Players, a Game object is created. The Game class implements the majority of the rules of poker. Some helper classes include Deck, which represents a shuffled array of Card objects; and RankedHand, which takes a set of Cards and returns a corresponding Rank object, representing the rank of that set of cards. Comparing Rank objects allows the winner of the hand to be determined.

Initially, we chose to use Unix named pipes for communication between the server and clients. However, during the implementation stage we were forced to switch to the use of TCP for communication. Details of this change and the reasons for it are found in the implementation section below.

Another consideration when designing the client-server interface was the number of clients we should allow to connect to the server at any one time. We considered allowing multiple clients to connect simultaneously and challenge each other to games. Since this would have complicated the design, and required the server to be multi-threaded, we instead opted for a much simpler solution. The server would only accept connections from two players; once two players had connected, the server would automatically begin a game between them. A potential downside of this solution is that it makes it difficult to allow TeaBot and CoffeeBot to play with humans over the internet, since a new server process must be spawned for every pair of players. However, we felt the speedup in development time outweighed this consideration.

We designed the communications protocol between the clients and the server to be in human-readable format. This was to facilitate debugging. During the implementation stage, we realized that the protocol required a slight redesign to increase the quantity and comprehensiveness of messages sent. This change was needed to provide the clients with more complete information about the current game state.

We also designed a web-based interface for the server. We had to consider both the user interface (UI), and the backend implementation. When designing the UI, we considered the Principles of User Interface Design [12], which includes the requirements that the UI should be structured and simple, and that relevant information should be visible while irrelevant and distracting information should be hidden. Details of the backend implementation for the web interface are included in the implementation section.

2.2. Analyze Poker AI algorithms

We looked at existing poker AI algorithms and decided which appeared most promising for our purposes, and we looked at limitations of those algorithms.

We confirmed out previous decision to make use of the miximax algorithm, as it is a relatively strong algorithm that has considerable room for improvement if a better opponent modeling strategy can be found.

It became clear during our background reading that there would be two areas of difficulty when designing a bot that utilized the miximix algorithm. The first, as mentioned above, would be finding a good opponent modeling function. The second problem, slightly related to the first, would be predicting a player's probability of winning at showdown, when both players' cards are revealed. [5]

2.3. Design Architecture of TeaBot/CoffeeBot

An object-oriented design was also used for the pokerbots, to ensure the separation of responsibilities between different classes. It also meant that TeaBot and CoffeeBot could easily share code; changes would only need to be made to the class that handled opponent modeling, since this was the only difference between the two bots. A description of the pokerbots' architecture follows.

A Client object handles the network interface with the server. Messages received from the server are handed to a Player object, which parses the messages and responds accordingly. Messages from the server fall into two main categories: information on the current game state, and prompts for action.

Information messages are used to update a GameState object, which represents the current state of the world as known to the pokerbot. If the messages are about opponent actions, an OpponentModel object is also updated. (OpponentModel is an interface, with two implementations; TeaBot uses the BasicOpponentModel class, and CoffeeBot uses the ImprovedOpponentModel class).

When a prompt for action is received, the Player object creates three new GameState objects, each representing the potential game state that will result if it bets, checks or folds. To calculate the expected value of each GameState object, the GameState class implements the miximax algorithm. In the process of calculating EV, the OpponentModel object is used to provide predictions of the opponent's actions in a given situation. Once the Player object has determined the optimum action to take, the Client object sends that action to the server.

Additional helper classes include GameInfo, which maintains state that may be shared between multiple GameState objects; ActionList, which provides several convenience methods for manipulating action histories; and CardArray, which represents a set of cards and computes the ranking of those cards.

Two external libraries were incorporated into the design. ImprovedOpponentModel makes use of the Weka Machine Learning library [11], and CardArray makes use of an external library that provides efficient ranking of a set of cards [8]. Details of the use of these external libraries are provided in the implementation section.

2.4. Research and Analyze Machine Learning Algorithms

We analyzed different machine learning algorithms and considered which ones are worth experimenting with. We decided to test the performance of naive Bayes classifiers and neural networks. We have chosen these two algorithms to experiment with as they are fairly dissimilar – for example, naïve Bayes classifiers assume the input features are independent, while neural networks make no such assumption. Further information on the working of these algorithms can be found at [10].

Another consideration was the choice of input features. The amount of potentially relevant information in predicting an opponent's actions is quite large - in machine learning parlance, we say the problem exhibits high dimensionality. Therefore, finding a subset of the data that is adequate for predictive purposes was be required to achieve good performance. A list of the variables used, with justifications for using them, follows:

- The current betting round (that is, preflop, flop, turn, or river). Useful as a player's strategy tends to vary over the course of a hand.
- The current "pot odds" - the ratio of the size of the pot to the cost of a call. Useful as the lower the pot odds, the more likely players are to fold.
- The opponent raise frequency – how often the opponent chose to bet or raise this hand. This usually indicates that they have a strong hand and are likely to raise again.
- Total raise frequency – considers how often both players chose to bet or raise this hand. Useful as a player's actions normally effect how their opponent plays.
- Whether the opponent's last action was a raise – when considered along with the opponent's raise frequency, this can help indicate if the opponent's behavior has suddenly changed.

We also considered using a supervised learning algorithm to learn things other than predictions of an opponent's actions. For example, a learning algorithm could be used to estimate what cards an opponent is holding, or what kind of strategy an opponent is following. However, we decided that this would create extra complexity and was outside the scope of the project.

2.5. Design Rule-based Opponents

To evaluate the performance of our pokerbots, we needed to find some opponents to test them against. There were three main options: playing against humans, playing against existing pokerbots written by others, or creating our own opponents.

Testing performance against human players is of obvious interest, but is not ideal for evaluating the performance of our pokerbots, since humans rarely have the patience to play enough hands to adequately reduce variance. In addition, humans often exhibit inconsistent performance between one game and another.

Playing against existing pokerbots may have been the best solution, in retrospect, since it would have enabled comparisons between strong opponents that exhibited various complex strategies. However, this would have required planning from the beginning of the project to write our pokerbots to interface with existing bots. By the time this requirement was realized, it was too late to change our implementation of the pokerbots and poker server, so we had to create our own simple opponents.

Three opponents were created. The first two, AlwaysCheck and AlwaysRaise, follow a very simple strategy of always checking or always raising. Though they are trivially easy to defeat, the degree to which a player can exploit them provides a useful metric for comparing more advanced players. The third, RulesBot, follows a rule-based strategy, where it acts depending on the strength of its current hand and the current betting round.

3. Implementation

3.1. Poker Server

Since the poker server was required to implement some complicated game logic, but was not expected to be a performance bottleneck, we chose to write it in the Ruby programming language. We chose Ruby for this task, as opposed to a faster language such as Java or C, as we believed that it would allow us to implement the server much more quickly. As a scripting language, Ruby enables rapid development time, at the cost of runtime efficiency. [6]

Memory requirements for the server were fairly low as it only needed to record a small, bounded amount of state. Runtime speed of the server was a more important consideration. If the server was significantly slower than the players, it would be a serious bottleneck to evaluating the player's performance, as it was considered desirable to play as many hands as possible between the players to reduce variance.

One issue that arose during the implementation phase was the difficulty in ranking a set of cards according the rules of poker. As there are 133784560 possible combinations of seven cards, and we must find the best 5 card subset, there are clearly many cases that must be considered. For this section of the project, a somewhat inefficient solution was developed in Ruby.

However, TeaBot and CoffeeBot also need the ability to rank hands, with the difference that they need to use this ability thousands of times per second. The search for a more efficient solution to this problem is detailed below, in the section on the implementation of TeaBot.

Another issue that arose was the need to choose an alternative to Unix named pipes for communication. We discovered that the standard library of Java, the language used for implementing TeaBot and CoffeeBot, does not include an interface for Unix pipes. Therefore, we switched to the use of TCP for communication between the clients and the server, as it is one of the few communication protocols supported by both the Java and Ruby standard libraries. This demonstrated the benefits of following the concept of separation of concerns; as the code for communicating with clients was encapsulated in one class, implementing this change was relatively simple.

When running experiments on our pokerbots, we decided to make a change to the implementation of the Deck class. Previously, the Deck had been randomly shuffled between every hand. However, to reduce variance, it was decided to keep the distribution of cards to one preset random sequence.

This was achieved by generating 10,000 random sets of nine cards (five for the board, plus two for each player), and writing them to a file. The Deck class can then take the option to read the cards sequentially from the file, rather than generating them randomly. This enables two players to play 10,000 hands; after the match is complete, the players can then be reset, reverse positions, and then play the same 10,000 hands with alternated positions. If the first player received a lucky series of cards during the first half of the match, the second player will receive the same lucky set of cards during the second half. The motivation for this change can be found in section 4.3, Experimental Design.

3.2. Web Interface

The backend of the web interface was also implemented in Ruby, and runs as a separate process to the poker server. The rationale for this was to facilitate debugging, and allow one process to fail without affecting the other.

Socket::TCPServer, a class contained in the Ruby standard library, contains basic functionality for managing TCP connections and is used as the basis for our web server. This web server also acts as a client to the poker server, sending and receiving messages as appropriate.

The web server therefore runs as two threads. The first thread handles the interface with the poker server. Messages received from the server are parsed and used to update the web server's internal representation of the current game state. If a prompt for action is received, the first thread waits until the second thread has received input from the player before sending this input on to the poker server.

The second thread handles the web interface. When a browser sends a request, the web server responds with the appropriate HTML, CSS, Javascript and AJAX to allow the browser to render the current game state. If a response is required from the player, the appropriate action buttons are also rendered as HTML forms. Clicking on one of these buttons submits a HTTP request with information about the player's action to the web server. The first thread then sends this action on to the poker server; in this way, the web server acts as an intermediary between the human player and the poker server.

3.3. TeaBot

TeaBot was implemented in Java, due to familiarity issues and its reasonably high performance. Another benefit was that Weka, a popular machine learning library, is also implemented in Java. As a result, experimenting with different machine learning algorithms was straightforward.

Since we had sketched out the miximax algorithm in detail during the design phases, there were relatively few problems converting it to code. As mentioned above, we still had to consider the problems of opponent modeling and estimating win probabilities. As TeaBot was intended to be a basic proof-of-concept for the miximax algorithm, we aimed to find the simplest solutions to these problems.

For estimating win probabilities, we used the same strategy that Vexbot employs. This requires maintaining a histogram of opponent hand strengths from previous rounds with similar betting sequences. [5] From this histogram, we calculate the probability that the player's hand will have a higher rank than the opponent's. If we have no observations for a particular betting sequence, we assume the opponent's hand strength will follow a uniform distribution.

Having a function that can efficiently calculate hand rankings is therefore important, as TeaBot must consider many possible hands before each action. Fortunately, numerous innovative solutions to this problem have been developed [7], and we decided to make use of one of these rather than "reinvent the wheel". We used a hand evaluator library for Java developed by users of the PokerAI.org forums [8].

TeaBot's opponent modeling strategy is very simple; it assumes that its opponents play a random strategy, choosing to bet, call or fold based on some fixed probabilities. For example, if it has observed the opponent fold once, call twice, and bet three times, it will assign a probability of one-sixth to the opponent folding, two-sixths to the opponent calling, and three-sixths to the opponent betting. Although this strategy is far too simplistic to be useful against a good human or AI player, TeaBot was successful when tested against the simple computer players AlwaysCheck and AlwaysRaise. Further details are provided in the evaluation section.

Although TeaBot is not a very strong poker player, it successfully implements the miximax algorithm and demonstrates its effectiveness. It also provided a sound foundation for developing a more effective poker AI, CoffeeBot.

3.4. CoffeeBot

CoffeeBot shared much of the same code as TeaBot; the only change was replacing the BasicOpponentModel class with an ImprovedOpponentModel class.

Altering CoffeeBot's opponent modeling function to make use of machine learning proved to be fairly straightforward, as the Weka library provides a convenient interface for many ML algorithms. A function to convert GameState objects to Weka Instances was written. When the client is notified of an opponent action, it passes this information along with the current GameState to the function, which outputs a corresponding Instance object. This is achieved by computing the input features described in section 2.4 from the GameState, and then setting the class attribute to be the opponent's move made – i.e., a raise, call or fold. The accumulated Instances are then used as training data for a supervised learning classifier.

When a prediction of an opponent's actions is required for a particular GameState, that GameState is first converted to an Instance, and then passed to the classifier. The classifier then outputs a probability triple representing the probabilities that the Instance should be classified as a call, a raise, or a fold – or equivalently, the probabilities that the opponent will call, raise or fold in that situation.

During the initial stages of evaluation, it was noticed that CoffeeBot's game performance consistently lagged behind RulesBot – this was somewhat surprising, as RulesBot followed a much simpler strategy. This necessitated making some changes to CoffeeBot's design and implementation.

Firstly, it was noticed that sometimes CoffeeBot would fold when there was no cost to check – this is always a suboptimum move, since there is no reason to give up a hand when there is no risk in continuing play. This was fixed by disallowing CoffeeBot from folding in such a situation (TeaBot also benefited from this change, since it was implemented in a class shared by the two bots).

Another change was implementing an improved strategy for the preflop stage of the game (preflop refers to the beginning of the hand, when each player can only see two cards). It was decided that using the miximax algorithm at this stage would be too computationally expensive. For this reason, CoffeeBot and TeaBot had originally followed the strategy of always calling preflop. However, this led to behavior that was far too loose – that is, the bots played too many weak hands. This was fixed by allowing CoffeeBot and TeaBot to consider hand strength before deciding how to act during the preflop stage.

A third tweak was made to the means for estimating win probabilities. TeaBot used a ten-interval histogram to store opponent's observed hand strengths. The external library used to compute hand ranks returned an integer between 1 and 7500 to represent the ranking of a set of cards, with lower integers representing higher ranks. During TeaBot's implementation, it was assumed that these ranks followed a uniform distribution, and so each cell in the histogram represented 750 ranks. E.g., cell zero represented ranks 1 to 750; cell one represented ranks 751 to 1500, and so on.

However, it was discovered that the ranks did not follow a uniform distribution, with the result that TeaBot's estimates of win probability were less accurate. CoffeeBot made use of a fifteen-cell histogram that used adjusted boundaries, so that its estimates of win probability were more accurate. In retrospect, this change should have also been implemented in TeaBot, to remove an unnecessary variable in subsequent experiments.

3.5. Rule-based Opponents

AlwaysCheck, AlwaysRaise, and RulesBot were implemented in Ruby. As these bots followed a relatively simple strategy, the implementation was straightforward.

4. Testing

4.1. Poker Server: Time Performance

We tested the server development throughout the implementation phase. For the poker server, the requirements were that it correctly enforced the rules of poker, and that it responded within a reasonable time frame. We used the Ruby programming language's built-in unit testing framework, Test::Unit, to perform automated testing on the server. We also wrote tests for each class to ensure that, even if the internal workings of the class are altered, the interface does not. This follows the software engineering principle of design by contract, where we ensure that the interface between each component is fixed.

The time performance of the server was tested manually – we found that the server could referee 10,000 hands within 1977 seconds (this includes the runtime of the pokerbots), which was reasonable, though we felt there was some room for improvement. We used the Ruby programming language to identify performance bottlenecks. The largest bottleneck was the code used to rank hands, since this involved numerous sorting and comparison operations. By implementing some performance optimizations to this code, we were able to increase time performance by approximately 15%.

4.2. TeaBot and CoffeeBot: Correct Behavior

During the development of TeaBot, the individual components were tested to ensure that each worked correctly under different inputs. In addition, if a component was intended to utilize a particular algorithm, we tested to ensure that the algorithm was correctly implemented. As each component was completed, regression testing was performed to ensure the whole system worked as expected. A similar strategy was followed during the implementation of CoffeeBot.

4.3. Gameplay Performance: Experimental Design

As comparing TeaBot and CoffeeBot's performance at the poker table was a major objective of this project, formulating a good strategy for evaluating that performance was important.

It should be noted that as Texas Holdem is a game with very high variance of scores, it is difficult to reliably ascertain which of two players is superior. Various approaches have been used by other poker AI developers [1]. These include:

- playing a very large number of games between the two players
- duplicate games, where the random seed is recorded, and the match is run twice with players alternating positions (so if one player receives a run of good cards, the other player will receive the same good cards in the second game)
- DIVAT analysis, a technique which compares the players' performance to that of a baseline strategy.

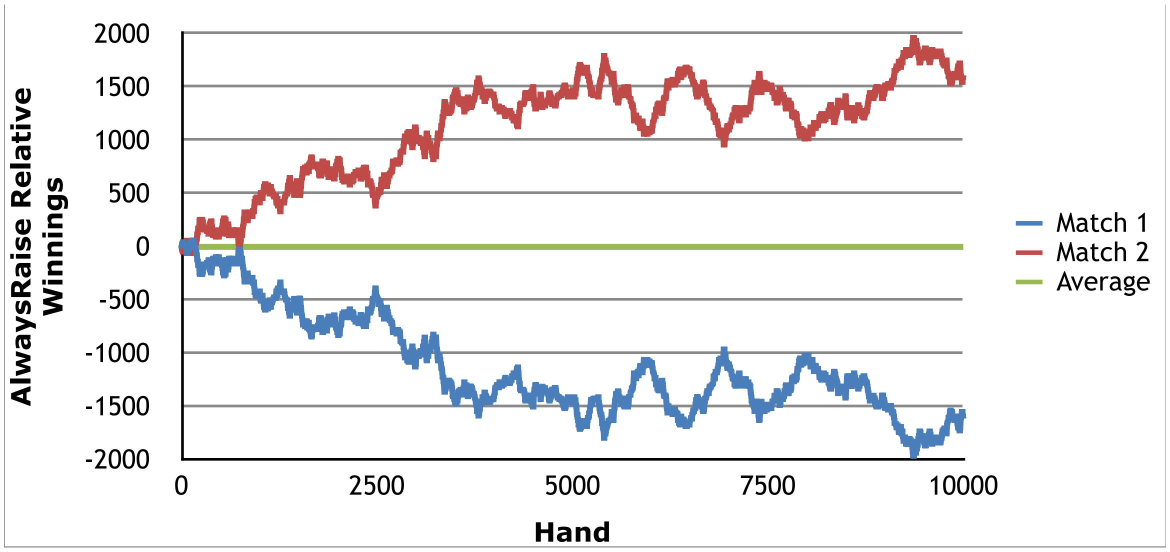
We chose to use a combination of the first and second approaches. To compare the performance of two players, they first play each other for 10,000 hands. Then the players reverse positions, and play a duplicate match for another 10,000 hands, with the cards reversed.

5. Evaluation

Twenty-four matches of 10,000 hands each were played in total. Graphs of the results of those matches follow. Note that there were two matches for each pair of opponents, with the cards revered between matches one and two.

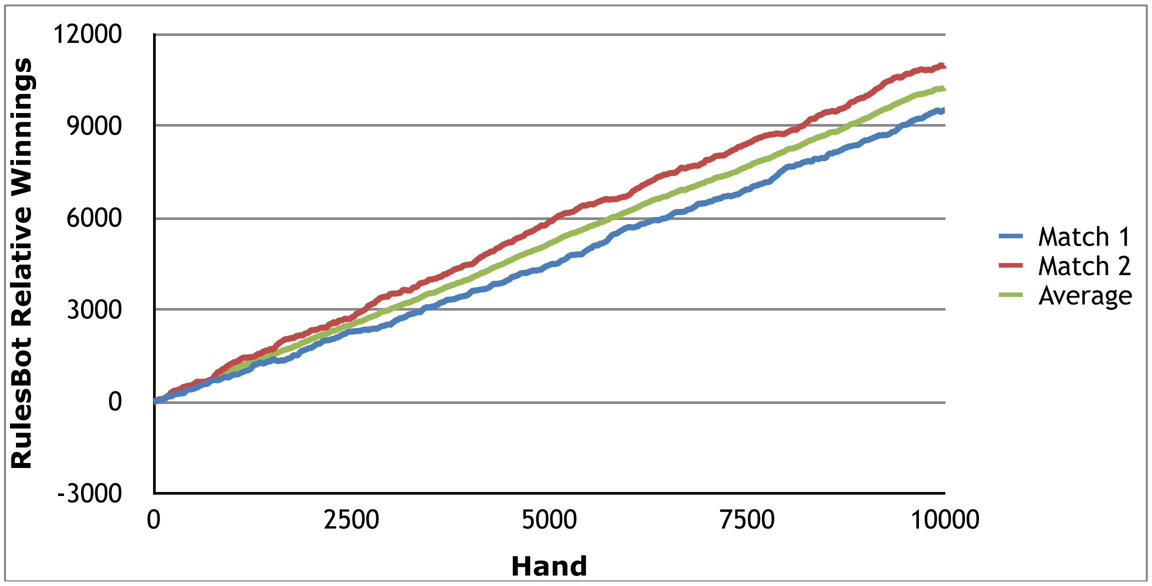
Each graph follows the same format. The x-axis indicates the first player's relative winnings (or, since two-player poker is a zero-sum game, the second player's relative losses). The y-axis indicates the hand number. The red line is the first player's performance in match one; the blue line is the first player's performance in match two (where they cards received by the players are reversed); and the green line is the first player's average performance between the two matches. It should be noted that the bots are reset between matches, so that the bots which adapt their strategies do not have an additional advantage during the second match.

5.1. AlwaysRaise vs. AlwaysCheck



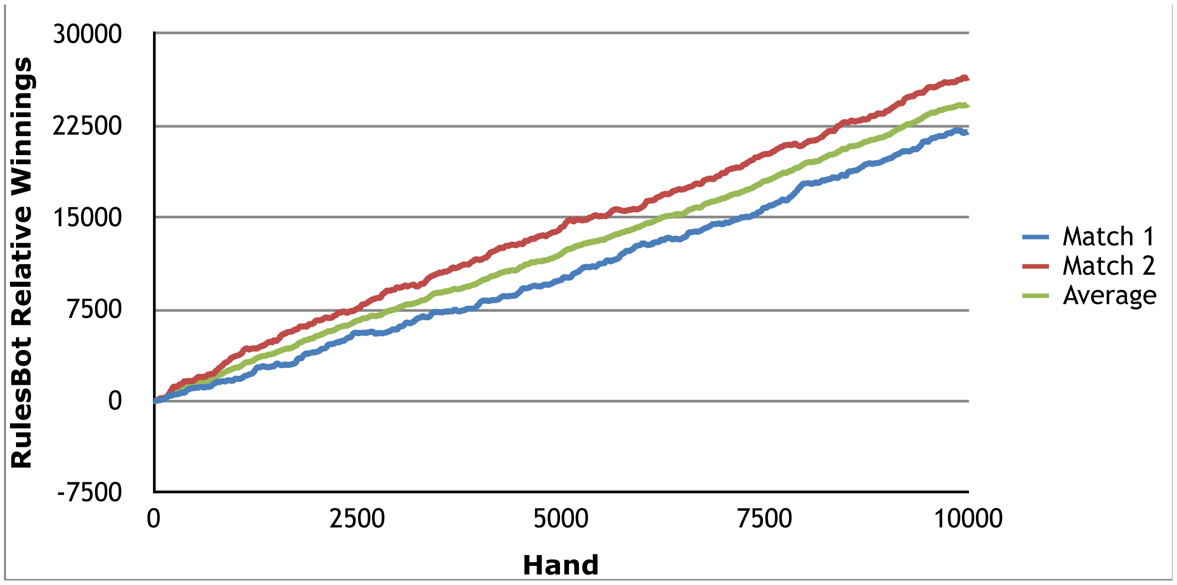
Since neither of these two players ever folds, this match was essentially a random walk. Note that AlwaysRaise's performance in the match one is the inverse of its performance in match two; this illustrates the effect of switching the cards dealt between the two matches. The fact that AlwaysRaise wins 1596 chips in 10,000 hands (~0.16 chips/hand) demonstrates the large role variance plays in poker.

5.2. RulesBot vs. AlwaysCheck



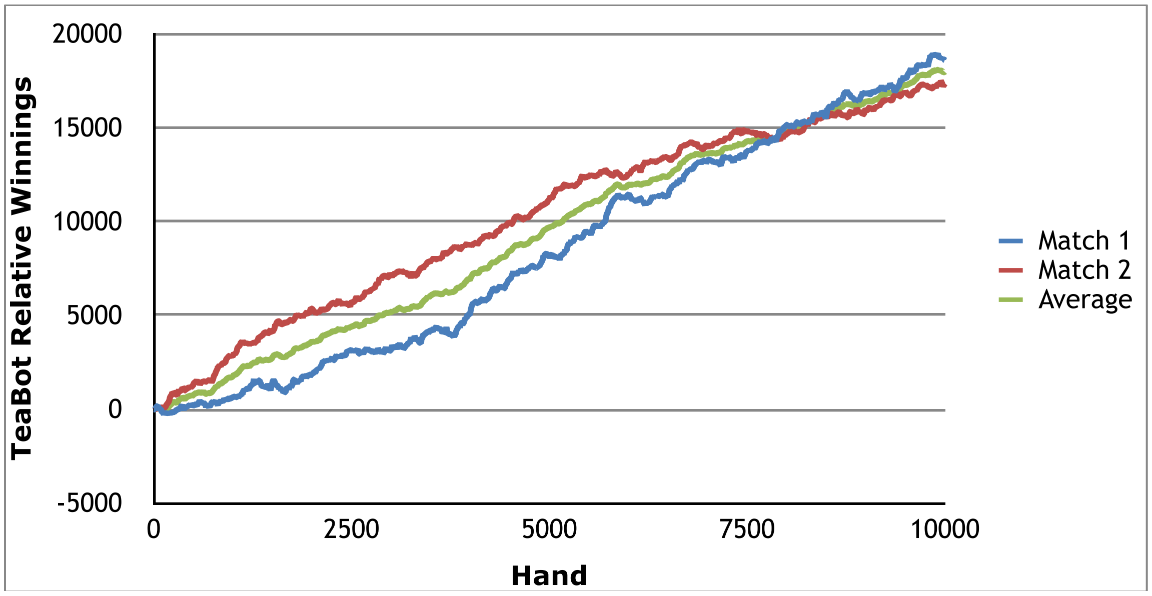
This demonstrates the massive advantage RulesBot has over the simpler bot AlwaysCheck; this difference is almost entirely due to RulesBot's ability to fold weak hands, and only risk money on hands with a strong chance of winning.

5.3. RulesBot vs. AlwaysRaise



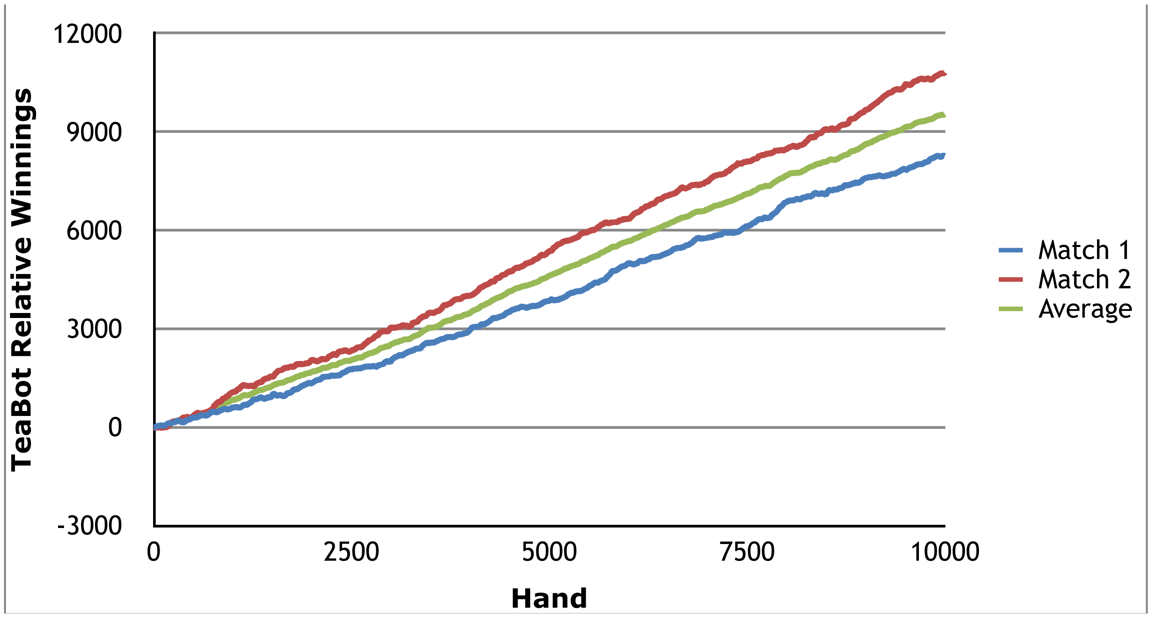
The shape of this graph is similar to the previous one. However, note the change in scale: RulesBot won by a much larger margin here. This is because it is possible to win much larger pots against a bot that consistently raises and increases its bet.

5.4. TeaBot vs. AlwaysRaise



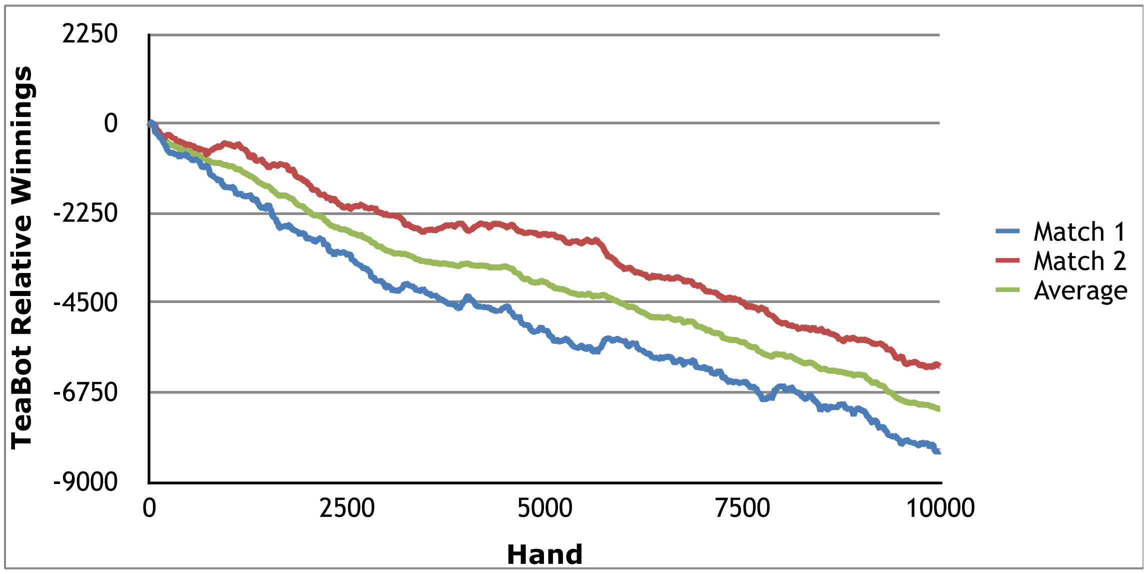
Note that in comparison with RulesBot's match against AlwaysRaise, TeaBot had a much less steady performance. It still won against AlwaysRaise, but not with a large a margin as did RulesBot.

5.5. TeaBot vs. AlwaysCheck



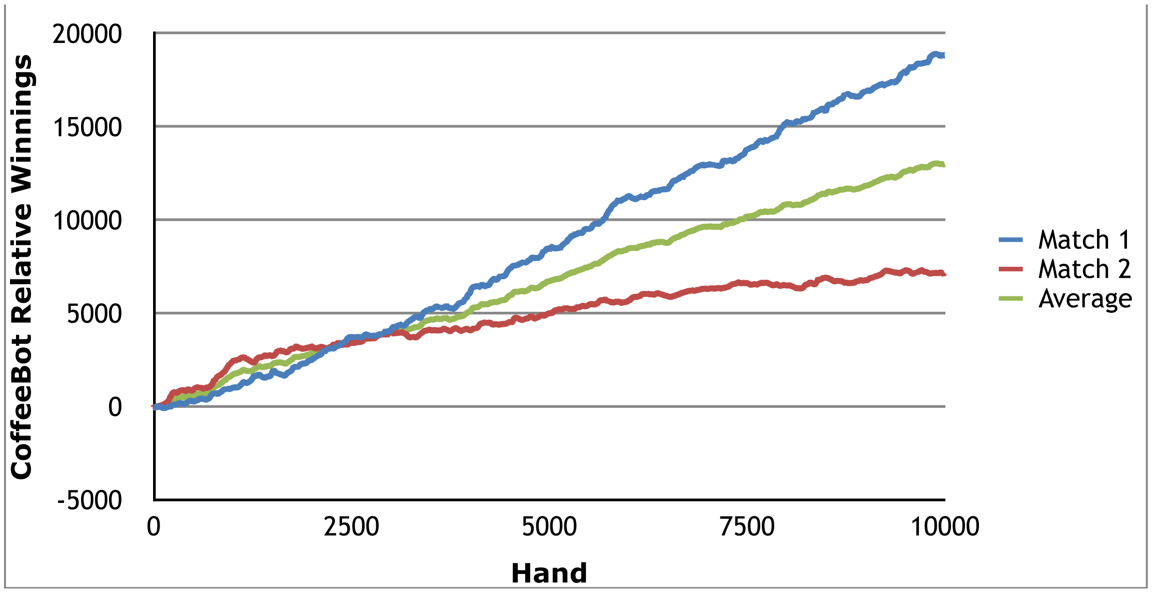
TeaBot's performance is more consistent here, and its relative winnings against AlwaysCheck are comparable to RulesBot.

5.6. TeaBot vs. RulesBot



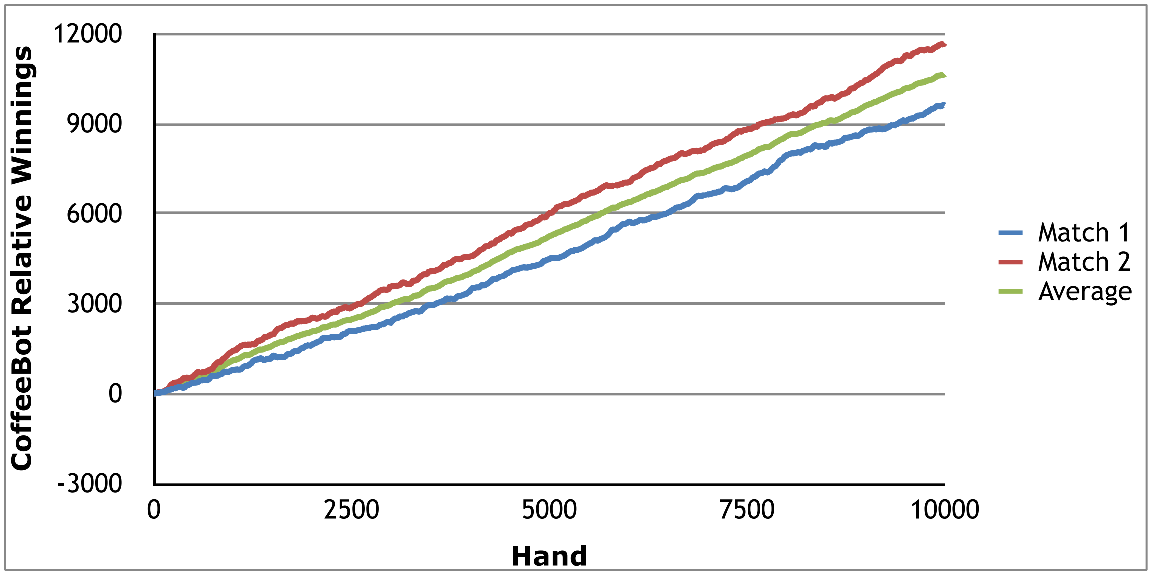
Even though RulesBot's rule-based strategy is rather simple, it clearly outperforms TeaBot's miximax-based strategy. This shows that miximax is ineffective without a strong opponent modeling function.

5.7. CoffeeBot (Naïve Bayes) vs. AlwaysRaise



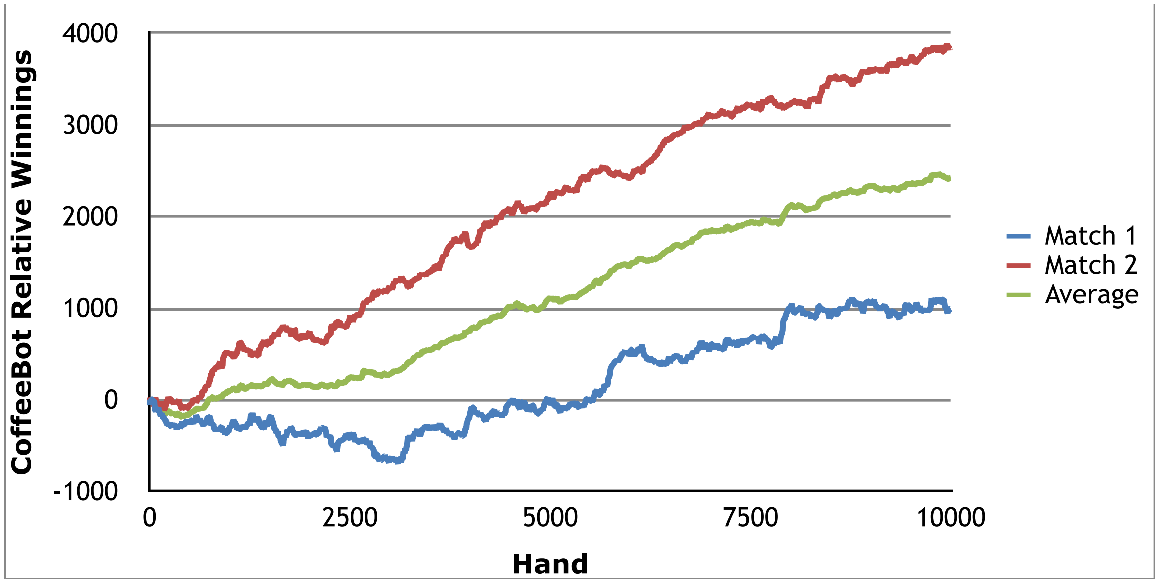
When using a Naïve Bayes classifier for opponent modeling, CoffeeBot's average performance against AlwaysRaise is comparable to that of TeaBot. However, there is a much wider discrepancy between the two matches. This is discussed in the evaluation section below.

5.8. CoffeeBot (Naïve Bayes) vs. AlwaysCheck



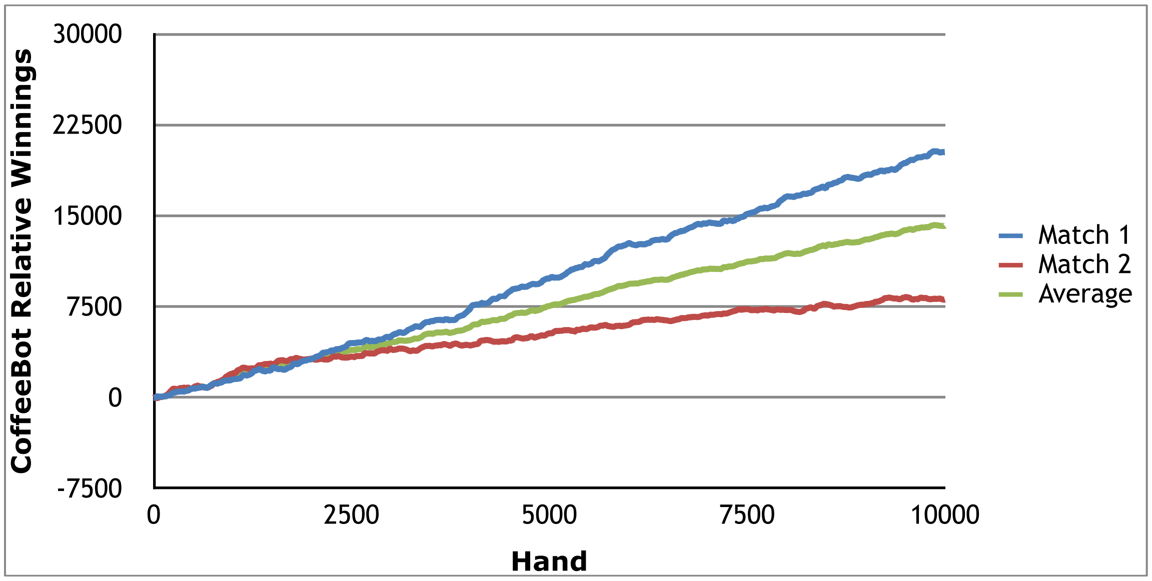
CoffeeBot plays more consistently here, and its performance against AlwaysCheck is comparable to that of TeaBot and RulesBot.

5.9. CoffeeBot (Naïve Bayes) vs. RulesBot



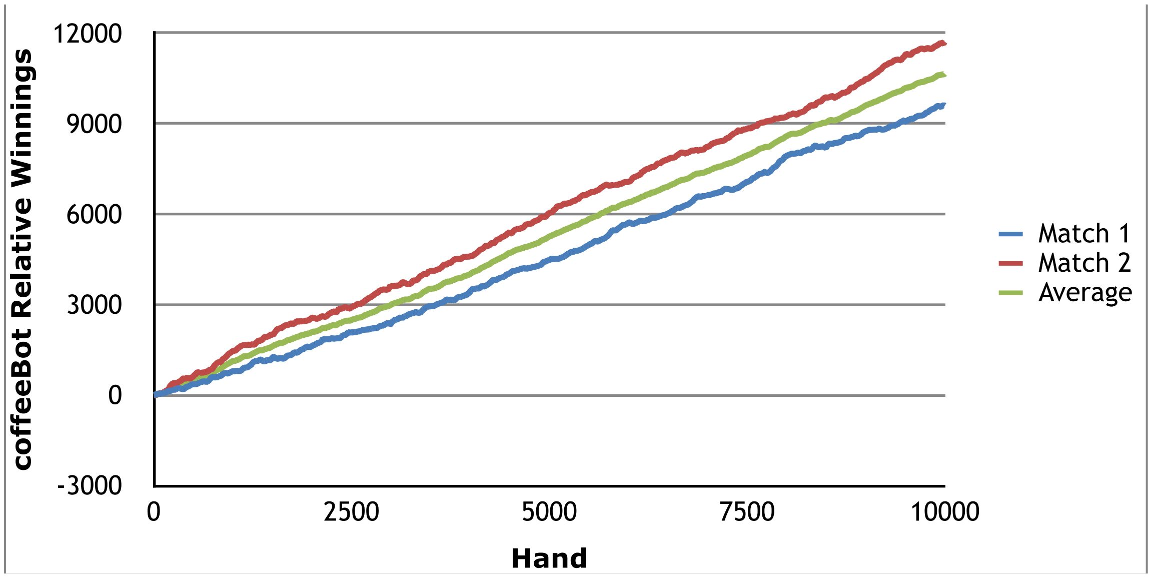
CoffeeBot defeats RulesBot, but its performance is not consistent between the two matches, and it does not win by a large margin.

5.10. CoffeeBot (Neural Network) vs. AlwaysRaise



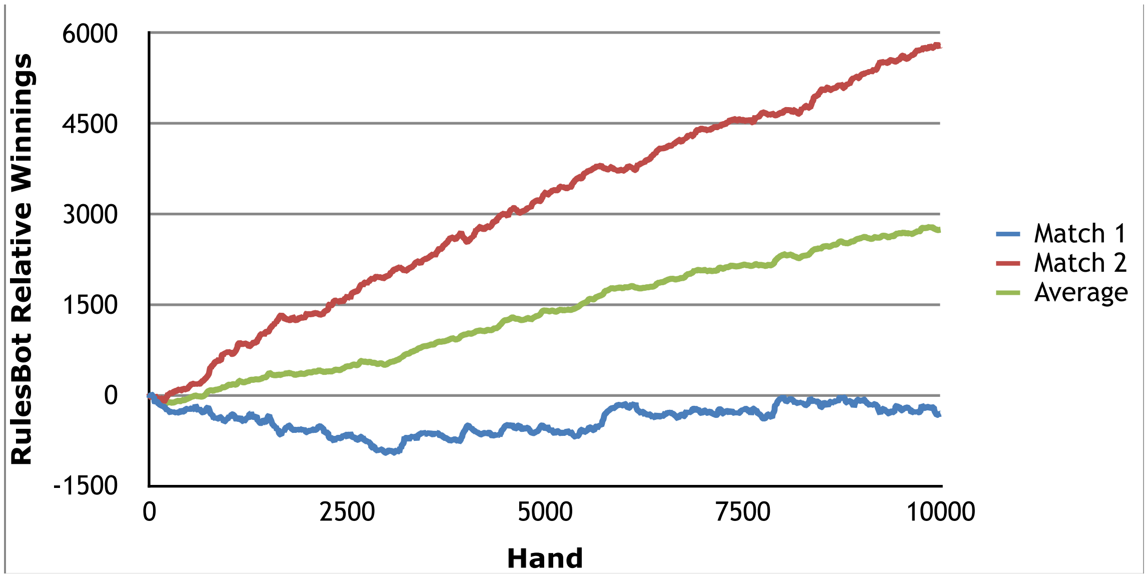
Although the win amounts are not exactly the same, CoffeeBot sees similar performance against AlwaysRaise regardless of whether it uses a neural network or a naïve Bayes classifier for opponent modeling. Note that the performance remains inconsistent between matches one and two.

5.11. CoffeeBot (Neural Network) vs. AlwaysCheck



Performance against AlwaysCheck is more consistent, and again there is little difference between using a naïve Bayes classifier or neural network for opponent modeling. The average win amount differed by only 25 chips (10650.5 chips with naïve Bayes, and 10675.5 chips with a neural network).

5.12. CoffeeBot (Neural Network) vs. RulesBot



Average performance against RulesBot was slightly better when using a neural network than when using a naïve Bayes classifier, but performance was much more inconsistent between the two matches. The first match was actually lost by a small margin; the possible reasons for this are discussed below, in the evaluation section.

5.12.1.Table of results

This table shows a common metric used for the performance of poker players, average small bets won (or lost) per hand. In the above games, a small bet size of two chips was used. All values are rounded to 2 decimal places.

	AlwaysCheck	AlwaysRaise	RulesBot	TeaBot	CoffeeBot(NB)	CoffeeBot(NN)
AlwaysCheck	x	0	-5.13	-4.78	-5.33	-5.33
AlwaysRaise	0	x	-12.13	-8.99	-6.52	-7.09
RulesBot	+5.13	+12.13	x	+3.58	-1.21	-1.38
TeaBot	+4.78	+8.99	-3.58	x	x	x
CoffeeBot(NB)	+5.33	+6.52	+1.21	x	x	x
CoffeeBot(NN)	+5.33	+7.09	+1.38	x	x	x

6. Discussion

During the testing process, we gathered data from 240,000 simulated hands. As this is a rather large amount of data, there are numerous points of entry for beginning an evaluation and deciding whether conclusions can be drawn that meet our stated objectives. To help focus our analysis, we chose six questions for discussion; the first four focus on notable features of the data gathered, and the last two attempt to bring the project full circle by comparing the data gathered with the original goals. The questions are:

- Why did AlwaysCheck lose by similar amounts to all opponents?
- Why did AlwaysRaise lose more to RulesBot and TeaBot than it did to CoffeeBot?
- Why did TeaBot lose so much to RulesBot?
- Why did CoffeeBot show such varying performance between the first and second matches of its games against AlwaysRaise and RulesBot?
- Does the data show that neural networks are a superior opponent modeling strategy to naïve Bayes classifiers?
- Does the data show that the miximax algorithm is superior to a rules-based strategy?

Why did AlwaysCheck lose by similar amounts to all opponents (with the exclusion of AlwaysRaise)?

Since AlwaysCheck follows such a primitive strategy, the optimum strategy against it is also simple (it is not possible to outthink an opponent that does not think). A player must simply be sure to only bet when they believe they have a greater than 50% chance of winning. Since CoffeeBot's winnings against AlwaysCheck were very similar for both opponent modeling approaches it used, this suggests that on both occasions it was playing a strategy that was close to optimum. In fact, CoffeeBot won the same number of hands (7102) with both opponent modeling approaches.

TeaBot won by the same number of hands as CoffeeBot, but with a slightly smaller win amount per hand than CoffeeBot. This is likely due to its inaccurate estimation of win probabilities, as mentioned in section 3.3, which led it to raise less than it should have.

RulesBot's hand strength based strategy also approached the optimum, though as it could not adapt its strategy, its performance lagged slightly behind that of CoffeeBot.

TeaBot's and CoffeeBot's performance remained constant over the course of the match. This suggests that their adaptive abilities were not a large factor in their performance against AlwaysCheck (had the bots adapted to their opponent over the course of the match, their performance would be expected to have improved correspondingly). It should also be noticed that all bots enjoyed greater performance in the second matches of the game, since the "left-side player" always receives a stronger run of cards during the second match – as can be seen most clearly in the match between AlwaysCheck and AlwaysRaise.

Why did AlwaysRaise lose more to RulesBot and TeaBot than it did to CoffeeBot?

The miximax algorithm does not perform as well against AlwaysRaise as it does against AlwaysCheck. RulesBot demonstrates both stronger and more consistent gameplay than CoffeeBot. TeaBot is less consistent, but also outperforms CoffeeBot. CoffeeBot also fares much worse in the second match than in the first, which is surprising, as it received better cards during the second match.

One possible reason for RulesBot's strong performance is that it cannot be "intimidated" into folding by its opponent raising aggressively. Once it has a sufficiently strong hand, it will play it until showdown. TeaBot and CoffeeBot will both normally revise their own expectations of victory downwards in the face of consistent opponent raises, making them more likely to cut their losses and fold.

It is surprising that TeaBot outperforms CoffeeBot here. However, it should be noted that for either opponent modeling strategy, CoffeeBot outperforms TeaBot during match one. It is only CoffeeBot's performance during the second match of either game that drags down its average winnings. Further evaluation of this discrepancy between matches one and two is found in the answer to the fourth question.

Why did TeaBot lose so much to RulesBot?

TeaBot's opponent modeling strategy assumes that its opponent's actions are drawn from a random distribution. This assumption works against AlwaysRaise and AlwaysCheck, since it quickly arrives at an accurate model of these opponents. However, this assumption does not work with RulesBot, and TeaBot loses by a large amount.

TeaBot loses more to RulesBot at showdown than by folding, suggesting it was betting on too many weak cards. This is likely because it overestimates the probability of its opponent folding, which leads to it overestimating the EV of raising. This also demonstrates the ineffectiveness of the miximax algorithm when it lacks a decent opponent modeling function.

Why did CoffeeBot show such varying performance between the first and second matches of its games against AlwaysRaise and RulesBot?

For both opponent modeling approaches, a wide discrepancy in performance was observed during these matches. However, in the games against AlwaysRaise, the second match saw worse performance. Against RulesBot, the worst performance was displayed in the first match of each game.

Analysis of the logs for the game against AlwaysRaise reveals that both CoffeeBot (Naïve Bayes) and CoffeeBot (Neural Network) fold much more frequently during the second match, when compared to the first. This proximate cause was CoffeeBot frequently underestimating the EV of calling or raising. However, the ultimate cause is unclear. It is possible that the AlwaysRaise showed down several strong hands during the early stages of the second match, leading to CoffeeBot consistently overestimating AlwaysRaise's chances of winning. However, this would contradict the observation that CoffeeBot received slightly stronger cards during the second match.

In the game against RulesBot, CoffeeBot wins by a comfortable margin during the second match; however during the first match, CoffeeBot either sees only a slight victory or loses outright. When looking at the logs, the major observable difference is that during the second match RulesBot wins more chips by CoffeeBot folding, but a lot less chips at showdown. CoffeeBot also exhibits the same pattern, although it is not as pronounced.

One possibility is that when a hand-strength based player such as RulesBot encounters an adaptive player such as CoffeeBot, the differences in hand strength between the two matches become more pronounced. During the first match, RulesBot gains a strong run of cards during the early stages, leading it to play aggressively. CoffeeBot then learns that RulesBot normally plays aggressively and frequently wins at showdown, leading CoffeeBot to play more cautiously as a result. However, it is difficult to determine if this is the real reason for CoffeeBot's poor performance during the first match, since the logic behind CoffeeBot's decisions is usually obscure to an outside observer.

Does the data show that neural networks are a superior opponent modeling strategy to naïve Bayes classifiers?

The data is unclear on this question. Against AlwaysCheck, the two ML algorithms gave very similar performance, for reasons outlined above.

Against AlwaysRaise, using neural networks led to slightly superior performance (a difference of around 0.5 small bets per hand). One disadvantage of naïve Bayes classifiers over neural networks is that the former assume independence of input features, meaning they cannot model certain functions. This means that neural networks may generally be expected to generate more accurate opponent models. However, it is not clear why this should be a significant difference when playing against AlwaysRaise.

Against RulesBot, neural networks again showed superior performance. However, they were also less consistent than naïve Bayes classifier, and actually led to CoffeeBot losing the first match. As mentioned in the previous question, the exact reasons for this happening are unclear, but it appears that the same factor impacted the performance of both naïve-Bayes-based and neural-network-based opponent modeling. However, that factor seems to have had a greater negative impact on the neural network classifier. If this factor could be identified and dealt with, one may expect to see comparatively stronger performance from a neural-network-based model as a result. Currently, however, neural networks only show slightly stronger performance than naïve Bayes classifiers at the task of opponent modeling.

Does the data show that the miximax algorithm is superior to a rules-based strategy?

The performance of TeaBot and CoffeeBot when matched against RulesBot suggests that the answer is yes, but only when the miximax algorithm is complemented with a sufficiently powerful opponent modeling function. This shows that we successfully met one of the core objectives of the project, which was to prove the worth of supervised learning for the task of opponent modeling.

However, when looking at the matches against AlwaysRaise, RulesBot exhibits higher relative winnings, which suggests that its rules-based strategy is superior against some opponents. It also suggests that TeaBot and CoffeeBot do not fully exploit such opponents. With further refinements to the miximax algorithm or to the opponent modeling used, it may be possible to reverse this outcome.

7. Conclusions

During our project we successfully implemented the miximax algorithm and demonstrated its effectiveness, along with the value of supervised learning for the task of opponent modeling. However, it is clear that there are areas where even a relatively simple rules-based strategy exhibits stronger performance. This implies that there is still much room for improvement. The following are suggestions for further work in this area:

- **Writing an interface for existing pokerbots, or extending the web server to allow play against humans online.** One major problem faced by the project was that there was a limited field of opponents to play against. Both of these approaches would allow a pokerbot to be tested against a wider range of strategies.

- **Experimenting with different input features to the classifiers.** As mentioned in section 2.4, there is a large amount of information available at any one time. The choice of which variables to use is a delicate balance; too much information, and the classifier will find it difficult to make generalizations and will be prone to overtraining. Too little, and the classifier may miss an important factor in the opponent's decision making.
- **Finding an improved means to calculate win probabilities.** After opponent modeling, this is the second important component of a strong miximax/miximax implementation. A more sophisticated means to compare hand strength is required here; for example, TeaBot and CoffeeBot do not currently consider the possibility that the strength of their hand will improve in subsequent rounds.
- **Experimenting with different machine learning paradigms, such as explanation based learning (EBL).** Explanation-based learning makes use of a domain theory to make generalizations from training examples. Advanced human players are often able to form accurate models of their opponents from very limited data, something that computer players find difficult. EBL may allow computer players to bridge that gap.

We have learned much throughout the project. As well as gaining a greater knowledge of technical areas such as poker AI and machine learning, we also learned some useful skills for managing large projects. Several times, we found that a great deal of work could have been saved with thorough research beforehand – a case of "several months in the lab can save whole hours in the library". We learned the importance of documenting the iterative process of design, implementation and testing. Perhaps most importantly, when attempting to improve the performance of our pokerbot, we found that thinking and analysis can often be more useful than mindless coding, and that the most important part of computer science is often the work done away from the keyboard.

8. References

[1] M. B. Johanson. (2007) *Robust Strategies and Counter-Strategies: Building a Champion Level Computer Poker Player*, M.S thesis, Dept. CS, U. o. Alberta, Edmonton. [Online]. Available: http://www.cs.ualberta.ca/~games/poker/publications/johanson.msc.pdf

[2] T. C. Schauenberg. (2006). *Opponent Modelling and Search in Poker*, M.S thesis, Dept. CS, U. o. Alberta, Edmonton. [Online]. Available: http://webdocs.cs.ualberta.ca/~darse/Papers/schauenberg.msc.pdf

[3] D. Kushner. (2005 Sept.) *On the Internet, Nobody Knows You're A Bot*, *Wired Magazine,* Issue 13.09. [Online]. Available: http://www.wired.com/wired/archive/13.09/pokerbots.html

[4] R. Caruana, A. Niculescu-Mizil. (2006). *An Empirical Comparison of Supervised Learning Algorithms*, Dept. CS, Cornell U., Ithaca. [Online]. Available: http://www.cs.cornell.edu/~caruana/ctp/ct.papers/caruana.icml06.pdf

[5] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaefer, and D. Szafron. (2004). *Game Tree Search with Adaptation in Stochastic Imperfect Information Games*, Dept. CS, U. o. Alberta, Edmonton. [Online]. Available: http://webdocs.cs.ualberta.ca/~darse/Papers/CG04-iigts.pdf

[6] Ruby community members. (2011). *About Ruby*, [Online]. Available: http://www.ruby-lang.org/en/about/

[7] Coding the Wheel. (2008, Sept. 5). *The Great Poker Hand Evaluator Roundup.* [Online]. Available: http://www.codingthewheel.com/archives/poker-hand-evaluator-roundup

[8] pokerai.org. (2011). *Spears adaptation of RayW LUT hand evaluator*. [Online]. Available ZIP archive: http://pokerai.org/pj2/code/eval.zip (within folder eval.zip/pokerai/game/eval/spears)

[9] pokerlistings.com (2010). *Texas Holdem Rules and Game Play*. [Online]. Available: http://www.pokerlistings.com/poker-rules-texas-holdem

[10] S. J. Russell, P. Norvig. (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall. pp. 163–171, 727, 808.

[11] University of Waikito. (2011). *Weka.* [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/

[12] L. Constantine, L. Lockwood. (2011). *Principles of User Interface Design.* [Online]. Available: http://www.foruse.com

Appendix A: Minutes

1. Minutes of the 1st Project Meeting

Date: August 27th, 2010

Time: 10:00 AM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

1.1. **Approval of minutes**

This was the first formal group meeting, so there were no minutes to approve.

1.2. **Report on progress**

Isaac read a number of papers on poker AI, and did some background reading on machine learning.

1.3. Discussion items

1.3.1. We discussed the development of the project

- Professor Mak recommended that for projects like this, a good approach is to first build a simple system that works, then attempt to find improvements.

- In this context, this means implementing an existing poker algorithm, then attempting to improve it through the use of machine learning techniques

1.3.2. We discussed the organization of the project

- What resources are available to students
- Need to set a time for fortnightly meetings

1.4. **Goals until next meeting**

4.1 Isaac will write the project proposal for September 21st

1.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 10:30 AM.

The date and time of the next meeting will be set later by e-mail.

2. Minutes of the 2nd Project Meeting

Date: 20th September, 2010

Time: 3:00 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

2.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

2.2. **Report on progress**

1. Isaac had almost completed a first draft of the project proposal, but wanted some feedback

2.3. Discussion items

2.3.1. Some questions of whether the writing style was appropriate
- e.g., whether use of words like 'likely', 'probably', 'maybe' is acceptable

2.3.2. Definitions should be provided for terms specific to poker, but not necessary for terms specific to machine learning.

2.3.3. Prof. Mak thought that the time allocated for attempting to improve the poker AI was too short, and therefore should aim to complete the earlier stages of the project sooner

2.3.4. Agreed on time for project meeting at 1:30 PM on alternate Fridays, beginning October 8th.

2.4. Goals until next meeting

2.4.1. Isaac will submit the completed proposal on the 21st September.

2.4.2. Isaac will begin work on the poker server.

2.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 3:30 PM.

The next meeting will be at 1:30 PM, October 8th.

3. Minutes of the 3rd Project Meeting

Date: 8th October, 2010

Time: 1:30 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

3.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

3.2. **Report on progress**

Isaac has begun work on the poker server.

3.3. Discussion items

There was little to discuss. Isaac said that progress on the server was continuing as expected. Professor Mak reminded him that midterms were approaching and that he might want to cease project work during that time.

3.4. **Goals until next meeting**

.As mentioned above, Isaac will cease work on the project during midterms, but will aim to implement the game logic component of the poker server before the next meeting.

3.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 1:40 PM.

The next meeting will be at 1:30 PM, November 5th

4. Minutes of the 4th Project Meeting

Date: 5th November, 2010

Time: 1:30 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

4.1. Approval of minutes

The minutes of the last meeting were approved without amendment.

4.2. Report on progress

Isaac has completed the game logic of the poker server, and now needs to implement a network (or other) interface.

4.3. Discussion items

4.3.1. Professor Mak recommended creating a graphical interface, to allow human players to compete with the AI and to produce a more impressive demo.

4.3.2. Prof Mak asked how it is possible to determine a strategy has improved, due to the random nature of poker – ie, one strategy may be superior to another, but get 'unlucky' during testing. Isaac said various papers talked about the same problem, and various approaches had been developed. One approach was simply to play a large number of games, in order to reduce variance – though other approaches exist.

4.3.3. Isaac said he was using Ruby to develop the server as it was a terse language that enabled rapid development time. He was concerned it might make the server too slow, but that turned out not to be an issue.

4.4. Goals until next meeting

Isaac will either begin work on a graphical interface or on the poker AI.

4.5. Meeting adjournment and next meeting

The meeting was adjourned at 1:50 PM.

The next meeting will be at 1:30 PM, December 3rd.

5. Minutes of the 5th Project Meeting

Date: 3rd December, 2010

Time: 1:30 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

5.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

5.2. **Report on progress**

Isaac demonstrated a web-based interface for the poker server.

5.3. Discussion items

5.3.1. Professor Mak suggested adding a debug mode to the interface or otherwise making it clearer what was happening, as the interface was somewhat confusing.

5.3.2. Isaac has been discussing ideas for his project on internet forums, does he need to cite this as a reference? Professor Mak said that this was not necessary.

5.3.3. As both Isaac and Professor Mak will have different schedules in the Spring semester, they need to organize another meeting after Winter break.

5.3.4. The progress report is due on the 11th February 2011.

5.3.5. The workings of the algorithm were discussed. Professor Mak asked where the training data will come from. Isaac said that it is possible to obtain training data from real games. It's also possible to simulate games, ie by having the AI play against itself.

5.4. **Goals until next meeting**

Isaac will work on the Poker AI over the winter break, and submit the progress report at the beginning of Spring semester.

5.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 1:50 PM.

The next meeting will be organized after the winter break

6. Minutes of the 6th Project Meeting

Date: 3rd March, 2011

Time: 1:30 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

6.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

6.2. **Report on progress**

Isaac demonstrated a improvments to the web-based interface.

6.3. Discussion items

6.3.1. Professor Mak asked whether fortnightly meetings would be preferable, as it would give Isaac more time to make progress. Isaac said that weekly meetings gave him more motivation to work harder.

6.3.2. Professor Mak had graded the progress report and said that in the final report he should elaborate the rules of poker and the working of the Miximix algorithm.

6.3.3. Since time was limited, Isaac should decide which AI techniques to experiment with.

6.3.4. Isaac should consider how to test that the algorithms used had been correctly implemented.

6.4. **Goals until next meeting**

Isaac will begin work on the AI.

6.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 3:15 PM.

The next meeting will be held on March 10th. (Due to Brian Mak being away on business, the meeting was postponed until March 17th).

7. Minutes of the 7th Project Meeting

Date: 17th March, 2011

Time: 3:00 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

7.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

7.2. **Report on progress**

Isaac had begun incorporating the Weka machine learning library into the AI.

7.3. Discussion items

Professor Mak said that as the end of the project was approaching, Isaac should aim to experiment with at least one of the ML algorithms as soon as possible.

7.4. **Goals until next meeting**

Isaac will implement at least one of the Weka ML algorithms into the AI.

7.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 3:15 PM.

The next meeting will be held on March 24th.

8. Minutes of the 8th Project Meeting

Date: 24th March, 2011

Time: 3:00 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

8.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

8.2. **Report on progress**

Isaac had successfully incorporated a Naïve Bayes classifier into the AI, and created a simple rules-based bot to test against.

8.3. **Discussion items**

8.3.1. Isaac said that a good experimentation strategy would be the next thing to work on. Once this was in place, incorporating different ML algorithms would not be a problem (as they use the same interface in Weka).

8.3.2. Professor Mak raised the issue of training from limited data. Isaac said that the AI would initially have poor performance until it had enough observations of an opponent to make accurate predictions. One target for experimenting could be to discover which ML algorithms learn the fastest.

8.3.3. Professor Mak asked which parameters would be used for the input. For example, if all the cards on the table were used as input parameters, the number of potential values would be too large and it would be difficult to gain enough data to adequately train. The choice of which parameters to use was discussed.

8.3.4. Isaac asked about entering the FYP for a conference on machine learning and data mining that was looking for undergraduate papers. Professor Mak said that he could try but the project may not be original enough.

8.4. **Goals until next meeting**

Isaac will try incorporating a Neural Network classifier into the AI, and think about which parameters to use as input and an experimentation strategy to test the AI's performance.

8.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 3:15 PM.

The next meeting will be held on March 31st.

9. **Minutes of the 9th Project Meeting**

Date: 31st March, 2011

Time: 3:00 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

9.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

9.2. **Report on progress**

As both parties were constrained for time, Isaac merely confirmed that the project was on track.

9.3. **Meeting adjournment and next meeting**

The meeting was adjourned at 3:05 PM.

The next meeting will be held on April 7th.

10. Minutes of the 8th Project Meeting

Date: April 7th, 2011

Time: 3:00 PM

Place: Room 3513

Present: Isaac Lewis, Brian Mak

Absent: None

10.1. **Approval of minutes**

The minutes of the last meeting were approved without amendment.

10.2. **Report on progress**

Isaac had mostly completed the project, he just needs to write the report.

10.3. **Discussion items**

10.3.1.Isaac asked some questions about the report format.

10.3.2.The format of the project presentation was discussed.

10.3.3.As the project is nearly completed, no more meeting will be necessary.

10.4.**Goals until next meeting**

Not applicable.

10.5. **Meeting adjournment and next meeting**

The meeting was adjourned at 3:15 PM.

There will be no further meetings.

Appendix B: Required Hardware & Software

Hardware

- Computer, for developing the server and AI
    - Dell Inspiron 1720 Laptop running Ubuntu/Windows Vista
    - 2GB RAM
    - Dual core processor

Software

- Ruby 1.8, for writing the poker server
- Java SE 6, for writing the poker AI
- Weka [11], for incorporating machine learning
- Spears adaptation of RayW LUT hand evaluator [8], for ranking hands
- Git, for version control

Appendix C: Rules of Texas Holdem Poker

Limit Texas Holdem Poker is a card game played between two and ten players, where the objective is to win as much money as possible from the other players. This project was focused on the two-player version, known as Heads-Up Poker.

A game of poker usually consists of many individual rounds, called *hands*. At the start of a hand, each player is dealt two hidden cards, called *hole cards*. During the course of the hand, five more cards are dealt to the board. At the end of the hand, the players reveal their hidden cards. Whoever can make the best five-card combination from their personal hole cards and the cards on the board wins the hand.

This would not make for an especially interesting game were it not for the importance of betting. At several stages during the game, players make bets to indicate their confidence in winning the hand; the player who eventually reveals the best set of cards at the end of a hand claims all the money bet during that hand. Normally, betting tokens called *chips* are used in lieu of actual money.

If a player believes she has a strong set of cards and therefore chance of winning, she may increase the size of her bet (known as *raising*), in order to increase her total winnings at the end of the hand. Her opponents must choose between matching this bet (*calling*), increasing the bet size even further (*re-raising*), or giving up the hand (*folding*). If all other players fold, the one player remaining is the winner by default. This enables the strategy of *bluffing*.

Bluffing is where a player with a weak set of cards makes large raises, hoping that her opponents will believe she has a strong set of cards and be intimidated into folding, leaving her the winner. A related strategy is *trapping* or *slowplaying*, where a player with a very strong hand will make little to no bets, giving the impression that her hand is weak and coaxing her opponents into making larger bets – bets that she hopes to win at the end of the hand.

A normal hand of poker proceeds as follows:

- Players place starting bets, called blinds.
- Each player is dealt two hidden cards.
- There is a round of betting.
- Three cards are dealt to the board (the flop).
- There is another round of betting.
- Another card is dealt to the board (the turn).
- There is a third round of betting.
- One more card is dealt to the board (the river).
- There is a final round of betting.

- Players reveal their hidden cards, and the player with the best set of cards wins.

The above paragraphs are only intended as an outline of the rules of poker; a more complete description may be found at [9]. Nevertheless, they illustrate some of the most salient features of the game from a game theoretical perspective. The random dealing of cards adds uncertainty to the game; a weak set of cards may become stronger as more cards are dealt to the board. The exact cards held by an opponent are unknown, and this makes the game one of imperfect information.

The fact that there is a relatively small amount of hidden information (just two hidden cards) makes it possible for observant players to make fairly accurate guesses about the cards an opponent is concealing. However, as soon as a player believes they have a good 'read' on an opponent, the opponent is able to alter their strategy to mislead and deceive the player. This dynamic, with players trying to mislead, outwit, guess and second-guess one another, is prominent in Texas Holdem, and leads to a game of great strategic depth. It is one of the reasons why poker champion Doyle Brunson declared "Texas Holdem is the Cadillac of poker games".