Project Name:
Texas Hold'em Poker Bot (with learned opponent models)
Team Name:
Team McGinnis (Zack McGinnis) – Fall 2015 - CS441

Introduction

Project Overview**:**
This project was meant to explore the effectiveness of using different artificial intelligence strategies in the game of No-Limit Texas Hold'em Poker. The strategies which I will be implementing and testing include:

i. A modification to the MCTS (Monte Carlo Tree Search) algorithm which implements a UCT (Upper Confidence bound applied to Trees) weighted to consider the SPR (Stack-to-Pot Ratio) of our bot as it decides which action to take at each stage of the game.
ii. Implementations of the NaiveBayes classifier to build opponent models based on learned hand histories of our opponent.

Description of Texas Hold'em Poker:
Texas Hold'em Poker is a game of incomplete information. This makes it especially difficult when attempting to design an intelligent poker bot capable of adjusting to stochastic elements within the game (unseen cards, unpredictable actions, bluffs, etc...). To account for this, we need to introduce features which assists in the bot's ability to recognize tendencies and probabilities based on previous actions taken by our opponent.

In my implementation, my bot will be playing against only one opponent. The format of the game will be no-limit, meaning that there is no maximum on the amount that a player can bet or raise (the maximum bet/raise is the total amount of the player's current stack size). This is in comparison to a fixed limit format, where bets and raise sizes at each stage are pre-determined and cannot be manipulated by the player.

Each player is dealt two hole cards (face down and unknown to your opponents). Then there is a round of betting. In a round of betting, each player must choose to perform one of the following possible actions: they can fold, and drop out of the game; they can call, and place into the pot the current outstanding bet required to continue playing (if applicable); or they can raise, and increase the current outstanding bet by at least the size of the current outstanding bet. Note that calling when the current maximum bet is zero is usually called checking. The betting round continues until all players have either folded or matched the current maximum bet.

Description of the OpenTestBed package:
OpenTestBed is an open-source Java project which allows users to create, modify, and test various poker bots and learning strategies. Included in OpenTestBed is the WEKA (Waikato Environment for Knowledge Analysis) software library. WEKA provides a collection of machine learning algorithms written in Java.

Description of Monte Carlo Search Tree (MCTS) algorithm:
Monte-Carlo Tree Search (MCTS) is a best-first search technique that estimates game tree node values based on the results from simulated gameplay. Its main goal is to replace exhaustive search through the game tree with well founded sampling methods. The most popular heuristic for the best-first selection in MCTS is UCT (upper confidence bound applied to trees).

The MCTS begins with only the root of the tree and repeats the following 4 steps until it runs out of allotted "thinking time":

Selection: Starting from the root, the algorithm selects in each stored node the branch it wants to explore further until it reaches a stored leaf. Note that my UCT implementation will weigh the selection process towards branches which produce lower stack-to-pot ratios in each game. The selection strategy is a parameter of the MCTS approach.
Expansion: One (or more) leafs are added to the stored tree as children of the leaf reached in the previous step.
Simulation: A sample game starting from the added leaf is played (using a simple and fast game-playing strategy) until conclusion. The value of the reached result at every reached game tree leaf is recorded.
Backpropagation: The estimates of the expected values $V*(P) = E[r(P)]$ (and selection counter $T(P)$) of each recorded node P on the explored path is updated according to the recorded result. The backpropagation strategy is also a parameter of the MCTS approach. Typically, the child with the highest expected value or highest visit count is chosen.

Description of the MCTS Bot (initial configuration):
In the OpenTestBed project, a MCTS bot is included along with 3 other bots. My modifications and implementations will apply strictly to this MCTS bot. Though I will not be modifying the creation of the game tree, it is important to understand the different nodes which represent it:

Decision nodes: These nodes represent states where the bot is in control of the game and must choose an action. Usually, the algorithm runs simulations which choose the child with the highest expected value.
Opponent nodes: These nodes represent states where the opponent is in control of the game and must choose an action. Usually, the algorithm run simulations which randomly choose a child to continue with (since we do not know the probability distribution of the opponents actions). By implementing an opponent model from learned hand histories, we will be able to run simulations which can predict the probability distributions of different actions which our opponent could make.
Leaf nodes: These nodes are created during the expansion phase to continue simulations after selection.
Showdown nodes: These nodes exend the leaf node class and represent the end of one simulated game. The count and value of these nodes are propogated back through the path taken.

Description of the test bot (our opponent):
The OpenTestBed project contains another bot, SimpleBot, which I will use as the opponent for my bot. The SimpleBot decides which action to take by using a rule-based system which only considers it's own two hole cards, the visible community cards, and the estimated expected value of each possible action. It does not utilize any opponent modeling or learning.

Description of Upper-Confidence bounds applied to Trees (UCT):
UCT essentially acts as a heuristic for the MCTS algorithm. In effect, a standard UCT implementation will limit the size of the game-tree which the MCTS algorithm has to search. This is useful in decreasing the computational requirements of the algorithm. In effect, we are limiting unneeded exploration moves, and maximizing exploitation moves.
After sampling all options once, it selects the option which maximizes:

$$\hat{V}(c_i) + C\sqrt{\frac{\ln \mathrm{T}(P)}{\mathrm{T}(c_i)}}$$

Where $T(P)$ = total number of samples made for node P, $V(c_i)$ = the expected value of the sample, $T(c_i)$ = number of all samples, and C = some constant which can be tuned to alter the exploration-exploitation tradeoff.

Description of my UCT implementation weighted for SPR:
The stack-to-pot ratio (SPR) is statistic which measures the amount of money in the pot at any given time relative to the amount of money in a player's stack. For example, if the pot contains $5.00 and we have a stack of $100.00, our SPR = 20. Ideally, we want our SPR to be low when we have a good probablity of winning the hand. Bluffing should also be discouraged in low SPR situations, since the probability that your opponent will fold is going to be reduced due to the large pot size. By choosing moves which result in a lower SPR, we will explore branches which contain more aggressive actions (betting and raising to grow the size of the pot). In my weighted UCT implementation, I will reflect our desire to explore lower SPR branches by using the following formula:

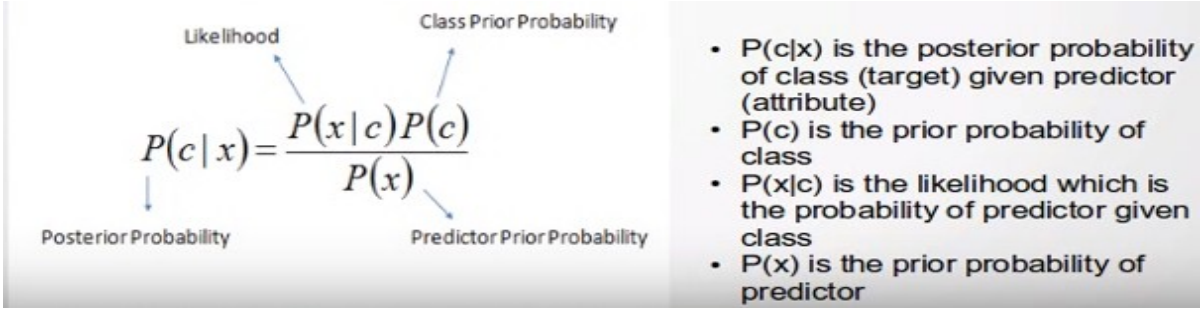
$$\hat{V}(c_i) + C\sqrt{\frac{\ln \mathrm{T}(P)}{\mathrm{T}(c_i)}} \quad \text{x (S)} \qquad \text{Where } S = 1/SPR$$

This formula will weigh our selection strategy toward game-tree branches which produce results made with lower SPR values throughout each individual game. The implication is that our bot will choose actions which result in lower SPR sizes (betting and raising rather than checking and calling).

Description of the Opponent Model:
In a regular MCTS algorithm without an opponent model, opponent nodes are often unable to be predicted accurately during sampling, leaving values to be selected in a random fashion. By implementing a learned opponent model, we have the ability to give estimated probabilites of each opponent action at each stage of an individual game simulation.

The opponent model I will be utilizing was trained on a set of 10,000 games played between two instances of SimpleBot (our opponent bot). The following data (gathered from the aforementioned 10,000 games in the forms of imported hand histories) was used to construct models with the NaiveBayes classifier: 820 instances of post-flop check or bet actions; 2363 instances of post-flop fold, call, or raise actions; 10 instances of pre-flop check or bet actions; 6973 instances of pre-flop fold, call, or raise actions.

Description of the NaiveBayes classifier:
The Naive Bayes classifier calculates the chance that a new case belongs to a certain category, based on the instances which we have already classified. This is known as prior probability. The more instances there are in one class, the more likely it is that a new case belongs to that class. With our gathered training data, we can gain insight on different attributes for each nominal action taken by our opponent (check, bet, call, raise, or fold). The NaïveBayes model will suggest an updated probability of opponent actions back to the MCTS algorithm as we encounter opponent nodes through each iteration of the game tree simulation.



This classifier operates in a supervised setting, such that all attributes and probabilities are classified to the nominal actions of fold, check, bet, call, or raise. For each of the pre and post flop models constructed with the NaiveBayes classifier, I used a 10-fold cross-validation. This means that the data was split into ten equal parts for the training, allowing the classifier to train separately on each.

Experimentation

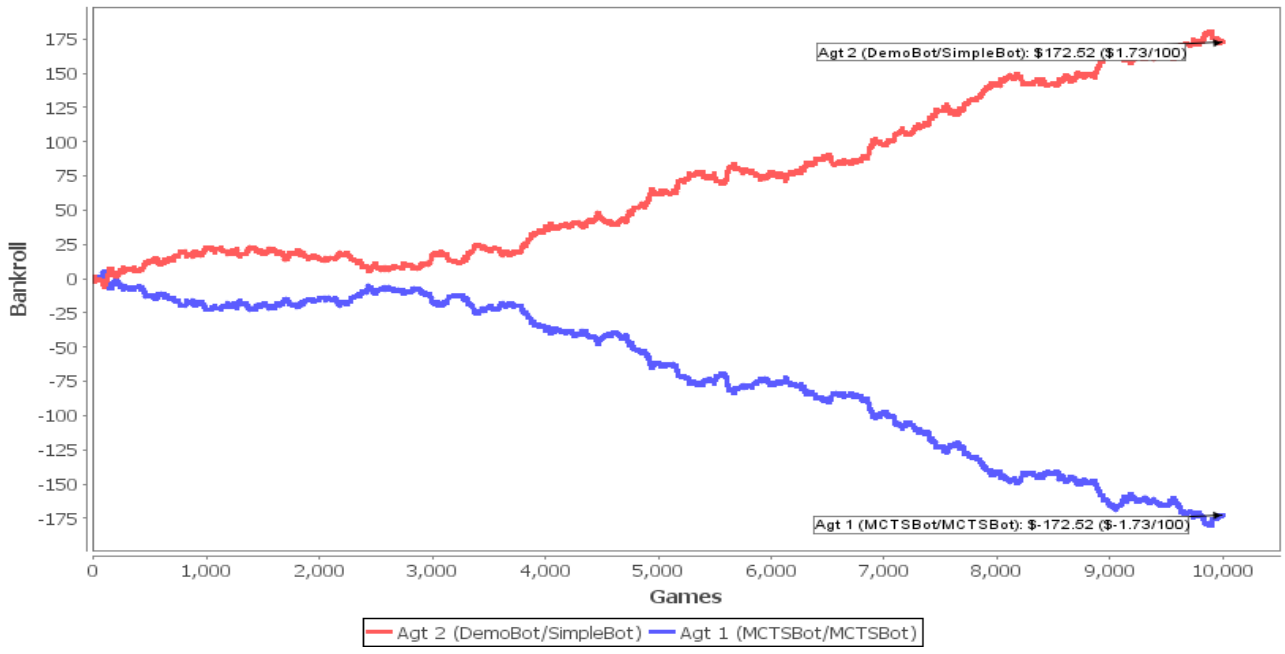
I ran four separate tests to determine which combination of features would produce the most lucrative results for my poker bot. Some features which remained constant during each of the four tests were:
Opponent: SimpleBot
Games: 10,000
Size of one bankroll: $1.00
Thinking time of each MCTS iteration: 900ms
Value of C within UCT implementation = 10

The four separate tests were labeled as follows:

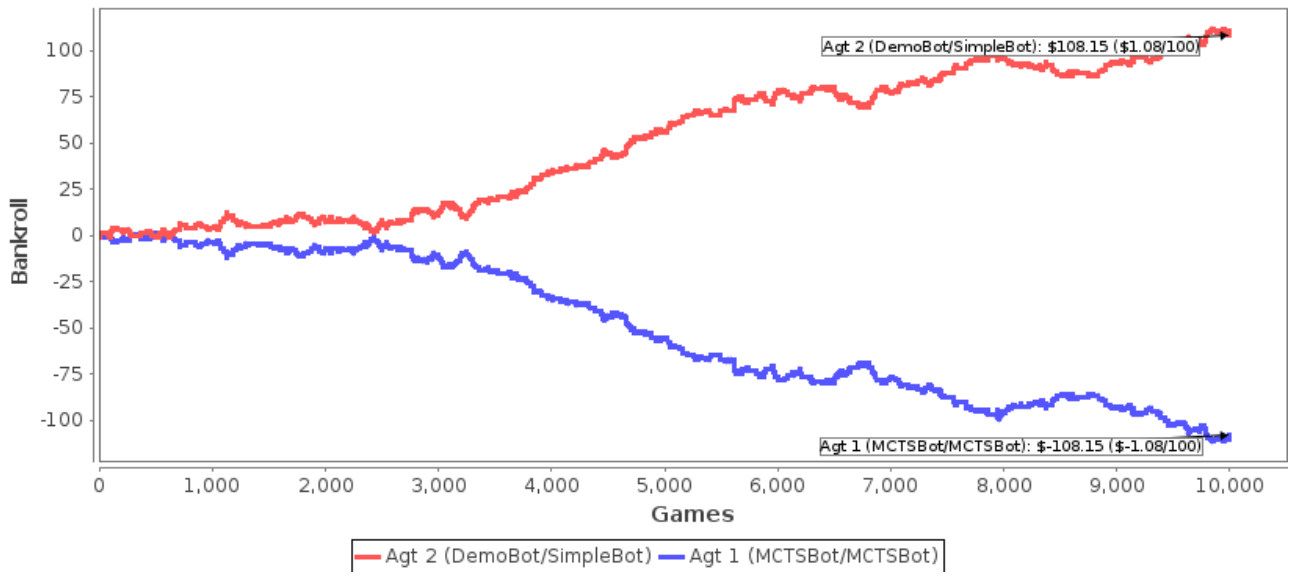
1. Standard UCT with no trained opponent model
2. Standard UCT trained with NaiveBayes trained opponent model:
3. SPR weighted UCT trained with no opponent model:
4. SPR weighted UCT trained with NaiveBayes trained opponent model:

Results

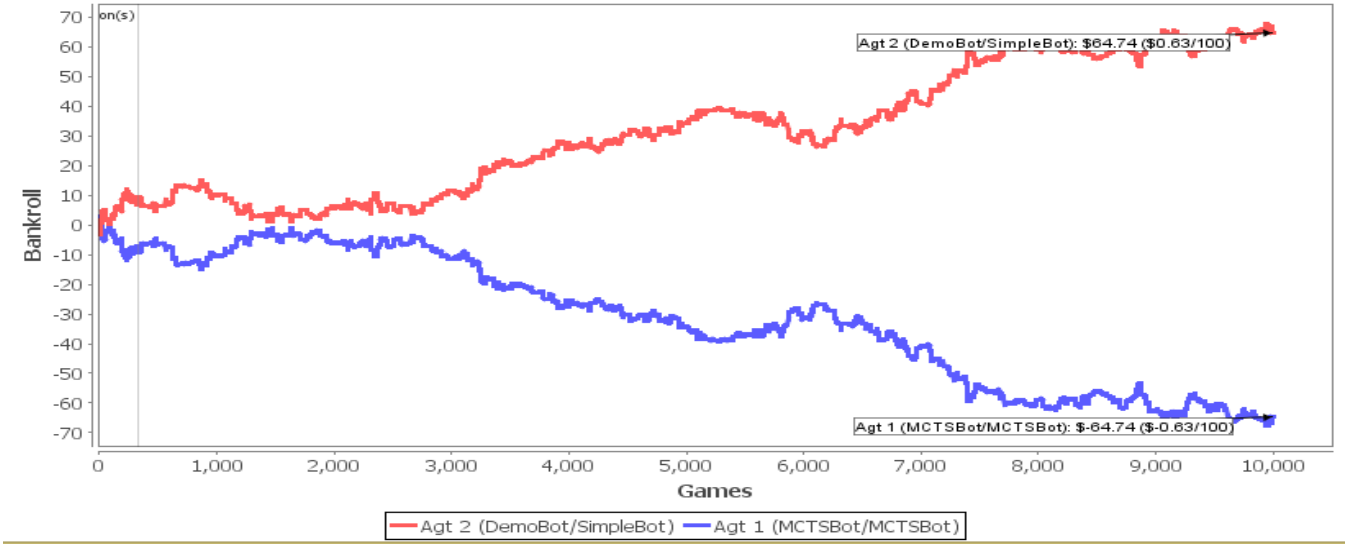
Standard UCT with no trained opponent model: -$172.52 ($-1.73/100)



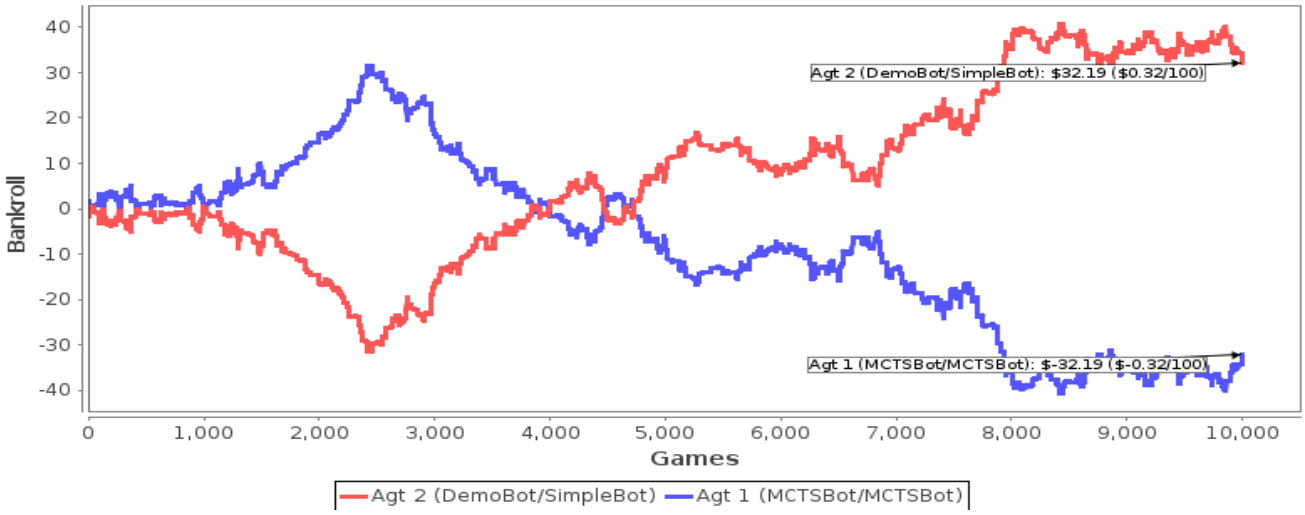
Standard UCT trained with NaiveBayes trained opponent model: -$108.15 ($-1.08/100)



SPR weighted UCT trained with no opponent model: -$64.74 ($-0.63/100)



SPR weighted UCT trained with NaiveBayes trained opponent model: -$-32.19 ($-0.32 /100)



Conclusion

The goal of this project was to determine which AI features I could add to an existing Monte Carlo Tree Search poker bot to produce better results. I chose to implement my own weighted UCT selection strategy within the MCTS algorithm, as well as an opponent model with a NaiveBayes classifier used to help predict opponent actions. It is important to note that continuous learning and online learning were not utilized during these experiments. Though these options are useful, the goal of the experiment was to test the opponent model on the training set rather than real-time gameplay.

The results of my experiments were fairly suprising. Although none of the tests were able to beat the SimpleBot over 10,000 games, the SPR weighted UCT implementations did appear to produce better results when compared to the regular UCT implementation. What is particularly strange is that in both of these tests, the results were fairly consistent until approximately 3,000 completed games where we notice a sharp decline in the MCTS bot game performance. Reasons for this would not be due to adaptation from the SimpleBot (our opponent), but could possibly be due to variance.

In looking at the data, it appears that the experiments which tested the opponent model faired moderately well compared to no opponent model. I can conclude that while the NaiveBayes classifier provides a simple and efficient way to classify a large amount of instances, it would have been beneficial to test multiple classifiers on the training data.

Through this project, I learned a great deal about the Monte Carlo Tree Search algorithm. There are many pieces which can be modified (selection strategies, backpropogation, etc.), making it especially interesting for those who wish to test and tune different techniques. I was also able to familiarize myself with data analysis through the usage of the WEKA toolkit. My implementation of the NaiveBayes classifier provided a real application in which I could apply training techniques to a game which I have an interest in (Texas Hold'em Poker).

One of the main obstacles faced during the development of this project was understanding how the MCTS algorithm and the opponent model were able to share information. This project contains over 53,000 lines of code and though I only contributed a very small portion, I needed to understand all of it. Additionally, I found it especially difficult to understand the results of my four experiments. Particularly, the consistent negative divergence of the MCTS bot in each test at around 3,000 games. I had also ran many more tests which were not included in this report, and nearly all of them contained a negative divergence of the MCTS bot at approximately 2,500-3,000 games. In the future, I would like to implement additional opponent models trained with different classifiers (SVM, decision trees, etc...) to test if this negative divergence occurs in these cases as well.