# Support Vector Machines in the Machine Learning Classifier for a Texas Hold'em Poker Bot

Jeremy Pfund
pfundj@seas.upenn.edu
Advisor: Dr. Mitch Marcus

University of Pennsylvania

April 10, 2007

## Abstract

This design project explores a fascinating application of artificial intelligence and machine learning to one of today's most popular mental challenges: Texas Hold'em poker. Recently, IBM's Deep Blue computer chess program managed to defeat the grandmaster Gary Kasparov utilizing an advanced brute-force search—a method dependent upon chess being a game of deterministic perfect information. This technique cannot be applied to the game of poker because poker is a game of incomplete information and chance outcomes, among other substantial difficulties.[5] In this case, one does not know the cards that his opponents are holding and one does not know whether your opponents are accurately representing the strengths of their hands or if they are just bluffing. Furthermore, future situations are non-deterministic and modeled probabilistically. This raises several interesting complexities regarding optimal strategy in the game of poker.

The approach taken in this project is to identify a set of attributes that can define the important features of a single game state. Given a value for each of these attributes, the bot, like any strong human player, has enough information to decide what the proper action is to

take. I use a database of game states and corresponding actions with which to train the bot, after which the engine is capable of classifying new game states into the proper action classes. An obvious platform upon which to administer this project is to interact with the well-known online poker application, PartyPoker.net®. This allows the bot to play with real people on a well-established platform for play money.

Following the strategies described in this paper, the bot that I developed achieves impressive results. Extensive play over time shows that the bot is losing an average of 3.27 big bets per 100 hands played. Regardless of the fact that it is a losing player, it is quite impressive that this value is so close to zero. There are plenty of foreseeable improvements, so its decent performance to this point shows promise that it has potential to succeed at these full tables pending future development.

## Related Work

There has been a lot of academic and professional work in the field of poker bots and poker artificial intelligence. One of the forerunners of this research is the Computer Poker Research Group at the University of Alberta, Canada. This group has released many research documents and has developed two poker bots—*Poki* and *PsOpti*.[2]

The most successful results of this work, including PsOpti's programming, involves a certain type of game known as *head's-up* poker, which simply signifies a two-player game of one-on-one poker. This is a significantly different situation from *full-ring*, or ten-person games in particular because bluffing becomes more dominant, substantially altering any winning strategies.[2] Furthermore, in poker the concept of opponent modeling is a significant aid in determining what style to play against an opponent. This involves keeping track of the opponent's style of play and tendencies in order to better predict the strength of his hand and his likelihood to respect your play. For example, if you observe, after many hands, that your opponent often calls to see the flop, but rarely calls to see the turn, you might bluff more often on the flop knowing that he is likely to fold.

The two most important categorizations of an opponent are the aggressiveness of the player and the player's willingness to put money into the pot. The respective extremes of these two categories are passive/aggressive and tight/loose. A player will see drastic improvement in their game play if they can categorize opponents with regards to these

distinctions and then adapt their own play in order to capitalize on the opponents' tendencies.

The University of Alberta team's bot, Poki, utilizes a method known as Selective Sampling in order to probabilistically predict the hole cards that its opponent is holding.[6] It remembers the opponent's previous actions and behaviors and models his tendencies in such a way that it can assign probabilities to the potential cards that the player is holding. This essentially is all that is needed to create a profitable entity and is an ability that all poker players aspire to. Once a bot can consistently determine the strength of its opponent's hand, the remainder is trivial—it need only adapt its play to force the opponent to bet more money in the pot when the opponent holds a weaker hand, and to refuse adding more money into a pot where the opponent holds a stronger hand. This last concept is in essence that of expected value. Whenever faced with a decision to make, the bot must determine how it can maximize expected value, which is contingent upon whether it believes it has a stronger hand than its opponent, which, finally, it manages to determine with its opponent modeling techniques.

One of the features that sets my project apart from Poki is that I am attempting to tackle the ten-person tables where opponent modeling becomes increasingly difficult. As a result, our strategies are quite different. Whereas Poki is meant to overcome the tendencies of a single opponent at any given time, my bot is meant to play a strong and solid game of poker against multiple opponents, and should be successful in defeating the average player found in an online poker room. I attempt to achieve this end through modeling pure game-states and not opponents directly.

# Technical Approach

### Texas Hold'em Poker

An introduction to the game of Texas Hold'em poker is required for a reader to fully understand the complexity of the game and the strategic initiatives behind the designs in this project. The game is played with 2-10 players at a single table. The first step is known as *pre-flop* and every player is dealt two cards face-down, known as *hole* cards. This is followed by the first round of betting. Next the dealer places three *community* cards face-up in the middle of the table, known as the *flop*. This is followed by the second round of betting.

Next the dealer adds one more card face-up to the community cards, known as the *turn*, following by the third round of betting. On the *river*, the fifth and final face-up card is added to the community cards and is followed by the final round of betting. After this betting, all remaining players make the best five-card poker hand that they can using the seven cards available to them (their two hidden hole cards and the five shown community cards). The player with the best hand wins all of the money previously bet in the *pot*. In a special case, if only one player remains prior to the conclusion of the final round of betting—i.e. all other opponents have folded to betting by this player—then this player wins all of the money in the pot and is not required to show his cards.

This project focuses on a specific variant of Hold'em known as Limit Texas Hold'em. This entails a structured betting system where all bets are in predetermined increments. There are two pertinent values, the small bet size and the big bet size, where the big bet size is typically double the small bet size. During the pre-flop and flop *streets*, or rounds of betting, bets and raises increment according to the small bet size, and on the turn and river streets, bets and raises increment according to the big bet size. All rounds of betting begin with the player immediately to the left of the dealer and move clockwise around the table. Upon his turn to bet, a player may *fold* (turn in his hold cards and forfeit previously bet money), *call* (match any outstanding amount bet by others and place it into the pot), or *raise* (call and place an additional bet increment into the pot that others must then match). If there is no outstanding bet, a call is referred to as a *check*, and a raise is referred to as a *bet*. In any round of betting, there is a limit of three total raises per street. It would be unfair for players to always act in the same order so, to level the playing ground, the dealer rotates clockwise around the table after every hand.

**Testing Platform Background and Implementation**

An obvious platform for this project is to interact with the well-known online poker application, PartyPoker.net®. This allows the bot to play with real people on a well-established platform, for play money. A substantial portion of this project involves coding the interface with this application. The bot needs to retrieve all necessary information from the PartyPoker.net® application and also needs to perform behaviors on the application simulating human interaction.

I feel that a short justification is required here because many online poker applications are explicit about disallowing the use of bots. I wish to stress that this project is

merely an academic exploration of artificial intelligence techniques and no commercial or gambling aims whatsoever. This project is morally acceptable given that it will only play for play-money and as such cannot cause any monetary damage. Furthermore, this project is not classified under the characteristics that PartyPoker.net® describes in its Unfair Advantage Policy as the types of bots that it does not allow,

> **…what we're talking about here are programs which: (i) advertise as a key feature that the buyer/user will gain an unfair advantage (quite often they even use the word "cheat") over the other players—in other words, they promise to help a player to cheat; or (ii) their use by the buyer/user is intended to remain concealed from the other players and from the "operator" (i.e., the online gaming room); or (iii) steal legally protected material (e.g., player identities) that violate International Copyright Laws as well as Privacy Laws.[4]**

This project is neither about cheating nor about collusion between players at the same table. This project is not concealed to the poker room and, as the terms of service state, the poker room may scan process lists and/or mine data on client computers to detect illegal activity and this bot will operate in plain view. This project does not steal any player identities. This project is purely an academic exploration of machine learning techniques and their applications to poker, and that is made clear here.

The interaction of the bot with the PartyPoker.net® application involves extensive windows programming. I utilize Microsoft Visual C++ coupled with the Microsoft Foundation Classes to program the application. There have been substantial challenges encountered in this portion of the project. My bot must grasp a handle of the window for a table that the user has joined. It needs to learn when a new hand is dealt out and to learn what two hole cards are dealt to the player. The bot must recognize player actions and discover when it is its turn to act. After deciding the proper action to take, it must grab a handle of the correct button on the table window and send a mouse-click window event to that button. The bot must continue on to recognize all community cards and further player actions on subsequent betting rounds. Finally, it must keep track of its *stack*

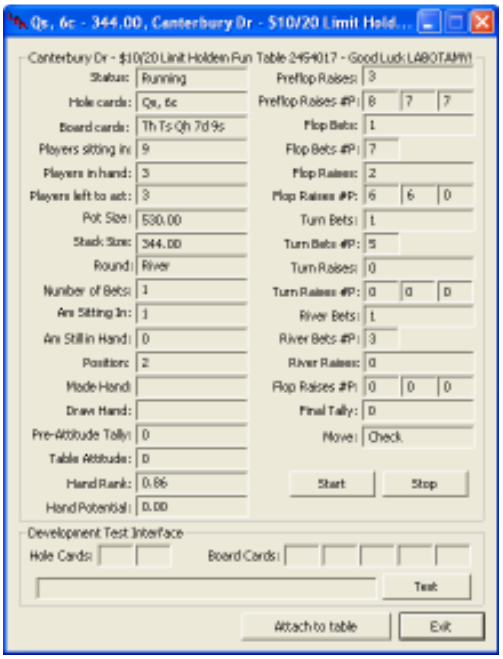
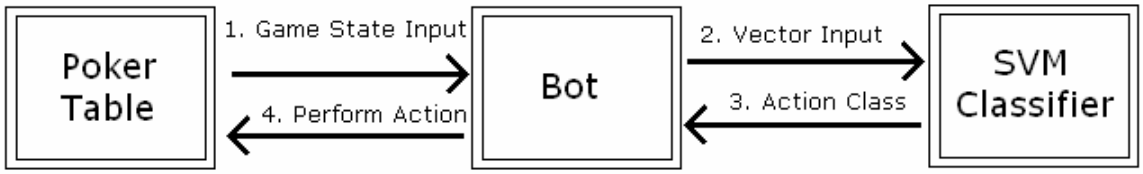



*Figure 1: Prototype Dialog Box Screenshot*

size (the amount of money it has available at the table to bet) so that it may recognize when this value reaches excessively low levels and behave accordingly. For example, it would not make sense to fold a hand when your stack size is only the value of one small bet, because after you have put in this bet, there is no further risk of being forced to fold later on in the hand due to a player pushing you out with a bet.

The current prototype has a very simple interface—it presents the user with details regarding current game-state in order to present the information it has gathered. Once opened, the bot allows the user to attach it to an open poker table. It presents the user with the open poker tables that it finds and prompts the user to select which table to attach to. The bot achieves this through traversing the MFC window classes and searching for appropriate table windows. Once the bot is attached to a table, it is ready to begin working. The user clicks the 'Start' button and from there on out, the bot is fully automated. It locates and clicks the appropriate buttons to sit the player in and post blinds. They are located through traversing the MFC windows to find the corresponding buttons and then are clicked automatically by sending windows event mouse-click messages to their handles. This same process is performed later on when an action is determined to be performed. Most of the game-play information is determined via the textbox console that declares player actions and table cards dealt. A handle of this textbox is attained and the bot parses through the text to retrieve this information in real-time. The player's hole cards, however, are not declared here and so the bot must perform screen scraping of the table window to identify what cards are dealt to the user for every hand. A bitmap of the table is copied and the volatile pixel coordinates are analyzed to determine card rank and suit. To facilitate this, I modify the application resource files to replace the card images that the application loads at runtime with more easily-identified images.




Once all of the proper information is identified and it is determined by the bot that it is the user's turn to act, the bot runs the current game state vector through the classifier, which returns the action class corresponding to this game state. Upon retrieving this action

class, the bot sends a mouse-click event to the corresponding button on the poker table interface to trigger the appropriate user action.

**Artificial Intelligence Strategy Overview**

A good place to begin strategic analysis is in determining what aspects of Texas Hold'em Poker correspond to artificial intelligence problems. The table in Figure 2 outlines these relationships and itself originates from a research paper published by the University of Alberta team.[3] The listing includes the likes of imperfect knowledge, multiple competing agents, risk management, agent modeling, deception, and unreliable information. Many of these problems actually run much deeper than they first might appear to. For instance, agent modeling includes the likes of pattern recognition, emotional aspects, and inference of intent. Another important problem that is left out of this table is that of dealing with probabilistic outcomes in expecting future cards.[6]

| General Application Problem | Problem Realization in Poker |
|---|---|
| imperfect knowledge | opponents' hands are hidden |
| multiple competing agents | many competing players |
| risk management | betting strategies and their consequences |
| agent modeling | identifying patterns in opponent's play and exploiting them |
| Deception | bluffing and varying style of play |
| unreliable information | taking into account your opponents' deceptive plays |

*Figure 2: Artificial intelligence problems and their corresponding place in poker*[3]

In any event, we make it clear that the task at hand is far from trivial from a computational standpoint. Keep in mind that the game of chess is essentially void of all of these technicalities and has itself already been proven as an incredibly complex computational challenge. As expressed earlier, the intent of this project is not to perfectly model every one of these problem realizations to create the perfect poker bot. Rather I develop a strong bot by generating and utilizing a training database coupled with recognition of basic poker principles. As a result, this project proves quite expandable in nature. Following this development of a strong decision-making foundation for the bot, a further task might include expanding my strategies to include some form of opponent modeling in an attempt to develop a more player-targeted system—that is to say, to develop the bot to modify its behavior according to the tendencies of other specific players at the table (e.g. making the correct call of a bet to which it otherwise might have folded on the basis of playing against a particularly loose player).

**Pre-flop Strategy**

The pre-flop actions should be treated as a separate case. This round is extremely simplified in comparison with later rounds, but also extremely important to consistent winning. It is tempting to call every raise pre-flop with the justification that you do not yet know how much the flop will improve your hand. However, overplaying pre-flop is well-known to be the common mistake of a beginner. A hand like A5o (the o stands for off-suited) will often lose a lot of money to an opponent with a hand like AKo simply because if an ace flops, both players now have significant hands, but the king is a better kicker. That is to say, both players would have a strong hand against a random opponent, but the opponent with AKo has the immense advantage in this case and is the heavy favorite to win. To make money in poker, you need to win more when you win, and lose less when you lose. In other words, it doesn't matter how often you win, all that matters is that you win the maximum when you have the best hand, and lose the minimum when you don't. By this reasoning, if you are a unfamiliar to the game, you will do better in poker simply by folding more—especially pre-flop. Another cause for pre-flop concern is the position you are in for a given hand relative to the dealer.

Since the complexity of pre-flop hands is very low, it is justifiable to develop a strong pre-flop system manually. I have devised a simple pre-flop strategy of only playing an elite group of hole cards to the flop and playing them with varying levels of aggression according to strength and position. This provides the bot with a tight enough attitude to spur it into a winning strategy for pre-flop play at a 6-10 person table. I qualify the table size because naturally, a player's hand strength is inversely proportional to the number of opponents, and for sake of discussion, we presume for the remainder of this analysis that the bot will play only at highly populated tables.

**Post-flop Strategy**

Post-flop, the complexity of game-states increases significantly. This portion of the game is substantially more interesting and warrants machine learning techniques to generalize action strategies. The classification system developed for this bot is to train on a database of game states and corresponding actions, so I must first define what a single game state constitutes. This involves identification of a set of important game state attributes, or features, which can sufficiently define all relevant information necessary to make a proper

decision. In turn, a vector consisting of values for each of these specified features will constitute a single game state.

The discussed strategy is contingent upon providing a database of training data. Thankfully, a real human hand history database is made available for public use by the University of Alberta and is known as Michael Maurer's IRC poker database. It is hosted by the University of Alberta Computer Poker Research Group website at http://games.cs.ualberta.ca/poker/IRC/. The portion of the database that contains useful information for this project is in the realm of several million game histories and from each game can be extracted multiple hand situations. A simple, perhaps obvious, way of improving the quality of data to train on is to include only those hand histories for the winning player of a given hand. This will ensure that the bot at least averages in on winning plays rather than losing plays, even if a small proportion of these plays were more likely to lose money than to win.

The proposed strategy also requires a machine learning technique with which to interpret the learning database and a corresponding classification technique with which to classify new game state instances as belonging to an action class. Implied here is that the domain of classes for a given situation is the set of three actions available to a user at any given game situation: Bet/Raise, Check/Call, and Check/Fold. That is to say that the class of a game state is the corresponding action to be performed. However, training on a hand history database will preclude this simplification into three easy classes because there is overlap between Check/Call and Check/Fold. Namely, if a player checks a hand, it is not clear which of these two classes he falls under—whether he would have called a bet before his action or not. Therefore, in classification for this bot, these action classes are slightly expanded to the four classes of Bet/Riase, Call, Check, and Fold. An exploration of classification methods and a conclusion on which classification method would best model this problem is discussed in the next section.

**Classification Training on Hand History Database**

Immediately it becomes apparent that a classifier is appropriate for this problem. Classifiers take as input an input vector and utilize a function to return a result, typically yes or no. This can be imagined as splitting the input space with a hyperplane to generalize not-before-seen inputs. Right off the bat, we can also rule out classifiers that make independence assumptions, such as the Naïve Bayes classifier, as our input vector will clearly

exhibit a degree of inter-dependence. The K-Nearest-Neighbor algorithm is another unfeasible classifier. The problem here is two-fold: 1) there is a high-dimensional space issue where a simple calculation discovers that for the proposed vector dimension of the problem coupled with the database size would require the hypercubic neighborhood spanning nearly half of the input space, and 2) this algorithm would require traversing the database of hand histories upon every classification.

A simple appropriate classification technique is to utilize a linear regression on the training database. This, however, would not exactly satisfy the need for classifying the data into distinct classes, and moreover is largely subject to skewed and erroneous anomalies in training sets. We can do better with Perceptrons and Support Vector Machines. Perceptrons divide the input space with a hyperplane and in the training phase attempts to eliminate errors in the training data. This requires more time to train the classifier, but the result is a very fast classifier because all that is required for training is computing the relationship between an input and the hyperplane. Support Vector Machines further the power of Perceptrons in additionally maximizing the margin between training points and the resulting hyperplane. SVMs work very well with large feature sets and can take advantage of optimization techniques and higher-order feature-sets utilizing the kernel trick.

Another peculiarity with this specific problem is that the domain for the classifier is not binary. This would suggest that the problem requires a multi-class classifier, and multi-class SVM classifiers do exist. However, recent research by Duan & Keerthi suggests that rather than using a direct multi-class SVM implementation, the project would be better off combining several binary SVM classifiers to attain a multi-class result.[9] This method involves learning via a one-against-the-rest classifier for each class. In our case, we first classify whether a given hand is classified as a fold or not, followed by whether it is classified as a check, followed by call, followed ultimately by bet and raise, which are treated equally. The learning data is converted for this reason into three learning files, each classified as yes/no for the given one-against-the-rest classification—there are three classification steps and not four because after three steps, the only remaining class is bet/raise. According to this setup, a given vector state must be classified according to the following preset order of fold, check, call, and bet/raise in order to properly utilize this learning technique.

My bot employs the publicly-available Support Vector Machine library SVM-Light. SVM-Light is an implementation of Support Vector Machines in C, authored by Thorsten

Joachim, and is provided free for non-commercial use at http://svmlight.joachims.org/. I have programmed a compilation of python programs to identify hand situation vectors from the database of hand histories and to use these vectors to transform the training database into readable input format for the SVM-Light training application. Running the trainer on this input will provide me with model files that encode each classification. Providing these models to the bot allows it to invoke the SVM-Light classifier for new input vectors to attain the appropriate action class.

**Feature Selection and Hand History Database Conversion**

Generally speaking, the strength of a player's situation in Texas Hold'em is a function of (1) that player's hand quality relative to the threats of better hands provided by the community cards, and (2) the strength of opposition measured by opponents' actions in the hand. A player's hand quality can be broken down further into made hand value (the quality of the player's hand given the cards that have already been dealt) and draw hand value (the quality of the player's hand based on its probability of improvement with future cards, e.g. a flush draw is a situation where a player has four cards of the same suit and hopes for a fifth card of the same suit either on a subsequent street). To be completely accurate, the made hand value and the draw hand value are both dependent upon opposition strength. Opposition strength can be further broken down into bets and raises during every round of betting, coupled with the number of people remaining in each round. Other previously unmentioned, albeit crucial, attributes include player position, pot size, and outstanding bets. The diagram in Figure 3 illustrates the proposed game state attributes and their relationships.

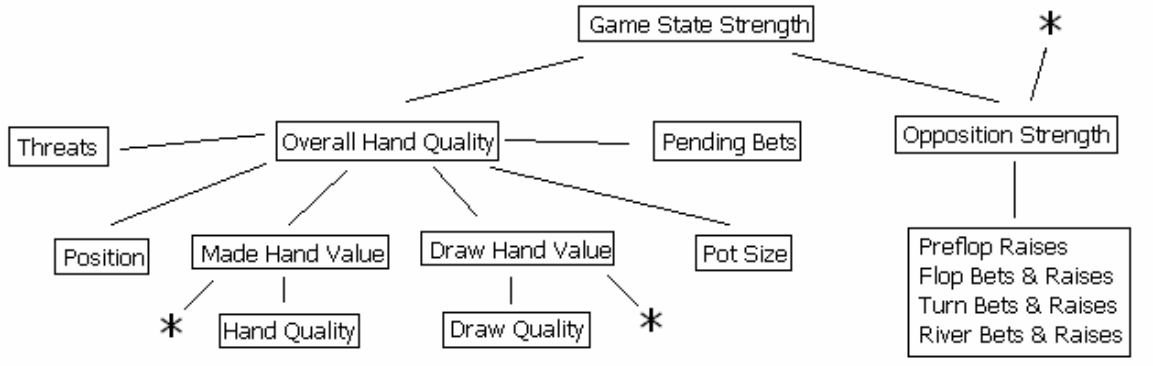



*Figure 3: The attributes of a given game state as they relate to the player's likelihood of winning a showdown*

Based on these criteria, the following list constitutes the vector features utilized in the training of my bot: hand rank, position, number of opponents, betting round, pot size,

number of bets per round, number of opponents remaining per round, hole card strength values. The hand rank and hole card strength features still require explanations regarding how they are calculated, how they are interpreted, and what significance they have in determining a corresponding action.

To determine the hand rank of a given hand, a brute-force, iterative calculation is performed in which for every possible pair of opponent hole cards coupled with the community cards, the best possible five-card hand is determined and compared with the rank of your own best possible five-card hand. The computation returns a decimal value between 0.00 and 1.00, where a value of 1.00 implies that you have an unbeatable hand. This value is incredibly pertinent in that it is naturally normalized across all possible hands, indicating the strength of your hand versus all possible opponents.

To determine the hold card strength, a pre-computed table is used. The table dimension is 13 x 13 x 2 representing respectively the rank of the first hole card, the rank of the second hole card, and whether the two are of the same suit or not. As a normalization improvement, the card of lower rank is considered as the first of the two cards, and the table is partially diagonal. When traversing the training data, each cell in this table accumulates values for the following crucial information regarding all pairs of hole cards witnessed: number of times won with these hole cards, number of times lost without folding, and number of times folded. Since the training data has imperfect information—i.e. it does not contain the hole cards of folded players—the third value, number of times folded, is determined as the difference between the expected number of times any pair of hole cards is witnessed (itself determined based on equal probability of any two cards) and the number of times the pair of hole cards was actually discovered. The value in collecting this information about all hole cards comes particularly into play to help the classifier learn whether it is holding a generally strong or generally weak starting hand. While the pre-flop strategy does not play weak starting hands, the bot must be prepared to handle the cases when it happens to enter the flop with weak cards when it is in the large blind and there is no raise to push it out pre-flop.

In preparing the hand history database for the learning program, the hand histories must be traversed and interpreted, and the pertinent features must be collected and outputted to an input file. Furthermore, every feature must be normalized between 0.0 and 1.0 to ensure equal weighting as well as propagated into the three learning files for one-

against-the-rest classification. This preparation process is very time consuming in the current configuration. Conversion of one such database file running on a single machine requires runtime on the order of roughly twelve hours and generates on the order of 25,000 (game-state, action) pairs. The bottleneck at large is due to the computing consumption utilized by the hand ranking algorithm. One such calculation requires on the order of two seconds. The entire database of hand histories holds seventy such files, resulting in a total training hand history learning file of 1.78 million entries. Due to this large training database, the learning process takes a substantial amount of computing power, but luckily is easily distributable across a computer cluster.

The previously mentioned issue of imperfect information comes as a significant challenge in the learning process. As the hand history database holds no information regarding hole cards of folded hands, the learner cannot in this way decide that any hands are to be folded. A couple of techniques are utilized in this bot to sidestep this dilemma. For one, any revealed hand in which a user did not fold their hand and lost, that game state is included with the fold action class to provide sample data. While there are surely cases in which the user should not have folded even though they wound up unlucky and losing, the majority should be properly classified and overwhelm these outliers. The second technique in alleviating this dilemma is the inclusion of the hole card strength values. These values inform the classifier of generally weak starting hands and provide data that would support hand strength or weakness in the general sense regardless of community cards. The hole card strength values, coupled with the fold histories, should provide the classifier with the much-needed information to determine weak enough hands to fold.

**Accounting for Hand Potential**

The entire decision engine of the bot cannot only be attributed to the machine learning implementation and classification—further mechanisms are also in place. For example, the previously discussed pre-flop algorithm is completely decoupled from the classifier. Another previously introduced addition to the decision engine is the accounting for hand potential. The concept of hand potential arises in special situations in early and middle game-play when a player's hand might have a significant probability of dramatically improving. More commonly, this class of hands is referred to as *draw hands* because as a player you hope to draw one of the desired cards. While any hand can be crudely considered a draw hand, since specific cards are likely to improve them, the key draws we are interested

in here are *straight draws* and *flush draws*. An *open-ended* straight draw arises when you have available to you four cards in a row by rank, and are hoping that on a later street you receive a card of rank that would complete your straight of five ranks in a row. In this case, there are eight cards that would achieve your straight. A *gut-shot* straight draw is the similar scenario, where you have four cards to a straight, but require one of the middle ranks, and therefore are drawing for one of four cards. The most substantial draw is the flush draw, where you have four cards of the same suit, and therefore drawing to one of eight cards that would complete your flush.

The concept of hand potential is an important one because it is easy to calculate the odds of completing your draw as well as to calculate your *pot odds*—that is, how much money you have to call to stay in the hand to see the next streets, relative to the amount of money already in the pot. If the pot odds are greater than your draw odds and the likelihood that if you make your draw that you will win the hand is close to 100%, then you should never fold because simply calling the bet would result in positive expected value. The bot incorporates this check into its system to profit from the additional advantage that it provides over opponents.

# Results

I have implemented the fully-automated bot application described in this paper from end to end. The bot successfully attaches to poker tables, retrieves all vital information in real time, performs the classification process to determine actions, and performs these actions in real time. The performance of the game-play strategy system has proven to be quite impressive. I have run the bot at play-money tables in real time over an extended period in order to collect the performance data displayed in Figure 4:

| # Hands | Voluntarily Put $ In The Pot | Won Money When Saw Flop | Amt Won per 100 Hands | Won $ at Showdown |
|---|---|---|---|---|
| 2343 | 5.63% | 17.01% | -3.27 Big Bets | 48.89% |

*Figure 4: The performance data of the bot on live play-money tables*

The value of *number of big bets won per 100 hands* is the best indication for performance of a player. This is a normalized value to the stakes that the user is playing at and indicates, all other statistics aside, whether that player is consistently winning. According to this result, the bot is losing an average of 3.27 big bets per 100 hands played. Regardless of the fact that it is a losing player, it is quite impressive that this value is so close to zero. The statistic for

whether the bot won money if it went all the way to a showdown—i.e. betting completed at the end of the river street and the player with the best hand wins—holds a value of nearly 50%. Since many of these showdowns result in more than two players, with a minimum of two players per showdown, the fact that this value far exceeds 1/n, where n is the number of players at the showdown, is a strong indication that the bot is not excessively bluffing nor excessively playing poor cards to showdowns. The value for how often the bot won money when it saw the flop is greater than 1/n, where n=9 is the number of players playing at a table, indicating that starting hand selection is resulting in positive play, granting confidence in the pre-flop strategy adopted by this project.

**Conclusion and Future Work**

As the first stage of a research project, this puts us exactly where we want to be. There are plenty of foreseeable improvements, so its decent performance to this point shows promise that it has potential to succeed at these full tables following future work. Foreseeable improvements include opponent modeling, reinforcement training and improved training data quality. The quality of training data is admittedly questionable. These hand history databases were collected on play-money IRC-chat poker tables. If I were to collect my own data, I could observe game-play at real-money tables at a rate of approximately 160 hands per hour per machine. Since each hand provides approximately ten (game-state, action) pairs, reaching the current quantity of training data would take just over 1000 hours on a single machine and be linearly distributable across machines. For a successful bot, this measure should certainly be employed as the collected data would veritably represent real-life performance. Meanwhile, opponent modeling is a significant aid in determining what style to play against any given opponent. This would involve developing the bot to modify its behavior according to the tendencies of other specific players at the table in order to provide a player-targeted system. The bot should keep track of opponents' styles of play and tendencies in order to better predict the strength of their hands and their likelihoods to respect your play. For example, if you observe, after many hands, that an opponent often calls to see the flop, but rarely calls to see the turn, you might bluff more often on the flop knowing that he is likely to fold.

The statistical results above suggest that most of the improvement that this bot would benefit from would be incurred by improving post-flop game-play. I witnessed some questionable decisions on the flop and turn that clearly exhibited negative expected value. I

believe these wrinkles could easily be ironed out if the bot were to adopt some sort of reinforcement strategy. This strategy would incorporate the following behavior: if the bot folds a hand that would have won if it had not folded then it should add the correct game-state vector to the set of learning data with the action class that it should have taken, and periodically, the classification model would retrain on the modified learning set. Similarly, the bot would add additional reinforcement data to amend mistakes where the bot played poor hands too aggressively and lost. This will help the bot learn from hands where it behaved poorly due to the model having been trained with insufficient training data.

# References

1. Spice, Byron. Carnegie Mellon Computer Poker Program Sets Its Own Texas Hold'em Strategy. http://www.cmu.edu/PR/releases06/060706_pokerbot.html. July, 2006

2. Schauenberg, Terence Conrad. Opponent Modeling and Search in Poker. M.Sc. thesis, 2006.

3. Darse Billings, Denis Papp, Jonathan Schaeffer and Duane Szafron. Poker as a Testbed for Machine Intelligence Research, Lecture Notes in Artificial Intelligence volume 1418, Springer Verlag, Robert Mercer and Eric Neufeld (editors), (Proc. 12th Bienniel Conference of the Canadian Society for Computational Studies of Intelligence, AI'98), Vancouver, Canada, June, 1998.

4. PartyGaming's Unfair Advantage Policy. http://www.partypoker.com/about_us/game_fairness/unfair_advantage.html. WPC Productions Limited, 2006.

5. Darse Billings, Neil Burch, Aaron Davidson, Robert Holte, Jonathan Schaeffer, Terence Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI), Acapulco, Mexico, 2003.

6. Pokerbot: A New Competition. http://simon.lrdc.pitt.edu/~iccm/pokerbot.html. International Conference on Cognitive Modeling. University of Pittsburgh. 2006.

7. D. Billings, D. Papp, L. Pena, J. Schaeffer, D. Szafron, Using selective-sampling simulations in poker, in: Proc. AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information, 1999.

8. Gilpin, A. and Sandholm, T. 2006. Finding equilibria in large sequential games of imperfect information. In Proceedings of the 7th ACM Conference on Electronic Commerce (Ann Arbor, Michigan, USA, June 11 - 15, 2006). EC '06. ACM Press, New York, NY, 160-169.

9. Duan, Kaibo and S. Sathiya Keerthi. Which is the Best Multiclass SVM Method? An Empirical Study. Technical Report CD-03-12, Control Division, Department of Mechanical Engineering, National University of Singapore, 2003.

10. Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, NJ, second edition, 2002.