Creating a NL Texas Hold'em Bot

Introduction

Poker is an easy game to learn by very tough to master. One of the things that is hard to do is controlling emotions. Due to frustration, many have made the wrong decision. Moreover, one can make a lot of money by playing multiple tables at once online. This is where an idea of a poker bot that makes quick decisions without the interference of emotions can be a very profitable one. Though it isn't strictly legal to use bots to play online poker in multi tables for you, the idea that one can create a bot is very exhilarating, just like it has been for many with the game of chess. Poker is a very complicated game and the machine learning techniques used to solve such a problem can be used in other useful areas that require reacting to different situations

Problem Definition

The goal of this project is to create a bot that plays poker in a profitable manner. The training data is my hand history that a software has saved from my games in online poker. The goal is to use this data to create a poker bot. To simplify the problem, some limitations to the game are made. Firstly, though this is NL Hold'em, the bot's bets will be fixed. A common raise made by many players is 3 times the bet made in front of them. If the bot is leading out, the bet will be 2/3 of the pot. Lastly, there will be no concept of bankroll, however, the winnings and the losses of the bot will be tracked.

Model Formation

To make computers play like humans, it has to estimate the hand of opponents based on its betting patterns. Imparting this wisdom to the bot is not a trivial task at all because there are so many situations of the game to consider. This is where machine learning becomes useful as it acts as a heuristic. The model's goal will be to determine based on a few features whether it will call, raise, check or fold in each street.

The training data available gives insight on the properties of the hands for which I called, checked, raised or folded. Hence, the training data lends itself to Naïve Bayes (NB). This Naïve Bayes model's job is to tell the bot which action is the most optimal. Hence, this model will be multinomial and not Bernoulli. By getting the probability that a certain action is the most optimal one, a preliminary approach may be to pick the action with the highest probability. A better approach would be to pick the action according to the distribution given, which adds randomness and is a crucial part of the game.

One of the added complexities of this model is the changing of the features for the decisions. One of the classical uses of this model is spam classification. The features to determine whether the email is spam or not are the same. In this case, the features used to determine whether the bot should fold in a later street (turn or river) will include features of the flop but not vice-versa.

Another alternative to NB is Support Vector Machines (SVM). This is an alternative because the classical applications that use NB uses many features, like the occurrence of a words in the dictionary. They are not that many features for this problem so a non-binary SVM could be a better model. However, getting the data to a format for SVM would be very difficult because there are copious training examples, One would have to decipher each one of them to summarize what actions were taken. Hence, for this project, with the available time, this model isn't as practical.

Features

The preliminary features selected for the study are listed below. For my training data, I filtered my hands based on these features and calculated the number of hands I called, raised, checked or folded.

1) Starting hand. Or in other words, the hole cards – the two cards given at the starting.
2) The opponent's action in the current street. This is not applicable if the bot acts first.
3) Current hand. The bot could have a pair, two pair, three of a kind, a straight, a flush or a certain of type of draw, which should affect its play
4) Position. In poker, the player who acts second has the upper hand.
5) The actions of the previous streets. In other words, what the bot should do in the turn may depend on what happened in the flop and pre flop.
6) Card texture. Many times what the flop, turn and river are should affect the bot's play.

Development process and Testing Procedure

Developing most software is an iterative process. This is crucial with machine learning so that the model is robust and can be implemented efficiently. With the proposed NB model, the pre flop was first implemented and the performance was evaluated. After necessary changes, the same is done for the flop and then for the turn and the river. For the intermediate testing, the interface was made in MATLAB. A table of 6 players was made, one of them being the bot. I manually played for the others and instruct the bot to respond. Poker players are a mix of certain styles, which include being tight, aggressive, loose and passive. Hence, I made the competitors play an extreme version of these styles. It wasn't a trivial task to set up the game and the interface and sample hands were shown in the poster session.

Changes to the NB model

The three main features of pre flop for the model are starting hand, position and bet state. Bet state corresponds to whether players before the bot had called or raised. Laplacian smoothing was implemented. However, it may not have had a significant impact as the training data had a wide variety of hands. After running the given properties of the model, the bot's pre flop performance was studied and concluded to be unsatisfactory. Firstly, if there were a bet in front of the bot, the bot would almost always fold. Lastly, many profitable players would open even with mediocre hands if they were sitting in late position to steal blinds. Though this is shown in the statistics, the bot wasn't doing this.

It was concluded that this model wasn't working well because NB assumption didn't hold for this problem. The features are conditionally dependent and now the conditional joint probability can't be expressed as a product of conditional probabilities. For example, given that a player raised and was in late position, the chance that he has a mediocre hand is high. Similarly, if a player bet from early position, the chance that he has a mediocre hand is low. If there is no raise in front of the bot and it is in late position, the bot should be raising almost always, regardless of starting hand. Hence, the solution to this was to model the conditional joint probability.

$$p(x_1, x_2 \ldots, x_n \mid a_s = k) \neq \prod_{i=1}^{n} p(x_i \mid a_s = k)$$

where x is the feature and a is the action. This states the invalidity of the assumption.

There are 169 possible starting hands for a player. It would take a lot of time for one to count the number of times a player performed a certain action from each position and for each hand. Hence, the conditional joint probability was modeled. This modeling also took two iterations. First, given that the bot performed a certain action and had a certain hand, the probability that the bot is in a certain position was modeled discretely using 6 bins of position: big blind, small blind, early, middle, late and dealer as shown below. To make the problem more manageable, the starting hands were given an integer from 1 to 5, with 1 denoting a premium hand.

$$p(x_i \mid a_s = k, h = l) = \alpha_{kli}$$

This distribution was solved using a method like least squares. The linear equation set up is shown below. What it denotes is that the distribution should be selected such that the global probability of an action is equal to the one measured. If N is total hands,

$$\frac{1}{N}\sum_{h} \alpha_{kli} h = \frac{\sum 1\{a = k\}}{\sum a}$$

The base constraint is that for each starting hand strength and action, the values that denote position should add up to 1 because this is a probability density function. There were other inequality constraints to capture the effect of position. More specifically, if the bot had a premium hand, the probability it should raise with it is evenly distributed among the positions because the position shouldn't have an effect. In contrast, if the bot has a mediocre hand, the bot should raise with that hand mostly from late position. The training data took into account the fact that one should be getting more mediocre hands than premium ones. (One can't always get aces). This problem is an example of quadratic programming. The techniques to solve such a problem, which include Lagrange multipliers and formulating a primal-dual problem is similar to the one used to solve SVMs. MATLAB's routines were used to solve this problem.

$$\min \left\| \vec{b} - A\vec{\alpha} \right\|^2$$

$$s.t. \sum \alpha_{kli} = 1, B\vec{\alpha} \leq \vec{c}$$

The bet state wasn't included in this system. This was for two reasons. First, it was unclear what constraints would model the effect of the bet state as was done for position. Secondly, the matrices were kept to a manageable size for ease of computation. Hence, for the next iteration, it was assumed that bet state was conditionally independent from

the other two features. From studying the performance of the bot, it was concluded that the bot handles position well. However, it doesn't handle bet states well. For example, with a drawing hand, like suited connectors, even if there is a raise before the bot, it is profitable for the bot to call and see the flop. To fix this issue, it was assumed that the bet state was the third and final feature the bot's play should depend on. With the previous method, the probability the bot performs a certain action with a certain hand in a certain position is known. With the probability the bot should perform a certain action given a certain bet state calculated from the training data, one can find out which hands the bot can have to perform that action in that position given that bet state. This was done by aligning the probabilities in a greedy way, with the maximum first. The cumulative distribution function was calculated and checked whether the CDF of that hand was lower or equal to the probability specified for that given bet state. This model works extremely well. However, it is less random than the previous model because many times, the model decides on one final action. For on another implementation, the bot can act outside the decision of this model with a given parameter to introduce more randomness.

Post Flop

The base NB algorithm was applied for the streets after pre-flop with the features being starting hand, action of previous streets and current made hand. Though there were correct moves, there were also a few blatantly wrong moves, skewed towards folding. After analyzing a few training hands that met the criterion and went as expected, it was concluded that the error was a high variance error and the number of features were reduced. Namely, instead of using the information of all the streets before the current street, the features were reduced to two, defense and offense. If the bot had bet in the previous street and got called, the bot was in offense and vice versa. The starting hand feature was also not used anymore. This got rid of lot of the noise and the bot performed much better. A lot of tuning as done for the pre flop was not done because now only the important features were used and the conditional dependence wasn't magnified.

Markov Decision Process (MDP)

With a table of players with different styles made, it was an interesting aspect to study whether the bot could adapt to the different players, which is a crucial part of poker. Instead of focus on making methods to test the model on other pre made data, this garnered more focus. This algorithm was used only in post flop given the robust method already implemented for pre flop. To use this model, I made the bot follow the NB model till the Markov decision process is trained after enough iterations. A mixture of unsupervised and supervised may be a good blueprint for other problems as well.

The states used for this problem are winning in the flop/turn/river (by making the opponents fold), losing in the flop/turn/river (by folding), seeing the next street and being in the defense/offense and winning at showdown for each specific hand and for each size of pot. Though the pot size is a continuous variable, it was discretized into 6 bins. This came up to 206 states. The rewards is the amount won or lost at the terminal states, which are the states at showdown and the losing and winning states at each of the streets. With

more hands, the average is stored. The actions are the four possible actions (call, raise, check and fold) for each of the street (flop, turn and river), which is a total of 12 actions.

The transition probability is now a function of player and this should help the bot finding the best action. For example, the probability of going to the next street being on the offense is low for an aggressive player even if the bot bets because it is likely that the bot will be re raised and is now forced to call. As the game goes on, hopefully the bot recognizes that being on the defensive is bad because if the opponent makes another bet, as given by the NB model, the bot should fold most of the time. Hence, against an aggressive player, it should check and then call. The bot is now not so much in a defensive state. Another example is the probability that a tight opponent folds to a bluff or a bet is high. So to increase the expected reward, it should raise against tight players. And with more iterations, it should know that the reward at showdown for bad hands is negative, which will induce the bot to bluff with bad hands and to bet with good hands to increase the pot and win more at showdown. In value iteration, to handle the different players, the update was normalized, where # is the number of players.

$$V(s) = R(s) + \frac{1}{\#}\sum_{p} \max_{a} \sum_{s'} P_{sa}(s')V(s')$$

Results

Using MATLAB, a poker session is simulated. At first, the NB model was used. The MDP was put into effect after the Markov model has a hand with each of the players. This took 48 hands, with more data on the aggressive players because they play more hands. By that number, fortunately, the rewards values were fairly accurate as this wasn't a function of player. I played as others with the bot for 50 more hands after that. In poker, many times there isn't a clear-cut answer to what the right play is as there are many factors to be considered. Hence, to test the models, from all the bots actions, I looked at the percentage of the bot's good, fair and wrong moves. Since most of those errors came post flop I looked at pre and post flop separately. PF = pre flop. F = post flop.

The bot's winnings are also reported, with its original stack starting at 1000. Except for the winnings, the rest are percentages. Some key statistics of the bot are also reported and compared to my play. I am looser pre flop because I didn't give the bot knowledge of pot odds, which encourages players to call with mediocre hands just because there is money in the middle. Having won huge pots before MDP, one with aces, MDP assigned a high value for going to showdown and so the bot became more loose and aggressive than NB, which, at times, was a good thing as the bot made some good moves. However, more iterations were needed for the bot to distinguish between the different players. At the end, it kept calling the aggressive players. For the next step, it would be good to devise a way to autonomously test so that the MDP can get trained in time.

Play	PF Good	PF Fair	PF Wrong	F Good	F Fair	F Poor	Saw Flop	Bet Frequency	Winnings
NB	91	9	0	44	35	21	11	28	700
MDP	91	9	0	61	11	28	11	34	950
Me	-	-	-	-	-	-	15	40	