Rafael Witten
Yuze Huang
Haithem Turki

Playing Strong Poker

1. Why Poker?

Chess, checkers and Othello have been conquered by machine learning - chess computers are vastly superior to humans and checkers and Othello computers play optimally. Part of what makes these games tractable is that both sides have perfect information. Poker is fundamentally different because the players do not know their opponent's cards and are instead left to create opponent models to "guess" what their opponent holds; it is a game of imperfect information. In chess, for instance, the standard assumption is that your opponent will make the best possible move – in poker that is not a reasonable assumption since your opponent lacks perfect information. Largely because of this added complexity, computers still lag behind humans in almost all poker variants.

The interesting variants of poker are sufficiently complicated as to be intractable – the computer cannot simply apply game theory tactics to create an in-exploitable solution. Creating programs to operate with incomplete information in such a complicated system is fundamentally challenging and techniques used are similar to those throughout other applications of machine learning.

2. The Rules of the Game

The variant of poker we chose was heads up limit Texas Holdem. It is a standard choice for machine learning because it is considered one of the "deepest" variants of poker and is quite commonly played.

Each of two players is dealt two face-down cards. First a group of three face-up cards are put in the "community" area (the flop), then a fourth one (the turn) and finally a fifth (the river), at any point these cards are called the board. Limit style betting is allowed before the flop, turn and river and finally after the river. Players make the best possible poker hand using five of the available seven cards. (If this summary is insufficient, the reader should read http://www.pokersavvy.com/Texas-holdem/Limit-Holdem-Rules.html because some understanding of the rules of poker is necessary to proceed.)

3. A Broad Overview of HaRDBot

A poker program, most broadly, is a function that takes a player's hand, the board, the action thus far in the hand and the opponent's history and returns a three-tuple, which represents the program's probability of folding, calling and raising; no strong poker program plays deterministically. Strong poker programs can be made that do not make any use of information about the opponent and instead attempt to play in a Nash Equilibrium, or in a way which is non-exploitable, meaning that the best the opponent can do is achieve parity.

The fundamental problem with attempting to play in Nash Equilibrium is that poker is intractable, so computing the Nash Equilibrium is impossible. However, it is possible, under some assumptions, to compute a e-Nash Equilibrium, or a strategy that is very close to being a Nash Equilibrium under those assumptions; however, how close the strategy actually is to the

Nash Equilibrium is subject to the accuracy of the assumptions. A general weakness with bots based purely on Nash Equilibria is that they tend to be bad at exploiting poor play by their opponents because they assume that their opponent is a rational actor similarly in a Nash Equilibrium.[1]

Instead of assuming that our opponent is in Nash Equilibrium, we instead base our original estimates of opponent strategy on empirical data – since no such data is available for a new opponent, we train from a corpus of high level machine play (about 200000 hands) and use Bayes Law to update our estimate of the probability distribution over our opponent's hands. This approach is unique in the literature and has so far produced exciting results. Our play, given a probability distribution over the opposing hands, is based on an expert system of similar complexity, but optimized for heads up play, to the prominent University of Alberta bot Poki, perhaps the best existing limit poker bot in multiway play.[2]

These two methods together define a full poker algorithm. Every time our opponent acts, we use our corpus and opponent data to update the probability distribution over our opponent's action. When it is our turn to act, we feed the probability distribution of our opponent's hand and the state of the game into the expert system and it probabilistically decides how to act.

4. Parts of HaRDBot

i. Updating Opposing Probability Distribution

In response to an opponent action, HaRDBot needs to update the probability of each possible opponent hand using Bayes Law. To train from a corpus of past poker hands, it is not enough to simply lookup how each possible opponent hand has been played in an identical situation. Even removing suit isomorphisms, we would encounter severe sparsity issues. For example, on the flop there are approximately 2 million different states and they increase exponentially on future rounds of betting.

Further, doing such calculations on a hand-by-hand level is computationally expensive and for the future analysis we assume we have bucketed opposing hands into hands that we believe will play similarly, a step which is discuseed in 4.iii. First we developed a technique for estimating the state of poker hand that can be compared across different hands and boards – essentially turning each opponent's hand into a vector of describing numbers. This vectorization is the key step that allows us to take advantage of the corpus of past poker hands. By consulting this corpus, we are able to, despite not having any knowledge of the opponent at the beginning of the match, update our probability distribution in a Bayesian way, assuming that our opponent is a "mean" opponent from the corpus.

To calculate P(action|bucket) we query a mysql database to find hands that have an identical betting history. Then we perform the following update, where the sum across entries means summing across each entry with an identical betting history. Here each bucket$_v$ is the vectorization of the bucket, entry$_v$ is our vectorization of the entry and entry$_a$ is the action that was taken in that case.

$$P(action|bucket) = \frac{\Sigma_{entries} 1\{entry_a = action\}\, e^{-|entry_v - bucket_v|}}{\Sigma_{entries} e^{-|entry_v - bucket_v|}}$$

This gives us exactly what we need to find P(bucket|action) using Bayes Law. Because our sample space is not discrete we need to decide the relevance of each of the entries in our database based on how far they are from our current term. This is what caused our use of the exponential term – it is a natural choice that goes from 1 (one the entries' vector is very close to the buckets) to zero (when they are very far away), thus similar to our intuition about how much to weight each example.

We built further on this formula to do opponent modeling. This formula works well at first when our knowledge of our opponent was limited, but by taking opponent specific knowledge into consideration we can make a better estimate. We decided upon this update:

$$K(action|bucket) = \Sigma_{entries} 1\{entry_a = action\}\ e^{-|entry_v - bucket_v|}$$

$$K' = K\frac{P(action|opponentmodel)}{P(action|database)}$$

Where we then normalized the K' terms to get a probability distribution. Here P(action | opponentmodel) is a smoothed Naïve Bayes modeled parameterized on the hand history. We used a generalization of Laplace smoothing, initializing our observations with instead of one of possible action with six total observations attempting to best fit our P(action | database) entry. Then when there were new actions we updated P(action | opponent), thus learning from our opponent's action.

Preflop vectorization proved difficult because the computations (even using Monte Carlo simulation) are somewhat expensive and unnecessary because of the relatively small preflop history space. Therefore, preflop we instead considered hands on a case by case basis. This was possible because there are only 1225 hands an opponent can have preflop and fewer than 10 betting sequences can occur on the flop that have the additional property that they do not end the hand. Given our huge corpus, we were therefore left with a meaningful number of hands after each betting sequence and had no sparsity issues.

ii. Vectorizing Hands

Although two poker hands and boards may be different, they may call for similar lines of play. The key step in the previous section was determining how to find these similarities between different situations. The task was equivalent to feature extraction.

E and E^2 are two particularly critical values we used in measuring poker hands. E describes the hand's chance of winning and E^2 is an information estimate of the value of our hand. Let F be the set of possible next observed cards, let Φ be a random variable distributed uniformly over F, and let h be a hand, i.e. two hole cards. We compute E(h) using the following formula (in each case where we sum over our space of opponent's hands):

$$E(h) = \Pr(1\{\text{win with } h\} = 1) = \sum_{f\in F} \Pr(\Phi = f)\Pr(1\{\text{win with } h\} = 1|\Phi = f)$$

and we compute E^2(h) using the following formula:

$$E^2(h) = \sum_{f\in F} \Pr(\Phi = f)\Pr(1\{\text{win with } h\} = 1|\Phi = f)^2.$$

In order to see why this formula for E^2 gives us an idea about how much information we would gain by playing the hand, consider an abstract variant of poker where there are only five possible (equally likely) flops, so F = {f1, f2, f3, f4, f5}. Consider two hands, h_1 and h_2. Suppose h_1 has a probability 1 of winning given f1, but probability zero of winning the rest of the flops, and suppose h_2 has zero probability of winning given f1 but has 1/4 probability of winning each of the rest of the flops. In our case, $E(h_1) = E(h_2) = 1/5$, but $E^2(h_1) = 1/5$, while $E^2(h_2) = 1/20$. This follows intuition, because after the flop, we will have already known whether we can win with h_1, but we are still uncertain on four of the five flops if we play h_2. Gaining knowledge early in the hand is very valuable from an information theory perspective and follows intuition on how humans play.

These two numbers, combined with NPOT and PPOT measures of the chance of a hand improving and getting worse, allowed us to find hands that had different hole cards or boards but would tend to be similarly played.[3] Other options for feature extraction were tried, such as "outs" (or number of ways to dramatically improve your hand) and a couple of other features that humans use to classify hands, but they were found to have little significance in the later bucketing step and were therefore eliminated to save computational time. It is worth nothing that E, E^2, PPOT and NPOT are all numbers that humans are unable to calculate in real time but appear to have greater significance than the metrics humans use.

iii. Bucketing

Running the Bayesian update rule for each hand is very slow and moreover completely unnecessary. Through the vectorization, we were able to find hands that would be played similarly. We then used K-means clustering to group hands that would have approximately the same update rule. Then, using the update rule established in 4.i on the updates, we are able to determine P(Bucket |Action). By our assumption that all of our hands in the bucket would be played the same way, we are able to define the hand update rule, P(Hand | Action) =(P(Bucket|Action)/P(Bucket))*P(Hand).

We chose to run clustering with K=5. The intuition behind this choice is weak, but the resulting buckets were very similar to how a human would classify them. Overall, the resulting system very much reflected how a human poker experts think about poker hands, classifying opposing hands into hands that have similar strategic implications and assigning probabilities to these hands.

iv. Expert System

Ultimately we used an expert system to decide how to act given a probability distribution over the opponent's hand. Our final design was based on the equivalent Poki algorithm. HaRDBot would consider a range of possible actions from most aggressive to least aggressive (or equivalently considering first those that would dictate a raise, then those that would dictate that we call) – for each, if the conditions necessary to take it were met we would consider taking the action probabilistically. Ultimately HaRDBot would fold if it chose none of the actions. This expert system was somewhat unsophisticated but worked quite decently.[1,4]

5. Conclusion

For our implementation, we used Meerkat-API to interface with a popular poker application often used by researchers, known as Poker Academy Pro. Because we were using the Meerkat-API we were able to compare HaRDBOT to Poker Academy Pro's bots, which are touted as being among the best in the world. However, simply playing a single bot is insufficient because strength at poker is not transitive (if A beats B and B beats C, A does not necessarily beat C), as can be seen in the data at the AAAI Annual Poker Competition Website.[6]

In order to test our bot, we devised a gauntlet of strong bots of different styles. HaRDBot played Sparbot, which attempts an e-Nash Equilibria strategy, Vexbot ,which relies on very aggressive opponent modeling and Poki, which was probably the best heads up limit bot when it was developed in 1998 (although it no longer has that distinction) and has a more balanced strategy. Only after checking out the expected earnings that our bot gets against all these bots can we determine our algorithm's strength.

The following tables measures HaRDBot's small bet gain per hand averaged over one thousand hands.

	SparBot	Poki	VexBot
HaRDBot	-.15 ± .1	.04±.1	-.23 ± .1

Although HaRDBot appears to have lost to SparBot and VexBot in a statistically significant way, it was a great victory to, even narrowly, beat Poki. Moreover our performance differences against SparBot and VexBot were hardly large, indicating that our poker bot was playing at a rather high level.

Our failure against VexBot demonstrates the primary weakness with HaRDBot. Our opponent modeling was the weakest part of our algorithm; HaRDBot effectively played statically (and not even in an estimate of a Nash Equilibrium). Therefore, VexBot was able to find exploitive strategies – the solution to this problem is for HaRDBot to do more opponent modeling, both to find weaknesses in the opposition strategy but also to update its own strategy, thus making opposing modeling less accurate.

Our corpus-based strategy for poker appears to have been rather successful. However, the corpus was, although large, only 200000 hands – because it is backed by a database, we could easily increase it to 2000000 hands without taking a performance hit. Moreover, HaRDBot suffered from a weak expert system. We devised it ourselves since Poki's is not completely available so it was probably somewhat subpar. Increasing the size of the corpus, optimizing the expert system further and doing more sophisticated opponent modeling would perhaps dramatically increase the success of our bot.

Sources:

1.) Schauenberg, Terence C. "Opponent Modelling and Search in Poker." *Opponent Modelling and Search in Poker*. University of Alberta, 2006. Web. 14 Nov. 2009. <http://poker.cs.ualberta.ca/publications/schauenberg.msc.pdf>.
2.) Billings, Darse. "Algorithms and Assessment in Computer Poker." *Algorithms and Assessment in Computer Poker*. University of Alberta, 2006. Web. 15 Nov. 2009.
3.) Papp, Denis. "Dealing with Imperfect Information in Poker." (1998): 1-9. *University of Alberta*. 1999. Web. 30 Nov. 2009. <http://poker.cs.ualberta.ca/publications/papp.msc.pdf>.
4.) We would like to extend our thanks to Ben Phillips, a Carnegie Mellon University CS major, class of 2012, and a semi-pro poker player who gave us considerable advice when designing the expert system.
5.) Sklansky, David. *The Theory of Poker*. Two Plus Two Pub., 1994. Print.
6.) "2009 Computer Poker Competition." *Department of Computing Science - University of Alberta*. Web. 11 Dec. 2009. <http://www.cs.ualberta.ca/~pokert/2009/index.php>.
Sklansky, David. *The Theory of Poker*. Two Plus Two Pub., 1994. Print.