Bayesian Range Estimator Bot

MIT Pokerbots Challenge: Andrew Kessler and Joseph Gross

Introduction

In this year's MIT Pokerbots Challenge, we developed a python-based bot to play a variant of no-limit heads-up Texas Hold Em called Swap Hold Em (SH). SH is a two-player game in which agents receive two hole cards and bet according to the strength of their hand, which can be made from the agent's hole cards and the communal cards available to both agents. In SH, unlike regular no-limit heads-up Texas Hold Em, there is a non-zero, independent probability of swapping each card in an agent's hole at all three flops of the communal cards. Depending on the value of said probability, this can change the game dramatically.

The central objective of our strategy was to achieve the best estimate of our equity against the opponent's range, or the expected value of our hand in the pot relative to the opponent's hand. Due to the natural information asymmetry in SH (we don't know our opponent's hand) and the randomness of swaps and draws, we cannot determine our equity with absolute certainty. We can, however, estimate equity from the board's structure and the opponent's actions, and act accordingly. When facing an opponent's bet, for example, if our equity is greater than the pot odds (our cost of continuation relative to the pot's size), we can comfortably call because the expected value of the call action is positive. We leveraged Monte Carlo simulation (MCS) and Bayesian logic to perform equity estimation.

Methods

For every hand, we first determined preflop equity with local memoization. There are 169 effectively distinct hole hands in SH[1], and for each hand we conducted a MCS of 100,000 iterations to determine the respective expected win rate. Our results were then saved in a CSV which is accessed prior to every hand in an effort to substitute computational time for memory allocation.

After determining our preflop equity, the bot then constructs a distribution of potential cards the opponent could possess, their range. To accomplish this, we store the MCS-estimated equity of each hand in the opponent's range in a python dictionary, with each hand key represented as a tuple, and methodically update the range in response to the opponent's actions. For example, a large raise action from the opponent will narrow their range to stronger hands, and check actions, particularly in position, will narrow their range to weaker hands. This 'narrowing' function is analogous to a Gaussian filter, whereby we refine our estimate of the opponent's range in proportion to the standard deviation of the win rates of the range.

```
if self.opp_range_mapping[potential_opp_hand_tuple] <=
get_mean_strength_from_range(self.opp_range_mapping) +
scaledMul*get_std_of_range_strengths(self.opp_range_mapping):
                    if potential_opp_hand in self.opp_range:
                        self.opp_range.remove(potential_opp_hand)
                        del self.opp_range_mapping[potential_opp_hand_tuple]
```

[1] Of the 13 ranks and 4 suits in poker, we can incongruously construct 13 combinations of pocket pairs, 78 combinations of suited cards, and 78 combinations of offsuited cards.

[2] Figure 1. Code snippet detailing the python implementation of our range update, scaled in proportion to the standard deviation of the range's component win rates. We simply delete the key-value (hand-win rate) pairs which are not consistent with the opponent's action. The scaledMul term corresponds to the strength of the action, such that higher raise actions, for example, result in a narrower estimate of the opponent's range.

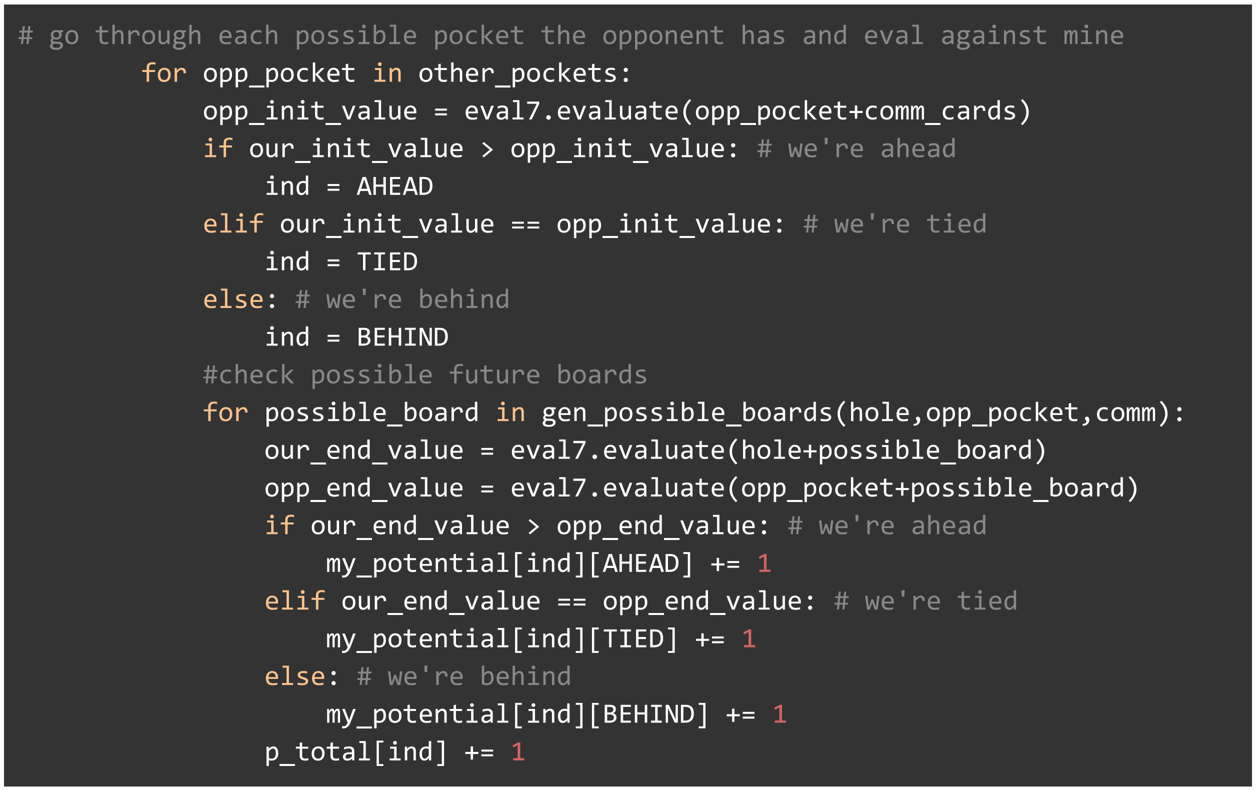
After updating our estimate of the opponent's range, we complete a Bayesian conditioning by determining the strength of our hand against the opponent's range. This is the most proximal step to determining our own action in the bot's workflow. We simply conduct another set of MCS against every hand in our estimate of the opponent's range. As indicated in the Introduction, if our equity is greater than the pot odds, we can justify calling. If it's significantly greater, the bot often implements a raise action. Notably, to prevent exploitation from other bots which could in principle outmaneuver pot-odds-based solutions, we also incorporate some randomness to our actions. Nevertheless, we weigh our actions according to their estimated expected value, so the bot probabilistically favors the 'right' decisions from an equity/pot-odds framework.

Results

The Bayesian Range Estimator Bot performed well against other bots, and particularly well against other pot-odds-based implementations, but suffered from computational limitations. Simply put, we just took too long on the tournament server. This is likely due to the repeated MCS which occurs for every hand in an opponent's range, after every range update. Because a range update occurs after every action of the opponent, significant range narrowing/filtering must occur at each stage in order to make MCS permissible under the tournament's time constraints. In order to rectify this, we introduced a number of features; these included increased randomness for inclusion of hands in the range (thereby resulting in a lower number of hand win rates to compute over the long term), fewer iterations for each MCS (from 100 to 20), and stricter range filtering by attributing more weight to the scaledMul term referenced in our Methods. We hope to further improve on the implementation side of bot.

Next Steps

In addition to implementation revisions, we also have a number of potential strategy developments. Chief among them involves accounting for equity realization factors, which capture the ability of a particular hand to realize, or actually capture, its equity. For example, while pocket deuces have higher equity than ace king offsuit preflop, the hand has poor equity realization as it can realistically win only a small subset of board constructions. Towards representing this factor, we wrote a MCS function to determine a given hand's potential. Specifically, for a given hole hand and subset of the five community cards, the bot creates a 3-by-3 matrix, with cells representing the number of possible boards where we end up ahead when first behind, and vice versa. This function definition was inspired by Hugh Pearse.[3]



```
# go through each possible pocket the opponent has and eval against mine
        for opp_pocket in other_pockets:
            opp_init_value = eval7.evaluate(opp_pocket+comm_cards)
            if our_init_value > opp_init_value: # we're ahead
                ind = AHEAD
            elif our_init_value == opp_init_value: # we're tied
                ind = TIED
            else: # we're behind
                ind = BEHIND
            #check possible future boards
            for possible_board in gen_possible_boards(hole,opp_pocket,comm):
                our_end_value = eval7.evaluate(hole+possible_board)
                opp_end_value = eval7.evaluate(opp_pocket+possible_board)
                if our_end_value > opp_end_value: # we're ahead
                    my_potential[ind][AHEAD] += 1
                elif our_end_value == opp_end_value: # we're tied
                    my_potential[ind][TIED] += 1
                else: # we're behind
                    my_potential[ind][BEHIND] += 1
                p_total[ind] += 1
```

[3] Figure 2. This details how we assign values to our matrix. For each pocket the opponent could possess, we simulate all possible board outcomes and use the eval7 python package to determine where we end up relative to the opponent's hand. We then update the matrix, and repeat until conclusion.

Another strategy addition we considered was adding our equity calculations to a comprehensive game state variable and deploying counterfactual regret minimization, a reinforcement learning-based method which is a current standard for two-player imperfect-information games. Essentially, after some necessary abstraction of SH,[4] we could assign regret values to each of our actions corresponding to how much we lose from such an action. The training objective of counterfactual regret minimization, then, would entail developing a policy to minimize some function of these regret values. We think that this strategy shows great promise.

Overall, the Bayesian Range Estimator Bot showed theoretical strength, but performed poorly under time constraints in practice. We hope to make improvements on this implementation side while iterating on our Bayesian strategy.

[4] Counterfactual regret minimization consists of an iterative traverse of the game tree. Because SH involves so many rounds with many possible actions, we'd need to abstractify SH in order to make the training feasible. We could, for example, bin similar hands under specific situations. For example, pocket deuces have similar equity to pocket threes on an ace king jack flop, so we could bin them together.