Monte Carlo Tree Search and Opponent Modeling through Player Clustering in no-limit Texas Hold'em Poker

A.A.J. van der Kleij

August 2010

Master thesis

Artificial Intelligence

University of Groningen, The Netherlands

Internal supervisors:
Dr. M.A. Wiering (Artificial Intelligence, University of Groningen)
Prof. Dr. L. Schomaker (Artificial Intelligence, University of Groningen)

university of groningen

Abstract

Texas Hold'em Poker is a challenging card game in which players have to deal with uncertainty, imperfect information, deception and multiple competing agents. Predicting your opponents' moves and guessing their hidden cards are key elements. A computer Poker player thus requires an opponent modeling component that performs these tasks.

We propose a clustering algorithm called "K-models clustering" that clusters players based on playing style. The algorithm does not use features that describe the players, but models that predict players' moves instead. The result is a partitioning of players into clusters and models for each of these clusters.

Learning to reason about the hidden cards of a player is complicated by the fact that in over 90% of the games, a player's cards are not revealed. We introduce an expectation-maximization based algorithm that iteratively improves beliefs over the hidden cards of players whose cards were not revealed. After convergence, we obtain models that predict players' hidden cards.

We built a computer Poker player using these algorithms and Monte Carlo Tree Search: an anytime algorithm that estimates the expected values of moves using simulations. The resulting program can hold its own in 2 player games against both novices and experienced human opponents.

Contents

Chapter 1

Introduction

The idea that we could build a machine that can beat us at our own games, has always been a fascinating one. Such ideas date back as far as the 18th century, when Baron Wolfgang von Kempelen exhibited the Maezal Chess Automaton – commonly known as *The Turk* – in Europe (Hsu et al., 1990). This "automaton" beat Chess players (including the likes of Napoleon Bonaparte and Benjamin Franklin) all around the world, drawing large audiences wherever it was publicly displayed. It was later uncovered as an elaborate hoax: the "automaton" was in fact a mechanical illusion, operated by a hidden human Chess expert (figure 1.1).

1.1 Artificial intelligence and games

It is this fascination amongst others, that turned computer science and early artificial intelligence researchers to the study of computer game playing. In the early 1950s, shortly after the invention of the first computer, initial efforts were already made to build computers that could play Checkers and Chess (Schaeffer and Van den Herik, 2002). Research into these topics continued, and for decades to follow, computer Chess would be the primary challenge in the field of artificial intelligence.

In the 1970s and 1980s, computers could not compete with human Chess experts, but remarkable progress had been made. At the time, computer Chess players based on *brute force* search techniques achieved an impressive level of play. As a consequence, the majority of research focused on developing faster data structures and search algorithms to further improve brute force performance.

From the late 1980s through the 1990s, many interesting things happened in the world of computer game playing. New techniques, modern computer hardware and innovative ideas led to a number of great breakthroughs. The first of these was the fact that Checkers was the first game to have a non-human world champion, after the program *Chinook* won the World Man-Machine Championship (Schaeffer et al., 1992). Chess followed shortly, when in 1997 50 years of hard scientific work in the field of computer Chess reached its apotheosis when IBM's *Deep Blue* defeated the then-reigning world chamption Garry Kasparov.

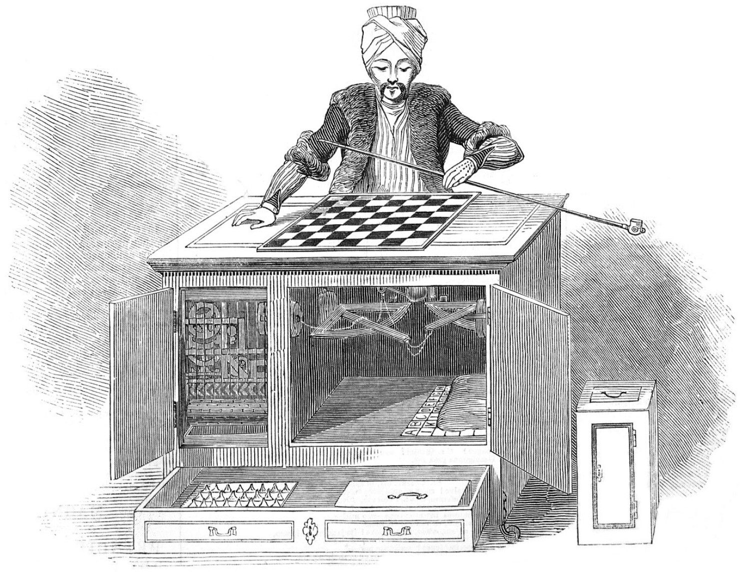


Figure 1.1: The Maezal Chess Automaton, better known as The Turk. This chess-playing "automaton" was displayed all around the world as the world's first chess playing machine in the 18th century. It was later uncovered as a mechanical illusion: it was secretly operated by a hidden human Chess expert.

1.1.1 Types of games

Chess is a *deterministic* game of *perfect information.* Essentially, this boils down to the fact that there is no element of chance and no deception in Chess. The players cannot hide anything from one another and the state of the game is solely determined by the sequence of moves of the players. To categorize games, we can use such a distinction between deterministic versus stochastic and perfect information versus imperfect information. The differences between these properties of games are as follows:

- **Perfect information** vs. **imperfect information**: in perfect information games, every player can observe the complete state of the game and every player thus has the same information. Examples of such games are Chess, Checkers and Go, in which the board and all players' pieces are visible to all players. In imperfect information games, some elements of the game are invisible to players, effectively introducing uncertainty about the state of the game. Most card games are games of imperfect information, as players have a set of cards that only they can see. An example of such a card game in which players have hidden cards is Texas Hold'em Poker.

- **Deterministic** vs. **stochastic**: in a deterministic game, the next state of the game is solely determined by the current state of the game and the action taken. An example of such a game is Chess, where the next position depends only on the current position and the player's move. In contrast, in stochastic games there are stochastic events that affect the new state of the game. For example, games that involve throwing dice or dealing cards are typically stochastic: the result of the stochastic event determines the next state of the game, and the players have no absolute control over what is going to happen.

These two properties allow us to categorize games in each of four combinations of these two properties. An overview of these categories and examples of some games in each of them is available in table 1.1.

Table 1.1: Some examples of games for each of the four combinations of perfect versus imperfect information games and stochastic versus deterministic games.

	Perfect information	**Imperfect information**
Deterministic	Checkers, Chess, Go	Battleship, Mastermind, Stratego
Stochastic	Backgammon, Blackjack, Monopoly	Poker, Risk, Scrabble

In a nutshell, the type of reasoning that is required for deterministic games is less complex than for stochastic games: the uncertainty introduced by stochastic events complicates the game and the decision-making process. Similarly, making a good decision in a perfect information game is typically easier than in a game of imperfect information, as no reasoning about the hidden elements of the game is required. Chess thus is a game of the "easiest" category when it comes to reasoning, as it is a deterministic game with perfect information.

As the grand challenge of computer Chess has essentially been completed since computers are beating the human grandmasters, researchers in artificial intelligence started looking for more complex games to study. Tesauro (1994) describes one of the most remarkable results of these studies: *TD-gammon* is a computer player for the stochastic, perfect information game of Backgammon that plays at human expert level. Similar success has been achieved for deterministic games of imperfect information, which can usually be approached with brute-force search methods. The most complex category of stochastic, imperfect information games remains as the main challenge for computer game playing.

1.2 Poker

One of the games of this category is the game of Poker. Poker is a family of card games with many variants that all share the concepts of betting rules and hand rankings. From the late 1990s till now, Poker has enjoyed a great increase in popularity. Online casinos offering Poker games started to emerge, television channels started

Table 1.2: The ten possible Poker hand categories. The worst Poker hand is a High card hand, whereas the best Poker hand is a Royal Flush.

	Hand	**Example**	**Description**
1	Royal flush	T♠ J♠ Q♠ K♠ A♠	A straight flush T to A
2	Straight flush	2♠ 3♠ 4♠ 5♠ 6♠	A straight of a single suit
3	Four of a kind	A♠ A♣ A♡ A♢ 5♠	Four cards of the same rank
4	Full house	A♠ A♣ A♡ K♠ K♣	Three of a kind + one pair
5	Flush	A♠ 4♠ 7♠ 9♠ K♠	Five cards of the same suit
6	Straight	2♠ 3♣ 4♣ 5♡ 6♢	Five cards of sequential rank
7	Three of a kind	A♠ A♣ A♡ K♠ Q♣	Three cards of the same rank
8	Two pair	A♠ A♣ K♠ K♣ 3♡	Two pairs
9	One pair	A♠ A♣ K♠ Q♣ 3♡	Two cards of the same rank
10	High card	A♠ 3♣ 7♣ 9♡ Q♢	None of the above

broadcasting Poker games and tournaments and the number of players greatly increased. The Poker variant of choice in both offline and online cardrooms, casinos and home games is no-limit Texas Hold'em.

1.2.1 Texas Hold'em Poker

In Texas Hold'em Poker, each player is dealt two private *hole cards* face down from a single standard deck of cards. The players use these cards to compose a five-card Poker *hand*. Each five card hand belongs to one of ten possible hand categories that determine its strength as compared to other hands. A listing of these ten hand categories is provided in table 1.2.

In a Poker game, one player is the dealer. This designation shifts one position clockwise every game and determines the order in which the players act. The game starts with the two players left of the dealer putting in the *small blind* and *big blind*. These are forced bets that effectively put a cost on sitting at the table: if there were no blinds, a player could just sit around and wait for the very best starting hand before playing.

When the blinds have been paid, the *pre-flop* betting round starts. During a betting round, the players in the game take turns in clockwise rotation. When there is no current bet, a player may *check* or bet. If this is not the case, a player may *fold*, *call* or *raise*. The effects of these actions are as follows:

- **Fold**: the player does not put in any more money, discards his cards and surrenders his chance at winning the pot. If he was one of two players left in the game, the remaining player wins the pot.

- **Call**: the player matches the current maximum amount wagered and stays in the game.

- **Raise**: the player matches the current maximum amount wagered and additionally puts in extra money that the other players now have to call in order to stay in the game.
- **Bet**: this is similar to a raise. When a player was not facing a bet by an opponent (and the amount to call consequently was 0) and puts in the first money, he bets.
- **Check**: this is similar to a call, except the amount to call must be 0: if noone has bet (or the player has already put in the current highest bet, which could be the case if he paid the big blind) and the player to act does not want to put in any money, he can check.

The betting round is finished when all players have either called the last bet or when there is only one player left (which is also the end of the game). When the pre-flop betting round is finished, the game progresses to the *flop* betting round and three *community cards* are dealt face up. When this round has finished, a fourth community card is dealt face up on the turn and another round of betting follows. The fifth and final betting round is the *river*, on which the fifth and last community card is dealt. When the river betting round has completed and two or more players are still in the game, the *showdown* follows.

Showdown

In the showdown, all remaining players reveal their hole cards. The player with the best five card Poker hand wins the pot. If both players have an equally strong hand, they split the pot.

Each player's five card Poker hand is formed using the player's private hole cards and the five community cards. A player can use none, one or both his hole cards to make a Poker hand. A Poker hand always consists of exactly five cards and is in one of the Poker hand categories that range from *high card* to *royal flush* (table 1.2). This category primarily determines the strength of a Poker hand: the player with the highest category Poker hand wins the pot. Should two players have a hand in the same category, then the ranks of the five cards in the Poker hand determine the winner.

Limit versus no-limit

In limit Poker, the amount a player can bet or raise is fixed: a player has no freedom to choose the amount of money he wants to put in. In contrast, in no-limit Poker a player can bet or raise any amount, as long as it meets the minimum required amount. The latter variant adds a lot of complexity and strategic possibilities to the game: a player no longer has at most three possible moves, but an enormous number of possible actions to choose from.

Example game

Let us now consider an example game of Texas Hold'em Poker to illustrate how the game is played. The example game is a 6 player game with a small blind of \$1 and a big blind of \$2. The cards are dealt, the player left of the dealer pays the small blind and the player left of him puts in the big blind of \$2.

Player A is the player left from the small blind, who is first to act and raises to \$8 with T♡T♢. One player folds, and player B calls the raise of \$8 with A♣9♣. The dealer folds and the two players who paid the blinds fold too. This concludes the pre-flop round, and we continue to the flop with a pot of $1 + 2 + 8 + 8 = 19$ dollar.

The flop comes 7♡2♣4♣, giving player A an *overpair* (a pair higher than the highest community card) and player B a flush draw (four cards to a flush) and two overcards (cards higher than the highest community card). Player A is first to act on the flop (because player B is closer to the dealer seat in clockwise rotation and therefore gets to act last) and makes a bet of \$15. Player B has \$70 dollar left on the table and decides to raise *all-in*: he puts in all his remaining money. Player B now has to call \$55 and decides to do so.

The turn and river cards are dealt and the final community cards are 7♡2♣4♣7♣T♠. Note that there are no betting rounds on the turn and river, as player B is all-in, and can therefore proceed to showdown without putting in any more money. In the showdown, player A reveals his A♣9♣ hole cards that together with the community cards make A♣9♣7♣4♣2♣, a flush. Player B then shows his T♡T♢ for a full house tens over sevens (T♡T♢T♠7♡7♣). Since a full house beats a flush, player B wins the pot of \$159.

1.3 Why study games?

So far, we have briefly discussed the history of games in the field of artificial intelligence, different types of games and Poker, which currently still proves to be very challenging for Computers. We have seen that remarkable success has been achieved in computer game playing and that some great challenges still remain to be completed in the future. What we have not discussed yet, is why games are interesting to study for artificial intelligence in the first place.

One of the primary reasons why games are so interesting to study, is that many real life situations can be modeled as games. Games can contain elements of negotiation, bidding, planning, deception and many other skills that are frequently used in real life. Studying how computers can play games, allows us to gain insight in how computers might mimic the skills that human players exhibit while playing games. The techniques developed in computer game playing can then hopefully be used in different applications.

Furthermore, the study of computer game playing is not only beneficial for computer science, it is also beneficial for human players of these games. They gain sparring partners that are available for a game any time of the day, don't mind starting over in the middle of a game when they are winning, can be adjusted to play a little weaker, and so on. As such, artificial players are a great practice tool and source of entertainment for human players.

1.3.1 Why study Poker?

As we have seen before, Poker is a non-deterministic game of imperfect information. Computers have had only limited success in games of this type so far, whereas they are playing at human grandmaster level in many deterministic or perfect information games. Poker as such poses a very challenging challenge with a large number of interesting properties. Billings et al. (2002) summarizes these properties as:

> *"[Poker's] properties include incomplete knowledge, multiple competing agents, risk management, opponent modelling, deception and dealing with unreliable information. All of these are challenging dimensions to a difficult problem."*

1.4 Contributions of this thesis

The great majority of published work on computer Poker has focused on the game of 2 player limit Texas Hold'em. This game has the beneficial property that the game tree is relatively small in comparison to other variants of Poker (yet is still very large in absolute numbers). The downside of studying this game is that the number of human players is limited: the current Poker game of choice is no-limit Texas Hold'em with 2 to 10 players. Some initial efforts have been made at artificial no-limit Texas Hold'em intelligence, and we would like to further investigate this game in this thesis.

The majority of recent work on limit Texas Hold'em Poker aims to find "optimal" ϵ-equilibrium strategies that are (nearly) inexploitable. The aim of such strategies is not to maximally exploit the opponent's mistakes, but to play in such a way that it is impossible to defeat it. While this is a valid and interesting approach, we will describe work towards using opponent modeling to maximally capitalize on the opponent's mistakes in order to win as much money as possible. Such an approach is interesting because Texas Hold'em Poker is a very complex game and many opponents will make major strategic errors: maximally exploiting such errors is very protitable. While such "maximal" approaches have been studied in the past for limit Texas Hold'em Poker, we feel that a lot of work remains to be done on this subject.

Applying machine learning algorithms on Poker games in order to learn models that describe a specific opponent's strategy is difficult because of data sparsity: we need accurate models as soon as possible, or we will have lost a lot of money by the time that we have accurate models. Previous work tries to resolve this problem by learning generic models that model "the average opponent'. While such an approach does not suffer from data sparsity, the resulting models are not very specific and will not describe most opponents very accurately. We propose a solution that aims to combine the best of both worlds by automatically clustering different playing styles observed in a database of Poker games. We can then learn accurate models for each of these playing styles. The task of accurately modeling a specific opponent during live play then reduces to figuring out which playing style most closely resembles the specific opponent's strategy.

Learning to reason about the hidden cards of a player is complicated by the fact that in over 90% of the games, a player's cards are not revealed. Most currently published work on opponent modeling in Poker resolve this by "postponing" reasoning about the hole cards until a showdown occurs in the game tree: during play prior to a showdown, the computer player has no idea whatsoever about the hole cards of its opponent(s). While this is a valid approach, it does not resemble the way human Poker experts play: they are constantly guessing the opponents' cards during a game and updating their beliefs after every move. We will present work that aims to mimic this behavior in an artificial Poker player.

1.5 Research questions

We can summarize the goals of this thesis as follows. We intend to develop methods for (I) an artificial player for the game of no-limit Texas Hold'em Poker, that uses (II) opponent modeling (III) for clusters of players in order to (IV) maximally capitalize on its opponents' mistakes. We can formulate these goals in a research question as follows:

> *How can we develop an artificial player for the game of no-limit Texas Hold'em Poker that uses opponent modeling for clusters of players in order to capitalize on its opponents' mistakes, and how do the individual components that compose the system and the system as a whole perform?*

This research question can be further specified by splitting it in a number of sub-questions:

1. How can we identify clusters of playing styles in databases of example games and what is the quality of the resulting partitioning?

2. How can we extract models that estimate the values of hidden information in imperfect information games from partially labeled data?

3. How can we evaluate the value of moves in non-deterministic games of imperfect information with very large game trees in limited time?

1.6 Structure of this thesis

In the next chapter, we will discuss some background knowledge that we will build on in the remainder of this thesis, along with relevant previous work on computer

Poker. Chapter three covers Monte Carlo methods in general and the Monte Carlo Tree Search (MCTS) algorithm for game tree search.

We propose a number of algorithms that may be used to (I) find clusters of player types in Poker games and (II) learn opponent models that predict players' moves given his cards. The former will be detailed in chapter four, whereas the latter will be discussed in chapter five.

In chapter six, we will consider experiments that we conducted on a complete computer Poker agent based on the proposed methods in this thesis. The computer Poker player is a complete implementation of Monte Carlo Tree Search and the algorithms and components that we will discuss in chapters four and five.

Chapter seven is the final chapter of this thesis and contains a thorough discussion of our findings, some directions for future work and a conclusion.

Chapter 2

Background

In this chapter we will discuss some background that will be built on in the remainder of this thesis. First, we will consider some general topics on computer game playing and machine learning. The final part of this chapter will summarize some important previous work that has been published on (computer) Poker.

2.1 Game theory: solving games

In this section we will discuss the related game-theoretic concepts of *solved games* and *optimal strategies*. In game theory, we consider a game solved when we know its outcome given that all players play an optimal strategy. An optimal strategy is a strategy that yields the best possible outcome, regardless of the response by the opponent. For example, an optimal strategy for the game of roshambo[1] is to always pick any option with a uniform probability (1/3). This strategy is unexploitable: there is no strategy that can defeat it in the long run. While this is obviously a great property, the downside of such an optimal strategy is that it does not exploit any weaknesses in the opponent's strategy either. The strategy cannot win from the extremely exploitable strategy of always playing rock, for example.

It is obvious that the long term result of two players playing the optimal roshambo strategy of uniform random choices, is a draw. This means that the game-theoretical *value* of the game roshambo is a draw. In game-theory we consider the game solved, since we know its value. Allis (1994) distinguishes three degrees to which a game may be solved:

- **Ultra-weakly solved**: we know the value of the initial position(s).
- **Weakly solved**: we know both the value of initial position(s) and a strategy to obtain at least the value of the game from these positions.

[1]Roshambo (or rock-paper-scissors) is a game in which players simultaneously pick one of three gestures: rock, scissors or paper. Any of these choices defeats exactly one of the others, making them equally strong. The winner is chosen according to the following rules: rock defeats scissors, scissors defeats paper, and paper defeats rock.

- **Strongly solved**: we know both the value and a strategy to obtain at least this value for all legal positions.

Note that all of these have the hidden additional requirement of reasonable resources. That is, we cannot say that a game is solved when we only know an algorithm that will generate a solution given infinite time. Instead, a game is solved to some degree when one of the abovementioned requirements can be met given reasonable resources. The techiques to find solutions to games are well known, the only reason why not every game has been solved yet is that our resources are limited and some games are just too complex to solve within reasonable time.

2.1.1 The prisoner's dilemma

The prisoner's dilemma is a thoroughly studied problem in game theory that illustrates why it may be theoretically optimal for two people not to cooperate even it in both their best interest to do so. Wooldridge (2002, pp. 114-115) phrases the problem as follows:

> "Two men are collectively charged with a crime and held in separate cells. They have no way of communicating with each other or making any kind of agreement. The two men are told that:
>
> 1. If one of them confesses to the crime and the other does not, the confessor will be **freed**, and the other will be jailed for **three** years.
> 2. If both confess to the crime, then each will be jailed for **two** years.
>
> Both prisoners know that if neither confesses, then they will each be jailed for **one** year."

There are four possible outcomes: i confesses, j confesses, both confess or neither confess. For convenience, let us from now on refer to confessing as defecting and to not confessing as cooperating. We can illustrate these possible outcomes and the associated rewards for both players in a *payoff matrix* (table 2.1)[2].

Table 2.1: The payoff matrix for the prisoner's dilemma. Since the prisoner's dilemma is about punishment, most payoffs are negative: a long jail time equals a negative reward.

	i **defects**	i **cooperates**
j **defects**	$j = -2, i = -2$	$j = 0, i = -3$
j **cooperates**	$j = -3, i = 0$	$j = -1, i = -1$

[2]Note that a payoff matrix illustrates positive payoffs or rewards that a player is happy to receive. Since the prisoner's dilemma is about jail sentences or punishment, the jail times have been converted to negative numbers: a long jail time equals a negative reward.

Note that neither defecting nor cooperating is always best: the results will depend on what the other prisoner decides. If we calculate the minimum guaranteed payoff for both strategies however, we find that defecting guarantees a minimum payoff of -2, whereas cooperating yields a minimum guaranteed payoff of -3. From a game-theoretic point of view, this observation implies that the rational thing to do in order to secure the greatest guaranteed payoff, is to always defect. The reason for this is that in game theory every agent is assumed to act optimally and we therefore look to maximize guaranteed payoffs. And since defecting yields us a guaranteed payoff of at least -2, it is preferred over the guaranteed payoff for cooperating, which is -3.

When a prisoner choses to defect, he will always do just as good as the other prisoner and possibly better, should the other prisoner cooperate. If either prisoner is defecting, the other prisoner should defect as well and he has nothing to gain by choosing any other strategy than defect. In game theory, such a stable state in which no player can gain by changing only his own strategy is called a *Nash equilibrium*. So, in the prisoner's dilemma, the set of strategies for both players that constitutes a Nash equilibrium is (*defect*, *defect*).

2.1.2 Exploitability

As we have seen, when a player is playing an equilibrium strategy, his opponents cannot gain anything by deviating from the equilibrium strategy. An equilibrium player's opponents can thus expect to do at best as well as the equilibrium player: the equilibrium player cannot be defeated and is *inexploitable*. Note that this does not imply that the equilibrium player will always win. As an example, the optimal, inexploitable strategy for Roshambo is to randomly pick any action with uniform probability as we have seen. This strategy is undefeatable as there is no strategy that can expect to beat it in the long run. But the random strategy cannot defeat *any other* strategy, not even the highly exploitable strategy of always playing the same single action.

We could say that equilibrium strategies or optimal strategies are *defensive* in nature. There is no strategy that can expect to defeat them, but they might not be very good at defeating other strategies. Playing an optimal strategy is therefore not always the best choice: if you know your opponent plays an extremely exploitable strategy such as always picking rock in Roshambo, you should probably play a strategy that is tailored towards maximally exploiting this strategy. Note that this implies that you have to deviate from the optimal unexplotable strategy and will be exploitable yourself.

2.2 Game tree search

2.2.1 Game trees

One commonly used technique in computer game playing is game tree search. In game tree search, moves and resulting game states are modeled as a game tree. In

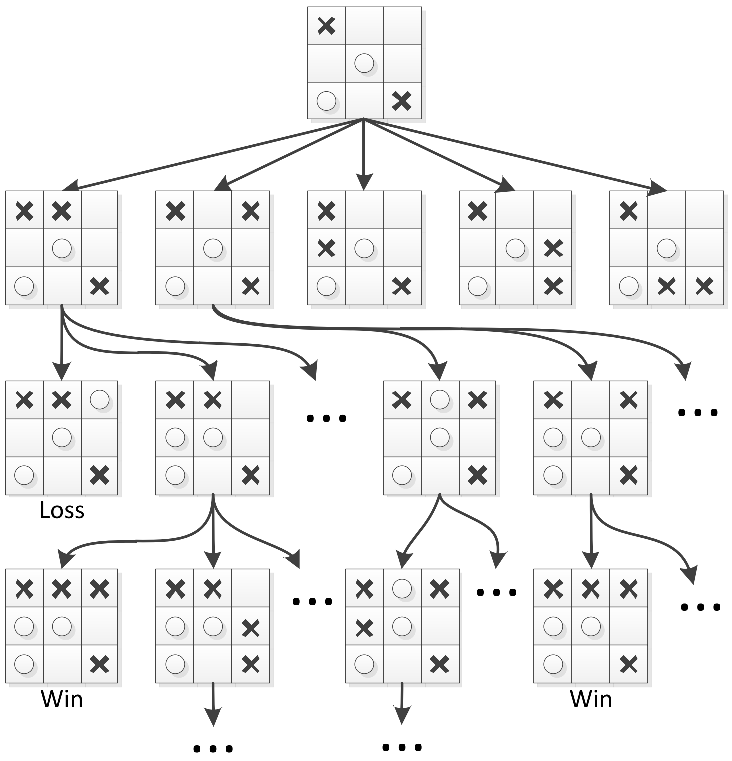


Figure 2.1: A (partial) game tree for the game of Tic-Tac-Toe. The X-player is to act at the root node. Nodes represent positions in the game and edges represent moves.

these trees, states (positions) of the game are modeled as nodes, and moves that players may make in these states are modeled as edges from these nodes, leading to other game state nodes. At the root of the game tree is the current state the game is in.

Figure 2.1 illustrates this idea for the well-known game of Tic-Tac-Toe. The X-player made the first move and therefore is to act in the game state for which the game tree is shown. Since there are five open squares, there are five possible moves for the X-player and thus five edges connecting the root node to the nodes representing the game states reached by making each of these moves. Play alternates between the X-player and the O-player taking turns until a leaf node – in which either player has three in a row or all squares are filled – is reached. Leaf nodes in the partial game tree shown in the figure have been marked "Win" or "Loss" accordingly, from the perspective of the X-player.

Game trees allow computers to reason about moves and their consequences. They clearly model which moves are possible in each game state and which new states may be reached by making any of these moves. There are several search algorithms that use game trees to decide on what move to play. The archetype of these algorithms is the well-known minimax algorithm.

2.2.2 Minimax

The minimax algorithm is an algorithm for finding an optimal strategy in a certain game state for deterministic two-player zero-sum games. The basic idea of the algorithm is that both players are assumed to play *optimally.* The optimality assumption implies that when the *MAX*-player (the player for which we are trying to find the best move) is to act, he will always select the move that yields him the largest utility eventually. Conversely, the *MIN*-player will select moves that will result in the lowest utility for the *MAX*-player. Since we know the values for leaf nodes (as these are simply their assigned utility from the perspective of the *MAX*-player), we can now use the following recursive definition to calculate the minimax value for all nodes in the tree (Russell and Norvig, 2002, p. 163):

$$\text{Minimax}(n) = \begin{cases} \text{Utility}(n) & \text{if } n \text{ is a leaf node} \\ \max_{s \in \text{Children}(n)} \text{Minimax}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Children}(n)} \text{Minimax}(s) & \text{if } n \text{ is a MIN node} \end{cases} \tag{2.1}$$

In this equation, Minimax(n) is the minimax value for node n, Children(n) is the set of child nodes of node n and Utility(n) is the utility of leaf node n. This function can be directly translated to a simple recursive algorithm that searches the game tree depth-first. When a leaf node is reached, the resulting utility is backed up the tree. Search terminates when the value of the root node has been calculated, which is by definition when every single node of the tree has been evaluated.

Figure 4.1 illustrates the minimax algorithm. Note that at *MAX*-nodes (circles), the largest of the values of the node's children is propagated. Conversely, at *MIN*-nodes (diamonds) the smallest of the values of the node's children is propagated. The illustration shows that apparently the best result the *MAX*-player can achieve is a draw (value 0), assuming his opponent plays optimally.

Note that when the assumption that the opponent plays optimally (and thus selects the move with the lowest utility) is violated and the opponent plays suboptimally, there may be strategies that perform better against this suboptimal opponent. These strategies will necessarily do worse against opponents that do play optimally however (Russell and Norvig, 2002, pp. 163-165).

2.2.3 Expectiminimax

The minimax algorithm is very basic and only applies to deterministic games with two players. Many games are not deterministic however and contain some element of chance, such as drawing cards or throwing dice. If these events might occur after the current position, we cannot use minimax to find an optimal solution and need to extend it to accommodate the stochastic events.

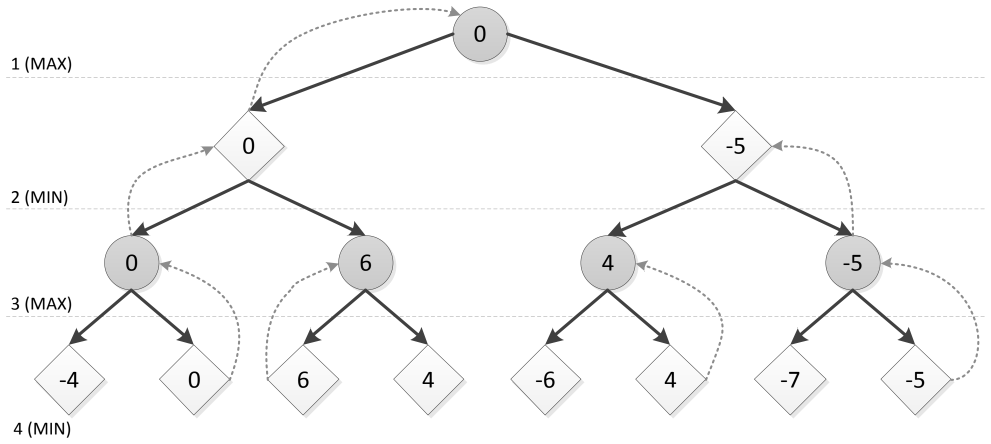


Figure 2.2: A minimax tree. The value of max nodes is equal to the largest of the values of its children. Conversely, the value of min nodes is the smallest of its children. Using this tree, we find that the best result for the *MAX*-player given that his opponent plays optimally, is a draw.

The game tree of a non-deterministic game contains a new type of nodes next to *MAX*-nodes and *MIN*-nodes: chance nodes. At chance nodes, one of its child nodes s is chosen with some probability $P(s)$. Chance nodes for rolling a single die would have six children with uniform probabilities $\frac{1}{6}$, for example.

We can now extend minimax to find optimal strategies in non-deterministic two-player games whose game trees contain chance nodes. For chance nodes, we define their values to be the weighted average or expected value over the values of their children. The extended version of minimax is called expectiminimax (Russell and Norvig, 2002, pp. 175–177). The node values for expectiminimax are defined as:

$$\text{Expect}(n) = \begin{cases} \text{Utility}(n) & \text{if } n \text{ is a leaf node} \\ \max_{s \in \text{Children}(n)} \text{Expect}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Children}(n)} \text{Expect}(s) & \text{if } n \text{ is a MIN node} \\ \sum_{s \in \text{Children}(n)} P(s)\text{Expect}(s) & \text{if } n \text{ is a chance node} \end{cases} \tag{2.2}$$

2.2.4 Searching large game trees

One problem with game tree search methods such as minimax and expectiminimax is that they search depth-first: search is proceeded all the way to the leaves of the tree, where the result is backed-up and search continues from a higher level. The great difficulty here is that the value of a node can only be determined accurately once all its child nodes have been evaluated. This implies that the results of the algorithms are only accurate when the entire search has completed. For games such as Tic-Tac-Toe this requirement poses no problems, as the complete game tree from any position may be searched in the blink of an eye. For other games with larger game trees, such as Go and No-limit Texas Hold'em Poker, the game trees are enormous and searching them completely in reasonable time is simply impossible with modern technology.

In attempts to overcome this problem, a number of techniques have been developed that aim to either (i) reduce search times by eliminating the need to search certain branches of the game tree or (ii) estimate the values of nodes instead of calculating them exactly. The former is usually referred to as *pruning*. We will discuss one frequently used type of pruning called Alpha-beta pruning, that has led to great success in games such as chess and checkers, in the next section. The latter will be discussed in section 2.2.4.

Alpha-beta pruning

Alpha-beta pruning is a technique that aims to reduce the number of nodes that has to be evaluated in minimax search. It is a very useful extension of minimax, because the results of the search are unaltered and thus optimal, while it may significantly reduce search times. The trick is that we can stop evaluating a node when we already know that it will always be worse than a previously examined move higher in the tree. There is then no point in searching the children of the node as we already know it will never be played.

To be able to perform alpha-beta pruning, two additional parameters α and β are introduced that describe the bounds on the backed-up values in the tree. They describe the current best move along the path for the MAX-player and MIN-player, respectively. These parameters are updated during the traversal of the game tree and are used to check if further evaluation of nodes is required. Let us now consider an example of the application of alpha-beta pruning.

Figure 2.3 illustrates alpha-beta pruning for a minimax game tree. We are interested in the value of node A, which is a MAX-node. At step a, we find that the first leaf node has a utility of 7. Since node B is a MIN-node, we can now infer that its value will be *at most* 7. At step b, we find that the next leaf node has a utility of 8, which does not change the range of values for node B. At step c we discover that node B's last child node has a utility of 4. Since node B is a MIN-node, we now know its value will be at most 4. But since we have now inspected all of its children, we now its value is actually exactly 4. We can now update the bounds for node A: since it is a MAX-node, it will be at least 4. At step d we find that node C's first child has a utility of 1. Since node C is a MIN-node, we now know that its value will be at most 1. And this is where alpha-beta pruning steps in: since we know that the value of node A is at least 4 and the value of node C will be at most 1, we already know that node C will never be selected and we can stop evaluating its children. We have pruned a part of the tree, reducing its size and thus the time that is required to search it.

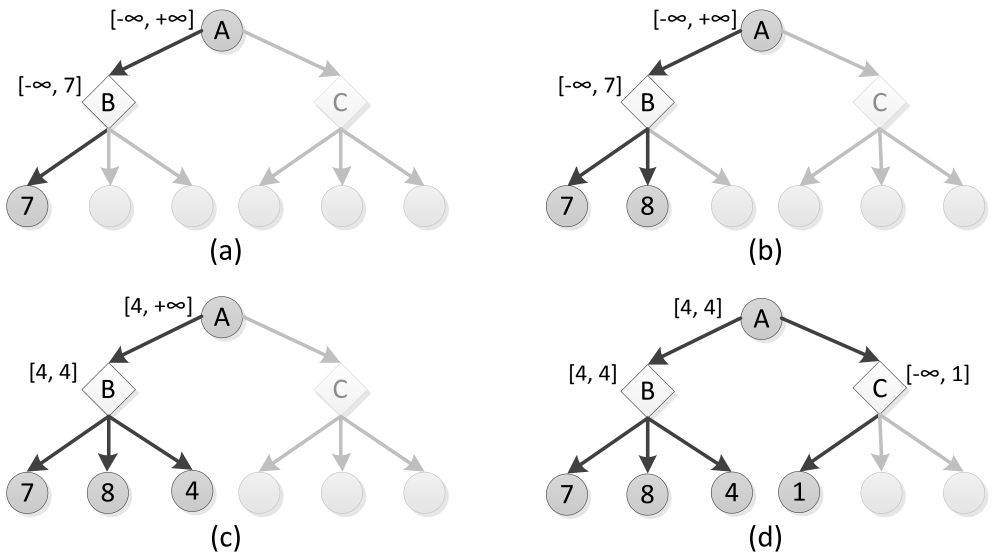


Figure 2.3: An example of alpha-beta pruning. Steps a through d show the four steps required to find the minimax value of node A. At step d we know that the value of node C will be at most 1. Since we also know that the value of node A will be at least 4, we can conclude that node C will never be selected and we can stop evaluating its children, pruning a part of the tree.

Limiting search depth

If pruning does not reduce the search tree to reasonable proportions, making exhaustively searching the tree impossible, we will have to find other means to extract strategies from game trees. One solution is to limit the depth of the search to some constant level. When the search reaches this depth, the values of nodes are no longer determined by exhaustive evaluation of all of their descendents, but are directly estimated instead. This means that the tree search algorithm can save valuable time by not considering the descendents of nodes at the maximum search depth. Some handcrafted or machine-learned evaluation function is used to directly assess the value of the node instead.

While this approach solves tractability problems to some extent, it also introduces a number of new difficulties. The definition of the evaluation function is one of them. It is often not trivial to come up with a good evaluation function that accurately depicts how favorable a position is. Since the evaluation function determines the playing strength of the resulting algorithm, it has to be spot on. One solution to this problem is to apply machine learning techniques to learn a value function automatically. Such an approach is often used in reinforcement learning, which we will discuss in section 2.3.1.

Another problem with limiting search depth is the horizon effect. The horizon effect arises when some move that causes significant damage to the MAX-player's position is inevitable but may be postponed for some time. An algorithm with limited search depth will select moves that avoid the damaging move for some time, pushing the inevitable hurtful move ”over the horizon” of the search. When the algorithm cannot see the move any longer because of its limited search depth, it will think it has avoided it while in fact it is only postponing the problem.

Monte Carlo tree search

Monte Carlo methods are a class of algorithms that rely on random sampling or simulations to approximate quantities that are very hard or impossible to calculate exactly. We can apply them to game trees that are too large to search exhaustively in reasonable time as well. Instead of completely evaluating (large parts of) game trees, we could try to simulate how the game will continue when we pick certain moves. The outcomes found in these simulations may then be used to assess the quality of the available moves. We will discuss Monte Carlo methods to search game trees in much greater detail in chapter 3.

2.3 Machine learning

Machine learning refers to the scientific field that is concerned with algorithms that allow computers to learn. Learning in this sense refers to learning in the broadest sense: any algorithm that allows a computer to improve its performance based on past experience may be considered machine learning. Mitchell (1997, p. 2) defines it as follows:

> ”A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.”

We can divide machine learning methods in categories based on the type of feedback available for learning. The field of machine learning generally distinguishes between supervised learning, unsupervised learning and reinforcement learning.

In supervised learning methods the goal is to learn some output function, given example pairs of inputs and desired outputs. For example, an insurance company might be interested in a system that could learn to decide whether it should accept someone as a new client or not. Supervised learning methods could be used to extract knowledge from a number of examples of applications and corresponding decisions that were made in these cases. Learning to label the input as one of a number of options (either ’accept’ or ’decline’ in the previous example) is known as *classification.*

In unsupervised learning, there are only unlabeled inputs and no desired outputs. The goal then is to find interesting patterns in the inputs. One typical example of unsupervised learning is clustering. In clustering, we’re trying to group together similar inputs into clusters based on some criteria.

In reinforcement learning, the only feedback available are rewards that are offered in some states. The system's goal is to choose actions in states to maximize its total long-term reward. We will discuss reinforcement learning in more detail now. Then, we will have a closer look at both supervised and unsupervised learning.

2.3.1 Reinforcement learning

Reinforcement learning (Sutton and Barto, 1998) is a machine learning technique concerned with how to take actions in an environment to maximize some long-term reward. The only feedback given by the environment are rewards for reaching certain states. The basic formulation of reinforcement learning is that there is some agent operating in an environment. The agent perceives some input state or pattern and produces an action for that state. The environment then responds by awarding the agent some reward that indicates how good or bad his choice was. The goal for the agent is to generate actions to maximize the rewards it is given. Often, the rewards are delayed in which case most of the states give a neutral reward of 0 and some terminal states give either a positive or negative reward.

These properties make reinforcement learning particularly suitable for learning how to play games, in which there typically are a limited number of clearly defined states and associated rewards. As an example, in Chess there are a great number of positions that offer no reward, but every game ends in some terminal position in which either a positive award (for winning), no reward (for draws) or a negative reward (for losing) is given. Playing good Chess then boils down to selecting actions that maximize the reward obtained at terminal states.

Delayed rewards introduce a difficulty called the *temporal credit assignment* problem (Tesauro, 1995). The problem is caused by the fact that delayed rewards have to be attributed to past actions that led to reaching the current state. We will somehow have to decide which actions contributed most to reaching the state in which the reward or punishment was received.

Tesauro (1994) used a solution to the temporal credit assignment problem called *temporal difference (TD)* (Sutton, 1988) learning to learn to play Backgammon. The resulting program *TD-Gammon*, is one of the most notable and successful applications of reinforcement learning to games. TD-Gammon plays at human grandmaster level and has actually changed some of the believes held by human Backgammon experts. For example, most human experts have changed some standard opening plays to match those of TD-Gammon as these plays are now believed to be superior to the earlier standard plays.

2.3.2 Supervised learning

In supervised learning, the goal is to deduce some function that accurately predicts desired outcomes from training data. The training data must be labeled: it must consist of pairs of inputs and corresponding desired outputs. We call this type of learning *supervised* because there is some "supervisor" that labeled the training data with desired outputs. The learner has to *generalize* from the training data to previously unseen inputs.

Inputs to supervised learning systems are often *feature vectors*. A feature vector is simply a list of values of *features*. A feature is a measurable description of some property of the input. Consider for example an insurance company that wishes to automate the process of deciding whether an application of a potential new client should be accepted or rejected. The training data that they have available is a large number of scanned paper applications and the corresponding decisions made in those cases. Because computer programs typically cannot interpret scanned documents unless a complicated system has been developed to do so, the documents will need to be transformed to some machine-readable representation so that supervised learning algorithms can operate on them.

For example, features that may be used to describe an application in the insurance company example are the age, sex or salary of the applicant. Those are features that might be meaningful in deciding whether to accept or reject the application. Features will often be numeric, but they need not be. Features may as well be nominal or ordinal, although not all machine learning methods can handle those well.

The desired outputs are typically either a continuous number or a class label. In the former case we speak of *regression*. In the latter case we speak of *classification*.

Classification

In classification, training data consists of inputs with desired class labels. The goal is to learn a function that accurately assigns class labels to previously unseen inputs. For example, an insurance company might be interested in an automated system that can accept or reject potential clients' applications. The training data for such a system would be pairs of previous applications and the corresponding decisions. Using this training data, a classifier may be trained that can accurately learn to accept or reject applications.

K-nearest neighbor classification

One of the simplest examples of a classification algorithm is the K-nearest neighbor algorithm. This algorithm is an example of *instance-based* or *passive* learning, because the training phase of the algorithm consists of simply storing the training examples. The majority of the work is done during the actual classification of an unseen instance.

When the algorithm is to classify a new input, it calculates the distance between the new input and all stored training examples. How the distance between two input feature vectors is defined, depends on the choice of a distance measure. Typical distance measures are the Euclidian distance or the Manhattan distance. Majority voting between the K nearest neighbors determines the resulting classification.

Figure 2.4 illustrates the K-nearest neighbor algorithm. In this abstract example there are two features and two classes (squares and triangles). We are trying to find a

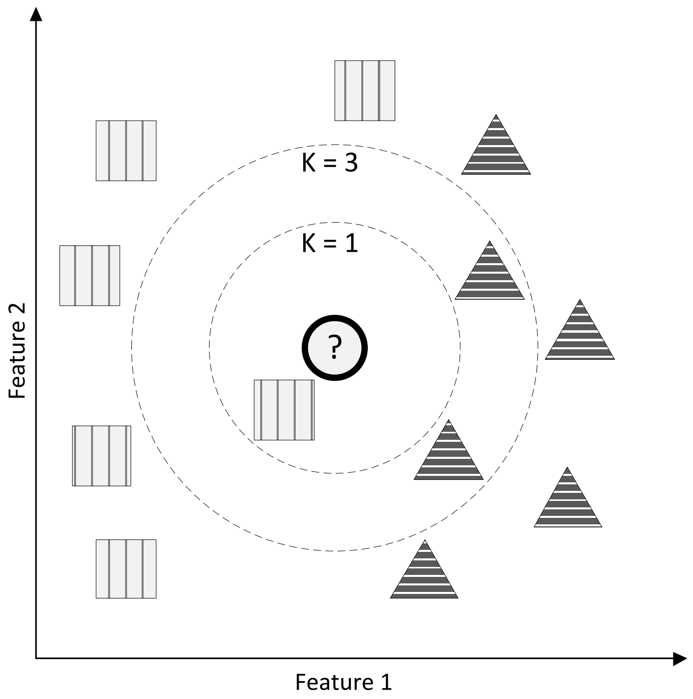


Figure 2.4: An example of the K-nearest neighbor algorithm. We are trying to decide whether the input (the circle with a question mark) should be classified as a square or a triangle. For $K = 1$, the classification will be square, whereas for $K = 3$, the classification will be triangle.

classification for the input, shown as a circle with a question mark. The dashed circles illustrate the nearest neighbors of the input, using Euclidian distance. If we choose $K = 1$, the classification is determined entirely by the input's single nearest neighbor, which is a square. For $K = 3$, the outcome is determined by majority voting between the 3 nearest neighbors and the resulting classification will be a triangle.

Decision trees

Decision trees are trees that model some type of decision-making process. At each inner node in a decision tree is some test that determines which edge coming from the node should be followed. The edges from inner nodes represent the possible outcomes of the test and leaf nodes contain outcomes of the decision-making process. A decision tree for classification typically compares the value of some feature in its input against one or multiple thresholds and has a classification at its leaf nodes. An example of a decision tree for the classification of fruit based on a small number of properties is shown in figure 2.5.

The general approach to deducing a decision tree (Quinlan, 1986) from a set of labeled training data is through *recursive partitioning.* Recursive partitioning refers to repeatedly splitting the training set into smaller subsets until some stopping

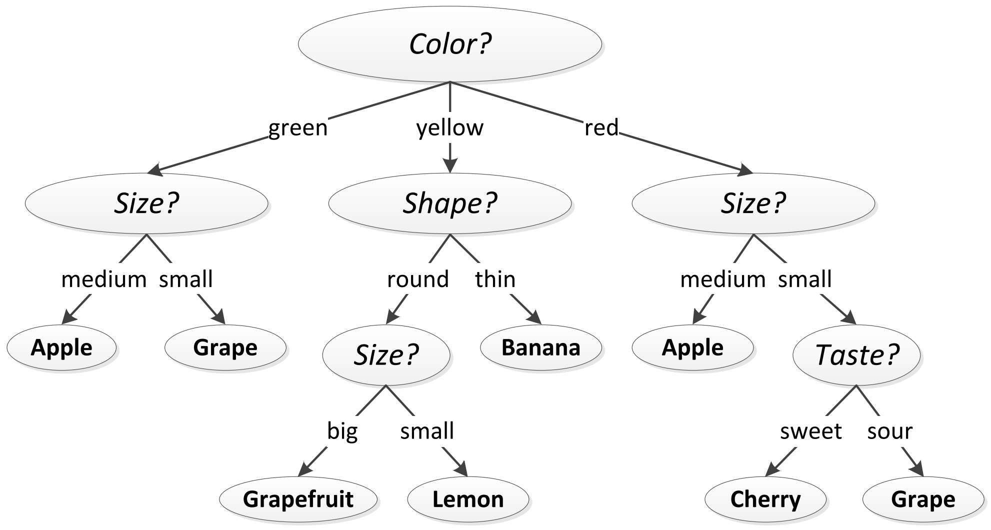


Figure 2.5: A decision tree for classifying fruits based on their color, size, shape and taste. As an example, we can use the tree to find that a fruit that is yellow, round and small is probably a lemon. Adapted from (Duda et al., 2000, p. 395).

criterion is met. Initially, the set consists of all of the training data. We then try to find a test for this set that splits the set into two (for binary tree learning) or more subsets so that each of the subsets is as homogeneous as possible: we are trying to find splits that separate the different classes in the data. Once a good split has been identified, the resulting subsets are split as well until the stopping criterion is met.

We then need a way to quantify how good a split is. One common measure for the quality of a split is the *impurity drop*. The impurity drop of a candidate split is the decrease in impurity (which we will discuss shortly) and may be defined as (Duda et al., 2000, p. 401):

$$\Delta i(N) = i(N) - P_L i(N_L) - (1 - P_L) i(N_R) \tag{2.3}$$

in which $i(N)$ is the impurity of the set before splitting, $i(N_L)$ and $i(N_R)$ are the impurities of respectively the left and right sets after splitting and P_L is the proportion of the set that goes to the left set after splitting. The impurity of a set may be quantified using a number of different methods. The choice of measure will typically not affect the resulting tree's accuracy but merely its size however (Mingers, 1989a). A popular impurity measure is the *entropy impurity*:

$$i(N) = -\sum_j P(\omega_j) \log_2 P(\omega_j) \tag{2.4}$$

where $P(\omega_j)$ is the fraction of set N that is in class j. The measure is related to the notion of *entropy* and is 0 when all samples in set N have the same label. It is positive when there is a mixture of class labels in the set and reaches its maximum value when all classes are equally likely.

A stopping criterion determines whether a subset should be split any further. Once splitting is stopped, the frequencies of the classes in the corresponding subset determine the classification for that branch. A simple stopping criterion is to continue splitting subsets until all sets are pure and contain only a single class label. This method is prone to *overfitting* however: the decision tree will be greatly tailored towards its training data and lack generalization. As a result, the decision tree will probably not classify unseen inputs very well.

There are a number of solutions to prevent overfitting in decision trees. They can roughly be divided in two categories: methods that stop the tree from growing and methods that fully grow the tree and subsequently prune branches. Since a thorough discussion of early stopping and pruning methods is beyond the scope of this thesis, please refer to (Mingers, 1989b; Esposito et al., 1997) for an overview of pruning methods and to (Duda et al., 2000, pp. 402–403) for an overview of early stopping criteria.

From a machine learning perspective, decision trees are interesting because in contrast to most models produced by machine learning algorithms, decision trees are comprehensible for humans. Furthermore, they handle nominal and ordinal data gracefully as opposed to most other learning methods. Constructing them is simple and classifying an unseen input requires only a small number of comparisons. This all comes at a cost of a slightly smaller accuracy than may be obtained using more advanced techniques, but the performance of decision trees is typically quite up to par (Dumais et al., 1998).

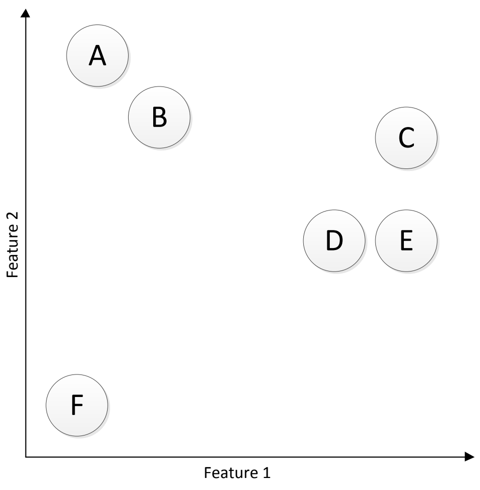
2.3.3 Unsupervised learning

In unsupervised learning, the training data is unlabeled and the goal is to find patterns or structure in the data.

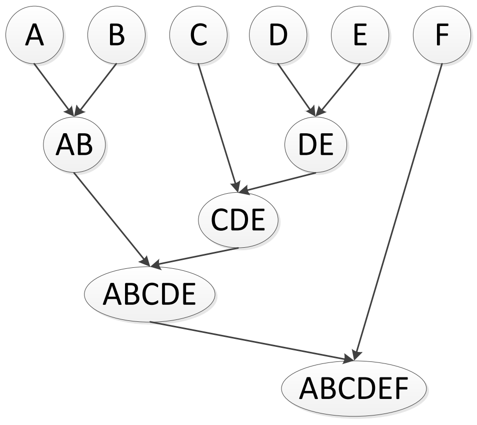
One common goal in unsupervised learning is to find *clusters* of instances in the data that are similar in some way. This form of unsupervised learning is called *cluster analysis* or simply *clustering*. We typically distinguish two types of clustering. In hierarchical clustering, every sample is initially in its own cluster by itself. Larger clusters are then formed by combining similar clusters into larger clusters. This repeated merging of cluster gives rise to a hierarchical structure of clusters, hence the name (figure 2.6). In partitional clustering, all clusters are determined simultaneously, typically in an iterative procedure. Let us now consider an example of partitional clustering called *K-means clustering*.

K-means clustering

The K-means clustering algorithm is a simple iterative procedure to find K clusters in a set of data. Initially, each data point is randomly assigned to one of the K clusters. Then, the following two-step procedure is repeated until convergence:



(a)



(b)

Figure 2.6: An example of hierarchical clustering. At the left the relative positioning of the data points A through F are shown. The resulting cluster hierarchy is shown at the right.

1. Each data point is reassigned to the cluster whose centroid it is closest to, according to some distance measure. The procedure has converged when the assignments no longer change.
2. The centroid of each cluster is updated to be the mean of all its members.

One difficulty with the K-means algorithm is that K has to be specified manually. If the natural number of clusters in the data is known, this is no problem. Often however, there is no such prior knowledge of the structure of the data and K has to guessed. A poor choice of K can cause poor clustering results.

The K-means algorithm is similar to the more general *expectation maximization (EM)* procedure for finding maximimum likelihood estimates of unknown parameters in models. We will discuss the EM procedure in more detail now.

Expectation Maximization

The expectation maximization (EM) algorithm can be seen as a generalization of the K-means algorithm. It is an iterative procedure to find *maximum likelihood (ML)* estimates of the parameter(s) of some model in the presence of hidden or missing data. Maximim likelihood refers to the fact that we would like to find values for the parameters for which the observed data are the most likely.

The EM algorithm consists of an expectation (E) step, followed by a maximization (M) step. Both these steps are repeated until convergence. In the expectation step, the missing or hidden data is estimated given the observed data and current estimate of the parameters. Note that for the first expectation step, the model parameters will have to be initialized, typically using an educated guess. The expectation step typically uses *conditional expectation*, hence the name. In the maximization step, the estimate of the missing or hidden data is used to find new estimates for the parameter(s) that maximize the likelihood of the data.

2.4 Previous work

In this section we will describe some previous work that has been published on (computer) game playing and Poker. We will first consider some early theoretic work on Poker that laid the foundations for modern game theory. Then, we will discuss more modern work on computer game playing and artificial Poker agents.

2.4.1 Early work

The famous mathematician and physicist John von Neumann included a chapter on a simplified Poker game in his groundbreaking work "Theory of Games and Economic Behavior"(von Neumann and Morgenstern, 1944). In this book, which many consider the book that created the field of game theory, he and the economist Oskar Morgenstern show that the optimal strategy to a simplified version of Poker includes bluffing. That is: without bluffing, one cannot win as much as one could with a limited amount of bluffing.

The reason for this is two-fold. First, since betting is often associated with holding strong cards, opponents might give up a stronger hand. By betting his weaker cards the bluffing player might win a pot that he would otherwise lose because he holds the weakest cards. Second and most interestingly: if a player is known to only bet with strong cards, his opponents can just fold whenever he bets and he will have a hard time winning money with his strong cards. Von Neumann and Morgenstern show that the optimal strategy for a greatly simplified Poker game that they describe therefore includes some bluffing to introduce some amount of uncertainty about the strength of the cards.

Because this finding and the rest of the book are not entirely relevant to the present research, we will not discuss them in further detail. Such an early study of Poker in the first great work on game theory deserves mentioning however.

2.4.2 Modern work

Billings (1995) renewed interest in researching Poker by putting it forward as an ideal candidate to succeed Chess as the primary focus of computer game playing research. Being a non-deterministic game of imperfect information, it offers a wealth of new challenges. Billings's master thesis led to the founding of the University of Alberta Computer Poker Research Group (UACPRG), whose members have published a large number of groundbreaking computer Poker articles. Researchers from other institutions quickly followed their lead, resulting in great progress at computer Poker with milestones such as defeating world-class human experts at heads-up limit Texas Hold'em Poker.

Near-optimal solutions for Poker

Koller and Pfeffer (1997) describe *Gala*, a system that may be used to find optimal strategies for imperfect information games. The authors successfully solved several games using this system and have investigated Poker, but conclude that the number of states for Poker is far too large to solve. They suggest that "[an] approach that may be useful for such games (as well as for others) is based on abstraction. Many similar game states are mapped to the same abstract state, resulting in an abstract game tree much smaller than the original tree. The abstract tree can then be solved completely, and the strategies for the abstract game can be used to play the real game".

It was this suggestion that resulted in the publication of (Billings et al., 2003). The authors used several abstraction techniques to reduce an $O(10^{18})$ *heads-up limit* search space to a manageable $O(10^{7})$ search space without losing the most important properties of the game. The resulting abstracted game was then solved using linear programming techniques. The resulting strategies are called *near-optimal* or *approximate Nash-equilibria* for the real, unabstracted game. The quality of these solutions depends on how well the abstracted game captures the properties of the unabstracted game.

Expert systems

Billings et al. (1998b) introduced the first of a series of modern computer Poker systems called *Loki*. Loki was designed to play limit Texas Hold'em Poker against two or more opponents. It uses a simple rule-based expert system to play during the preflop phase. For the postflop phase, Loki uses a simple formula that picks an action based on the strength of the cards it is holding, given the board. Since it incorporates no opponent modelling and is entirely based on rules and formulas, it is essentially an expert system.

against 2 or more opponents. It uses expert rules to play in the preflop phase and switches to a system based on opponent modeling and expert rules for the postflop phase.

Opponent modeling

Papp (1998); Billings et al. (1998a) describe efforts to incorporate opponent modeling in Loki. The resulting opponent modeling system models specific opponents by counting their actions in a small number of distinct *contexts*. A context is defined by the values of a small number of simple features, such as "number of bets" (1, 2 or 3) and "street" (preflop, flop, turn or river). Clearly, such a system allows for no generalization over contexts. The simple opponent models were used to calculate improved estimates of the relative strength of Loki's cards.

Pena (1999) describes the resulting system called Loki-2, which also incorporates Monte Carlo simulations to select actions during the postflop phase. These simulations use the newly added opponent modeling module to estimate the expected values of the moves available to Loki-2. The simulations replace the previously used simple formula to select actions based on the relative hand strength.

Davidson et al. (2000) tried to improve the simple opponent modeling by investigating the use of artificial neural networks (ANNs) to predict opponents' actions. Their basic backpropagating ANN achieved about 85% accuracy at predicting actions and resulted in significant improvement of the system's playing quality. Davidson (2002) describes the new system, which from then on was labeled *Poki* instead of Loki. Davidson discusses a number of classification systems that may be used in opponent modeling for Poker, amongst which are ANNs, decision trees and simple statistics. Poki contains a multi-classifier system that uses a number of different classification techniques and combines their results into a single classification.

Billings et al. (2004) introduced *Vexbot* as a first attempt at using game tree search with opponent modeling in Poker. To this end, it proposes two variants of Expectiminimax (see section 2.2.3) for Poker game trees called Miximax and Miximix. Through clever use of data structures and caching, they manage to perform an exhaustive depth-first search of game trees for two-player Texas Hold'em Poker in reasonable time. The opponent modeling in Vexbot is rather basic and uses frequency counts in a small number of contexts with an additional system of "abstractions" to allow for some generalization between different contexts (Schauenberg, 2006).

Monte Carlo methods for no-limit Poker

Van den Broeck (2009) and Van den Broeck et al. (2009) were the first to apply Monte Carlo ideas that are very successful in computer Go to no-limit Texas Hold'em Poker [3]. Their work focused on exploitative play in no-limit Texas Hold'em Poker with any number of players. The game trees for these games (particularly for games with 10 players) are enormous and cannot possibly be approached with any kind of traditional game tree search. Monte Carlo Tree Search estimates the values of moves in these large trees using an *anytime* algorithm: the longer the algorithm is allowed to run, the better its results will be. We will discuss Monte Carlo Tree Search in greater detail in chapter 3.

2.5 Conclusion

In this chapter, we have covered some topics that we will build on in the remainder of this thesis. We have discussed some concepts from game theory and artificial intelligence, such as solving of games, game trees and Monte Carlo methods. In the next chapter, we will consider Monte Carlo Tree Search (MCTS) in more detail.

[3] `http://code.google.com/p/cspoker/`

Chapter 3

Monte Carlo Tree Search

3.1 Introduction

In the previous chapter we have discussed some tree search methods that allow computer programs to reason about moves and their consequences in games. These algorithms have successfully been applied to a great number of games and have led to remarkable performance of computer programs in these games. We have already seen that there lies a problem in the fact that the game tree grows exponentially in the number of moves, however. While this problem may partially be avoided by employing smart pruning techniques, for some games such as Go and full scale No-limit Poker the game tree is just too large the be searched exhaustively using the aforementioned tree search algorithms.

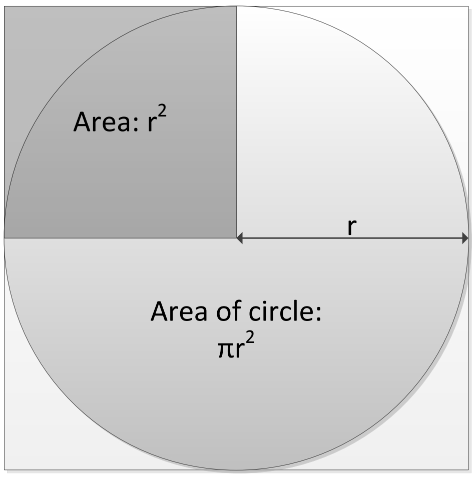
In this chapter we shall consider the Monte Carlo Tree Search algorithm, a method that uses simulations (also known as continuations) to estimate the values of moves by sampling the tree. The accuracy of these estimates depends on the number of simulations they are based on: the more simulations, the better the estimate.

3.1.1 Monte Carlo methods

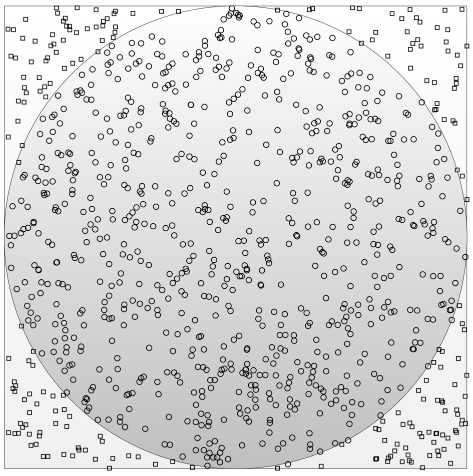
The term Monte Carlo method was first coined by von Neumann and Ulam during World War II as a codename for secret work at Los Alamos National Laboratory on nuclear weapon projects (Rubinstein and Kroese, 2007; Kalos and Whitlock, 2008)[1]. They used it to refer to a method of using repeated random sampling to estimate the solutions to problems that are very hard or impossible to find analytically.

Let us consider an example to illustrate the basic idea behind Monte Carlo methods. The value of π may be estimated as follows. From plane geometry, it is known that the area of a circle with radius r is πr^2. Since the area of a square whose sides have length r is r^2, the ratio of the area of a circle with radius r to such a square is equal to π. We can estimate this ratio by randomly placing n points in a drawing of a square with an inscribed circle and counting the nunber of points that lie in the

[1]The name Monte Carlo refers to the Monte Carlo Casino in Monaco.



(a) The relationship between the area of a circle with radius r and the area of a square whose sides have length r.



(b) We can estimate π by drawing a square with an inscribed circle and n randomly positioned points. The ratio of points within the circle to the total number of points is an estimate of $\pi/4$. In this case the estimated ratio is 786 / 1000.

Figure 3.1: Estimating π using random sampling.

circle. Dividing the number of points inside the circle by the total number of points n, yields an estimate for a quarter of the aforementioned ratio[2] and thus for $\pi/4$ (figure 3.1b).

In figure 3.1b, 1000 points have been randomly positioned on such a drawing of a square with an inscribed circle. Counting tells us that 786 of the points lie within the circle. This gives us a value of $786/1000 = 0.786$ for $\pi/4$, and thus a value of $4 \times 0.786 = 3.144$ for our estimate of π, which we know to be a decent estimate of the actual value.

3.1.2 Monte Carlo methods and game trees

The idea that you can use random sampling to estimate solutions to problems that are too hard to handle analytically, may be extended to game tree search. For example, we can evaluate some game state S by running a (preferably large) number of simulations of how the game might continue from state S. The outcomes of these simulations may then be combined to form an estimated value of state S.

Figure 3.2 illustrates how we can use Monte Carlo evaluation to perform a simple stochastic tree search. First, we expand the root node, adding its five child nodes to the tree. Then we estimate the value of each of these child nodes by running

[2]Note that the square that contains the inscribed circle is actually four times as large are the square whose sides have length r

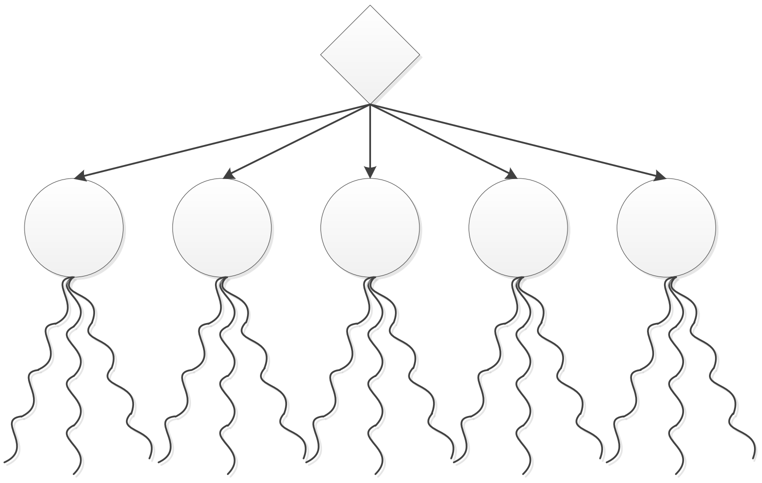


Figure 3.2: A depth-one greedy algorithm to find the best move using Monte Carlo evaluation. The root node is expanded, revealing five child nodes. Each of these moves is evaluated by running a (preferably large) number of simulations and averaging the outcomes. The best move is the move with the highest estimated value.

simulations. The value of each of the child nodes may then be estimated to be the combined results of the simulations, the average of the outcomes for example. We now have estimated values for the child nodes of the root node and can play the move with the highest estimated value.

While Monte Carlo evaluation saves us from having to exhaustively search large branches of the game tree, it introduces the new problem of how to simulate continuations of the game. We now have to select moves for all players in the game until a leaf node is reached. One way to handle this new problem is to simply select random moves for all players. For some simple games this might suffice and result in reasonable approximations of the values of nodes. For some others random moves might not suffice and we will have to come up with something smarter. Since using tree search to select moves in simulations effectively somewhat defeats the purpose of the simulations, we cannot use those.

3.1.3 Progressive pruning

Progressive pruning (Bouzy et al., 2003) is a refinement of the aforementioned simple greedy Monte Carlo method for move selection. In progressive pruning, each move has an associated mean value μ and standard deviation σ. Simulations are used to estimate the value of each move and their associated μ and σ are updated after each simulation. Using simple statistics, it is then possible to calculate whether a move is statistically inferior to another move with reasonable confidence. In progressive pruning, this idea is used to stop considering moves that can be proven to be statistically inferior to others. This way, precious time for simulations can be spent on more promising moves. This progressive pruning of moves continues until either one move is left or time runs out, in which case the currently most promising move should be selected.

3.2 Monte Carlo Tree Search (MCTS) algorithms

The world of computer Go was revolutionized when two programs called Crazy Stone (Coulom, 2006) and MoGo (Gelly and Wang, 2006) emerged in 2006. These programs proved to be the absolute state of the art at the time of their introduction, and were the first to defeat professional Go players on a 9×9 board. The publications revealing their internals learned that they used a combination of conventional game tree search and Monte Carlo simulations.

Monte Carlo Tree Search cleverly merges tree search and Monte Carlo methods by gradually building a tree in memory that grows with the number of simulations. The tree contains bookkeeping information that may be used to look up the values of moves obtained from previous simulations. These values may be used to guide the simulations towards promising branches in the tree. This way, effort may be concentrated on simulating branches in the tree with high potential, while inferior moves will receive fewer attention.

Note that there are two distinct trees in the context of MCTS. First there's the slowly growing tree used to store the information on nodes that is used to guide simulations. Second, there's the more abstract game tree that describes the moves and positions in the game being played. To disambiguate between the two, we will use *search tree* for the former, and *game tree* for the latter.

The MCTS starts with initializing the search tree. This step typically consists of just adding a single root node representing the current game position to the tree. Once the search tree is initialized, the following four steps are repeated for as long as there is time left:

- **Selection**: a *selection function* is recursively applied, starting from the root node. Selection continues until a leaf node of the search tree has been reached. Such a leaf of the search tree is not necessarily also a leaf of the game tree, although it could be.

- **Expansion**: one or more children of the selected node are added to the search tree.

- **Simulation**: a simulation is started that simulates play from the earlier selected node until a leaf node of the game tree is reached.

- **Backpropagation**: the value obtained from the simulation is backpropagated through the search tree up to its root node. The bookkeeping information stored at each node in the path from the selected node to the root node is updated to include the new information.

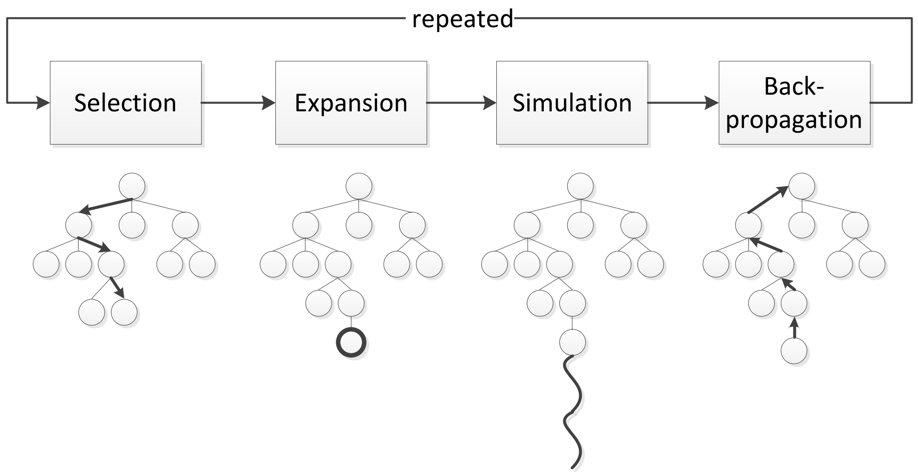


Figure 3.3: The Monte Carlo Tree Search algorithm. Note that the tree shown is the search tree that is gradually built with the MCTS algorithm and not the abstract game tree of the game being played. Adapted from (Chaslot et al., 2008).

Figure 3.3 illustrates this process. While the four abovementioned steps and the figure are frequently used in MCTS literature, it may be easier to think of the entire process (all four steps as a whole) as one simulation. This is the case because the selection step effectively boils down to simulating moves to play, just like the simulation step. The only difference between the two is that in the selection process, information from the search tree is used to decide which moves the players will choose, whereas in the simulation step, heuristics or random selection determine what moves are chosen.

Note that the algorithm is *anytime*: the algorithm provides an approximate solution whose accuracy depends on the number of completed iterations. This is a great property when searching trees that are too large to search completely, as it allows us to obtain a solution within any amount of time.

We will now discuss the steps of the algorithm in more detail.

3.3 Selection

In the selection step, a selection function is recursively applied in order to select a leaf node of the search tree where expansion and simulation will continue. Note that a leaf of the search tree is not neccesarily also a leaf of the game tree. As a matter of fact the contrary applies: for large game trees leaf nodes of the search tree typically are not also leaves of the game tree.

Because how nodes are selected depends on the type of node under consideration, let us now consider the types of nodes in the Poker game tree and their corresponding MCTS selection methods.

3.3.1 Opponent nodes

At opponent nodes, an opponent is to act. At these nodes we want to select child nodes with the probability that the opponent will play the corresponding move. If the opponent would never play a certain move, there is no point in it being selected for further investigation. Conversely, if we know an opponent is very likely to play a particular move, we should select it very frequently. Typically, we should therefore use some heuristic or model to estimate the probability of the opponent playing each move available to him.

3.3.2 Chance nodes

At chance nodes, some stochastic event takes place. For Poker, this boils down to drawing new community cards for the flop, turn or river. We should select cards here with the probability that they will be drawn. Since cards that are in the players' hands cannot also be in the deck and be drawn at chance nodes, some probabilistic inference may be used to estimate the probability of each card being drawn at chance nodes. Most currently published methods however assume the effect of such inference is negligable and just assume each card has an equal probability of being drawn (Billings et al., 2004, p. 6).

3.3.3 Player nodes

Player nodes are an interesting kind. At these nodes we have two opposing goals that we want to satisfy. First, we would like to frequently select moves that yielded good results in previous iterations of the algorithm to confirm their strength. Second, we would also like to occasionally select some weaker nodes. We should occasionally select weaker nodes because their perceived weakness may be the result of random events in previous simulations: the branch might actually be quite strong.

The trade-off between picking nodes that we found to be strong and picking weaker nodes may be considered a trade-off between *exploitation* and *exploration*. Luckily, balancing these two is a recurring problem in scientific research and a great body of literature has been published on the subject. The problem is interesting because there are a number of interesting real-life applications, such as trying new medical treatments. New treatments need to be tested, while minimizing patient loss. The canonical problem studied in the context of exploration-exploitation trade-offs is the *multi-armed bandit problem.*

The multi-armed bandit problem and Upper Confidence Bounds (UCB)

The multi-armed bandit problem, named after the traditional one-armed bandit slot machine, models a gambler trying to maximize his profits in a series of trials. The gambler is playing on a slot machine with K levers (or K slot machines with one arm). When pulled, each of the levers provides a reward that is drawn from a distribution specific to that lever. The gambler has no prior knowledge about the reward distributions associated with the levers and will need to find a sequence of levers that maximizes the sum of the rewards. Clearly, he will have to find a strategy that carefully balances acquiring new knowledge about the levers (exploration) with pulling levers previously observed to yield large rewards (exploitation).

A number of strategies for playing the multi-armed bandit have been developed, the simplest of which is the ϵ-greedy strategy. This strategy selects the lever with the largest expectation for a proportion $1 - \epsilon$ of the trials. In the remaining proportion ϵ of the trials, a random lever is pulled with uniform probability. While simple, this strategy turns out to be hard to beat (Vermorel and Mohri, 2005).

Auer (2003) proposed the Upper Confidence Bounds (UCB) strategy for the bandit problem. In this strategy each lever is played once initially. Then, the next arm is selected for which the following equation is maximal:

$$\overline{V_i} + C\sqrt{\frac{2\ln \sum_j n_j}{n_i}} \tag{3.1}$$

In which $\overline{V_i}$ is the average reward obtained by pulling lever i and n_i is the number of times lever i was pulled. C is a constant that balances exploitation and exploration. The first term causes exploitation as it is large for levers with a high average reward. The second term is an estimate of the upper bound of the Chernoff-Hoeffding confidence interval on the average returned reward (van den Broeck et al., 2009) and causes exploration.

UCB applied to Trees (UCT)

Kocsis and Szepesvári (2006) introduced UCB applied to Trees (UCT), an extension of UCB that applies ideas from the multi-armed bandit problem to Monte Carlo planning in Markov decision processes and game trees. In UCT, each node selection step is considered as an individial multi-armed bandit problem.

One problem with this approach is that UCT assumes that the returned rewards for each option are drawn from stationary distributions. This assumption is violated in MCTS, as with each iteration the average rewards and selection counts of some nodes in the tree are updated, causing changes in the reward sampling distributions.

Another problem with UCT for MCTS lies in the fact that they have subtly different goals. The goal for UCT is to select the best option as often as possible. The goal of MCTS selection is to sample the options in such a way that the best option will be selected at the root node of the tree (van den Broeck et al., 2009). While these goals are quite similar, they are not identical and the ideas of UCT do not completely apply to MCTS selection. Despite these problems, UCT has been shown to work quite well as a selection function in MCTS (Gelly and Wang, 2006; van den Broeck et al., 2009).

3.4 Expansion

In the expansion step, one or more nodes are added to the MCTS search tree. This step determines how quickly the search tree grows. The only decision that has to be made in this step is the number of nodes to add. We could add all child nodes of the selected node, a fixed number of sampled nodes or just a single node. A common strategy is to add just a single new node to the tree with each iteration.

3.5 Simulation

In the simulation step of MCTS, the remainder of a game is simulated from a leaf of the search tree to a leaf of the game tree. In case the selected leaf node of the search tree is also a leaf of the game tree, this step may obviously be omitted. In the simulated continuation of the game, all players take turns to make simulated moves. Chance events such as drawing cards are simulated as well. The outcome of the simulation yields an estimated value for the game state where the simulation started, that may be used to update expected values higher in the search tree.

The problems we are facing here is how to simulate players' moves and stochastic events. For some games such as Go, it may be acceptable to simply simulate random moves for all players. For other games such as Poker, such an approach most likely won't suffice. In these games, we will have to use models or heuristics to simulate players' actions instead. We will discuss the subject of simulating moves and stochastic events for Poker in section 3.7.

3.6 Backpropagation

In the backpropagation step, the result of the simulation step is propagated back to the root of the search tree. This means that the nodes whose values should be updated – the nodes on the path from the root node to the selected node – are visited in order to update their estimated expected values. Since there is no single correct way to perform backpropagation for all types of nodes in the search tree, we will now discuss backpropagation for different node types in more detail now.

3.6.1 Opponent and chance nodes

The intuitive way to calculate a new expected value for any node that has children is to use a weighted average of the values of its child nodes. We should take great care when selecting weights to use for this weighted average however: using the probabilities with which the nodes are selected introduces a bias in the calculation of the expected value.

Van den Broeck (2009) provides a great example that illustrates this problem. Let $\overline{V}_n$ be the expected value of node n. Node n has two children, nodes a and b that are selected with probabilities $P(a|n)$ and $P(b|n)$, respectively. Let N be the number of samples we draw to estimate $\overline{V}_n$. The expected value of $\overline{V}_n$ after drawing N samples is given by:

$$\begin{aligned} E\left(\overline{V}_n\right) &= P\left(a|n\right)^N \times \overline{V}_a + \\ & P\left(b|n\right)^N \times \overline{V}_b + \\ & \left(1 - P\left(a|n\right)^N - P\left(b|n\right)^N\right)\left(w_a \times \overline{V}_a + w_b \times \overline{V}_b\right) \end{aligned} \tag{3.2}$$

In this equation, w_a and w_b are the weights that we will be using for nodes a and b in the weighted average. Note that the formula describes three cases:

1. We only draw node a in N trials with probability $P(a|n)^N$. In this case we simply propagate the value of node a, $\overline{V}_a$.
2. We only draw node b in N trials with probability $P(b|n)^N$. In this case we propagate $\overline{V}_b$.
3. We draw a combination of nodes a and b in N trials with probability $1 - P\left(a|n\right)^N - P\left(b|n\right)^N$. In this case we propagate the weighted average of both nodes $w_a \times \overline{V}_a + w_b \times \overline{V}_b$.

Let us now consider what would happen if we would simply use $p(a|n)$ and $p(b|n)$ for w_a and w_b respectively. Suppose that $\overline{V}_a = 0$, $\overline{V}_b = 1$, $P(a|n) = 0.1$ and $P(b|n) = 0.9$. For $N = 2$, the expected value for $\overline{V}_n$ is given by:

$$0.1^2 \times 0 + 0.9^2 \times 1 + (1 - 0.1^2 - 0.9^2)(0.1 \times 0 + 0.9 \times 1) \approx 0.972$$

This value overestimates the actual expected value for node n which is $0.1 \times 0 + 0.9 \times 1 = 0.9$ because nodes with a high probability introduce bias. The solution to this bias is not to use the probability of the nodes being selected, but use the sampling frequency instead. Let $T(n)$ be the number of times node n was sampled. If we use $T(n)/N$ to calculate the weights w_a and w_b, we obtain the following unbiased expected value for $\overline{V}_n$:

$$0.1^2 \times 0 + 0.9^2 \times 1 + (1 - 0.1^2 - 0.9^2)(\frac{1}{2} \times 0 + \frac{1}{2} \times 1) = 0.9$$

This example illustrates that we should use the sampling frequency and not the selection probabilities to weigh the values of child nodes when propagating values at chance and opponent nodes. This gives us the following formula to calculate the new expected value $\overline{V}_n$ for node n:

$$\overline{V}_n = \sum_i \frac{T(i)\overline{V}_i}{N} \tag{3.3}$$

In which $\overline{V}_n$ is the expected value of node n, $T(i)$ is the number of times the child node i was selected and $N = \sum_i T(i)$ is the total number of times children of n were selected.

3.6.2 Player nodes

At player nodes, values may be backpropagated in a number of ways. The only requirement imposed is that the propagated value should converge to the maximum expected value in the limit of infinitely many samples. Two backpropagation strategies that meet this requirement are:

- **Sampling frequency**: this is the same backpropagation strategy as for chance and opponent nodes. This strategy meets the requirement because a proper selection strategy will only select the maximum child in the limit of infinitely many samples. Equation 3.3 then converges to the maximum expected value of the player nodes' child nodes.
- **Maximum child**: we can also use simple classic minimax backpropagation. We then propagate the maximum value of the player node's children. Note that for stochastic game trees this is likely to overestimate the true expected value, as any noise introduced by stochastic events is maximized as well.

3.6.3 Leaf nodes

For leaf nodes, the backpropagated values should simply be their utilities. For games where there is no hidden information, the utility is simple to determine, as it is clear which player wins. For games with hidden information such as Poker, determining the utility of leaf nodes is not always this simple and might require the use of heuristics or models to estimate the value of the leaf node. We will discuss backpropagation of Poker leaf nodes in more detail at the end of this chapter.

3.7 MCTS and Poker

In this section, we will discuss how Monte Carlo tree search may be applied to no-limit Texas Hold'em Poker. Van den Broeck (2009) and Van den Broeck et al. (2009) describe a MCTS computer Poker player, which we will build on in the work presented here.

As we have seen in the previous sections of this chapter, MCTS requires a number of heuristics or models to estimate some probabilities that are used in the selection, simulation and backpropagation steps. We will briefly address the types of models that are required now and describe practical implementations of these models in more depth in the following chapters.

3.7.1 Opponent modeling

For the selection and simulation step, the probability distribution over the possible actions needs to be estimated. To this end, an opponent modeling component is required that estimates these probabilities given the current state of the game. During the simulation step, the computer player's own actions also need to be predicted. One solution there is to play random moves or to use the opponent modeling component to do self-modeling.

Note that for no-limit Poker as opposed to limit Poker, the number of allowed actions is extremely large because of all the different amounts of money players can bet. One simple solution to this problem is to perform discretization on the bet amounts, effectively limiting the number of actions to a constant number. A simple way to implement such discretization that is actually often used by human Poker experts, is to map the bet amounts to certain fractions of the pot. For example, all possible actions could be discretized to $\frac{1}{4}$ pot or smaller, $\frac{1}{2}$ pot, $\frac{3}{4}$ pot, and 1 pot or higher.

3.7.2 Expected value of showdowns

During the backpropagation step of MCTS, the utility of leaf nodes should be determined. For leaf nodes where the game ended because all but one player chose to fold, determining the utility is simple: the single player left in the game wins the entire pot, while all others win nothing. For leaf nodes where the game ends with a showdown of cards, matters are more complicated. Since we know only our own cards, we will need to somehow estimate the strength of the opponents' cards in order to determine the utility of the showdown leaf node.

Note that if we could predict an opponent's actions given his cards, we could use these models to estimate the probability distribution over all possible cards using Bayes' rule. Developing opponent models that can estimate the probability of an opponent taking an action given each of the possible hole cards could thus solve the problem of opponent modeling and predicting the cards he is holding simultaneously.

3.8 Conclusion

In this chapter we have discussed how Monte Carlo ideas may be applied to game tree search for complex games with large game trees. The Monte Carlo Tree Search (MCTS) is an algorithm that proved to be very successful in computer Go and initial attempts have been made to implement this algorithm for Poker too. In the final section of this chapter, we have discussed what components are required to build a computer Poker player using MCTS. In the following chapters, we will discuss our work that implements these components in an effort to build a no-limit Texas Hold'em Poker computer player.

Chapter 4

Opponent modeling for Poker: predicting moves

As we have discussed in the previous chapters, we will need to model opponents in order to predict (i) their actions and (ii) the cards they have at showdowns. In this chapter, we will consider the former. The latter will be covered in chapter 5.

4.1 Learning to predict moves

In order to play good Poker, a Poker player needs to reason about future actions of both himself and his opponents. For example, if you have a weak holding and are contemplating a bluff, you will need to estimate how often the bluff will be successful and win you the pot. Similarly, if you have a very strong hand on the river and are first to act you will need to figure out how you can win the most. If you think your opponent is likely to call a big bet, betting is probably the best option. If you think your opponent is more likely to fold to a bet but might bluff when you check, a check might be the best choice.

As we have seen in chapter 3, this quality of a good Poker player is also reflected in the components required to build a computer Poker player using Monte Carlo Tree Search. During both the selection and simulation steps, actions need to be predicted for both the computer player and his opponents. Note that we are not interested in what action a player is most likely to take, but rather in the probability distribution over all possible actions. For example, if we would only sample a player's most likely move during simulation, all simulations from a position would result in the exact same sequence of actions. Consequently, we would like to know that a player will fold 60%, call 30% and raise 10% of the time instead of only finding that the player will most likely fold.

We would like to learn models that estimate the probability that a player will take action A as the nth action, given the current state of the game. The description of the current state of the game should include all actions (0 through $n-1$) leading to the current position and other revelant information such as the community cards on the board and the stack sizes (the amount of money left to bet) of all players. More formally, we are trying to find models that estimate:

$$P(a_n = A | a_{n-1}, ..., a_0, \vec{b}, \vec{s}) \tag{4.1}$$

Where a_n is the nth action in the game, $\vec{b}$ are the community cards (the board) and $\vec{s}$ are the stacksizes. Note that the moves that players will select will also greatly depend on the cards they are holding. But since these are not visible to anyone but the player himself, we cannot use this information to predict moves.

We will now consider how such models could be learned by a computer. First, we will discuss possible machine learning algorithms that are suitable for the task at hand. Then, we will consider how the set of training data for these algorithms may be composed.

4.2 Selecting a machine learner

In chapter 2 we have covered a number of different machine learning algorithms. The ones we have covered are just a small sample of the available machine learners: there are many more, each with its own strengths and weaknesses. For our opponent modeling task, we will have to choose a suitable machine learner. In order to make a good choice, let us now summarize some of the properties a machine learner for the task at hand should have:

- **Fast evaluation**: since our models will be used in Monte Carlo methods, the speed with which the models can be evaluated is important. The faster the models are evaluated, the more iterations can be squeezed in the limited time available, and in Monte Carlo methods more iterations equal better results.

- **Posterior probability estimates**: the machine learner should be able to output probability estimates over all possible actions instead of just the most likely action. While most machine learners can be configured to output estimates of these probabilities, it is much easier and more intuitive for some algorithms than for others.

- **Weighing of training instances**: in chapter 5 on predicting the cards of opponents, we will present algorithms that require weighing of individual training instances. We should be able to express how representative an example in the training set is.

- **Fast training**: whereas fast training is not extremely important since all training is done *offline*, fast training is convenient. In a number of algorithms which we will present later, iterative training of models is used. That is, we will be training models over and over again, which could get terribly slow if the machine learner needs a lot of time to learn a single model.

4.2.1 Decision trees

Decision or regression trees meet the requirements imposed on the machine learner for opponent modeling in Poker nicely. The resulting decision trees are extremely fast to evaluate, since a small number of comparisons is enough to get to a leaf node and obtain a result. Decision trees can be used to obtain posterior probabilities instead of single classifications by storing the number of instances of each class at leaf nodes instead of only the most frequent class. Decision tree learners also support weighing instances. The training times largely depend on the number of numerical features in the input, as these take the majority of processing time because splits have to be tested for many different values of the feature. Typically, the time required to train a decision tree model is acceptable. Decision trees also have the additional benefit of yielding human comprehensible models.

4.3 Composing a training data set

Once we have chosen a machine learning algorithm, we should compose a set of training data. Since we are trying to predict actions, such a training set should consist of pairs of resulting actions and descriptions of the current state of the game, preceding actions and other bits of information that may be useful. We will now discuss how a Poker game can be represented in a form that machine learning algorithms can handle, while maintaining all or most of the relevant information.

4.3.1 Representing a Poker game

The current state of a Poker game can be fully described by its initial state and all subsequent actions leading up to the current state. The initial state contains information such as who the players in the game are, who is sitting where, who the dealer and the blinds are and how much money every player has. From this initial state, an enormous amount of other states can be reached by a sequence of actions following it. It seems logical to just include all this information in a description that will be used to learn to predict actions.

From a machine learning perspective, there is a problem with such a description however. The problem lies in the fact that the sequence of actions leading to the current state has an unknown length. There could be 0 actions preceding the current state, there could be 5, there could be 20. Since feature vectors for machine learning typically have a fixed length, we will have a hard time representing a sequence of actions whose length is variable.

The usual solution to such a problem is to include a number of features that describe the sequence of actions, instead of the actual sequence. Examples of such features for a sequence of moves in a Poker game are the length of the sequence, the number of raises in the sequence and the number of calls in the sequence. Finding suitable features that accurately represent the actual sequence of actions is a fairly hard task that requires expert knowledge. The features that we used to represent the current state of a Poker game and all actions leading up to it in our experiments are listed in appendix B.

4.3.2 Representing moves in no-limit Poker

In limit Poker, there are at most three actions available to a player at any time. In no-limit Poker, this number can be extremely large since the player can usually choose to bet a very large number of different amounts. Most machine learners – and decision tree learners are one of them – will have a hard time working with numerical desired outputs that can take on a large number of values, so we will need to find a way around this problem. One solution is to map all possible moves onto a small number of classes representing these moves.

For Poker, human experts already often use such a mapping naturally. When they suggest advise for certain positions to other players, the advise is often phrased along the lines of "I would bet half the pot" or "a pot-sized bet would be best here". Human Poker experts tend to think in fractions of the pot, and not in very specific amounts. We can adopt a similar approach for computer Poker players and map all possible moves to fractions of the pot.

In all experiments that we will discuss in the remainder of this thesis, we use the following mapping of amounts to abstracted bets:

- **Minimum bet/minimum raise**: the minimum amount the player is allowed to bet or raise.
- **1/4 pot**: a quarter of the pot. A raise of a quarter of the pot means that the player first calls and then raises an additional 1/4 of the *new* amount in the pot, including the call. Should this amount be smaller than the minimum raise, the action is mapped to the minimum raise.
- **1/2 pot**: half of the pot.
- **3/4 pot**: three quarters of the pot.
- **Pot**: a pot-sized raise or bet.
- **One and a half pot**: a raise of bet of one and a half pot.
- **Two pot raise and higher**: a bet or raise that is twice the pot or greater.

An opponent's move may be mapped onto any of these abstracted moves in multiple ways. One way would be to map onto the move for which the amount is closest to the actual amount of the action. Schnizlein (2009) suggests to use a different approach:

> *"If b is the real bet and c, d are the two closest abstract bets such that $c < b < d$, then we map b to c or d according to whether c/b or b/d is largest. For instance, if we observe a bet of size* 20 *and the closest*

legal bets are 12 *and* 88*, then we will interpret this as a bet of* 12 *since* $\frac{12}{20} = 0.6 > \frac{20}{80} = 0.25$*."*

In the experiments in the remainder of this thesis, we will use the method suggested by Schnizlein to map real bets to abstract bets.

4.3.3 Selecting training examples

Once we have decided on a machine learning algorithm and a representation for the input to this machine learner, we should select the actual training data that we will be using. A large database with Poker games is a great start, but it will contain many different situations and types of players. We can aid the machine learning algorithm by carefully selecting sets of data from this database for which we will be learning models.

Distinguishing game states

One distinction that we can make in a data set of Poker games, is to distinguish between the different streets (preflop, flop, turn and river). The play during these streets usually differs significantly, and training a specific model for each of these streets will probably yield better models than training a single model that covers every street. Splitting the data in different streets and learning a model for each of these also has the additional advantage that we can use a specific set of features for each street. Some features will be very informative on one street, and meaningless on another.

A second distinction in situations in a Poker game is whether or not a player is faced by a bet or a raise by an opponent. If this is the case, the player can call the bet or raise and he is not allowed to check. On the contrary, when a player is not facing a bet he cannot call, but he has the option of checking instead. Furthermore, while folding is allowed without facing a bet, it is a move that is game-theoretically dominated by the better choice of checking. Knowing whether or not a player is facing a bet is therefore extremely informative: it limits the possible actions a player can take. If we split the data in cases where the player can call, fold or raise (CFR) and cases where he can check or bet (CB), we have greatly reduced the complexity of the problem compared to if we would train models that try to distinguish between all five possible actions.

Since there are four streets and two sets of allowed actions, we can distinguish $4 \times 2 = 8$ different situations that a Poker player can be in. In the remainder of this work, we will use these distinctions and will therefore learn eight models dealing with every possible situation.

Distinguishing players

Ideally, we would like to learn a model that is perfectly tailored towards the player whose actions we are trying to predict. To this end, we could learn a different model for each individual player. These models could then be trained using data from the previous games that we have played with that player. While this approach yields models that model the specific player that we are interested in, it also introduces some problems that are all related to the fact that we typically have a very limited amount of data for each single player. For example, how should such a model predict actions for situations that the targeted player has never been in before? And how should it accurately predict actions during the first few games with this player? Because Poker is a complicated game with an extremely large number of states, you would need thousands of hands before such a model would become remotely accurate. During these thousands of hands the opponent is modeled poorly and will probably have taken all your money already.

One solution to this problem of data sparsity is to go the opposite direction and to use all available data from any Poker game ever seen and use these to learn a single model. The resulting model would be very generic and model some average playing style that might not be very representative for most of the players encountered. This approach has the advantage that there is no lack of data: any Poker game ever played may be used to learn such generic models. The obvious disadvantage is that the resulting models might not model most opponents very accurately.

Davidson et al. (2000) discuss these two opposite ways to select training data for opponent models and calls the former *generic opponent modeling* and the latter *specific opponent modeling*. We have seen that there are some problems associated with both methods, which is why we propose a novel method that hopefully alleviates most major problems associated with the previous methods. Whereas these previous methods are all-or-nothing, we propose a method that combines ideas from both methods called *group-specific opponent modeling*.

4.3.4 Group-specific opponent modeling

We propose a novel method for selecting training data for opponent modeling based on ideas from *collaborative filtering* (Adomavicius and Tuzhilin, 2005). In collaborative filtering, the goal is to filter information using collaboration amongst multiple agents. For example, one application of collaborative filtering are recommendation systems that are frequently used in online stores. These systems recommend products to the user based on previous purchases or page views. The key idea behind collaborative filtering is: those who have agreed in the past, tend to agree in the future.

This idea is typically implemented in collaborative filtering systems as a two-phase process. First, users with similar interests are identified. Then, data from similar users may be used to find filter information (recommend some products, for example) for the current user.

We propose to apply these ideas to Poker opponent modeling. Instead of using only data from a single user or using all data of all users to train opponent models, we could use data from similar users. We call our method group-specific opponent modeling, as the method learns models that are specific to a group of users instead

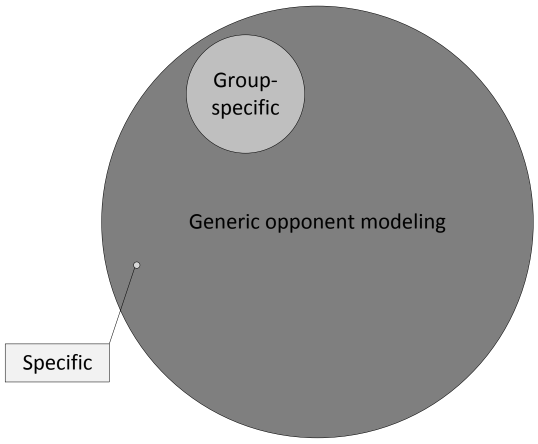


Figure 4.1: Types of opponent modeling. Generic opponent modeling uses all available data to train a generic model that models the average player. Specific opponent modeling uses only data from a single opponent to learn a model specific to that opponent. We propose a new approach called group-specific opponent modeling that uses data from a group of similar players to learn a model that models all these similar players.

of to a single user. The idea of group-specific opponent modeling is to partition all players in some large data set of previous games in n groups based on similarity in playing styles. We can then learn models for each of these groups.

The new problem we are facing now, is how to identify players with similar playing styles. One solution would be to find clusters of player types using the clustering algorithms that we have discussed in section 2.3.3. But typical clustering algorithms will need a machine-comprehensible description of each player, which requires that we define features that describe each player. It is not trivial to find good features that accurately represent the different player styles found in Poker. To circumvent the problem of having to define features that describe Poker players, we propose a novel technique to cluster players that uses the models that predict players' moves to define similarity between them, instead of vectors of features.

4.4 K-models clustering

We want to find k clusters of players so that we can train a model for each of these clusters. We expect the average accuracy of these k models to be better than the accuracy of a single model trained on all available data. That is, we expect a performance gain by clustering similar players compared to generic opponent modeling.

The ideal assignment of players to clusters therefore is one that optimizes the accuracy of the k resulting models for each of these clusters, so that these models outperform a single model trained on all data on average. This formulation of optimizing for accuracy of the k models leads to the proposal of a novel clustering technique we like to call *K-models clustering*.

K-models clustering is similar to expectation maximization and K-means clustering techniques. However, whereas in K-means clustering some distance measure such as the Euclidian distance is used to calculate the distance between an input feature vector and a cluster centroid, in K-models clustering the nearest cluster centroid is found using models. Initially, each input is randomly assigned to any of the k clusters, after which a model is trained that predicts the moves of the players in the cluster.

To train a model for a cluster, a training set should be composed that contains examples of positions in a Poker game and resulting actions for the current players in the cluster. One way to do this, would be to use all the data in the data set and weigh each example with the probability that the player in the example belongs to the cluster that we are trying to learn a model for. A simpler approach to compose a training set for a cluster is to simply select only the actions of players currently in the cluster. In the experiments that we will describe in the remainder of this thesis, we chose to use the simpler method for convenience, as we expect the resulting clusters to be of similar quality.

When a model has been trained for every cluster, the next step in the algorithm is to update the assignments of players to clusters. The new assignment for a player is found by calculating the likelihood that the model for each cluster generated the observed moves for the player. That is, the new cluster for player j can be determined as follows:

$$\begin{aligned} Cluster(j) &= \underset{i\in\{0,...,k-1\}}{\operatorname{argmax}}\ P(C_j = i | a_{j,0}, ..., a_{j,n}) \\ &= \underset{i\in\{0,...,k-1\}}{\operatorname{argmax}}\ \alpha P(a_{j,0}, ..., a_{j,n} | C_j = i) P(C_j = i) \\ &= \underset{i\in\{0,...,k-1\}}{\operatorname{argmax}}\ P(a_{j,0}, ..., a_{j,n} | C_j = i) P(C_j = i) \end{aligned} \tag{4.2}$$

Where $Cluster(j)$ is the new cluster of player j, C_j is a random variable that holds the cluster assignment of player j and $a_{j,n}$ is the n-th action of player j (counting over all games, not a single game). Note that the cluster prior probabilities $P(C_j = i)$ may be assumed to be uniform.

Just like in K-means clustering and expectation maximization, the algorithm repeats an update and assignment step until convergence. In the update step, the model for each cluster is updated. During the assignment step, all players are reassigned to the cluster that has the greatest likelihood of generating the observed data for that player. The algorithm is summarized in algorithm 1.

The K-models clustering algorithm described here has a number of advantages compared to other clustering algorithms such as classical K-means clustering. First, it does not require the definition of features that describe players: we only need to represent the actions that the players take. Second, the algorithm optimizes the

Algorithm 1 K-models clustering

```
1: Randomly assign players to $k$ clusters
2: repeat
3:    models = {}
4:    for $i = 0$ to $k - 1$ do
5:       models.append(trainModel($i$))
6:    end for
7:    for all $p \in$ players do
8:       actions = loadActions($p$)
9:       likelihoods = {}
10:      for $i = 0$ to $k - 1$ do
11:         likelihoods.append(calculateLikelihood(models[$i$], actions))
12:      end for
13:      reassignPlayer($p$, $\text{argmax}_i$ (likelihoods[$i$]))
14:   end for
15: until No changes in cluster assignments
```

cluster assignments for good action prediction accuracy. Since the reason we would like to cluster players is to train models that predict actions for each of these clusters, optimizing the cluster assignment in this way guarantees that the resulting clusters actually improve prediction accuracy.

4.5 Evaluation

In this section, we will describe experiments that we have conducted to evaluate the performance of the move prediction and player clustering techniques discussed in this chapter. We will first consider the experimental setups that we used to measure the performance of the techniques. Then we will discuss the results of these experiments and finally summarize our findings.

4.5.1 Setup

To evaluate the performance of (I) the models that predict actions and (II) the K-models clustering algorithm introduced in this chapter, we conducted a number of experiments that measure the performance of these algorithms. Since our method is intended to be suitable for computer Poker in games with any regular number of players (2 up to 10), we have set up a number of different experiments aimed at games with 2, 6 and 9 players.

Table 4.1: Sizes of the different data sets in number of games.

	Number of hands	**Number of moves**
2 player set	2,000,000	7,549,628
6 player set	1,000,000	9,122,523
9 player set	500,000	5,795,892

Table 4.2: Composition of moves in the training sets. For the 6 player set, there is a total of 9122523 moves in the set, of which the majority (4655521 or 51.03%) are fold actions. This indicates that a very simple baseline classifier that always predicts fold would get an accuracy of 51.03% on the 6 player set.

	2 player set	**6 player set**	**9 player set**
Fold	1,720,389 (22.79%)	4,655,521 (51.03%)	3,772,092 (65.08%)
Check	2,031,732 (26.91%)	1,531,169 (16.78%)	672,864 (11.61%)
Call	1,344,970 (17.82%)	1,338,930 (14.68%)	599,653 (10.35%)
Bet	989,460 (13.11%)	742,174 (8.14%)	316,356 (5.46%)
Raise	1,463,077 (19.38%)	854,729 (9.37%)	434,927 (7.50%)
Total	7,549,628 (100%)	9,122,523 (100%)	5,795,892 (100.00%)

Data sets

The website PokerAI.org provides a large database of Poker games for scientific purposes[1]. The database is composed of 600 million hands with a wide range of game structures: Texas Hold'em and Omaha Poker, limit and no-limit and all kinds of blind levels. We sampled 3 data sets with 2, 6 and 9 players from this database. All sampled games are from low stakes ($0.05/$0.10 blinds up to $1.00/$2.00 blinds). The sizes of the data sets are shown in table 4.1. The composition of each of these data sets is provided in table 4.2.

Experiments

We ran the K-models clustering algorithm three times on each of the 2, 6 and 9 player data sets. In order to obtain fair measurements of the performance of the resulting models, only 90% of the players in a cluster were used to train models for that cluster on each iteration. This allows us to use the other 10% of the members of each cluster to measure the performance of the models trained for that cluster for that iteration.

This way, we obtain a set of models, prediction accuracies on the train and test sets and a list of members for each cluster for each iteration. We would like to use these to assess (I) how well the clustering algorithm has grouped players with similar playing styles and (II) how well the models predict moves for players in its cluster.

[1]`http://pokerftp.com/index.htm`

We will discuss methods to measure the algorithm's performance at both of these tasks now.

Evaluating cluster quality

To be able to evaluate how good the proposed algorithm performs at grouping similar playing styles, we will need some measure to quantify the quality of an assignment of players to clusters. Human Poker experts often use a set of features to describe the playing style of an opponent when discussing Poker strategy. We will use these features to describe the average playing style of a certain cluster. We can then quantify whether the algorithm has found sensible clusters of players by comparing the average values of these features for different clusters. The features that human Poker experts often use to describe a player's playing style are:

- **Voluntarily put money in pot % (VPIP)**: the percentage of the games in which the player has voluntarily put any amount of money in the pot. Voluntarily in this sense means that posting the small or big blind does not count towards this percentage, as these actions are obligatory.
- **Pre-flop raise % (PFR)**: the percentage of the games in which the player has made a (re-)raise before the flop.
- **Aggression factor (Aggr)**: the number of times the player has bet or raised divided by the number of times the player has called. For the results presented in section 4.5.2, we set this value to 100 if the player has bet or raised but never called.

Furthermore, we expect that when sensible clusters are being formed by the proposed algorithm, the performance of the models that predict actions for each of these clusters should increase with each iteration of the clustering algorithm: as the composition of each of these clusters becomes more homogeneous, it should be easier to learn good models that predict actions for the players in the cluster. This implies that we can indirectly evaluate the quality of the resulting clusters by comparing the performance of models generated for an initial random clustering with the performance of the models generated for the final clustering.

Evaluating model quality

To measure the performance of the models learned for each cluster by the proposed algorithm, we can use typical machine learning evaluation techniques such as accuracy measures or confusion matrices. As we have mentioned earlier, we separate the players in each cluster in each iteration in a train group and a test group. The former is used to train the models that predict actions for the cluster, the latter may be used to evaluate the trained models. We will compare the performance of the models for each of these groups using their accuracy and confusion matrices.

4.5.2 Results

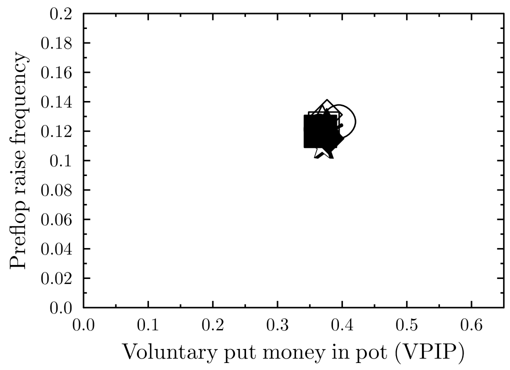
Average feature values for clusters

In this section we will consider how the average values of a number of features that human Poker experts often use to describe playing styles changes for different clusters with iterations of the clustering algorithm, as we have discussed in section 4.5.1. Figure 4.2 illustrates the average values of these features for 10 different clusters for one run of the clustering algorithm for 6 player games. The left column shows scatter plots of the mean values of the voluntary pot in pot (VPIP) feature versus the preflop raise frequency (PFR) feature. The right column shows similar plots for VPIP versus aggression factor. The rows show the values of these statistics after different numbers of iterations of the clustering algorithm. The size of the markers is based on the sum of the standard deviations of both statistics for the associated cluster: the larger the standard deviations, the larger the marker. Each particular cluster is indicated with the same marker symbol in all six scatter plots.

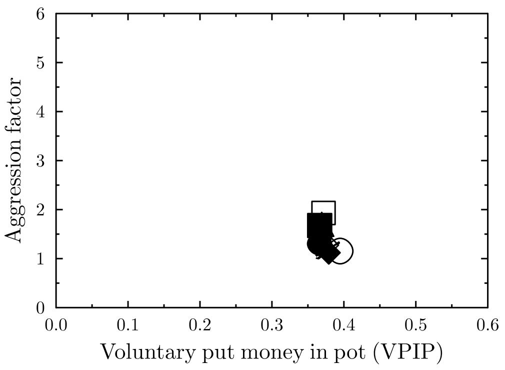
After 0 iterations (after random initialization), all markers are lumped together and the markers have a relatively large size when compared with later iterations (which is indicative of large standard deviations within clusters). After 3 iterations, the markers have spread out and decreased in size. After convergence, the markers have spread out even further and have become smaller: the average values of the features are different for different clusters, and the standard deviation within clusters has decreased.

Model accuracies per iteration

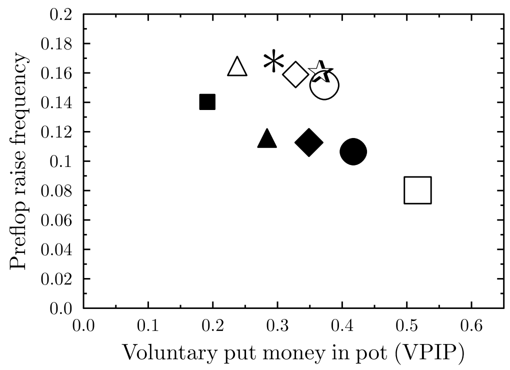
Figure 4.3 shows how the average accuracy of models improves with the number of iterations of the clustering algorithm. The figure contains graphs for two, six and nine players respectively. All graphs were generated by averaging the accuracy of all clusters over three runs of the algorithm. The errors bars show the 95% confidence interval for the accuracy of that iteration.



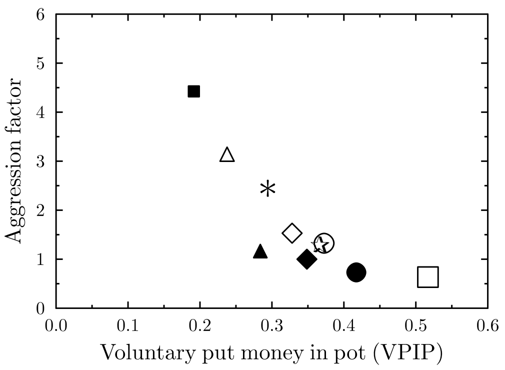
(a) VPIP versus PFR after 0 iterations.



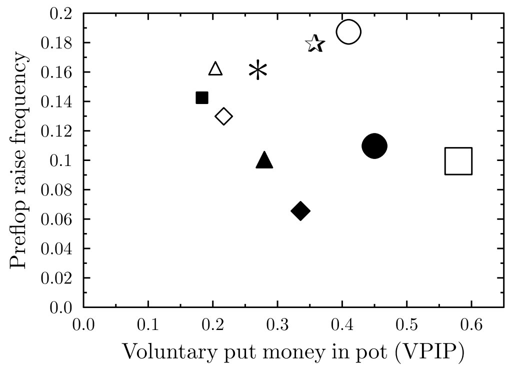
(b) VPIP versus aggr. after 0 iterations.



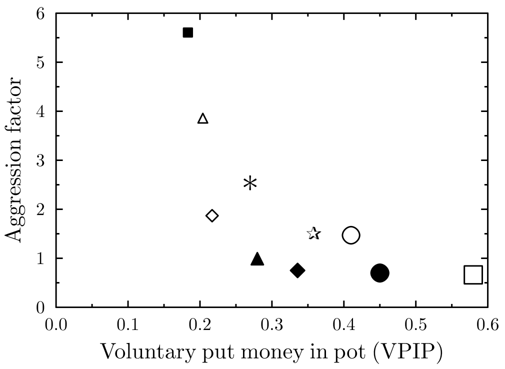
(c) VPIP versus PFR after 3 iterations.



(d) VPIP versus aggr. after 3 iterations.

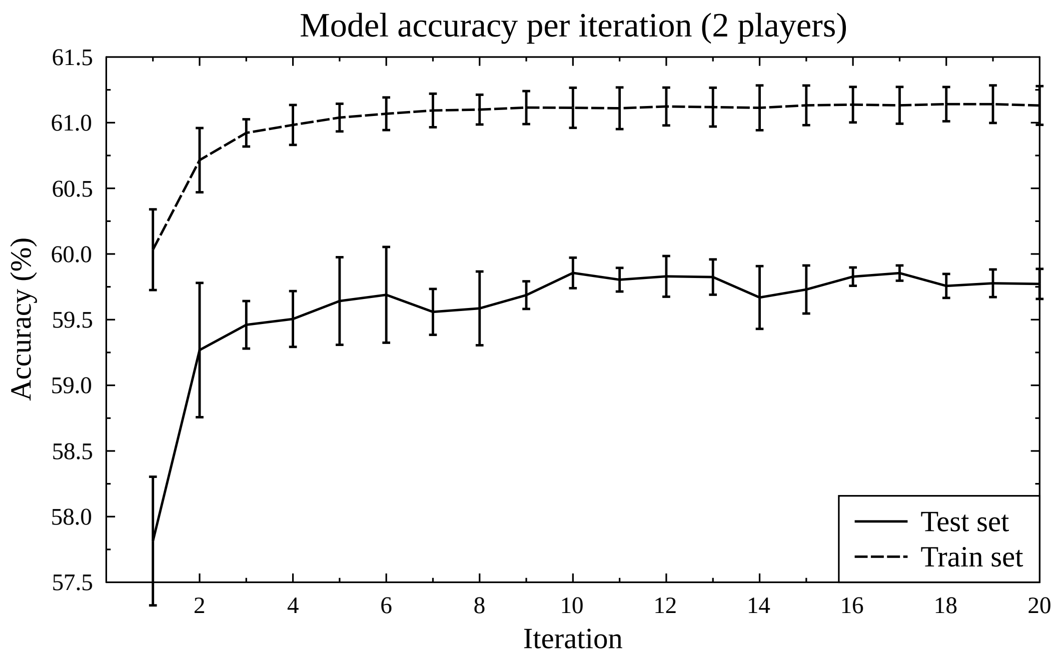


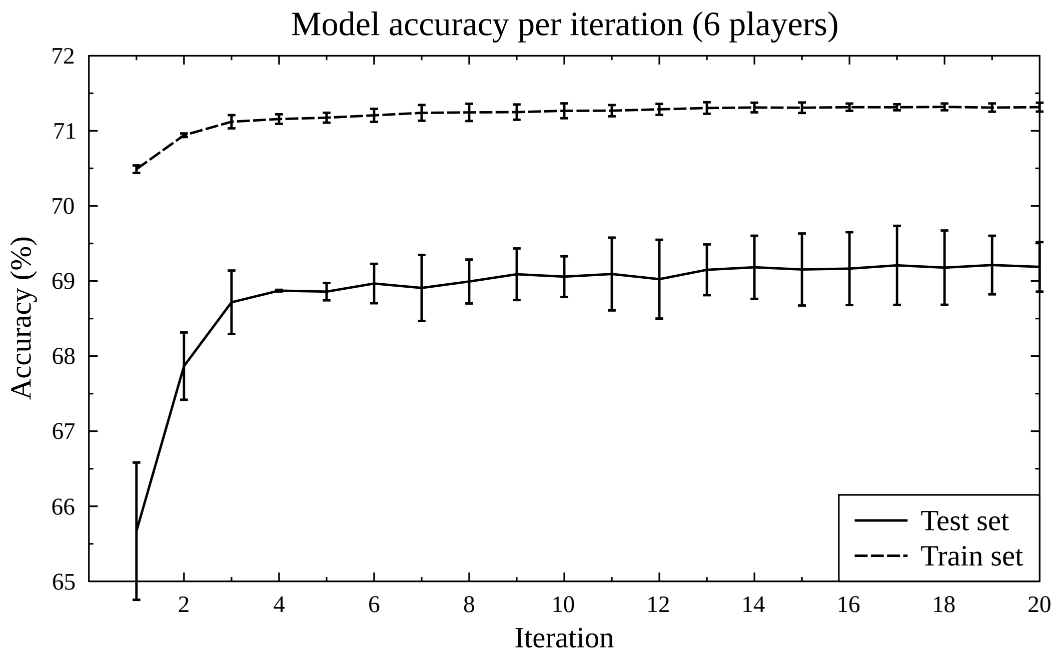
(e) VPIP versus PFR after convergence.



(f) VPIP versus aggr. after convergence.

Figure 4.2: The left column shows scatter plots of the mean values of the voluntary put money in pot (VPIP) and preflop raise frequency (PFR) statistics for 10 clusters. The right column shows similar scatter plots where the aggression factor is plotted versus VPIP. The rows show how these values change with iterations of the algorithm. The size of the markers depends on the sum of the standard deviations of both statistics for the associated cluster.


Model accuracy per iteration (2 players)
Accuracy (%)
61.5
61.0
60.5
60.0
59.5
59.0
58.5
58.0
57.5
2
4
6
8
10
12
14
16
18
20
Iteration
Test set
Train set


Model accuracy per iteration (6 players)
Accuracy (%)
72
71
70
69
68
67
66
65
2
4
6
8
10
12
14
16
18
20
Iteration
Test set
Train set

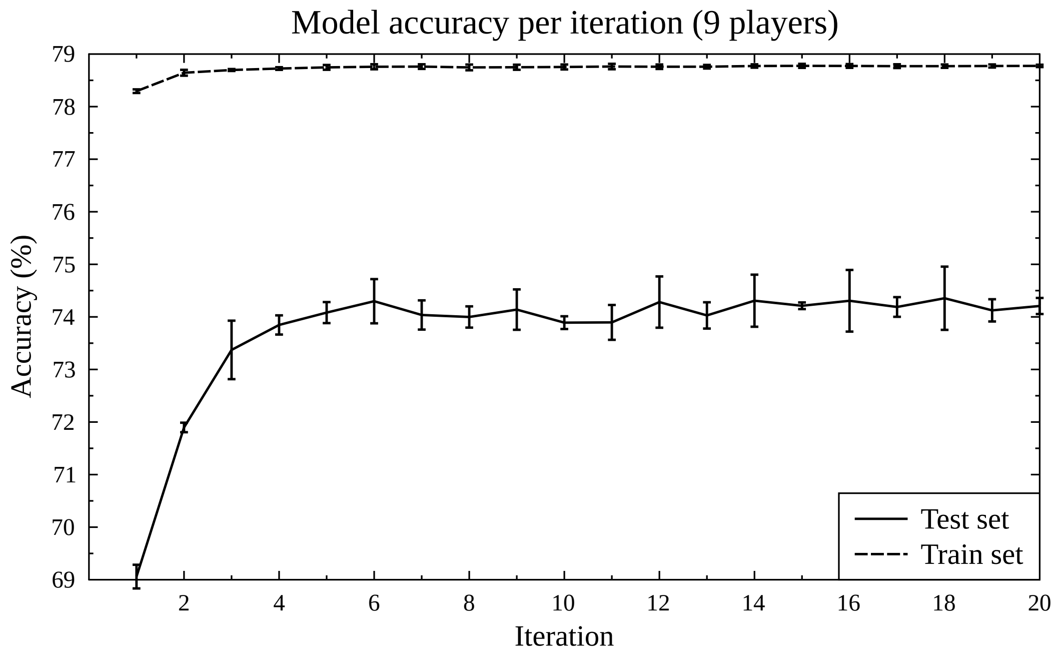


Figure 4.3: The three graphs shown here illustrate how the accuracy of the models trained for each cluster improves with more iterations of the clustering algorithm. Each figure was generated using three runs of the clustering algorithm. The error bars indicate the 95% confidence intervals for the mean accuracy over these three runs. The naive baselines for these accuracies are 26.91%, 51.03% and 65.08%, respectively (table 4.2). These baselines have been ommitted in the graph to ensure visibility of fluctuations in test and train accuracies.

Move prediction confusion matrices

Table 4.3, table 4.4 and table 4.5 provide confusion matrices that illustrate how well the models obtained during the clustering algorithm predict players' moves after convergence for games with two, six and nine players respectively. The error ranges shown are 95% confidence intervals. The results are the averages over all clusters over three runs of the clustering algorithm.

Bet size accuracy

As we have discussed in section 4.3.2, we map all bet and raise moves to one of seven abstract moves. Table 4.6 shows how accurately the models predict the correct abstract bet after convergence. Similarly, table 4.7 illustrates how accurately the models predict the right abstract raise move. These tables were built by considering all bet and raise moves in the test parts of the data sets. We then consider a model's predicted action to be the the abstract bet or raise action for which the model outputs the largest probability.

Table 4.3: Confusion matrix for move prediction for 2 player games (accuracy = 0.596 ± 0.001). The bounds shown are 95% confidence intervals. Note that the sums of the columns are not identical to the figures in table 4.2. This is caused by the fact that these confusion matrices are calculated on test sets sampled from the complete data set.

		Actual					
		Fold	**Check**	**Call**	**Bet**	**Raise**	**Sum**
Predicted	**Fold**	$9.73 \pm 0.00\%$	$0.00 \pm 0.00\%$	$6.26 \pm 0.00\%$	$0.00 \pm 0.00\%$	$2.97 \pm 0.00\%$	18.96%
	Check	$0.02 \pm 0.00\%$	$24.99 \pm 0.01\%$	$0.00 \pm 0.00\%$	$11.18 \pm 0.01\%$	$0.89 \pm 0.00\%$	37.09%
	Call	$7.14 \pm 0.01\%$	$0.00 \pm 0.00\%$	$12.62 \pm 0.01\%$	$0.00 \pm 0.00\%$	$4.72 \pm 0.01\%$	24.47%
	Bet	$0.00 \pm 0.00\%$	$1.11 \pm 0.00\%$	$0.00 \pm 0.00\%$	$2.49 \pm 0.00\%$	$0.00 \pm 0.00\%$	3.60%
	Raise	$4.59 \pm 0.00\%$	$0.00 \pm 0.00\%$	$1.48 \pm 0.01\%$	$0.00 \pm 0.00\%$	$9.79 \pm 0.01\%$	15.86%
	Sum	21.48%	26.11%	20.37%	13.67%	18.37%	100%

Table 4.4: Confusion matrix for move prediction for 6 player games (accuracy = 0.692 ± 0.003).

		Actual					
		Fold	**Check**	**Call**	**Bet**	**Raise**	**Sum**
Predicted	**Fold**	$43.53 \pm 0.34\%$	$0.00 \pm 0.00\%$	$10.01 \pm 0.07\%$	$0.00 \pm 0.00\%$	$7.34 \pm 0.06\%$	60.89%
	Check	$0.03 \pm 0.00\%$	$16.88 \pm 0.13\%$	$0.00 \pm 0.00\%$	$7.23 \pm 0.05\%$	$0.37 \pm 0.00\%$	24.52%
	Call	$3.73 \pm 0.03\%$	$0.00 \pm 0.00\%$	$7.00 \pm 0.05\%$	$0.00 \pm 0.00\%$	$1.34 \pm 0.01\%$	12.07%
	Bet	$0.00 \pm 0.00\%$	$0.67 \pm 0.01\%$	$0.00 \pm 0.00\%$	$1.83 \pm 0.01\%$	$0.00 \pm 0.00\%$	2.50%
	Raise	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.01 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.01 \pm 0.00\%$	0.02%
	Sum	47.30%	17.55%	17.02%	9.07%	9.06%	100%

Table 4.5: Confusion matrix for move prediction for 9 player games (accuracy = 0.742 ± 0.001).

		Actual					
		Fold	**Check**	**Call**	**Bet**	**Raise**	**Sum**
Predicted	**Fold**	$55.89 \pm 0.02\%$	$0.00 \pm 0.00\%$	$10.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$6.55 \pm 0.00\%$	72.44%
	Check	$0.02 \pm 0.00\%$	$12.54 \pm 0.00\%$	$0.00 \pm 0.00\%$	$5.36 \pm 0.00\%$	$0.22 \pm 0.00\%$	18.13%
	Call	$2.18 \pm 0.00\%$	$0.00 \pm 0.00\%$	$4.19 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.84 \pm 0.00\%$	7.21%
	Bet	$0.00 \pm 0.00\%$	$0.62 \pm 0.00\%$	$0.00 \pm 0.00\%$	$1.58 \pm 0.00\%$	$0.00 \pm 0.00\%$	2.20%
	Raise	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.01 \pm 0.00\%$	0.01%
	Sum	58.10%	13.16%	14.19%	6.94%	7.62%	100%

Table 4.6: This table shows the accuracy with which the models predict the right amount of a bet move. The error bounds are 95% confidence intervals.

Data set	**Bet accuracy**	**Baseline**
2 players	$77.1 \pm 0.5\%$	68.04%
6 players	$63.5 \pm 0.2\%$	45.38%
9 players	$59.9 \pm 0.2\%$	41.10%

Table 4.7: This table shows the accuracy with which the models predict the right amount of a raise move. The error bounds are 95% confidence intervals.

Data set	**Raise accuracy**	**Baseline**
2 players	$88.5 \pm 1.1\%$	85.00%
6 players	$58.4 \pm 0.8\%$	55.25%
9 players	$72.3 \pm 4.9\%$	54.55%

The confusion matrices that these summary tables are based on are provided in appendix C. One interesting observation found in these tables is that some of the abstracted moves that we chose to use (cf. 4.3.2) are very uncommon in some types of games, while they are seen quite frequently in other types of games. For example, $\frac{3}{2}$ pot bets or raises are very uncommon in all games and $\frac{1}{2}$ pot raises are very uncommon in 2 player games, but fairly common in other games.

4.6 Conclusion

In this chapter we have discussed how we can learn models that predict moves for players. The proposed solution is based on group-specific opponent modeling: we introduce a method based on clustering playing styles and learning models for each of these clusters. The resulting method uses a novel clustering algorithm called "K-models clustering". This clustering scheme does not require descriptions of individual players as input. Instead, the algorithm generates an assignment of players to clusters using sets of moves for each player and a machine learner that can learn to predict moves for clusters of players based on these players' actions.

This approach has a number of advantages. First, it avoids the hassle of finding suitable features that accurately describe players' playing styles. Second, the method effectively optimizes the cluster partitioning for accurate models: since our goal is to obtain accurate models for each of the clusters, this is a great property. Third, we automatically obtain models that predict moves for the players in each cluster as a by-product of the clustering process.

The results of our experiments indicate that the resulting clusters indeed represent different playing styles. In section 4.5.2, we considered the average values of three features that are frequently used by human Poker experts to describe playing styles for different clusters. We found that the average values of these features for different clusters indeed diverges with subsequent iterations of the clustering algorithm. We also showed that the standard deviations of these features decreased within clusters with iterations of the algorithm. Furthermore, we found that the resulting models outperform baseline models and that the performance of the models indeed increases with iterations of the algorithm (section 4.5.2).

Note that aside from the features that describe the state of the game, all of the methods developed here make no Poker-specific assumptions. The K-models clustering algorithm can therefore be used in other games or completely different fields, such as collaborative filtering. Another interesting use of the clustering algorithm would be in feature validation: once we have established that the algorithm indeed finds meaningful clusters of entities without using any features that describe these entities, we can compare the values of features that *do* describe these entities for the obtained clusters and validate whether the features indeed captures differences between clusters.

We may conclude that the methods developed in this chapter perform their tasks nicely: we obtain meaningful clusters and accurate models that predict the actions for players in each of these clusters.

Chapter 5

Simulating showdowns: predicting holdings

5.1 Introduction

In chapter 4, we have discussed how players' moves can be predicted. Using decision trees, we obtain a probability distribution over all possible actions, given a description of the current state of the game and the actions leading to it. We could use these probability distributions to simulate Poker games for a Monte Carlo computer Poker agent, but one problem remains. When two or more players reach the showdown phase in which they reveal their hidden cards to determine the winner of the pot, how do we simulate who wins the game?

To solve this problem, we will need to develop models that can estimate the probabilities of each player winning a showdown. That is, we want to estimate the probability that player i wins the pot, given a description of the state of the game and the actions leading to the showdown. We could try to estimate these probabilities directly using a machine learning algorithm as we have done in the previous chapter. A different and probably better solution would be to try to predict the actual cards that all players will show down. That is, we could try to estimate the probability distributions over all possible holdings for each player involved in the showdown. We could then use these distributions over hidden cards to estimate the chances of each player holding the best hand.

5.2 Predicting players' holdings

There are several options that will allow us to calculate a probability distribution over all holdings for a player, based on the moves that the player selected during a game. One would be a direct machine learning approach in which the input of the system is a description of the game and the actions that all players took, and the output is a probability distribution over all 1326 holdings. There are a number of problems with such an approach however. For instance, not all machine learners handle problems with 1326 possible output classes very well. Another problem is how we should compose a training set: if all we have is a database of Poker hands, how do we extract pairs of game states and desired probability distributions over holdings from it?

A totally different approach would be to try to estimate the probability distribution over holdings indirectly. Bayes' formula allows us to calculate a posterior probability for a holding given a player's actions, using the "opposite" probability of observing a player's actions given that holding. Since we already have methods that estimate the probability of a player taking a certain action, all we need to do is extend the models used in these methods to incorporate a player's holding in the input.

5.2.1 Formalization

More formally, we want to find $P(H|a_n, \ldots, a_0, \vec{f_n}, \ldots, \vec{f_0})$ where H is the holding under consideration, a_n is the n-th action in the game, and f_n is a feature vector that describes the state of the game at the time of action a_n. Since ideally f_n describes the entire state of the game and all that has happened so far, f_n encapsulates all of $f_{n-1}, \ldots, f_0$ and $a_n - 1, \ldots, a_0$ and may be used to replace these. Applying these ideas, we find:

$$\begin{aligned} P\left(H|a_n, \ldots, a_0, \vec{f_n}, \ldots, \vec{f_0}\right) &= P\left(H|a_n, \vec{f_n}\right) \\ &= \frac{P\left(H, a_n, \vec{f_n}\right)}{P\left(a_n, \vec{f_n}\right)} \\ &= \frac{P\left(a_n|H, \vec{f_n}\right) P\left(H, \vec{f_n}\right)}{P\left(a_n, \vec{f_n}\right)} \\ &= \frac{P\left(a_n|H, \vec{f_n}\right) P\left(H\right) P\left(\vec{f_n}\right)}{P\left(a_n, \vec{f_n}\right)} \\ &= \alpha P\left(a_n|H, \vec{f_n}\right) P\left(H\right) \end{aligned} \tag{5.1}$$

In which α is a constant that results in normalization, so that the sum of all probabilities for all holdings is 1. The derivation shows that we can indeed calculate a probability distribution over all holdings if we can estimate the probability that a player will take some action given a description of the game and his holding. One problem with this approach is that our feature vector is not perfect and does not capture every possible detail of the game. The first step in the derivation in which $P(H|a_n, \ldots, a_0, \vec{f_n}, \ldots, \vec{f_0})$ is reduced to $P(H|a_n, \vec{f_n})$ is rather coarse and will probably introduce quite some error.

Luckily, we can do better. Let us consider the following derivation in which the reduction of $a_{n-1}, ..., a_0, f_{n-1}, ..., f_0$ to f_n is used in later stages, making the reduction less coarse.

$$
\begin{aligned}
P\left(H|a_n, \dots, a_0, \vec{f_n}, \dots, \vec{f_0}\right) &= \frac{P\left(H, a_n, \dots, a_0, \vec{f_n}, \dots, \vec{f_0}\right)}{P\left(a_n, \dots, a_0, \vec{f_n}, \dots, \vec{f_0}\right)} \\
&= \alpha P\left(a_n| \dots, a_0, \vec{f_n}, \dots, \vec{f_0}, H\right) \\
&\quad P\left(a_{n-1}, ..., a_0, \vec{f_n}, \dots, \vec{f_0}, H\right) \\
&= \alpha P\left(a_n|..., a_0, \vec{f_n}, \dots, \vec{f_0}, H\right) \\
&\quad P\left(a_{n-1}|..., a_0, \vec{f_{n-1}}, \dots, \vec{f_0}, H\right) \\
&\quad P\left(..., a_0, \vec{f_n} - 1, \dots, \vec{f_0}, H\right) \\
&= \alpha P\left(a_n|..., a_0, \vec{f_n}, \dots, \vec{f_0}, H\right) \\
&\quad P\left(a_{n-1}|..., a_0, \vec{f_{n-1}}, \dots, \vec{f_0}, H\right) \\
&\quad \dots \\
&\quad P\left(a_0|f_0, H\right) \\
&= \alpha P\left(a_n|\vec{f_n}, H\right) \\
&\quad P\left(a_{n-1}|\vec{f_{n-1}}, H\right) \\
&\quad \dots \\
&\quad P\left(a_0|f_0, H\right) \\
&= \alpha P\left(H\right) \prod_{i=0,\dots,n} P\left(a_i|\vec{f_i}, H\right)
\end{aligned}
\tag{5.2}
$$

Note that α is a normalization constant again. Furthermore, the derivation uses the fact that a_{n-i} is conditionally independent of $f_n, ..., f_{n-i+1}$. Again, we find that we can obtain an estimate of the probability for each holding in terms of $P(a_i|\vec{f_i}, H)$ and the prior $P(H)$, which we know to be uniform. In the following sections, we will discuss how we can learn models that approximate $P(a_i|\vec{f_i}, H)$.

5.3 Predicting actions given cards

As we have discussed in the previous section, if we could learn models that predict a player's actions given his cards and a feature vector, we could use these to calculate a probability distribution over all holdings. In chapter 4, we have considered how models can be learned that predict actions given the current position in the game (without considering a player's hidden cards). We would like to expand the methods developed there to learn models that additionally use a player's holding to estimate the probability distribution over possible actions.

First, we will have to change the input the decision tree learner receives to incorporate the player's cards. One way to do this would be to include features that describe the player's cards to the set of features that describe the game state (cf. section 4.3.1). A good numerical feature that immediately comes to mind is the current hand strength. Another way to incorporate a player's cards, is to train separate models for each holding, just like we trained separate models for each street and player type in section 4.3.3. While both methods are valid, we will focus on the latter because that method will prove to be more convenient for use in techniques that we will discuss in the remainder of this chapter.

5.3.1 Training data

The training set for a machine learner that will learn models that predict actions given some holding, should be composed of data of players who held that specific holding. The great problem here is that players' holdings are hidden in Poker: they are only revealed at showdown. If we observe a large number of Poker games with the intention of using these games as training data for the problem at hand, the majority of the data will lack essential information (the player's holding) and be unusable. For example, in the training sets for 2, 6 and 9 players used in section 4.5.1, the player's cards are known for only respectively 10.1%, 5.3% and 3.2% of the data.

To worsen our problems, the problem is not just that we will have to observe more games because a large part of our data cannot be used. We cannot simply use the part of the data where we do know the players' cards either. The reasons for this are twofold.

First, the models that we would learn from this part of the data would never predict a fold, because the data contains no fold actions. Since such a data set consists solely of actions of players who went to showdown, they cannot have folded (or they would not have reached the showdown).

Second, the data would be greatly biased towards mostly very strong cards and some very weak cards. The fact that the player reached a showdown implies that he either had a hand worth going to showdown with, or a hand that he did not want to bet and luckily got to showdown with because no opponent made any bet either. The play in games that went to showdown is not representative for the average Poker game because of this bias.

We will revisit this problem and propose a solution in section 5.5 of this chapter. First, we will address a different problem that is caused by the fact that a holding alone is not sufficient to make accurate action predictions during the post-flop phase of a Poker game.

5.4 Bucketing

The observant reader may have noticed that during the post-flop phase – in which there are community cards showing – the holding of a player alone is not very informative anymore. For example, during the pre-flop phase the fact that a player has 4♣8♡ is enough to conclude that the player has a weak holding. During the post-flop phase, the fact that a player has 4♣8♡ is not enough information to conclude whether he has a strong or a weak holding. On a 5♣6♢7♡ flop, the holding turns into a very strong one, whereas on a board of A♠A♣Q♠ it is still a very weak holding.

If we want to predict actions given a player's holding during the post-flop phase, we will thus have to include more information than just the holding to get meaningful predictions. One solution to account for this is to include the community cards in our description of the game, along with the holding cards. The weakness of this solution is that a machine learner will also have to learn how certain combinations of holdings and community cards affect the outcomes it should predict: it would have to learn the rankings of Poker hands, introducing more difficulties in an already rather complex problem. A better solution therefore is to do this job for the machine learning and to replace the combination of a holding and the community cards with a variable that describes the strength of the combination of hole cards and community cards.

One such approach is called *bucketing* and is frequently used in Poker literature (Johanson, 2007, pp. 23–28). As the name implies, holdings that are strategically similar are grouped together in a "bucket" and this bucket is used to represent all the holdings it contains. For example, we could define two equally-sized buckets for the flop: one for weak holdings and one for strong holdings. If the flop then came A♣K♠4♡, A♡A♠ and K♣K♡ would be in the bucket with strong hands, whereas 2♣3♡ and 2♣5♣ would be in the bucket with weak hands. If the flop came 2♠2♡3♣ however, the bucket assignments would be the exact opposite.

Note that if we are using bucketing, a player no longer has a holding. Instead, he has a bucket that represents the community cards and a set of holdings, one of which is his actual holding. So, instead of predicting actions given a holding as discussed in the previous section, the goal would be to predict actions given a bucket. Post-flop bucketing also effectively removes the community cards altogether, since they are included in the buckets.

Bucketing can also be useful during the pre-flop phase, in which there are $\binom{52}{2} = 1326$ different combinations of 2 cards. These 1326 possible holdings can be grouped in 169 different buckets without any loss of information. For example, there is no strategic difference between A♣K♣ and A♡K♡ when there are no community cards yet: neither is better than the other. By grouping all such *suit-symmetrical* holdings in buckets, the problem space for pre-flop Poker can be decreased from 1326 to 169 classes without any loss of information.

5.4.1 Hand strength

The strength of a combination of holding cards and community cards in Poker is typically represented by a measure called *hand strength.* The hand strength may be calculated by counting how often the holding under consideration wins, ties and loses against all possible other holdings at the current board. The actual hand strength is then given by:

$$\text{HS} = \frac{\text{wins} + \text{ties}/2}{\text{wins} + \text{ties} + \text{losses}} \tag{5.3}$$

Note that this measure only considers the current set of community cards and does not account for cards that are to come on future streets. For example, the hand strength of 2♣3♣ on a flop of 4♣5♣J♠ is extremely low, because it currently is only jack high. The hand has a lot of potential however, because it has good odds of making a straight or a flush on the turn or river, which would skyrocket its hand strength. It is therefore that the expected value, $E(HS)$ of the hand strength is frequently used to express the strengh of a holding on a board instead. This expected value is calculated by *rolling out* all combinations of community cards and calculating the hand strength measure on these completed boards. The expected value of the hand strength over all rollouts is then used as a measure of strength of the holding.

5.4.2 Types of bucketing

In the computer Poker literature, a fairly large number of different bucketing strategies have been proposed. One of the simplest is *uniform bucketing*, in which there are N buckets and every bucket receives a fraction $1/N$ of the total number of holdings. Which holding goes to which bucket is determined by calculating the expected hand strengths of every holding and ranking all holdings according to their hand strength. The lowest $1/N$ fraction of the holdings go to the first bucket, the next $1/N$ fraction to the second, and so on.

Other types of bucketing include multi-dimensional bucketing in which the bucket assignment depends on more than 1 measure, *percentile* bucketing and *history* bucketing. A thorough review of these techniques is beyond the scope of this thesis, but it available in (Johanson, 2007).

5.4.3 Weighted bucketing

The type of bucketing that we choose to use for all our experiments is a weighted form of bucketing. This type of bucketing is based on how human Poker experts reason. For example, on a flop of A♣K♠4♡, they would consider hundreds of holding such as 2♣3♡ and 2♣5♣ as extremely weak and uninteresting, a larger number of holdings as medium strength, and a very small number of holdings such as AA, KK, 44 or AK as extremely strong. Note that the stronger the holdings are, the smaller the number of holdings that are similar in strength.

We chose to adopt a similar bucketing system in which weak holdings go to large buckets and strong hands go to smaller buckets called weighted bucketing. Every bucket b_i has an associated weight w_i that determines the fraction of holdings it should receive. The larger the weight, the more important the bucket is and the fewer holdings it receives. The fraction of all holdings that bucket i receives is given by:

$$\frac{1}{w_i} / \sum_j \frac{1}{w_j} \tag{5.4}$$

In all our experiments that use this bucketing, we use 20 flop buckets, 20 turn buckets and 10 river buckets. We use an exponential function for the weights of the buckets, so that the buckets for strong hands have a significantly larger weight than the first bucket, which has a very small weight and receives all weak hands.

Expected hand strength squared

Instead of the expected value of the hand strength ($E(HS)$), we use a related metric called expected hand strength squared ($E(HS^2)$) (Johanson, 2007). This is the expected value of the *square* of the hand strength. The advantage of this metric is that it assigns higher values for hands with a lot of potential. For example, consider two holdings a and b. Holding a results in a mediocre hand on the river twice with hand strengths 0.5, 0.5. Holding b is a drawing hand that results in either a weak hand or a strong hand, 0.2, 0.9 for example. The $E(HS)$ values for holding a and b are $(0.5+0.5)/2 = 0.5$ and $(0.2+0.8)/2 = 0.5$, respectively. The $E(HS^2)$ metric assigns the values of $(0.5^2 + 0.5^2)/2 = 0.25$ and $(0.2^2 + 0.8^2)/2 = 0.4$: the $E(HS^2)$ metric prefers hands with potential over mediocre hands with a small chance to improve, which more closely resembles human play.

5.4.4 Soft bucketing

A problem with bucketing methods is that they typically perform a *hard* mapping: every holding is assigned to a single bucket and every bucket thus contains a set of n holdings. This is a problem because it does not handle edge cases where a large number of holdings has an equivalent strength gracefully.

Consider for example the extreme case where the board is T♠J♠Q♠K♠A♠ and every player in the game has a royal flush. In this case, it does not matter which holding you have and each of the 1326 possible holdings thus has the exact same strength. Suppose that we use k buckets and uniform bucketing, in which each of the k buckets receives $1326/k$ holdings. How should we assign each holding to exactly one bucket?

Figure 5.1 illustrates this problem for three buckets and three holdings. We would like to assign each holding to exactly one bucket based on strength. It is obvious that there is no sensible way to do this, because every holding equals the exact same final hand: a royal flush.

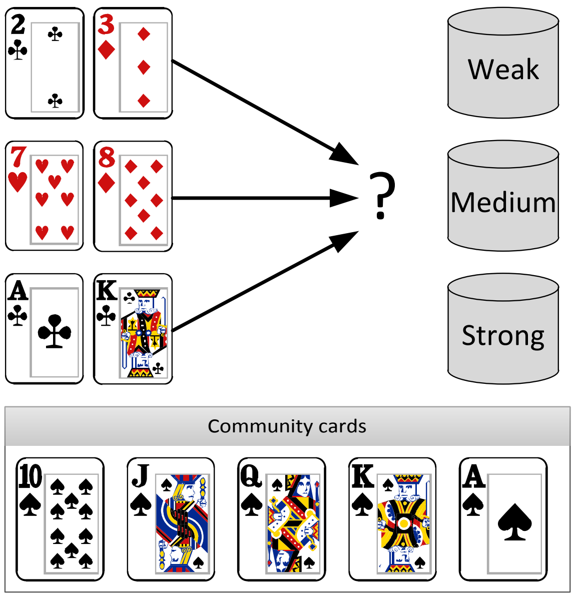


Figure 5.1: An edge case that causes problems for traditional hard bucketing. The goal is to pair each of the holdings 2♣3♢, 7♡8♢ and A♣K♣ with exactly one bucket based on the strength of the holding. Since all holdings are equally strong (as they all result in a royal flush, which is on the board), there is no sensible assignment of holdings to buckets.

One obvious solution to this problem would be to just assign all holdings with the same strength to a single bucket. For example, in the case of the royal flush on the board, we could just assign every holding to the strongest bucket. While this seems an effective and simple solution, it is not valid because it changes the meaning of the buckets accross different boards. The idea behind buckets is that they represent a group of holdings on some board that are very similar for different combinations of sets of holdings and boards. For example, in a uniform bucketing setup with 10 buckets, the strongest bucket should always contain the top 10% of the holdings on *any board*. Were we to assign all 1326 possible holdings to this bucket, it no longer represents the top 10% of the hands but rather all hands.

In our experiments, we adopt a technique called *soft bucketing*, in which there is no hard assignments of holdings to buckets, but rather a soft assignment in which every holding can partially be in a number of buckets. Every holding then has an associated probability distribution over all possible buckets. Let us reconsider the problem illustrated in figure 5.1, in which we are trying to assign three holdings to three buckets with a royal flush on the board. Using soft bucketing, we find that every holding has a probability distribution of $1/3, 1/3, 1/3$ over the three buckets (figure 5.2). Using this approach, every bucket retains its meaning for all possible boards, and we can find meaningful assignments of holdings to buckets for boards in which many holdings have an identical strength.

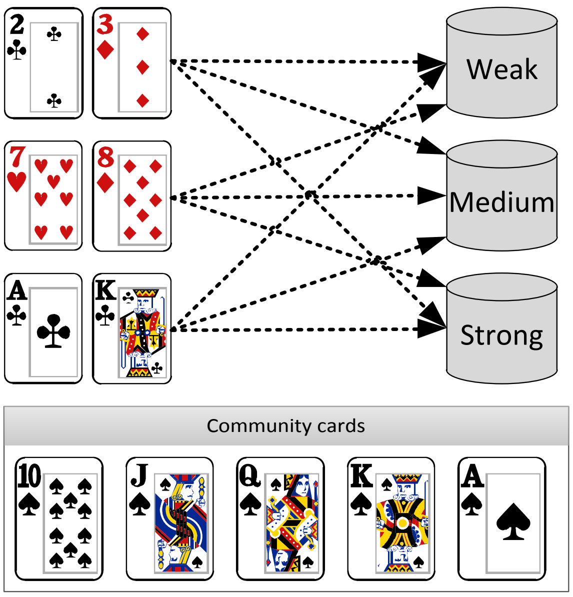


Figure 5.2: A solution to the problem illustrated in figure 5.1 using soft bucketing, in which every holding is partially assigned to every bucket. In this case, every holding is for 1/3 in every bucket.

5.4.5 From buckets to holdings and back

In a real Poker game, there are no buckets and only holdings. If we want to use buckets in our algorithms for Poker games, we will therefore need to be able to convert holdings to buckets and back. Most importantly, we will need to be able to convert between the probability that a player has a certain holding and the probability that he has a certain bucket.

The probability that a player has a certain holding h_i, given a probability distribution over buckets and bucket membership probabilities may be obtained usign:

$$P(H = h_i) = \sum_j m_{i,j} P(B = b_j) \tag{5.5}$$

Where $P(H = h_i)$ is the probability that he player has holding h_i, $m_{i,j}$ is the membership probability that holding h_i belongs in bucket j and $P(B = b_j)$ is the probability that the player has a holding in bucket j. Note that this is a simple weighted summation.

Similarly, the probability that the player has a holding in a certain bucket b_j may be obtained using the following equation:

$$P(B = b_j) = \sum_i m_{i,j} P(H = h_i) \tag{5.6}$$

5.5 Algorithm

As we have discussed in section 5.3.1, we are facing the problem that the small portion of the data for which we know the hole cards is not sufficient to train unbiased models that predict actions for a bucket. In this section, we will present an algorithm that aims to circumvent this problem by using all of the data to train models.

The basic idea of the algorithm is that we can use the data for which we do know hole cards (the labeled data from now on) to iteratively improve an estimate of the hole cards for all players in the part of the data for which we do not know the hole cards (the unlabeled data from now on). To this end, we keep a probability distribution over all holdings for every player in every game in the unlabeled data. These probability distributions are all initialized to an educated guess that is calculated as follows.

Since we know that every holding has an equal chance of being dealt to a player, we know that the distribution over all holdings in all of the data (the unlabeled and labeled data combined) must approximately be uniform. We can determine the distribution of all holdings for the labeled data since these are observable. By subtracting these two distributions and applying weighing to account for different sample sizes, we can infer the distribution over all holdings for the unlabeled data. We can then use this distribution as our initial probability distribution for all players in the unlabeled data set.

By guessing an initial probability distribution over the holdings of all players whose cards were not revealed, we effectively label the large unlabeled part of the data. After this initialization step, all data can be considered as labeled and we can treat it as such. We now have a distribution over all holdings for every player in the data set and can use the data to learn models that predict actions, given some description of the state of the game and a holding or bucket.

Unfortunately, the models that we would obtain from this data set in which 90 to 97 percent of the holdings have been initialized to the same rather inaccurate educated guess will probably not be very accurate. However, as we have seen in section 5.2, we can obtain estimates of the probability distribution over all holdings for each player from the moves that the player selected and a set of models that predict these moves given that the player held a particular holding. This means that we could use our newly obtained models to estimate a probability distribution over all holdings for every player in the unlabeled part of the data.

The assumption is that these inferred probability distributions will be better than the initial distributions that we had guessed, as at least they will be based on the actual actions that the player performed and will not be identical for every player in every game. These improved estimates of what holding each player held in the unlabeled part of the data may then be used to train improved models that predict actions given holdings. These models may then be used to obtain further improved estimates of the probability distributions over holdings. This process may be repeated until there is no further improvement.

The aforementioned process to iteratively improve the estimates of the probabil-

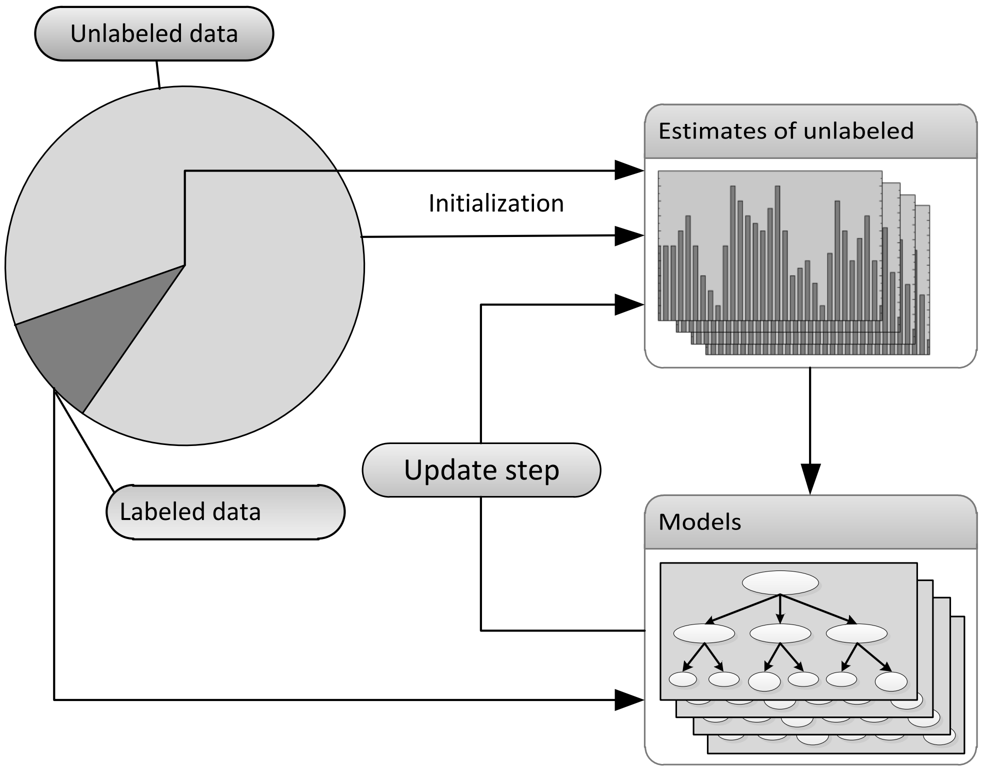


Figure 5.3: Overview of the proposed algorithm to infer probability distributions over all holdings for players' whose holdings have not been revealed. The idea is to guess initial probability distributions for each player based on the labeled part of the data and the fact that we know that each holding has an identical probability of being drawn. We can then use this guess in combination with the labeled data to train models that predict actions given holdings. These models may then be used to improve our estimate of the probability distributions over holdings for the unlabeled data. This process may then be repeated until convergence.

ity distributions for the unlabeled data and the models that predict actions given holdings is summarized in figure 5.3. Note that it is effectively an expectation maximization (cf. section 2.3.3) process, in which an expectation (estimating probability distributions over holdings for the unlabeled data) and a maximization (learning new models that predict actions given holdings) step are alternated until convergence.

5.5.1 Step by step

Let us now describe the algorithm step by step, in some more detail:

1. **Initialization**: infer the distribution of holdings for the unlabeled data using the fact that all holdings have an equal chance of being dealt and that we know

the distribution for the labeled data. Assign this distribution to every player in every game of the unlabeled data.

2. **Models training**: train a model for every bucket for every street. The data set for the model for bucket b is composed of both labeled and unlabeled data. The labeled data has a weight of 1 in the training set if the player had a holding belonging to bucket b (and 0 if the player had a different holding). The weights for the unlabeled data are taken from the associated probability distribution over holdings for each player: the more likely it is that a player has a holding in bucket b, the greater the weight of his actions in the training set.

3. **Probability distribution updates**: update the probability distributions over holdings for every player in every game of the unlabeled data. In equation 5.2, we found that we can obtain a probability estimate for every holding using the following formula:

$$P\left(H|a_n,\ldots,a_0,\vec{f_n},\ldots,\vec{f_0}\right) = P\left(H\right) \prod_{i=0,\ldots,n} P\left(a_i|\vec{f_i},H\right)$$

   Where $P(H)$ is the prior for the holding (which we know to be uniform), and $P(a_i|\vec{f_i},H)$ is the probability that the player takes action a_i given a feature vector f_i and holding H. Since the models that we learn predict models for a bucket and not for a holding, we need to translate between buckets and holdings (cf. section 5.4.5), and the equation becomes:

$$P\left(H=h_i|a_n,\ldots,a_0,\vec{f_n},\ldots,\vec{f_0}\right) = P\left(H=h_i\right) \prod_{j=0,\ldots,n} \sum_{k} m_{i,k} P\left(a_j|\vec{f_j},B_k\right)$$

   Where $m_{i,k}$ is the membership probability that holding h_i belongs in bucket k and $P(a_j|\vec{f_i},B_k)$ is the probability that the player takes action a_i given a feature vector f_i and bucket B_k. $P(a_j|\vec{f_i},B_k)$ may be obtained using the models that were learned in step 2.

4. **Repeat step 2 and step 3 until convergence.**

5.6 Improved player modeling

The primary goal in this chapter was to find methods to simulate showdowns by predicting players' holdings. In the process, we have expanded the models that predict actions given the state of the game to additionally take the player's holding as input. This side effect is rather interesting, since a player's holding will usually be the most influential factor in deciding his next move. While a player's decision is based on a wide range of different factors, the fact whether he has a weak or strong hand will be the decisive factor.

Using the methods developed in this chapter, we can now update a probability distribution over all holdings for every player after every action. We can then use these distributions and the models that predict actions given a game state and a holding to predict the player's next move. Since this method includes knowledge about the holding of a player, we expect this method to yield more accurate action predictions than the predictions made by the models described in chapter 4.

5.6.1 Self modeling

The new action prediction models are also beneficial for self modeling (modeling the computer player's own behavior). Self modeling is used during the simulation phase of Monte Carlo Tree Search, in which the computer player's actions are simulated using some heuristic that samples the computer player's future moves. Since the computer player always knows his actual cards, he can always use the models developed in this chapter that take the player's holding into account. We expect that the new models developed in this chapter will therefore result in more accurate self modeling.

5.7 Evaluation

In this section, we will evaluate the performance of the algorithm discussed in this chapter. We will describe the experiments that we conducted and present the results of these experiments.

5.7.1 Experiments

Evaluating the models that predict actions for a given bucket is a challenging problem. Ideally, we would like to measure how well the models predict actions for a test set in which we know the hidden cards of every player. Sadly, we do not have such a test set: the part of the data for which we know players' hole cards contains no fold moves and is biased towards situations where the players had good cards. We therefore have no data that we can use to directly evaluate the accuracy of the models that predict actions given holdings.

Evaluating hole cards prediction accuracy

Therefore, we evaluated the performance of the models obtained with the methods discussed in this chapter indirectly. We first identified clusters in each data set using the K-models clustering algorithm that we have covered in chapter 4. We subsequently ran the algorithm that learns models that predict moves given holdings for each of these clusters[1]. The input data for the algorithm consisted of a test test

[1]Note that it is unnecessary to do repeated runs of the algorithm, since the algorithm is completely deterministic and the results would always be exactly the same for particular inputs. Whether it is a good thing that the algorithm is deterministic or not will be covered in chapter 7.

of 95% of the games played by players in the cluster. We used the remaining 5% of the data to evaluate the performance of the resulting models.

Since the goal of this chapter was to be able to predict a player's hole cards at showdown, we will evaluate how well the models predict the hole cards of players in the test set. We can do this by using the models obtained with the algorithm and the method from section 5.2.1 to calculate a probability distribution over all 1326 hole card combinations for all players whose cards were revealed in the test set. We can then evaluate the resulting probability distribution by considering the probability that we obtained for the holding that the player in the test set actually had. We can then assess how well the resulting models perform at predicting cards (and therefore indirectly at predicting actions) by considering the average probability that the procedure outputs for the actual hole cards that a player had in the test set.

Tractability

The algorithm discussed in this chapter introduces some tractability problems. On the one hand we want to use a very large database with Poker games, since more data will result in more accurate models. Furthermore, since only at most 10% of the data is labeled and can be used to "steer" the process in the right direction, we need a large amount of data to get sensible results. On the other hand, large input databases introduce tractability problems for the proposed algorithm.

This is caused by the fact that the algorithm has to maintain a probability distribution over 1326 hole card combinations for every player whose cards were not revealed in the train set. If we use a 32 bits floating point number to represent the probability of each holding, the memory requirement for one such hole cards distribution is $1326 \times 4 = 5304$ bytes. Since we needed to maintain several hundreds of thousands of these distributions, the memory requirements of the algorithm were too large to fit in the average computer's internal memory.

We resolved this problem by storing these distributions on disk, which significantly increased the amount of time that was required for a single run of the algorithm to approximately 4 days on average. Because the time available for this thesis was limited, we were only able to run the algorithm on 10 clusters for the data set of 2 player games. For the 6 player and 10 player games, we ran the algorithm for respectively 4 and 3 randomly selected clusters.

5.7.2 Results

Performance at estimating players' cards

Figure 5.4 presents the results of the experiment described section 5.7.1. The three graphs plot the average value of the (average) performance at predicting hole cards of different clusters for games with 2, 6 and 9 players against iterations of the proposed algorithm. For example, the graph for 6 player games is based on 4 runs of the algorithm for 4 different clusters and plots the average performance achieved by these 4 clusters for consecutive iterations.

We measured performance by measuring the average probability that the resulting models for each cluster assigned to the hole cards a player actually held in a test set. For example, if we have an example game in the test set where a player has A♣A♢ and the models predict the player to have A♣A♢ with 1% probability, the performance for that example is 0.01. The gray area around the "test set" line shows the minimum and maximum scores achieved by any of the clusters' models. We provide these values instead of confidence intervals or standard deviations because the measured scores for different clusters are not normally distributed: the models for different clusters of players represent different playing styles that differ in how predictable they are. As a result, the scores of the models associated with each of these clusters are not normally distributed.

Taking a look at the resulting models

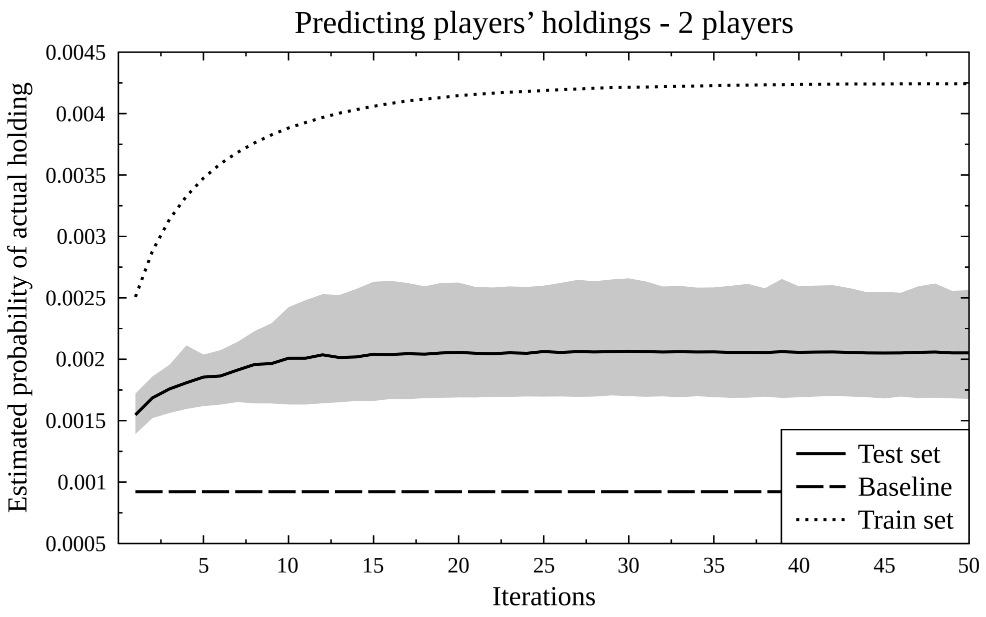
Upon manual inspection of the resulting models obtained after convergence of the proposed algorithm, we noticed something concerning. The decision trees that predict moves for the strongest of holdings predict that a player will fold with a fairly large probability. For example, the models predict that a player will fold A♠A♣ before the flop or 7♣8♣ on a flop of 4♣5♣6♣. In reality, players will never do this as it makes no sense.

Figure 5.5 illustrates this problem. The figure shows plots of the call, fold and raise probabilities in the decision trees for 20 flop buckets for 2 and 6 player games (note how the lines sum up to 1 for every bucket). The values plotted are averages over the models for all clusters (10 clusters for 2 player games, 4 clusters for 6 player games and 3 clusters for 10 player games). Bucket 1 is a large bucket that contains a lot of weak holdings, whereas bucket 20 is a very small bucket that contains only the few strongest holdings. A player would never fold a holding in bucket 20, yet the models still predict folds with a fairly large probability.

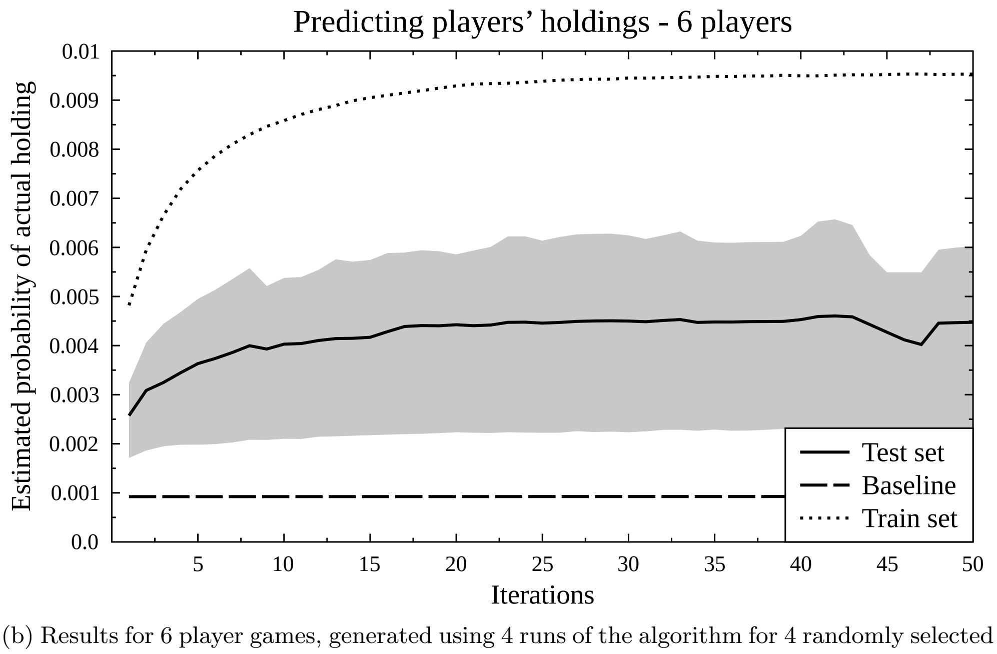
The graps also include a baseline approximation of what the fold line should instead look like (the dotted line labeled 'expected fold'). This line was generated by assuming that the probability that a player will fold a holding is a linear function of its strength. That is, if we order all holdings according to strength so that the weakest holding is at position $p = 1325$ and the strongest is at $p = 0$, the probability that a player will fold the holding is $p/1325$. The plotted line for this baseline is not linear because the number of holdings differs for buckets (cf. section 5.4.3).

This effect can be observed in models for all streets and in the models for 2, 6 and 9 player games and is indicative of a fundamental problem. We believe that the problem is one of *local maxima*. It is clear that the models are learning sensible things (as the models are significantly better than the baseline at predicting players' cards), but they could do better. While the algorithms converged, they apparently converged to a state that is locally maximal and does not represent the real world very well.

Further inspection of the generated models showed that the models make reasonable predictions, except for the fact that they overestimate fold probabilities for

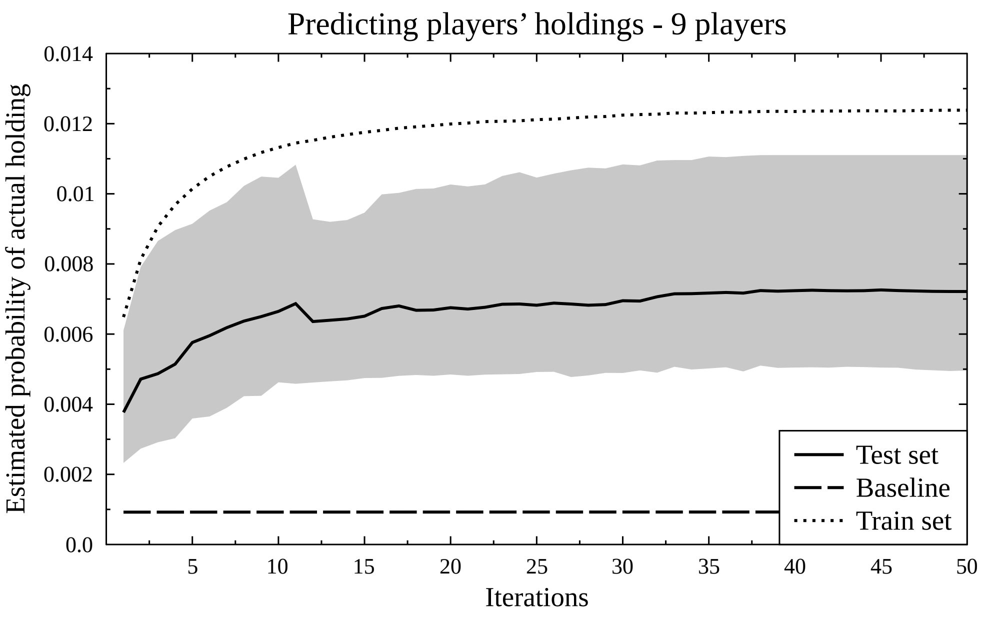


(a) Results for 2 player games, generated using 10 runs of the algorithm for 10 clusters.



(b) Results for 6 player games, generated using 4 runs of the algorithm for 4 randomly selected clusters out of 10.

Figure 5.4



(c) Results for 9 player games, generated using 3 runs of the algorithm for 3 randomly selected clusters out of 10.

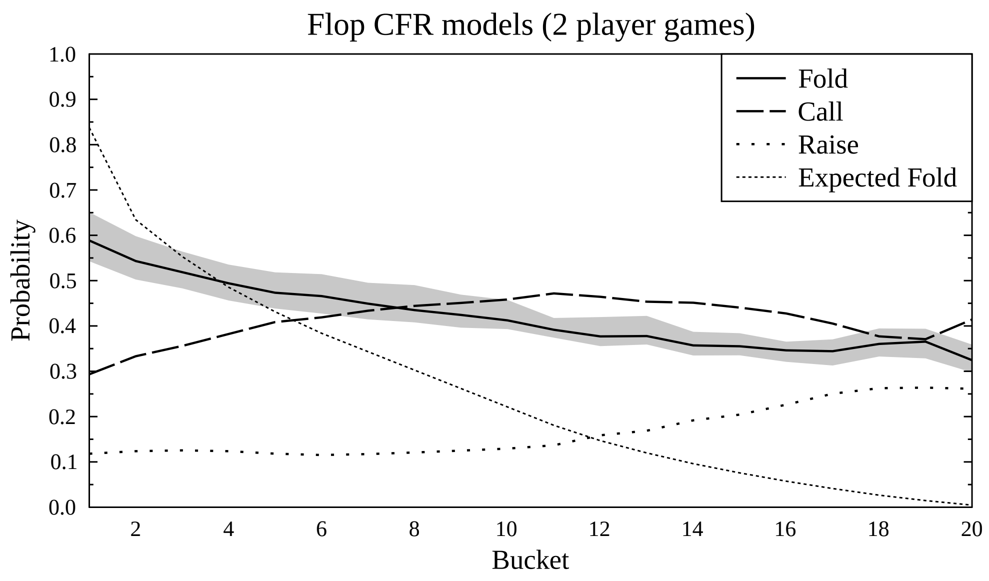
Figure 5.4: Plots of the performance of the models at predicting players' hole cards against iterations of the inference algorithm for 2 (a), 6 (b) and 9 (c) player games. The gray area around the test set line shows the minimum and maximum accuracy measured for clusters at that iteration.

strong holdings and underestimate them for weak cards: the inference process appears to fail to perfectly attribute fold moves to weak cards instead of strong cards. The fact that the models achieve impressive performance at predicting hole cards for showdowns confirms this idea: these are based solely on call and raise moves, and apparently the models do fairly accurately estimate the relative probabilities of these moves.

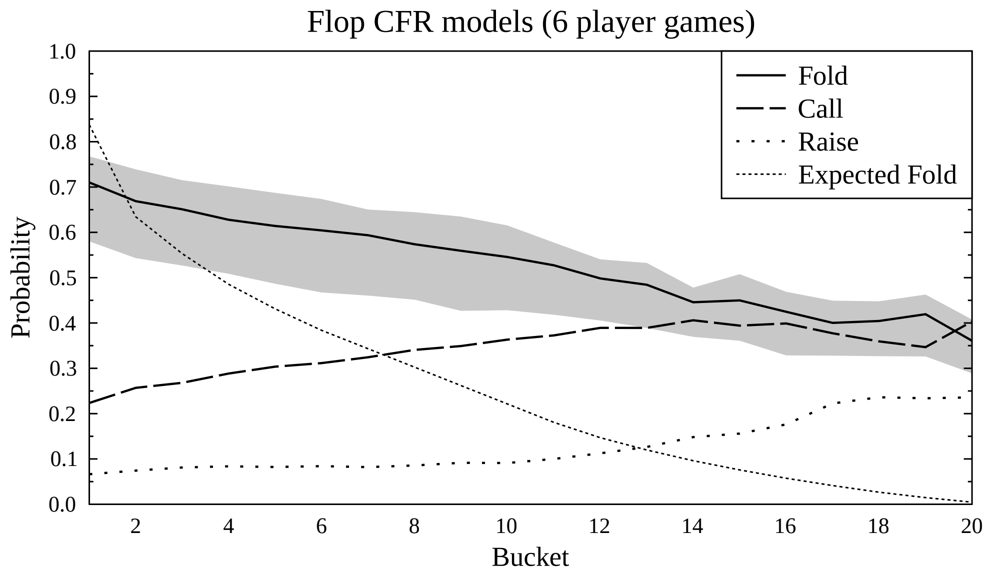
5.8 Conclusion

In this chapter, we have considered how we can develop models that predict the hidden hole cards of players. Learning to do so is complicated by the fact that in over 90% of the games, a player's cards are not revealed. This effectively makes the data partially labeled and partially unlabeled, which makes for an interesting problem. We introduce an expectation-maximization based approach that iteratively improves beliefs over the unlabeled data, starting from a very basic initialization.

While the resulting models achieve impressive performance at predicting players' hole cards (with average scores of over 7 times the baseline for 9 player games), they output unreasonable predictions in some cases. For example, the models predict that



(a)



(b)

Figure 5.5: Plots of the call, fold and raise probabilities in the decision trees for 20 flop buckets for 2 player (a) and 6 player (b) games. The values plotted are averages over the models for all clusters. The gray area illustrates the minimum and maximum values of the fold probabilities of different clusters. Bucket 1 is a large bucket containing many very weak hole cards. Bucket 20 is a small bucket that contains only best few hole cards of a flop: a player would never fold these cards on the flop. Still, the models predict a certain percentage of folds. The lines labeled 'expected fold' show an approximation of what we would expect the fold line to look like.

a player who has the very best possible hole cards on the flop will fold his cards to a bet with a non-zero probability. In reality, no Poker player would ever fold these cards, as it makes no sense to do so. This behavior of the models is probably caused by convergence to local maxima: while the models did learn, they fail to perfectly attribute fold moves to the correct holdings.

The proposed algorithm makes no Poker-specific assumptions, aside from the fact that cards are dealt with uniform probability. This property allows the ideas developed here to be extended to other games of imperfect information or similar machine learning problems where data is only partially labeled. While this is an interesting property, it may have been better to use more Poker-specific knowledge in the initialization step of the algorithm: currently, we assign an extremely simple educated guess to every single player in the data set whose cards were not revealed (cf. section 5.5). This is very simplistic and makes the algorithm deterministic: the same input will always result in the exact same output.

Using more Poker-specific knowledge – such as that players never fold very strong holdings – during the initialization step along with some randomization may prevent the algorithm from converging to local maxima. Sadly, we were unable to confirm these ideas for this work because of the very long running times of the proposed algorithm and the limited time available.

Chapter 6

Building and evaluating a computer Poker player

In order to evaluate whether the different techniques presented in this thesis together would actually result in a good computer Poker player, we built a fully functional implementation. In this chapter, we will consider how a computer Poker player may be constructed using the previously described algorithms for finding clusters of players and models that predict moves given hole cards.

6.1 Putting the pieces together

In chapter 4 and chapter 5, we have considered how we can find clusters of similar players, models that predict actions for players in these clusters and how we can extend these models to predict actions for a given holding (so that we can predict the cards that players will have in a showdown). In this section, we will discuss how these individual pieces may be combined to build a fully functional computer Poker player.

6.1.1 General steps

1. **Composing a database**: first, a database should be composed with games that we want the resulting computer player to play. For example, if our target game is a 2 player (heads-up) game with blinds of \$0.25 and \$0.50, we should compose a database with such games. Should the amount of data for that specific game be insufficient, it is acceptable to add some similar games, such as 2 player games with blinds of \$0.50 and \$1.00, as the strategies used in both types of games should be quite similar.

2. **Finding clusters**: the next step is to identify clusters of players in the database using the methods described in chapter 4. The result of this step is an assignment of players in the database to clusters and a set of models that predict moves for players in each of the clusters.

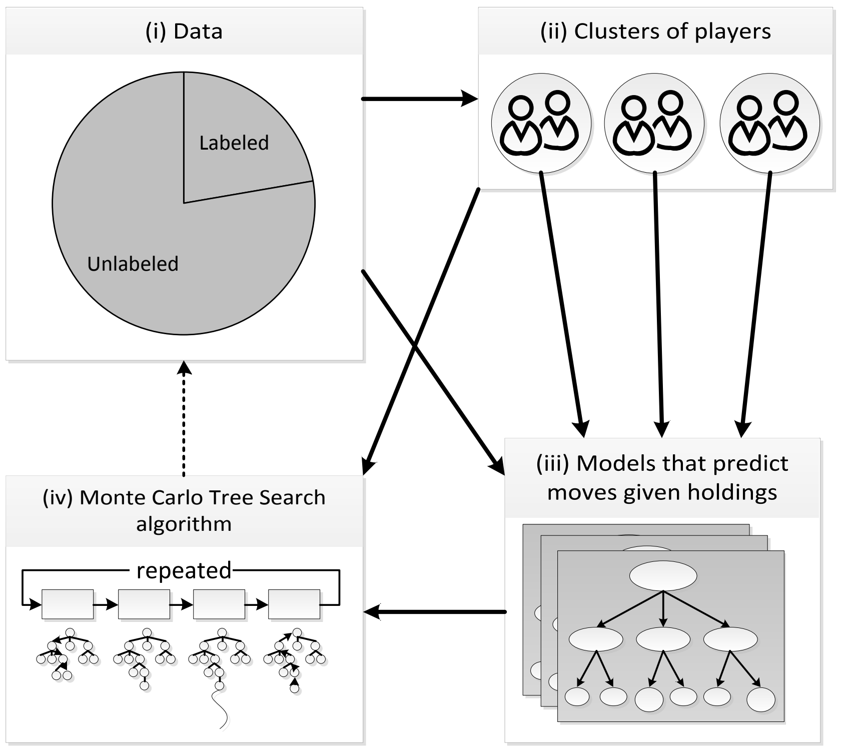

(i) Data
Labeled
Unlabeled
(ii) Clusters of players
(iii) Models that predict moves given holdings
(iv) Monte Carlo Tree Search algorithm
repeated

Figure 6.1: The general steps required to build a fully functional computer Poker playing using the methods described in this thesis. (I) A database of data from the target game is composed. (II) Each player in the database is assigned to one of K clusters. (III) A set of models that predict actions given holdings is trained for each of the clusters found in step (II). (IV) The models that we learned in step (III) are used in an online game tree search algorithm such as the Monte Carlo Tree Search (chapter 3) algorithm to determine which move has the highest value.

3. **Learning models that predict moves for given holdings**: once a number of clusters of players have been identified, the next step is to learn a set of models that predict moves given a certain bucket for each of these clusters. The methods described in chapter 5 may be used to obtain such models.

4. **Play using the models and cluster assignments**: once we have a set of models that predict moves for different clusters, we are ready to play. An implementation of Monte Carlo Tree Search (chapter 3) can use the set of models to estimate the expected values of possible moves by simulating continuations of the game using these models. We will discuss how MCTS may be implemented for the specific process described here in more detail later.

This process is summarized in figure 6.1. The steps in the figure correspond with the aforementioned steps: database composition, finding clusters, learning models and actual automatic play using the obtained models.

6.1.2 Using the models and clusters to play Poker

In chapter 3, we have discussed the Monte Carlo Tree Search algorithm. In this section, we will consider how the MCTS algorithm may be used in conjunction with the models obtained using the algorithms in chapter 4 and chapter 5.

Assigning players to clusters

The first thing that we should do, is finding out which cluster(s) of playing styles resembles the current opponent's strategy the most. To do so, we can calculate a probability distribution over all the clusters that we have defined. Since we have a set of models that predict moves and a set of models that predict moves *given buckets* for each cluster, we can use these models to calculate the likelihood that the models for a certain cluster generated the player's observed actions:

$$P(C = c_i|a_0, ..., a_n) = \alpha P(a_0, ..., a_n|C)P(C = c_i) \tag{6.1}$$

Where $a_0, ..., a_n$ are the player's moves in a series of games and $P(C = c_i)$ is the prior probability that the player is in cluster c_i. These priors may be obtained by counting how many players were assigned to each cluster in the K-models clustering algorithm. Note that a new opponent that did not make any moves yet is assigned to clusters based on these priors. The membership probability distribution calculated here should be updated regularly. We chose to update the distribution after every game.

A thing to consider here is that a player might change his playing style: he could play very conservative and passive for his first few games and later start to play really aggressive. We would like to recognise such changes in playing style as soon as possible and shift our beliefs accordingly. One way to do this, is to only consider a player's last n moves instead of all his moves.

Implementing MCTS

Whereas implementing the MCTS algorithm is fairly straightforward, we are faced with a number of design decisions. We have two sets of models that predict moves: one whose models are independent of the player's hole cards and one whose models are dependent of hole cards. We could use either one to predict players' moves in the MCTS algorithm. If we chose to use the former however, we would still need to infer a probability distribution over holdings when we reached a showdown in MCTS simulations. We therefore decided to use the latter set of models and implement MCTS as follows.

The input to the MCTS algorithm as we have implemented it, are cluster membership probabilities for every player in the game, sets of holding dependent models for every cluster and probability distributions over hole cards for every player. These probability distributions over holdings may be obtained by starting with a uniform distribution over all 1326 possible holdings for every player at the start of the game and updating these distributions as players make moves (cf. chapter 5). Since different models for different clusters predict different move probabilities and thus result in different holding probabilities, we update a holding distribution for every cluster for every player.

One iteration of the MCTS algorithm as we have implemented it using the techniques developed in this thesis then consists of the following steps:

1. **Drawing clusters**: draw a cluster for each player from their associated cluster membership probability distributions. The player is assumed to play according to that cluster's strategy for this iteration.

2. **Drawing cards**: draw a holding for each player from their associated probability distribution over holdings for the cluster drawn for that player in step 1. Ensure that each card is only dealt once.

3. **Monte Carlo Tree Search**: now that every player has an associated cluster and holding, we can perform the normal Monte Carlo Tree Search steps (cf. chapter 3). Note that there is no hidden information left because we guessed cards for every player. Resolving showdowns therefore boils down to simply comparing the five card Poker hands of the players. This also allows us to use the models that predict actions for a given holding.

Action selection

The Monte Carlo Tree Search algorithm yields the expected values of available moves. We then need to decide which move to play. The logical thing to do seems to be to simply play the action with the highest expected value. While this is indeed a sensible choice, it can result in somewhat predictable play. In Poker, it may be beneficial to sacrifice a little expected value and to occasionally play a move with a slightly smaller value in order to be less predictable.

One way to implement such a scheme, is *softmax* action selection (Sutton and Barto, 1998). The softmax function maps the expected values of moves to action selection probabilities. The probability of playing move a is:

$$P(a) = \frac{e^{V(a)/\tau}}{\sum_{i=1}^{n} e^{V(i)/\tau}} \tag{6.2}$$

Where $V(a)$ is the expected value of move a and τ is the *temperature*: this parameter determines whether the action with the greatest expected value should be preferred or whether the action should be chosen randomly. For high temperature ($\tau \to \infty$), all moves have the same probability and the choice is random. For low temperature ($\tau \to 0$), the probability of the action with the highest expected value goes to 1.

6.2 Evaluation

In this section, we will consider how we evaluated the performance of the implementation described in the beginning of this chapter. We were primarily interested in how the system performs against human opponents.

6.2.1 Online experiment

We have conducted an experiment in which human participants could play the computer player through a web interface. Potential participants could sign up on a website where they had to fill in a short form which amongst others asked them how much Poker experience they have. This allows us to gain insight in how the system performs against both novices and experienced players.

The web interface was implemented in Adobe Flash and connects to a game server that runs the Poker games. The actual Monte Carlo Tree Search program that makes Poker decisions was implemented as a C++ server program that other programs can connect in order to obtain a suggested action. A very simple Python program connects to both the game server and the decision server and plays Poker games against the human participants. Whenever the computer players controlled by this program are to act, it queries the decision server for the best move and returns this move to the game server. An overview of this experimental setup is provided in figure 6.2.

All games played during the experiment were 2 player games where both players start with 100 big blinds. To minimize the difference between games, the amount of money of both players was reset to 100 big blinds after every game. Furthermore, the Monte Carlo Tree Search algorithm was set to run for a fixed number of iterations, so that each decision was based on the same number of iterations. Since we express the expected values of moves in hundreds of big blinds, their minimum value is -20000 and their maximum value is 20000. We chose to use softmax action selection with a temperature of 50, which was emperically found to nicely balance exploitation and deception for this range of possible action values.

While the methods developed in this thesis support games with more than 2 players, we chose to evaluate games with 2 players only. The reason for this is that evaluating games with more than 2 players is complicated. We would have to decide whether to put one or more computer players at a table, we would need multiple human opponents before a game can start and we would have difficulty obtaining unbiased measurements of performance of the computer player.

Level of play of participants

One difficulty with a large-scale online evaluation is that the participants' money is not really at stake. Therefore, they might not play to the best of their abilities at all times (which might be putting it mildly). We tried to prevent this by explaining to the participants that they are not helping out when they are not playing seriously.

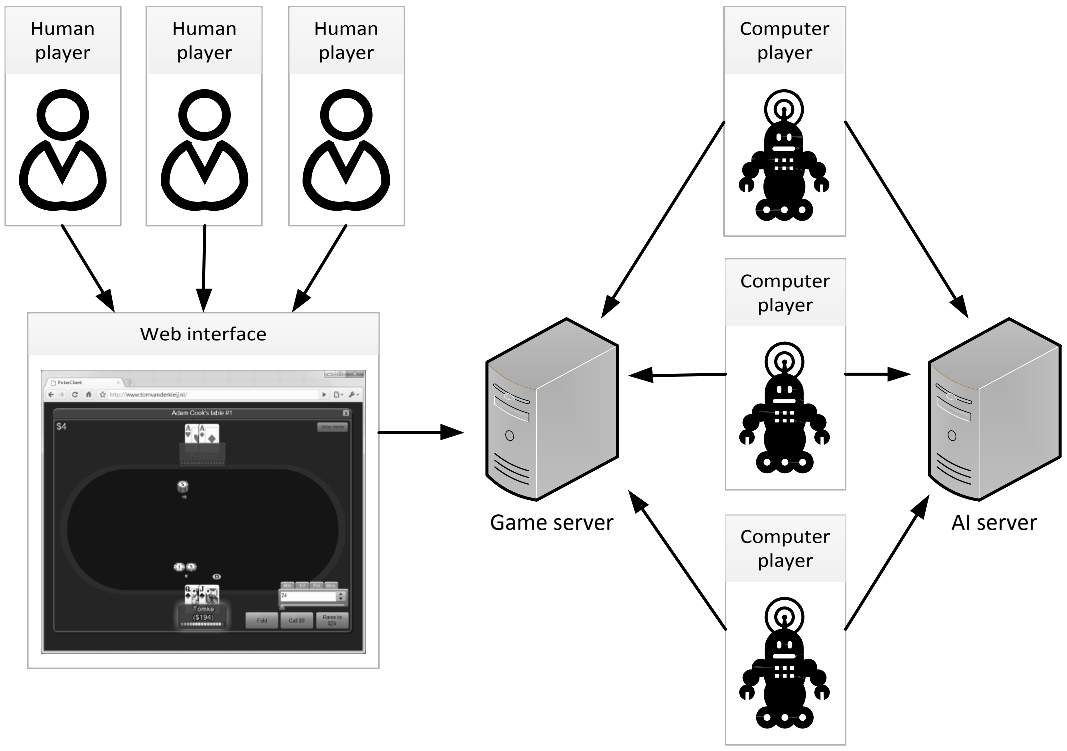


Figure 6.2

Furthermore, we monitored the experiments and excluded players that were clearly not taking the experiment seriously (by going all-in pre-flop every hand for example).

Post-processing of models

In section 5.7.2, we have discussed some problems with the models generated by the algorithm introduced in chapter 5. These models overestimate fold probabilities for strong holdings and underestimate them for weak holdings.

For the online experiment, we have post-processed the generated models by adjusting the probabilities of leaf nodes. For every leaf node, we set the probability of the fold move to be a weighted average of the original fold probability and a probability estimated by a baseline that approximates more reasonable fold probabilities (cf. section 5.7.2). The ratio of the weights for both probability estimates is given by an exponential function of the bucket. For strong buckets, the weight of the baseline is much larger than the weight for the original fold probability. For weaker buckets, the weight of the baseline is only slightly larger. Once the fold probability has been adjusted, the probabilities for other moves are normalized so that the sum of probabilities for all moves is 1.

6.2.2 Baseline agent

Many publications about computer Poker agent implementations include results against one or more baseline agents that use very simple rules such as "always check when possible, otherwise call" to play. The system described in this chapter is specifically designed to play against human opposition and will not perform extremely well against artificial strategies, as none of the clustered playing styles the system learned to recognize will resemble such a strategy. For completeness sake however, we will include the system's performance against a baseline agent that always checks or calls.

6.2.3 Questionnaire

We have asked the human players that participated in the online experiment and played 100 games or more to fill out an online questionnaire about their experiences with the computer player. The questions that we asked are primarily about the playing experience the player had against the computer players. We are interested in how human opponents perceived the computer player's play: are there any major strategic mistakes? How humanlike are the computer players playing? The complete questionnaire is provided in appendix D.

6.2.4 Reducing variance

The game of Texas Hold'em Poker – especially the no-limit heads-up variant – is a game with a lot of variance. You could play great Poker and still lose, or you might play terribly and win nevertheless. This property of the game is very inconvenient when you are trying to measure how well a certain player is playing. If we base our judgement solely on profit made, we will need a very large sample size before we can draw any meaningful conclusions.

Luckily, there are some techniques that can help us obtain more objective measurements of the quality of a Poker player. Billings and Kan (2006) describe DIVAT, a "tool for the direct assessment of Poker decisions". This tool is meant to evaluate the quality of decisions in hindsight given full information (that is, it can only be used to analyze games where you know everyone's hidden cards) by comparing the decisions to a pseudo-optimal baseline strategy. Unfortunately, the DIVAT method is only suitable for limit Texas Hold'em games and cannot be used to analyze no-limit games.

One of the largest sources of noise in profit measurements in no-limit Texas Hold'em Poker – especially in heads-up games – are all-ins situations before the river. A player is all-in when he has put all his money in the pot and has nothing left to bet. He then no longer has to call any bets and is guaranteed to go to showdown with whomever else is left in the pot[1]. For example, if one player has Q♠Q♡ and

[1]It is a common misconception (probably caused by spectacular Hollywood movies) that whenever someone raises an enormous amount, his opponent has to somehow match that bet in order to stay in the pot. In movies, one frequently sees people bringing the key to their car or the deed to their house to the table in order to call a very large bet. In reality, the opponent only has to call

another has A♣K♣, it is not unlikely that they will both raise and re-raise before the flop until one or both of them has no money left. They are then all-in and five community cards will be dealt to determine the winner of the pot.

These events, particularly in situations such as the example where both players have approximately equal chances (Q♠Q♡ versus A♣K♣ is approximately fifty fifty) are greatly influenced by luck and have a large impact on a player's profits, since the pot will typically be very large (as one or more players put in all their money). In live Poker games, the players came up with a solution that somewhat reduces the variance associated with these situations called "dealing it twice": instead of dealing the remaining community cards only once, they will deal them twice (typically without putting any cards back in the deck). If either player wins on both boards, he gets the entire pot. If both players win once, the pot is split between them.

We can take this idea from live Poker to the extreme in order to obtain a less luck-influenced measure of profit. For any all-in situation before the river, we can disregard the actual outcome of the situation and instead pretend that the players won a share of the pot based on their equity instead. For example, if two players are all-in with Q♠Q♡ and A♣K♣ before the flop and the pot is \$200, each player receives approximately \$100. Similarly, if player A has 6♡7♡ and player B has 4♠4♡ and they go all-in for a total pot of \$100 on a flop of 3♣4♡5♡, player A receives \$70 and player B receives \$30, because their respective odds to win are 70% versus 30%. In all the results mentioned in the following section, we have applied this method to somewhat reduce the influence of luck on the results.

6.3 Results

6.3.1 Online experiment

Approximately 10,000 games were played by 51 participants in the online experiment. Of these 51 participants, 35 players were "novices" and the remaining 16 players were "experienced players"[2]. The 35 inexperienced players played approximately 2200 games, whereas the 16 experienced players played about 7700 games. It is quite remarkable that the experienced players played the great majority of the games, as there were twice as many inexperienced players. This can probably be explained by the fact that experienced players are used to playing online and playing long sessions.

Figure 6.3 shows the profit of the computer player over all games in small blinds per game. The x-axis shows the game number for which the small blinds per game value is shown at the y-axis. Initially, the values fluctuate quite heavily as the sample size over which the small blinds per game value is being calculated is small. As the number of games increases, the values converge. While the sample size of 10,000 is reasonably large, it is not large enough to draw any strong conclusions. Looking at

whatever amount of money he has left on the table.

[2]This is obviously a very coarse distinction to make. The distinction is made based on the participants own judgement of their Poker qualities.

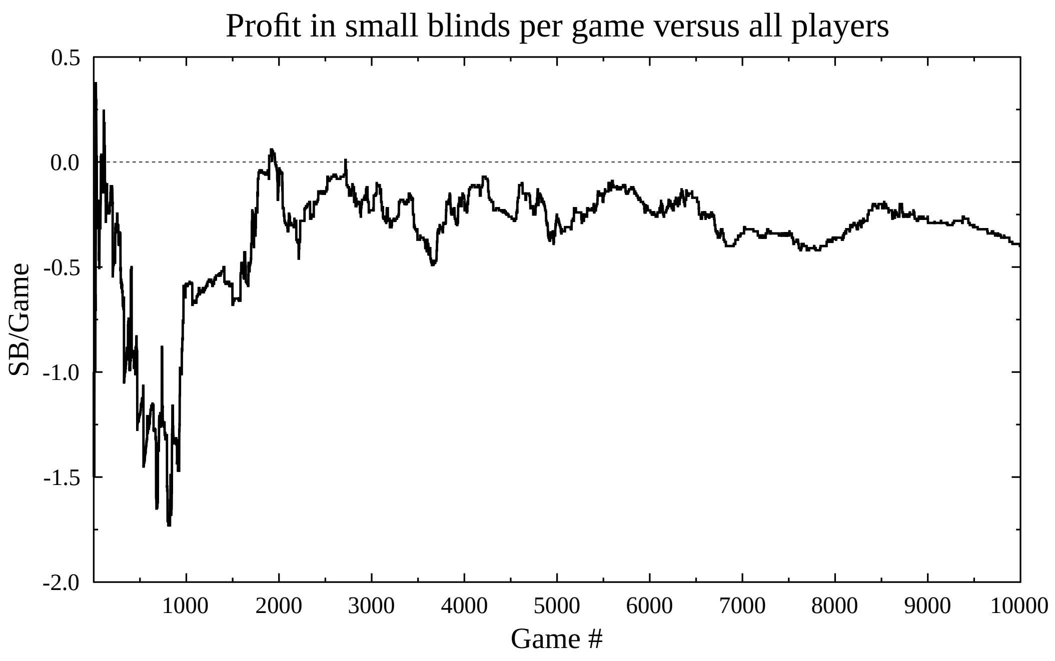


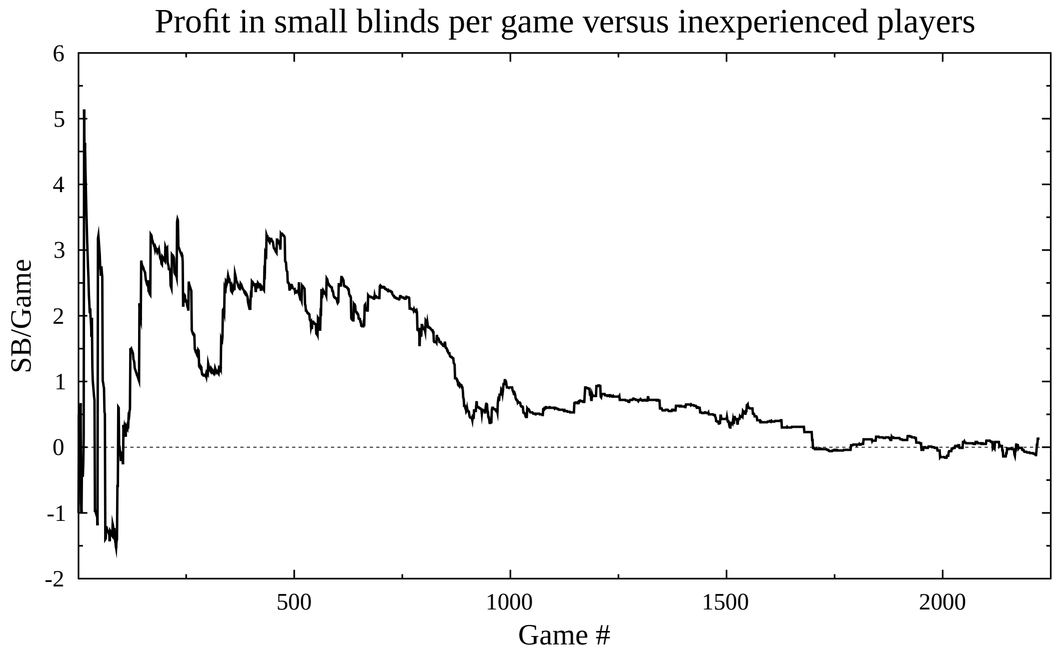
Figure 6.3: The win rate of the computer player in small blinds per game (SB/game) over all games versus human opponents in the online experiment. The x-axis shows the game number for which the small blinds per game value is shown at the y-axis.

the graph, it is reasonable to expect the computer player's average win rate to be somewhere between 0 and -1 however.

Figure 6.4 provides similar graphs for the games against novices and experienced players respectively. The computer Poker player performed slightly better against the inexperienced players, with an approximated win rate of 0 small blinds per game: it is playing break-even. Its losses are caused by playing the experienced players, against whom its winrate is approximately -0.7 small blinds per game.

Observations

One thing that became clear during the experiment, is that the computer player has a rather predictable strategy. This predictability is probably the main cause of its negative win rate against experienced players. Figure 6.5 illustrates this by plotting the win rate that the computer player would achieve if we only include the first n games of each player in the results against n. For example, this graph shows that when we only include the first 400 games for each player, the computer player achieved a win rate of approximately 0.18 small blinds per game. The graph illustrates that playing many games against a single opponent has a detrimental effect on the computer player's winrate: when human opponents figure out its strategy, it starts losing money at a rapid pace.



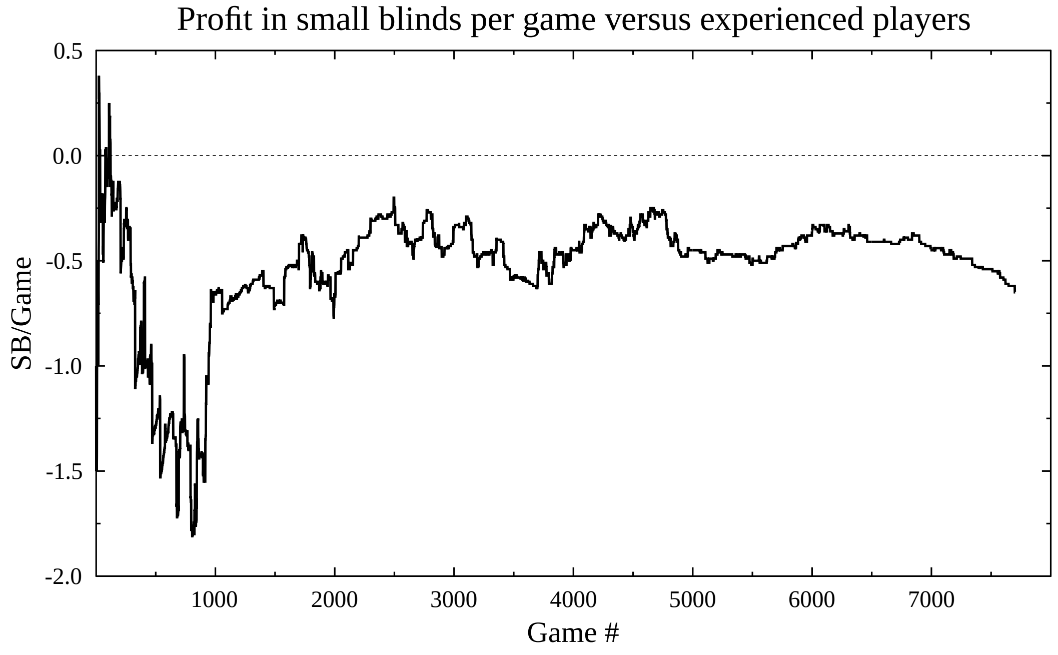


Figure 6.4: The win rate of the computer player in small blinds per game for the online experiment against inexperienced and experienced players respectively. The x-axis shows the game number for which the small blinds per game value is shown at the y-axis. Against inexperienced players, the computer player played approximately break-even. Against the experienced players, the computer player achieved a win rate of approximately -0.7 small blinds per game.

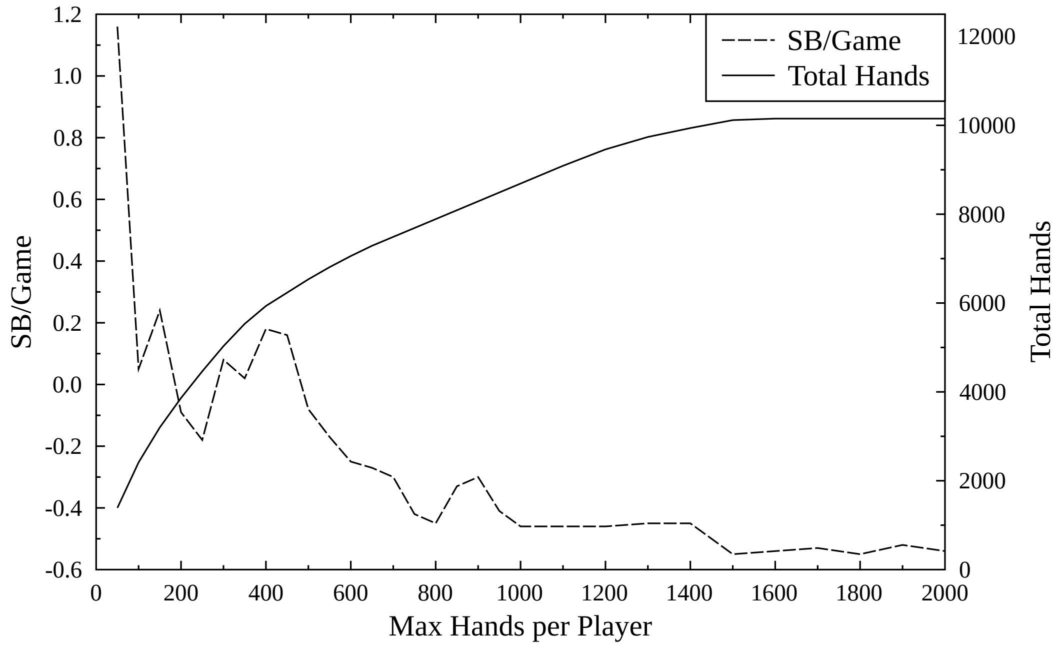


Figure 6.5: The win rate of the computer player in small blinds per game if we only include the first n games against each player. The x-axis shows the number of games against each opponent that is included in the calculations. The left y-axis shows the win rate of the computer player in small blinds per game. The right y-axis shows the total number of games over which the win rate statistic is calculated. It is clear that playing more games against a particular opponent has a detrimental effect on the computer player's win rate.

In section 4.5.2, we found that some abstracted bet and raise moves that we chose to use are very uncommon in some game types. For example, a raise of $\frac{1}{4}$ pot is essentially never used in 2 player games, whereas 14.17% of the raises in 9 player games are $\frac{1}{4}$ pot. We noticed that some players discovered that using bet or raise sizes that are very uncommon in the training data could throw off the computer player. Table 6.1 presents how often participants in the experiment used each abstracted bet or raise move.

The observed bet moves are different than we would expect based on the training data, $\chi^2(6, N = 1326) = 2941.30, p < 0.001$. The observed raise moves are also different than we would expect based on the training data, $\chi^2(6, N = 5845) = 177114.33, p < 0.001$. This is indicative of the fact that participants were using uncommon moves that the computer player does not respond well to.

Table 6.1: The expected and observed abstracted bet and raise moves used by participants in the online experiment. The expected counts are based on the frequencies in the training data (appendix C). We only include the participants' moves, not the computer player's moves.

Bet	**Expected**	**Observed**
Minbet	76.51	271
1/4 pot	75.05	31
1/2 pot	216.93	338
3/4 pot	902.21	312
Pot	41.63	301
3/2 pot	6.23	15
2+ pot	7.56	58

Raise	**Expected**	**Observed**
Minraise	572.81	1141
1/4 pot	0.00	50
1/2 pot	1.17	312
3/4 pot	12.86	1094
Pot	4968.25	2538
3/2 pot	247.83	398
2+ pot	42.08	312

Significance

As we have mentioned in section 6.2.4, Texas Hold'em Poker is a game with a rather large element of luck. To make matters worse, heads-up no-limit Texas Hold'em is probably the most luck-influenced Texas Hold'em variant because players tend to play extremely loose and aggressive. Unfortunately, there are no good academic publications that detail the statistic analysis of Poker games. We therefore do not know the amount of games that is required to obtain statistically significant results. However, the consensus amongst human Texas Hold'em Poker experts is that tens of thousands of games are required to determine if you are a winning player.

Billings and Kan (2006) provide some academic insight on the subject for *limit* Texas Hold'em:

> " *The* signal to noise ratio *is low – a player can play extremely well and still lose over an extended period of time, just by bad luck. The normal variance in the game is so high that many thousands of games need to be played before obtaining even a modest degree of confidence in the result.*"

It is clear that the results provided in this section are not significant and only provide an indication of the system's performance. This is a fundamental problem in

Poker research: while 10,000 games are a lot of games in absolute numbers, we will need an even larger experiment to draw statistically significant conclusions.

6.3.2 Questionnaire

The questionnaire form (appendix D) was filled out by 13 participants that played 100 games or more. Figure 6.6 shows box plots of the answers that the participants gave on questions about the bluffing frequency, valuebet size (the size of the bets that the computer player made when holding a strong hand) and bluffing size. For these questions, 1 means "too infrequent/small', 5 means "too frequent/large" and 3 means "good". Note that the answers indicate that the computer player did well on these elements, according to the participants.

Figure 6.7 shows box plots of how participants estimate the playing strength, predictability, anthropomorphism and aggressiveness of the computer Player on a scale from 1 to 5. The participants believe that the computer's playing strength is fairly weak. Furthermore, it is rather predictable, not very anthropomorphic and not very aggressive.

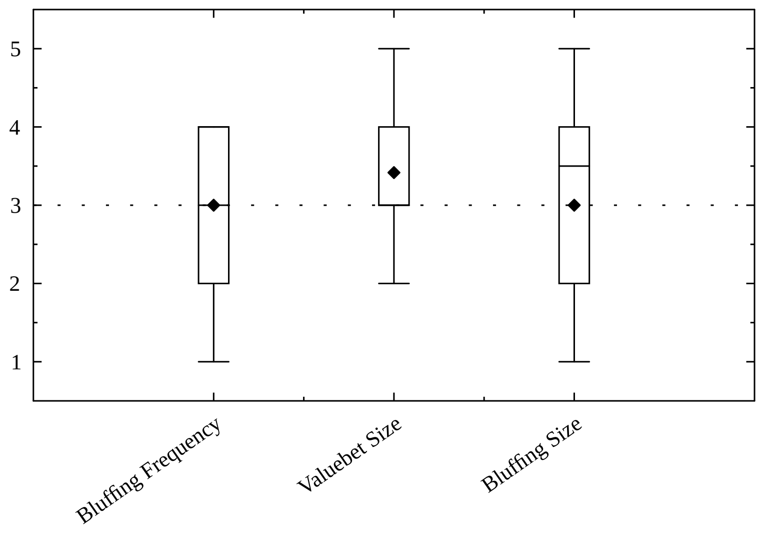


Figure 6.6: Box plots of the answers that the participants gave on questions about the bluffing frequency, valuebet size and bluffing size. An answer of 1 means "too infrequent/small', 5 means "too frequent/large" and 3 means "good".

6.3.3 Baseline agent

The computer player's results against the baseline that always check or calls is shown in figure 6.8. It is clear that the computer player's profit per game against this baseline agent is approximately 17.5 small blinds per game. Considering that the

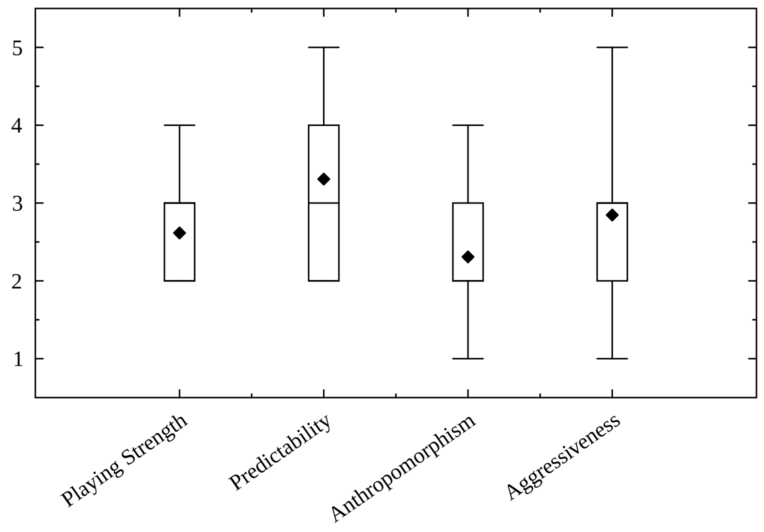


Figure 6.7: Box plots of how participants estimate the playing strength, predictability, anthropomorphism and aggressiveness of the computer Player on a scale from 1 to 5.

system was not designed to play against artificial strategies that in no way mimic a human player's playing style, this is an impressive win rate.

6.4 Conclusion

In this chapter, we have discussed the implementation and evaluation of a computer Poker player based on the methods developed in the previous chapters. The player was evaluated using an online experiment in which participants could play against the computer in one versus one games. Additionally, we have used a questionnaire and a match against a simple baseline agent for further evaluation.

The evaluation showed that the computer player holds its own in games against both inexperienced and experienced players. It plays solidly and is typically fairly hard to beat. This is illustrated by the results of the questionnaire, which indicate that the players bluffing frequency, bluffing size and valuebetting size are good.

In terms of profit, the computer player plays only break-even against the novices and loses slightly form the experienced players. This can be explained by the fact that the computer player is rather predictable. This observation is confirmed by the results of the questionnaire. During its initial hundreds of games against human opponents, the computer player does well and gives its human opponents a run for their money. After a few hundred hands however, the human players adapt to the computer player's strategy, but the computer player fails to adapt its game appropriately. As a result, it starts to lose money at a fairly rapid pace.

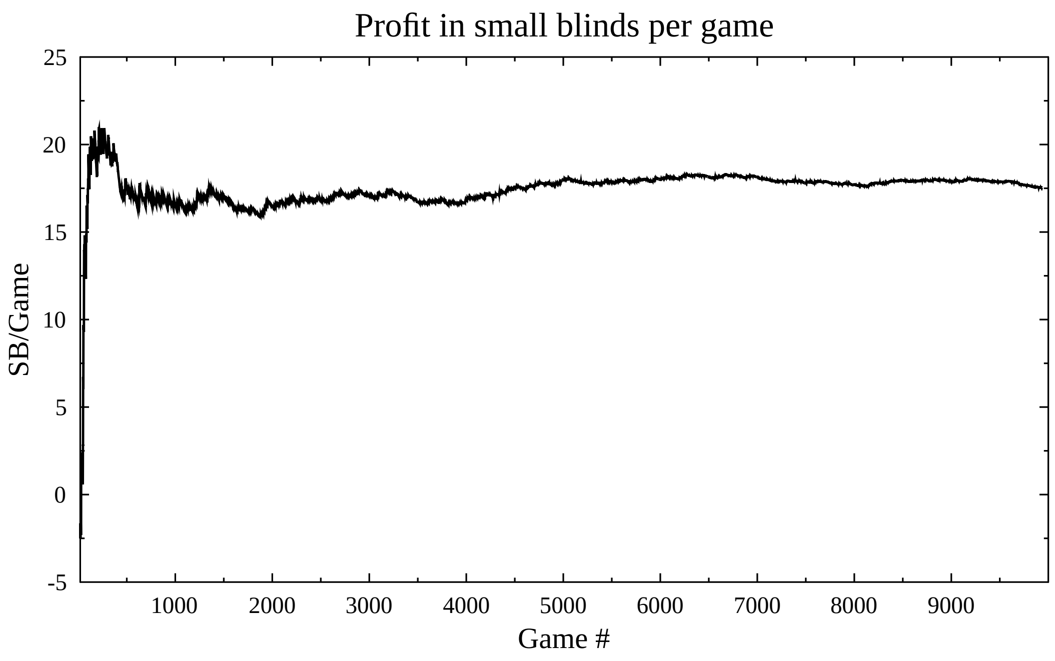


Figure 6.8: The results in small blinds per game of a match versus a baseline computer player that always checks when it can and otherwise calls. Looking at the graph, we can conclude that the computer player's win rate against this baseline agent is approximately 17.5 small blinds per game.

The computer player also contained some obvious weaknesses that some of its opponents managed to find and exploit successfully. These soft spots can be traced down to the fact that the computer player reasons about its opponents hole cards but does not do a lot of online learning. For example, the models that were obtained during offline inference indicate that a small raise of about $\frac{1}{2}$ pot typically equal a weak hand. Players soon found out that they can abuse this by making raises of that size with their very strong hands, throwing off the computer player. As a result, the computer player would think its opponent has a weak hand and try to bluff, while in fact the opponent held a very strong hand. Because the computer player's online learning is limited, it will hold on to its false beliefs despite a lot of evidence of their falsities.

Chapter 7

Discussion

7.1 Overview

In this thesis, we have considered algorithms and methods for the game of no-limit Texas Hold'em Poker. This is a complex and challenging game, that due to non-deterministism, imperfect information and an enormous game tree cannot be approached with traditional methods. We set out to study algorithms and methods towards an adaptive computer player that maximally capitalizes on its opponents' mistakes in games with 2 to 9 players.

7.1.1 K-models clustering

We introduced a clustering algorithm called *K-models clustering* that clusters players based on playing style. This algorithm does not use features that describe players and their strategies, but models that predict players' moves instead. The advantage of the K-models clustering algorithm in comparison to traditional clustering methods is that we do not have to come up with features that describe players or a distance function, that we obtain models that predict players' moves as a by-product of the clustering and that the assignment of players to clusters is optimized for model performance.

Our experiments showed that the clustering algorithm finds clusters that indeed represent different playing styles according to human Poker experts' standards. Furthermore, the accuracy of the models for individual clusters is greater than the accuracy of models trained on unclustered data. Since the only Poker-specific assumptions that we make are the features that describe the context in which players make moves, the clustering algorithm generalizes to other games and even beyond the scope of games.

7.1.2 Guessing hidden hole cards

Learning to reason about the hidden cards of a player is complicated by the fact that in over 90% of the games, a player's cards are not revealed. The labeled part of the data is extremely biased and is not suitable to be used on its own. We propose an expectation-maximization based inference algorithm that iteratively updates beliefs about the hidden hole cards of players whose cards were not revealed in the partially labeled data set. After convergence, we obtain models that predict players' moves given hole cards.

These models achieve impressive performance of over 7 times the baseline at predicting hole cards, which is a difficult $\binom{52}{2} = 1326$ class problem. Despite this, we found some problems with the models. For instance, the models output non-zero probabilities for fold moves when the player is holding extremely strong cards. In reality, players would never fold with such cards. We believe this problem is largely caused by the fact that we again make no Poker-specific assumptions in the algorithm (except from the fact that cards are dealt with uniform probability at the start of a game). Although this allows the generalization of the method to other games or completely different applications, it may be the cause of convergence to local optima for our specific case. We believe the problem can be overcome with the addition of some basic Poker-specific knowledge to guide the algorithm towards global optima.

7.1.3 Monte Carlo Tree Search

In the final chapter of this thesis, we have evaluated an implementation of a computer Poker player based on Monte Carlo Tree Search (Coulom, 2006; Gelly and Wang, 2006) and the algorithms introduced in this work. In one experiment, human Poker players played a total of over 10,000 heads-up games against the computer player. This experiment showed that the computer player plays approximately break-even against inexperienced players and loses slightly from experienced players. The computer player appeared to play a fairly solid strategy however, which was confirmed by its human opponents in a questionnaire: its bluffing frequency, bluffing size and valuebetting size were considered good by its opponents.

The fact that the computer player does not win despite playing solidly for most of the time, can mostly be attributed to predictability. It became clear that the computer player's strategy contains some weaknesses that its human opponents learn to exploit sooner or later. While the human players adapt their strategies accordingly, the computer player has only limited ability to do so and starts to lose money. Our results indicate that the computer player maintains a positive win rate during its initial hundreds of games against a single opponent. Once its opponents figure out its strategy however, the computer player starts to lose money at a fairly rapid pace.

7.2 Future work

7.2.1 Improving opponent modeling

In chapter 4 we have discussed group-specific opponent modeling and the K-models clustering algorithm. The basic idea underlying these methods is that any Poker player's strategy will at any time fairly closely resemble one of many prototypical strategies present in a training database. We proposed to find these prototypical strategies and learn models for them, after which opponent modeling during live play reduces to deciding which prototypical strategy most closely resembles a player's actual strategy.

While the underlying assumptions are probably true and the proposed clustering algorithm does indeed find meaningful clusters of playing styles, we observed some problems that can be attributed to poor opponent modeling in the experiments described in chapter 6. We will now propose some solutions to address this problem.

Mixing specific and group-specific opponent modeling

One interesting improvement would be to learn specific models for a single opponent during live play. These models will initially be inaccurate, but should slowly start to outperform the group-specific opponent models after a large number of games. Ideas from ensemble classification (Dietterich, 2000) could be used to combine predictions from both types of models. This way, a smooth transition from group-specific to specific opponent modeling should occur as more games are played.

This also introduces more sophisticated online learning than the current system is capable of, which might relief some of the predictability problems that we have discussed earlier. If the computer player can adapt more appropriately to its opponent's strategy, it might stop making similar mistakes over and over again.

Improving models that predict moves given hole cards

The inference algorithm that we have discussed in chapter 5 may be improved by using a smarter initialization. The current initialization method is extremely simple and assigns a single identical distribution to all players whose cards were not revealed in all games. This can be improved by using some very simple Poker-specific guidelines (such as that players never fold with extremely strong hole cards) to improve our initial guesses. Furthermore, some randomization could be applied in the process, so that that algorithm is no longer completely deterministic. This will allow repeated runs of the algorithm, after which the best resulting models can be selected.

7.2.2 Improving bet abstractions

We map every no-limit Texas Hold'em bet or raise move to an abstracted move based on the amount that the player bet or raised (cf. section 4.3.2). The particular mapping that we used is inspired on how human Poker experts discuss moves and maps bets and raises to 7 different fractions of the pot.

The largest abstracted bet or raise that we used was a bet of '2 times the pot or more'. This turned out to be exploitable: the computer player treats bets of 2 times the pot and enormous pre-flop allins of over 60 times the pot the same, and some players found strategies to cleverly exploit this. Furthermore, some abstracted bets are very uncommon in some types of games (section 6.3.1). For example, $\frac{3}{2}$ pot bets or raises are very uncommon in all games and $\frac{1}{2}$ pot raises are very uncommon in 2 player games, but fairly common in other games. When an abstracted move is very uncommon, the amount of training data for that move will be very limited. This in turn causes the inference algorithms to occasionally draw some strange conclusions when such an action is observed. Some human opponents found successful strategies based on uncommon betting sizes.

This situation can easily be improved on by either picking smarter abstracted moves that are specific to a type of game or using an algorithm that finds suitable abstracted bet and raise actions for a specific data set. The abstracted moves should be chosen in such a way so that all abstracted bets represent some some minimal proportion of the data. Furthermore, sufficiently large abstracted moves should be added, so that enormous all-ins of multitudes of the pot are treated appropriately.

7.2.3 Evaluating games with more players

In chapter 6, we have discussed the implementation and evaluation of a computer Poker player that uses the techniques described in this work. The specific Poker variant that we chose to evaluate the player against was 2 player no-limit Texas Hold'em, in which the computer player faces a single opponent. We chose this variant because it was convenient: evaluating against more players has some difficulties (how to fill the tables, how to measure the computer player's performance when facing multiple opponents that are computers and/or humans themselves, and so on).

Two-player no-limit Texas Hold'em Poker is one of the most difficult Poker variants however: players typically play extremely aggressive and unpredictable compared to games with more players. This is reflected by the fact that the results presented in section 4.5.2 and section 5.7.2 are consistently worse for 2 player games when compared to 6 player or 9 player games: the algorithms clearly perform worse on 2 player games.

It would therefore be very interesting to see how the computer Poker player performs in games with 6 or 9 players. We expect the computer player to do remarkably better than in games with 2 players, since these games are easier and the models that we use throughout this work are more accurate for these games.

7.3 Conclusion

We have discussed our work towards an adaptive, exploitative, artificial no-limit Texas Hold'em Poker player for games with 2 to 9 players. Our main contributions consist of the K-models clustering algorithm that clusters players based on playing style and an expectation-maximization based algorithm that iteratively improves beliefs of the hidden hole cards of players.

We have shown that these methods may be used to develop a fully functional computer player for the game that can hold its own against novices and experienced players. Despite the fact that the resulting computer player's strategy is deemed fairly solid by human experts, it contains some weaknesses that human players eventually spot and exploit successfully. Luckily, there is a lot of room for improvement, and we therefore consider this work a first step in a very promising direction.

Appendix A

Poker glossary

- **Aggressor**: the player that put in the last bet or raise on a street. Different streets can have different aggressors.
- **All-in**: a player in a Poker game is said to be all-in when he has put all his money in the pot and consequently cannot invest any more money. A player that is all-in is guaranteed to go to showdown. Note that a player that is all-in can only win the share of the pot that he participated in.
- **Blind**: a forced bet put in by a player before the cards are dealt. In Texas Hold'em, there is a small blind and a big blind. The small blind should be paid by the player after the dealer and the big blind by the player after the small blind. Note that in a 2 player game, the dealer is the small blind.
- **Board (cards)**: see *community cards.*
- **Button**: typically refers to the player that has the dealer button (a button that is used in live Poker games to indicate who the dealer is in that game): see dealer.
- **Community cards**: in Texas Hold'em, the community cards are the cards that any player can use to compose their five card Poker hand (along with their two hidden cards). Before the flop round, three community cards are dealt. Another card is dealt before the turn round. The final fifth community card is dealt before the river round.
- **Dealer**: the player that is in the dealer position. This is considered an advantageous position because the dealer gets to act last during post-flop rounds, and thus has information about what the other players did prior to his own decision.
- **Effective stack**: the effective stack of one player relative to another player is the minimum of both their stacks. If player A has \$100 behind and player B has \$50 at the table, the effective stack of A relative to B and vice versa is \$50.

- **Flop**: the second of four rounds in Texas Hold'em Poker. The flop follows the pre-flop round and precedes the turn an river rounds. Before the flop round, three community cards are dealt.
- **Game**: a single hand of Poker (in which one pot is built and distributed to the winner(s)).
- **Hand**: can refer to multiple concepts, depending on the context in which it used.
  - "Yesterday, we played some *hands* of Poker": see game.
  - "I was dealt a strong *hand*": see holding.
  - "A full house is a strong *hand*", "I flopped a good *hand*": one of the 10 hand categories in Poker.
- **Heads-up**: a Poker game is said to be heads-up when it is a one-on-one (2 player) game.
- **Holding**: the two hidden cards that each player receives in a game of Texas Hold'em Poker.
- **Hole cards**: see *holding.*
- **Implied odds**:
- **Post-flop**: this refers to the total of all three rounds that follow the pre-flop phase. That is, the flop, turn and river.
- **Pot odds**:
- **Pre-flop**: this is the first of four rounds in a Texas Hold'em Poker game, in which there are no community cards showing yet.
- **River**: the fourth are last round of a Texas Hold'em Poker game.
- **Showdown**: in a showdown, two or more players reveal their hidden cards to see who has the best hand. The person with the best hand then wins the pot. A showdown is reached when the river round is completed.
- **Stack**: short for "chip stack". The amount of money that a player has left to bet (and stacked in front of him during a live game) at a Poker table.
- **Turn**: the third of four betting rounds in a game of Texas Hold'em Poker. On the turn, there are four community cards showing.

Appendix B

Features

This appendix provides a list of all the features that we used to describe the current state of a Poker game. They are used to train models that predict moves given the current state of the game. The descriptions provided for some features contain basic Poker-specific lingo, please refer to appendix A when unfamiliar with some of the terminology.

Note that some features can only be used in certain contexts. For example, the features 'amount to call' can only be used in situations where a player is facing a bet that he has to call. Furthermore, features such as 'player was the aggressor on the flop' are only meaningful on the turn and river, where the flop round has completed. As we have discussed in section 4.3.3, we distinguish 8 different states that a Poker game can be in at any time, based on the betting round and whether the current player has to call a bet. We adjust the set of features that is used for models for each of these game states accordingly. Which features can be used in what contexts should be easy to deduce from the descriptions.

Name	**Type**	**Description**
ActivePlayers	Numeric	The number of players that haven't folded.
ActiveToTotalPlayersRatio	Numeric	The number of players that haven't folded divided by the number of total players in the game.
AggressiveToPassiveRatioFlop	Numeric	The ratio of aggressive (bet, raise) to passive (check, call) moves for the flop round.
AggressiveToPassiveRatioPreflop	Numeric	The ratio of aggressive (bet, raise) to passive (check, call) moves for the pre-flop round.
AggressiveToPassiveRatioRiver	Numeric	The ratio of aggressive (bet, raise) to passive (check, call) moves for the river round.

Name	**Type**	**Description**
AggressiveToPassiveRatioTurn	Numeric	The ratio of aggressive (bet, raise) to passive (check, call) moves for the turn round.
AllInWithCall	Boolean	Whether the player is all-in if he calls.
AmountToCall	Numeric	The absolute amount a player has to call.
AverageRank	Numeric	The average rank of the cards on the board (deuce = 0, trey = 1, ..., ace = 12).
CallersSinceLastRaise	Numeric	The number of callers since the last bet or raise on the current street.
EffectiveStackVersusActivePlayers	Numeric	The maximum of the effective stack of the current player against all other players that didn't fold.
EffectiveStackVersusAggressor	Numeric	The effective stack that the current player has relative to the current aggressor in the current round.
HighestCardOnBoardFlop	Nominal	The highest card showing on the board at the flop round (deuce, ..., ace).
HighestCardOnBoardRiver	Nominal	The highest card showing on the board at the river round (deuce, ..., ace).
HighestCardOnBoardTurn	Nominal	The highest card showing on the board at the turn round (deuce, ..., ace).
ImpliedOddsVersusAggressor	Numeric	Implied odds versus the current aggressor: amount to call / (amount to call + size of pot + effective stack versus aggressor)
InPositionVersusActivePlayers	Boolean	Whether the player will be the last to act at post-flop rounds of all players that didn't fold.
InPositionVersusAggressor	Boolean	Whether the player will get to act after the current aggressor on post-flop rounds.
MaxCardsFromSameSuitFlop	Numeric	The maximum number of cards of a single suit showing at the board on the flop.
MaxCardsFromSameSuitRiver	Numeric	The maximum number of cards of a single suit showing at the board on the river.
MaxCardsFromSameSuitTurn	Numeric	The maximum number of cards of a single suit showing at the board on the turn.
MaxCardsOfSameRank	Numeric	The maximum number of cards of the same rank showing at the board on the flop.

Name	**Type**	**Description**
NumberOfBetsFlop	Numeric	The total number of bets and raises on the flop round.
NumberOfBetsPreflop	Numeric	The total number of bets and raises on the pre-flop round.
NumberOfBetsRiver	Numeric	The total number of bets and raises on the river round.
NumberOfBetsTurn	Numeric	The total number of bets and raises on the turn round.
NumberOfDifferentHighCards	Numeric	The number of distinct cards with a rank of jack or higher showing on the board.
OffTheButton	Numeric	The number of players that get to act after the current player on post-flop streets.
OwnPreviousAction	Nominal	The player's previous move, where bet and raise moves are both mapped to 7 abstract moves based on the amount bet or raised.
OwnPreviousActionCategory	Nominal	The player's previous move, where bet and raise moves are not abstracted.
PlayersActed	Numeric	The number of players that has acted since the last bet or raise on the current round.
PlayersLeftToAct	Numeric	The number of players that are left to act on the current round.
PotOdds	Numeric	Amount to call / (Amount to call + amount in pot).
PotSize	Numeric	The absolute amount of money in the pot.
StackSize	Numeric	The amount of money a player has left to wager.
StraightPossibilitiesFlop	Numeric	A number that represents how likely it is that a random holding will have a straight given the board on the flop.
StraightPossibilitiesRiver	Numeric	A number that represents how likely it is that a random holding will have a straight given the board on the river.
StraightPossibilitiesTurn	Numeric	A number that represents how likely it is that a random holding will have a straight given the board on the turn.
WasAggressorFlop	Boolean	Whether the player was the player that made the last bet or raise at the flop.
WasAggressorPreflop	Boolean	Whether the player was the player that made the last bet or raise pre-flop.

Name	**Type**	**Description**
WasAggressorTurn	Boolean	Whether the player was the player that made the last bet or raise at the turn.

Appendix C

Bet and raise size accuracies

The following confusion matrices illustrate how well the models that predict moves (independent of hole cards) predict the correct bet or raise amount. Refer to section 4.5.2 for more details.

Table C.1: Confusion matrix for bet sizes for games with 2 players (accuracy = 0.771 ± 0.005).

		Actual							
		Minbet	**1/4 pot**	**1/2 pot**	**3/4 pot**	**Pot**	**3/2 pot**	**2+ pot**	**Sum**
Predicted	**Minbet**	$5.06 \pm 0.00\%$	$0.90 \pm 0.00\%$	$1.21 \pm 0.00\%$	$0.51 \pm 0.00\%$	$0.20 \pm 0.00\%$	$0.12 \pm 0.00\%$	$0.13 \pm 0.00\%$	8.13%
	1/4 pot	$0.34 \pm 0.00\%$	$3.78 \pm 0.00\%$	$1.60 \pm 0.00\%$	$0.39 \pm 0.00\%$	$0.20 \pm 0.00\%$	$0.10 \pm 0.00\%$	$0.12 \pm 0.00\%$	6.52%
	1/2 pot	$0.10 \pm 0.00\%$	$0.40 \pm 0.00\%$	$2.43 \pm 0.01\%$	$1.33 \pm 0.00\%$	$0.28 \pm 0.00\%$	$0.08 \pm 0.00\%$	$0.10 \pm 0.00\%$	4.72%
	3/4 pot	$0.27 \pm 0.00\%$	$0.58 \pm 0.00\%$	$11.12 \pm 0.01\%$	$65.80 \pm 0.03\%$	$2.47 \pm 0.00\%$	$0.18 \pm 0.00\%$	$0.21 \pm 0.00\%$	80.63%
	Pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	3/2 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	2+ pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Sum	5.77%	5.66%	16.36%	68.04%	3.14%	0.47%	0.57%	100%

Table C.2: Confusion matrix for raise sizes for games with 2 players (accuracy = 0.885 ± 0.011).

		Actual							
		Minraise	**1/4 pot**	**1/2 pot**	**3/4 pot**	**Pot**	**3/2 pot**	**2+ pot**	**Sum**
Predicted	**Minraise**	$5.73 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.13 \pm 0.00\%$	$2.76 \pm 0.01\%$	$0.28 \pm 0.00\%$	$0.12 \pm 0.00\%$	9.02%
	1/4 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	1/2 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.02 \pm 0.00\%$	$0.01 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.02%
	3/4 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Pot	$4.05 \pm 0.01\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.08 \pm 0.00\%$	$82.00 \pm 0.05\%$	$3.27 \pm 0.01\%$	$0.51 \pm 0.00\%$	89.91%
	3/2 pot	$0.02 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.01 \pm 0.00\%$	$0.24 \pm 0.00\%$	$0.69 \pm 0.01\%$	$0.09 \pm 0.00\%$	1.05%
	2+ pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Sum	9.80%	0.00%	0.02%	0.22%	85.00%	4.24%	0.72%	100%

Table C.3: Confusion matrix for bet sizes for games with 6 players (accuracy = 0.635 ± 0.002).

		Actual							
		Minbet	**1/4 pot**	**1/2 pot**	**3/4 pot**	**Pot**	**3/2 pot**	**2+ pot**	**Sum**
Predicted	**Minbet**	$4.65 \pm 0.00\%$	$1.29 \pm 0.00\%$	$1.57 \pm 0.00\%$	$0.49 \pm 0.00\%$	$0.24 \pm 0.00\%$	$0.08 \pm 0.00\%$	$0.07 \pm 0.00\%$	8.40%
	1/4 pot	$0.81 \pm 0.00\%$	$9.01 \pm 0.00\%$	$3.69 \pm 0.00\%$	$0.84 \pm 0.00\%$	$0.37 \pm 0.00\%$	$0.18 \pm 0.00\%$	$0.17 \pm 0.00\%$	15.06%
	1/2 pot	$0.41 \pm 0.00\%$	$2.52 \pm 0.00\%$	$10.88 \pm 0.00\%$	$5.39 \pm 0.00\%$	$1.30 \pm 0.00\%$	$0.34 \pm 0.00\%$	$0.45 \pm 0.00\%$	21.29%
	3/4 pot	$0.10 \pm 0.00\%$	$0.50 \pm 0.00\%$	$13.17 \pm 0.00\%$	$39.07 \pm 0.00\%$	$2.05 \pm 0.00\%$	$0.15 \pm 0.00\%$	$0.22 \pm 0.00\%$	55.25%
	Pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	3/2 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	2+ pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Sum	5.97%	13.32%	29.30%	45.80%	3.95%	0.75%	0.91%	100%

Table C.4: Confusion matrix for raise sizes for games with 6 players (accuracy = 0.584 ± 0.008).

		Actual							
		Minraise	**1/4 pot**	**1/2 pot**	**3/4 pot**	**Pot**	**3/2 pot**	**2+ pot**	**Sum**
Predicted	**Minraise**	$4.16 \pm 0.00\%$	$0.88 \pm 0.00\%$	$0.76 \pm 0.00\%$	$0.53 \pm 0.00\%$	$0.29 \pm 0.00\%$	$0.35 \pm 0.00\%$	$0.41 \pm 0.00\%$	7.39%
	1/4 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	1/2 pot	$1.47 \pm 0.00\%$	$7.68 \pm 0.00\%$	$54.49 \pm 0.00\%$	$25.28 \pm 0.00\%$	$2.35 \pm 0.00\%$	$0.35 \pm 0.00\%$	$1.00 \pm 0.00\%$	92.61%
	3/4 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	3/2 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	2+ pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Sum	5.63%	8.56%	55.25%	25.81%	2.64%	0.70%	1.41%	100%

Table C.5: Confusion matrix for bet sizes for games with 9 players (accuracy = 0.599 ± 0.002).

		Actual							
		Minbet	**1/4 pot**	**1/2 pot**	**3/4 pot**	**Pot**	**3/2 pot**	**2+ pot**	**Sum**
Predicted	**Minbet**	$2.34 \pm 0.00\%$	$0.75 \pm 0.00\%$	$0.67 \pm 0.00\%$	$0.22 \pm 0.00\%$	$0.12 \pm 0.00\%$	$0.03 \pm 0.00\%$	$0.03 \pm 0.00\%$	4.16%
	1/4 pot	$0.57 \pm 0.00\%$	$6.66 \pm 0.00\%$	$3.19 \pm 0.00\%$	$0.72 \pm 0.00\%$	$0.33 \pm 0.00\%$	$0.12 \pm 0.00\%$	$0.13 \pm 0.00\%$	11.73%
	1/2 pot	$0.79 \pm 0.00\%$	$3.74 \pm 0.00\%$	$21.87 \pm 0.00\%$	$11.16 \pm 0.00\%$	$2.49 \pm 0.00\%$	$0.70 \pm 0.00\%$	$0.56 \pm 0.00\%$	41.30%
	3/4 pot	$0.27 \pm 0.00\%$	$0.50 \pm 0.00\%$	$11.49 \pm 0.00\%$	$28.99 \pm 0.00\%$	$1.35 \pm 0.00\%$	$0.09 \pm 0.00\%$	$0.12 \pm 0.00\%$	42.81%
	Pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	3/2 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	2+ pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Sum	3.95%	11.66%	37.22%	41.10%	4.29%	0.94%	0.84%	100%

Table C.6: Confusion matrix for raise sizes for games with 9 players (accuracy = 0.723 ± 0.049).

		Actual							
		Minraise	**1/4 pot**	**1/2 pot**	**3/4 pot**	**Pot**	**3/2 pot**	**2+ pot**	**Sum**
Predicted	**Minraise**	$54.01 \pm 0.13\%$	$8.39 \pm 0.05\%$	$3.81 \pm 0.02\%$	$1.52 \pm 0.01\%$	$0.86 \pm 0.00\%$	$0.45 \pm 0.00\%$	$0.57 \pm 0.00\%$	69.62%
	1/4 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	1/2 pot	$0.53 \pm 0.00\%$	$5.77 \pm 0.03\%$	$17.65 \pm 0.13\%$	$4.83 \pm 0.05\%$	$0.98 \pm 0.01\%$	$0.33 \pm 0.00\%$	$0.29 \pm 0.00\%$	30.38%
	3/4 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	3/2 pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	2+ pot	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	$0.00 \pm 0.00\%$	0.00%
	Sum	54.55%	14.17%	21.46%	6.35%	1.84%	0.78%	0.86%	100%

Appendix D

Questionnaire form

On a scale from 1 to 5, how well do you think the bots played?
o 1. Very poor o 2. Poor o 3. Okay o 4. Well o 5. Very well
On a scale from 1 to 5, how aggressive do you think the bots played?
o 1. Very passive o 2. Passive o 3. Medium aggressive o 4. Aggressive o 5. Very Aggressive
On a scale from 1 to 5, how predictable do you think the bots played?
o 1. Not predictable at all o 2. A little predictable o 3. Quite predictable o 4. Predictable o 5. Very predictable
On a scale from 1 to 5, how humanlike do you feel the bots played?
o 1. Not humanlike at all o 2. A little humanlike o 3. Quite humanlike o 4. Humanlike o 5. Very humanlike

On a scale from 1 to 5, how often do you think the bots were bluffing?
o 1. Hardly ever o 2. Not often o 3. Not too often, nor too infrequently o 4. A bit too often o 5. Far too often
When the bots were bluffing, how large were their bets on a scale from 1 to 5?
o 1. Very small o 2. Small o 3. Neither too small, nor too large o 4. Too large o 5. Far too large o I don't know
When the bots had a good hand, how large were their bets on a scale from 1 to 5?
o 1. Very small o 2. Small o 3. Neither too small, nor too large o 4. Too large o 5. Far too large o I don't know
What do you think the bots' main strategic mistakes are?
On the contrary, what do you think the bots did well?

Bibliography

G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749, 2005.

L. Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, Maastricht University, 1994.

P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2003.

D. Billings. Computer Poker. Master's thesis, University of Alberta, Edmonton, Canada, 1995.

D. Billings and M. Kan. A tool for the direct assessment of Poker decisions. *International Computer Games Association Journal*, 29(3):119–142, 2006.

D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in Poker. In *Proceedings of the 10th conference on AAAI*, pages 493–499. AAAI Press, Menlo Park, CA, USA, 1998a.

D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Poker as a testbed for machine intelligence research. In *Proceedings of Advances in Artificial Intelligence Research*, pages 1–15, 1998b.

D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of Poker. *Artificial Intelligence Journal*, 134:201–240, 2002.

D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale Poker. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 661–668, 2003.

D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. H. Bowling, R. C. Holte, J. Schaeffer, and D. Szafron. Game-tree search with adaptation in stochastic imperfect-information games. In *Computers and Games*, pages 21–34, 2004.

B. Bouzy, B. Helmstetter, and T. Hsu. Monte Carlo Go developments. In J. v. d. Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games: Many Games, Many Challenges.*, pages 159–174, 2003.

G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search: A new framework for game ai. In C. Darken and M. Mateas, editors, *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference.* The AAAI Press, Menlo Park, CA, USA, 2008.

R. Coulom. Efficient selectivity and backup operators in Monte Carlo Tree Search. In *Proceedings of the 5th International Conference on Computer and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, Berlin, Germany, 2006.

A. Davidson. Opponent modeling in Poker: Learning and acting in a hostile and uncertain environment. Master's thesis, University of Alberta, Edmonton, Canada, 2002.

A. Davidson, D. Billings, J. Schaeffer, and D. Szafron. Improved opponent modeling in Poker. In *Proceedings of the 2000 International Conference on Artificial Intelligence*, pages 1467–1473, 2000.

T. G. Dietterich. Ensemble methods in machine learning. *Lecture Notes In Computer Science*, 1857:1–15, 2000.

R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* Wiley-Interscience, New York, NY, USA, 2nd edition, 2000.

S. Dumais, J. Platt, D. Heckerman, and M. Sahami. Inductive learning algorithms and representations for text categorization. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 148–155, 1998.

F. Esposito, D. Malerba, G. Semeraro, and J. Kay. A comparative analysis of methods for pruning decision trees. *IEEE transactions on pattern analysis and machine intelligence*, 19(5):476–491, 1997.

S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte Carlo Go. In *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*, 2006.

F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A grandmaster chess machine. *Scientific American*, 263(4):44–50, 1990.

M. B. Johanson. Robust strategies and counter-strategies: Building a champion level computer Poker player. Master's thesis, University of Alberta, Edmonton, Canada, 2007.

M. Kalos and P. Whitlock. *Monte Carlo methods*. Wiley-VCH, Weinheim, Germany, 2nd edition, 2008.

L. Kocsis and C. Szepesvári. Bandit based Monte Carlo planning. In *Proceedings of the 15th European Conference on Machine Learning*, pages 83–90, 2006.

D. Koller and A. Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94:167–215, 1997.

J. Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine learning*, 3(4):319–342, 1989a.

J. Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4(2):227–243, 1989b.

T. Mitchell. *Machine Learning*. McGraw-Hill Education, 1st edition, 1997.

D. R. Papp. Dealing with imperfect information in Poker. Master's thesis, University of Alberta, Edmonton, Canada, 1998.

L. Pena. Probabilities and simulations in Poker. Master's thesis, University of Alberta, Edmonton, Canada, 1999.

J. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo Method*. John Wiley & Sons, Hoboken, NJ, USA, 2007.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

J. Schaeffer and H. Van den Herik. Games, computers, and artificial intelligence. *Artificial Intelligence*, 134(1-2):1–8, 2002.

J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

T. Schauenberg. Opponent modelling and search in Poker. Master's thesis, University of Alberta, Edmonton, Canada, 2006.

D. P. Schnizlein. State translation in no-limit Poker. Master's thesis, University of Alberta, Edmonton, Canada, 2009.

R. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. The MIT Press, Cambridge, MA, USA, 1998.

G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

G. van den Broeck. Een studie van algoritmes en technieken voor artificiële no-limit pokerspelers (a study of algorithms and techniques for aritificial no-limit Poker players). Master's thesis, Catholic University of Leuven, Leuven, Belgium, 2009.

G. van den Broeck, K. Driessens, and J. Ramon. Monte Carlo Tree Search in Poker using expected reward distributions. In Z.-H. Zhou and T. Washio, editors, *Proceedings of the 1st Asian Conference on Machine Learning: Advances in Machine Learning*, volume 5828 of *Lecture Notes in Computer Science*, pages 367–381. Springer-Verlag, Berlin, Germany, 2009.

J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of the 16th European Conference on Machine Learning (ECML)*, pages 437–448. Springer-Verlag, Berlin, Germany, 2005.

J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, 1944.

M. Wooldridge. *An introduction to multiagent systems.* Wiley & Sons, Chichester, UK, 1st edition, 2002.