MACHINE LEARNING METHODS FOR OPPONENT MODELING IN GAMES OF IMPERFECT INFORMATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

VOLKAN ŞİRİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2012

Approval of the thesis:

MACHINE LEARNING METHODS FOR OPPONENT MODELING IN GAMES OF IMPERFECT INFORMATION

submitted by **VOLKAN ŞİRİN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Prof. Dr. Fatoş Tünay Yarman Vural _____
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Göktürk Üçoluk
Computer Engineering Dept., METU _____

Prof. Dr. Fatoş Tünay Yarman Vural
Computer Engineering Dept., METU _____

Prof. Dr. Faruk Polat
Computer Engineering Dept., METU _____

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Dept., METU _____

Dr. Onur Pekcan
Civil Engineering Dept., METU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: VOLKAN ŞİRİN

Signature :

ABSTRACT

MACHINE LEARNING METHODS FOR OPPONENT MODELING IN GAMES OF IMPERFECT INFORMATION

Şirin, Volkan
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Fatoş Tünay Yarman Vural

September 2012, 62 pages

This thesis presents a machine learning approach to the problem of opponent modeling in games of imperfect information. The efficiency of various artificial intelligence techniques are investigated in this domain. A sequential game is called imperfect information game if players do not have all the information about the current state of the game. A very popular example is the Texas Holdem Poker, which is used for realization of the suggested methods in this thesis. Opponent modeling is the system that enables a player to predict the behaviour of its opponent. In this study, opponent modeling problem is approached as a classification problem. An architecture with different classifiers for each phase of the game is suggested. Neural Networks, K-Nearest Neighbors (KNN) and Support Vector Machines are used as classifier. For modeling a particular player, KNN is found to be most successful amongst all, with a prediction accuracy of 88%. An ensemble learning system is proposed for modeling different playing styles and unknown ones. Computational complexity and parallelization of some calculations are also provided.

Keywords: Machine Learning, Poker, Artificial Intelligence, Games, Ensemble Learning

ÖZ

MAKİNE ÖĞRENMESİ YÖNTEMLERİ İLE EKSİK BİLGİ OYUNLARINDA RAKİP MODELLEME

Şirin, Volkan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Fatoş Tünay Yarman Vural

Eylül 2012, 62 sayfa

Bu tezde, eksik bilgi oyunlarında rakip modelleme problemine ilişkin bir makine öğrenmesi yaklaşımı sunulmuştur. Çeşitli yapay zeka yöntemlerinin bu alandaki verimliliği incelenmiştir. Sıralı bir oyunda eğer oyuncular oyunun o anki durumuna ilişkin tüm bilgiye sahip değillerse buna eksik bilgi içeren oyun denir. Popüler bir örnek olan Texas Usulü Poker tezde önerilen metodların gerçeklenmesi için kullanılmıştır. Bir oyuncunun rakibinin davranışlarını tahmin etmesini sağlayan sisteme Rakip Modelleme Sistemi denir. Bu çalışmada, rakip modelleme problemi bir sınıflandırma problemi olarak ele alınmıştır. Oyunun her bir fazı için değişik bir sınıflandırıcı içeren bir mimari önerilmiştir. Sınıflandırıcı olarak Yapay Sinir Ağları, K-Yakın Komşu (KYK) ve Destek Vektör Makineleri yöntemleri kullanılmıştır. Belirli bir oyuncuyu modellemede %88 doğru tahmin oranıyla KYK yönteminin daha başarılı olduğu gözlenmiştir. Değişik ya da bilinmeyen oyun stillerini modellemek için bir topluluk öğrenme sistemi önerilmiştir. Bazı hesapların karmaşıklığı ve paralelleştirilmesine de yer verilmiştir.

Anahtar Kelimeler: Makine Öğrenmesi, Poker, Yapay Zeka, Oyunlar, Topluluk Öğrenmesi

To my family

ACKNOWLEDGMENTS

First, I want to thank my supervisor, Prof. Dr. Fatoş Yarman Vural for her trust, guidance and encouragement. Apart from her exceptional supervision in this thesis, she is a great mentor in understanding complex problems of life and science. Special thanks should go to my dear friend Ömer Ekmekci who was always there whenever I needed help. I am thankful to Dr. Onur Pekcan whose help is invaluable in organizing my thoughts related to my thesis. I also thank my thesis committee members Prof. Dr. Göktürk Üçoluk, Prof. Dr. Faruk Polat and Prof. Dr. İsmail Hakkı Toroslu for their valuable commentaries and suggestions. I thank to The Scientific and Technological Research Council of Turkey for the scholarship during my master's studies.

Finally, I am very thankful to my family for believing in me and providing full support.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Since the birth of the field, games have been used for testing Artificial Intelligence (AI) techniques[1]. For example, chess game has been studied intensively during the history of AI research[2]. Another well-known example is the Samuel's work[3] on checkers game in which he employed machine learning techniques to create an average player. Several properties of games are the reason for this popularity. First of all, games have well-defined goals and rules. Besides, it is easy to measure success and compare against human experts. Imperfect information games are more challenging because one does not have all the information about the game. Thus, probabilistic reasoning should come into place.

In this thesis, we study one of the most popular card games for our experiments: Texas Holdem Poker. With the introduction of online casinos and TV shows featuring the game, Texas Holdem received massive interest worldwide. At the same time, it gained popularity in the academia. Beginning from 2000s University of Alberta Poker Research Group pioneered the research. With the contributions of researchers worldwide, Poker is now recognized as a useful research domain for AI[4]. A short introduction of the game is provided at Section 1.2.

Poker is a game of imperfect information[5] in which players has only partial knowledge about the current state of the game. In Texas Holdem, each player has 2 hidden cards that opponents can not see until the end of the game. There are also community cards which can be seen and used by all players. These cards are put on the table one by one and at each step a betting round is conducted. The actions in the game has stochastic outcomes since future cards to be opened to the table are not known. Finally, it's a multi-agent game in which exploiting opponent's weaknesses is essential for profit.

The above mentioned properties makes the poker game difficult. For perfect information

games such as chess, game tree search, which is the act of analyzing all possible game sequences, provides an adequate solution. However, these methods have not been sufficient in games of imperfect information. Dealing with imperfect information is the major cause that developments in artificial poker and bridge players have lagged behind progress on other games[6].

1.1 Opponent Modeling

The importance of opponent modeling in zero-sum games, in which win of one player means the loss of the other, has long been observed. Edgar Allan Poe (1902) tells the story of a child whose ability in opponent modeling made him famous[7]:

> I knew one about eight years of age, whose success at guessing in the game of 'even and odd' attracted universal admiration. This game is simple, and is played with marbles. One player holds in his hand a number of these toys, and demands of another whether that number is even or odd. If the guess is right, the guesser wins one; if wrong, he loses one. The boy to whom I allude won all the marbles of the school. Of course he had some principle of guessing; and this lay in mere observation and admeasurement of the astuteness of his opponents. For example, an arrant simpleton is his opponent, and, holding up his closed hand, asks, 'are they even or odd?' Our schoolboy replies, 'odd,' and loses; but upon the second trial he wins, for he then says to himself, 'the simpleton had them even upon the first trial, and his amount of cunning is just sufficient to make him have them odd upon the second; I will therefore guess odd;' –he guesses odd, and wins.

We provide the payoff matrix of this game in Table 1.1. Here, it's obvious that the optimal strategy is randomly guessing even or odd. This strategy guarantees that one does not lose any marbles in the long run. However, one also can not win any marbles. Most of the poker players are not optimal players. A practical measure of expertise of a human poker player is how much money he wins in the tables. Thus, if AI research aims to match up with human experts, we must create artificial players with opponent modeling capabilities.

Table 1.1: Odd-even game

		Poe's Student	
		Even	Odd
You	Even	-1	+1
	Odd	+1	-1

1.2 Description of the Texas Holdem Poker

There are different kinds of Texas Holdem Poker. As most of the previous research on poker, we focus on heads-up type in which there are only two players. There are also variations in the betting structure. We focused on limit poker in which maximum betting amount is limited. The AAAI Computer Poker Competition held in July 2006 also used this betting structure. Basic rules of the game are as follows:

- There are four phases of a game: pre-flop, flop, turn and river. Last three of these called the post-flop phases.
- There are three possible actions in the game: fold, call and raise. Fold means the player withdraws from the game and forfeits previously committed money. Call (or check) means the player commits the same amount of the money with his opponent and continues the game. Raise (or bet) means the player commits more money than his opponent. In this case, if the opponent wants to continue the game, he should call or raise this action.
- In the pre-flop phase, first player (small blind) contributes 1 unit of money and the other player, which is also the dealer, contributes 2 units of money automatically. Then, each player is dealt two hidden (hole) cards. Next, first betting round begins. In a betting round, in order to stay in the game, a player should put at least an equal amount of money as the opponent to the table. Then, the small blind can either call (by putting 1 unit of money), or raise (3 units) or fold. When a player folds, the game is over and the other player wins all the money. When a player calls, the betting round finishes and game continues to the next phase. When a player raises, his opponent is asked whether he will call, fold or raise in response. Four raises are allowed in each round.
- In the flop phase, three community cards are dealt to the table. Every player can see

and use community cards. Another betting round begins in this phase. Dealer asks the other player about his action. Recall that there are three possible course of action at each point: withdrawing, continuing without increasing the bet and continuing with increasing the bet. If second action is performed not in response to the bet of opponent, it is called check instead of call. If third action is performed not in response to the bet of opponent, it is called bet, instead of raise. Thus, if the player checks, the opponent can fold, check or bet; otherwise the opponent have the options of fold, call or raise.

- In each of the turn and river phases, one more community card is dealt and a new betting round is conducted.

- If none of the players folds until the end of the game, it is called show-down. Each player composes a five card hand from a total of seven cards (two hidden cards and five community cards). Both players show their hands and the strongest hand according to the predefined ranking of hands, wins all the money at the table. In poker game, specific patterns in a hand makes it better. For example, if all of the cards in the hand belongs to the same suit (one of hearts, clubs, diamonds and spades), it is called flush. If the cards are sequential like 6, 7, 8, 9, 10, it is called straight. There is a sorting among these patterns. For example, a flush hand beats a straight hand.

1.3 Problem Statement and Scope

Poker is an extensive form game like chess and back-gammon. This game type allows explicitly representing different aspects of the game such as decisions of players, their information about other players and payoffs by a game tree. In the game tree, nodes represent players, edges represent actions and leaves represents outcomes[8]. There are two types of extensive form games: Perfect Information and Imperfect Information. For games including chance element such as back-gammon and poker, chance players are added to the basic model. Chance events are represented by chance nodes in the game tree[5].

For imperfect information games, players may not be aware of exact game state. For example in case of poker, opponent hands are unknown to the player. The set of game states which can not be differentiated by a player is called an information set. Figure 1.1 shows what information set corresponds to in poker context. We can not differentiate between the choice

nodes following the chance nodes that assign cards to our opponent. Thus, all these states corresponds to an information set.

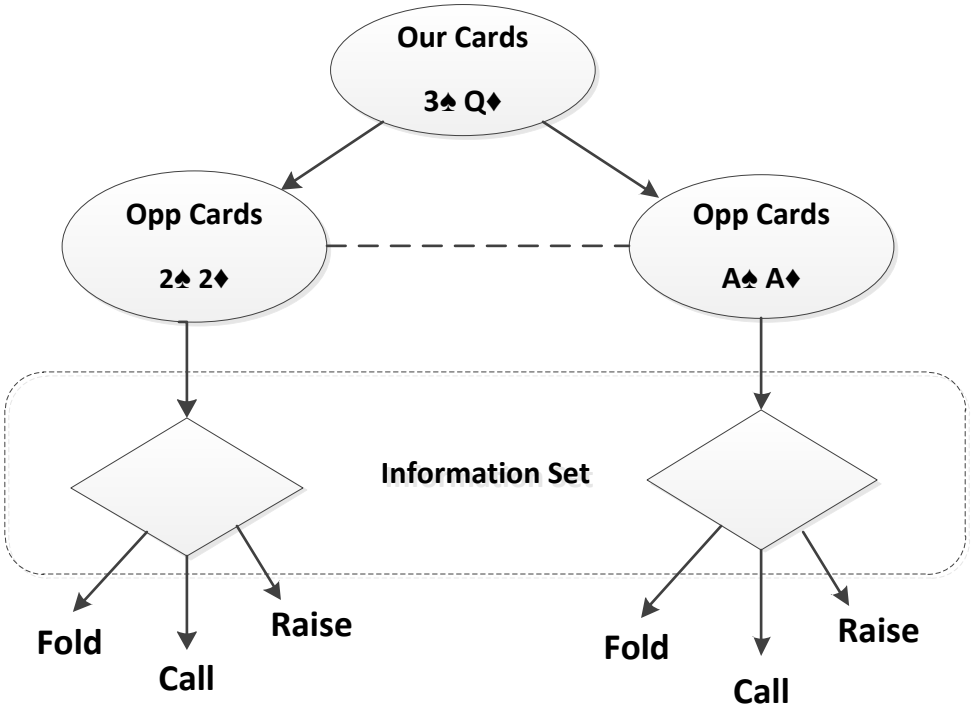

Figure 1.1: Information set concept in poker

Given an information set in the game, the strategy of the player determines how it will respond. Now we can define the objective of the artificial agent as: to find a strategy for a given information set so that expected value of its payoff is maximized. As we will see in the next chapters, predicting how the opponent will react to our particular action is very valuable to determine the strategy of an artificial agent. Hence, in this thesis, our main scope is acquiring this ability, i.e.: building an effective opponent model.

1.4 Motivation and Objectives

Poker research up to now has created powerful agents. However, state-of the art artificial players are still behind human experts. We still lack exploitive agents with human-like capabilities. Previous research shows that opponent modeling is critical for exploitive agents[9]. However we observe that research up to now has used simple opponent modeling techniques such as memorizing previous actions of the opponent. Modern machine learning techniques

such as SVM and KNN has not been studied adequately for this domain. Hence, our research will be focused on studying the efficiency of state-of-art pattern recognition techniques for opponent modeling. Our objective is, after analyzing several machine learning methods for this domain, to propose a robust opponent modeling system that can be used to exploit weak players and match-up with strong players.

1.5 Data Set and Methodology

Starting from 2006, AAAI organizes a tournament for poker bots[10] each year. Academics from different universities and independent researchers participates in the tournament. Thousands of matches are played between players and all the games are logged. Game logs are provided freely for researchers. This data is a great opportunity for building learning machines and investigating playing patterns of state-of-the-art artificial players. Data for different variants of the game is published. We worked on heads-up limit variant of the Texas Holdem game in which there are only two players. When we started our experiments, results of 2011 competition was available, so we worked on these. 2011 tournament was held at the Twenty-Fifth Conference on Artificial Intelligence in San Francisco, California[11]. From 19 participants, we worked on the most successful 8 of them. Their names are: Calamari, Sartre, Hyperborean, Feste, Slumbot, ZBot, 2Bot and LittleRock in the order of their performance in the competition. To give some background about the top performers: Calamari was developed by Marv Andersen of UK, Hyperborean was developed by University of Alberta Computer Poker Research Group and Sartre was developed by University of Auckland Game AI Group.

For each of the above mentioned player, we have data which contains around 30 thousands games. In each game there are at most four phases. Since players often fold early in the game, later phases has less data than the early ones. In each phase, there may be zero, one or more decision points in which the player performs an action. We treat each of these decision points as a sample. As a result, typical size of the data for a player is around 30 thousand samples for pre-flop phase, 20 thousands for flop, 10 thousands for turn and 5 thousands for river.

We try to build a learning system and model the most successful artificial players up to date. Prediction performance of the system will be our main metric. Precision and recall scores will be calculated for all classes. These metrics will be defined in chapter 3. Related performance

scores from previous work will be reported and compared with our scores. Apart from modeling a particular player, we will also measure the generalization performance of our system to other players.

1.6 Contributions

Major contributions of this thesis is listed below:

- The effect of various features is analyzed and an effective feature subset is determined for a particular player.
- Instead of one learning machine for the whole game, we propose four different learners for each phase of the game. This approach is found to be successful.
- Prediction performances of various machine learning algorithms such as Neural Networks, K Nearest Neighbors and Support Vector Machines are analyzed. Their parameters are optimized for maximum performance. KNN is found to be most suitable for this domain.
- By observing a single player, we are able to build an expert modeler which can predict actions of that player. We proposed a way of combining different experts in an ensemble to build a single system that is able to provide predictions about different players, including unknown ones.
- GPU programming is found to be an important tool speeding up calculations.

1.7 Organization of the Thesis

This work is composed of 5 chapters first of which is this introduction. Other chapters are explained below:

- Chapter 2: In this chapter, major approaches to the problem is outlined.
- Chapter 3: After introducing the machine learning techniques that will be used in the chapter, we render the main work of this thesis. Several machine learning methods are applied to the problem and results are discussed.

- Chapter 4: In this chapter, we will discuss the computational complexity of some of the calculations and propose GPU programming to speed them up.
- Chapter 5: Here, we draw some conclusions and talk about possible future directions.

CHAPTER 2

RELATED WORK

In this chapter, we will introduce several different approaches to the problem. Some of the suggested systems are static, which means the strategy of the player does not change. Some of them are dynamic in which the agent adaptively changes its strategy as it gets more information about the playing style of its opponent. We will discuss the weaknesses of the static players and the need for opponent modeling in dynamic players. Some of the metrics previously introduced for assessing the situation of a player are also described in this chapter.

2.1 Knowledge Based Strategies

Knowledge-Based strategies incorporate domain knowledge of experts to rules or formulations. Rule based bots play according to a collection of if-then rules. *SoarBot*, developed by Follek[12], is a good example for a rule-based system. He built his system on top of the SOAR. SOAR is a symbolic cognitive architecture created by Laird et. al.[13]. Follek states that SoarBot performs much worse than human experts. A similar approach is to use formulas inside the rules to decrease the number of rules. Formula based rules generally include metrics such as hand strength and hand potential. These metrics will be described in Section 2.2. Figure 2.1 shows an example formula based rule. It is extremely simple for the sake of illustration. Real rules used in knowledge based players have to account for various complexities in the game. The winner of the 2008 ACPC 6-Player limit Holdem competition[11] was a formula-based bot, named Poki[6], developed by University of Alberta CPRG.

Knowledge Based bots encode expert knowledge in logic or cognitive systems. These systems are sufficient for pre-flop play since the problem size is small at that stage. However, after pre-

```
FlopAction(Hand hand)
     if(hand.strength > 0.5)
          return "Raise"
     else if (hand.strength > 0.2)
          return "Call"
     else
          return "Fold"
```

Figure 2.1: A hypothetical rule for Texas Holdem

flop, game size is very large to be encoded in rules, i.e.: massive rule sets are needed. Besides, because of their static strategy, these systems are found to be exploitable by professional poker players[14]. That is, an expert human can predict the possible responses of these players by guessing the underlying rules and play accordingly to maximize his profit.

2.2 Hand Evaluation Techniques

Hand Evaluation Algorithms aim to assess the strength of the hand of the agent. There are different approaches to evaluate the hand in pre-flop phase and post-flop phases.

2.2.1 Pre-Flop Hand Evaluation

There are $C(52, 2) = 1326$ possible hidden hand pairs in poker. The expected income rate of each hand can be predicted by using a simple technique that is called roll-out simulation. The simulation consists of playing several million trials. In each of these trials, after hidden cards are dealt, all other cards are dealt without any betting and the winning pair is determined. Each trial in which a pair wins the game, increases the value of that particular hand. Of course it is an oversimplification of the game, but it provides an accurate description of relative strengths of the hands at the beginning of the game.

David Sklansky, a professional poker player and author of the leading books about Poker, gives hand rankings for this phase of the game. His table is based on expert experience. It has been observed that there is a strong correlation between his rankings and the results of the roll-out simulation [15]. Table 2.1 shows some example results.

Table 2.1: Some of the pre-calculated simulation results for pre-flop hand evalutaion

Cards	Winning Probability
AAo	85%
KKo	82%
QQo	79%
JJo	77%
TTo	74%
99o	71%

2.2.2 Post-Flop Hand Evaluation

The hand strength, *HS*, is the probability that a given hand is better than that of an active opponent. To calculate HS, all of the possible opponent hands are enumerated and checked whether our agent's hand is better, tied or worse. Summing up all of the results and dividing the number of possible opponent hands give the hand strength. The algorithm for hand strength calculation, taken from[15], is shown as Algorithm 1 in the next page.

Hand potential calculations are for calculating the winning probability of a hand when all the board cards are dealt. The first time the board cards are dealt, there are 2 more cards to be revealed for each round. For the hand potential calculation, we look at the potential impact of these cards. The positive potential, *PPot*, is the chance that a hand which is not currently the best, but improves to win at the showdown. The negative potential, *NPot*, is the chance that a currently leading hand ends up losing.

PPot and NPot are calculated by enumerating all the possible cards for the opponent like the enumeration in hand strength, in addition by enumerating all the possible board cards to be revealed. For all the combinations of possible opponent cards and possible board cards we calculate the PPot and NPot as:

PPot: Count the number of times our agent's hand is behind, but ends up ahead.

NPot: Count the number of times our agent's hand is ahead, but ends up behind.

Algorithm 1 Hand strength calculation

```
function HandStrength(ourCards, boardCards)
    ahead ← 0, tied ← 0, behind ← 0
    ourRank = Rank(ourCards, boardCards)
    for all oppCards do
        oppRank = Rank(oppCards, boardCards)
        if ourRank > oppRank then
            ahead ← ahead + 1
        else if ourRank = oppRank then
            tied ← tied + 1
        else
            behind ← behind + 1
        end if
    end for
    handStrength ← (ahead + tied/2)/(ahead + tied + behind)
    return handStrengh
end function
```

Pseudo code of Hand Potential calculation is shown as Algorithm 2 in the next page, which is adapted from[15].

There are 52 cards in a deck. We hold two of them. At flop there are three cards on the board. For simulation, we should go over all possible hole cards combinations that our opponent may have been dealt. This number indicates the complexity of hand strength calculation. Hand Potential calculation adds one more inner loop as seen in the Algorithm 2 This adds another multiplicative term for the two future cards to be opened to the table after the flop phase of the game. Thus, the hand potential calculation requires $C(47,4) \times C(4,2) = 1070190$ calculations. That much of calculations are generally not feasible to compute on a single CPU, so Monte Carlo simulation is an often used method for decreasing the computation time. In Monte Carlo simulation, instead of going over all the possibilities we only consider a subset of them. We will discuss parallelization of these computations in GPU in chapter 4.

Please note that, in these calculations probability of each car pair belonging to our opponent is assumed to be equal. That would be true if we had no information about our opponent.

Algorithm 2 Hand potential calculation

```
function HandPotential(ourCards, boardCards)
    HP[3][3] ← 0, HPTotal[3] ← 0
    ourRank = Rank(ourCards, boardCards)
    for all oppCards do
        oppRank = Rank(oppCards, boardCards)
        if ourRank > oppRank then
            index ← ahead
        else if ourRank = oppRank then
            index ← tied
        else
            index ← behind
        end if
        HPTotal[index] ← HPTotal[index] + 1
    end for
    for all (turnCard, riverCard) do
        ourPossible ← Rank(ourCards, boardCards, turnCard, riverCard)
        oppPossible ← Rank(oppCards, boardCards, turnCard, riverCard)
        if ourPossible > oppPossible then
            HP[index][ahead] ← HP[index][ahead] + 1
        else if ourPossible = oppPossible then
            HP[index][tied] ← HP[index][tied] + 1
        else
            HP[index][behind] ← HP[index][behind] + 1
        end if
    end for
    PPot ← HP[behind][ahead] + HP[behind][tied]/2 + HP[tied][ahead]/2
    PPot ← PPot/(HPTotal[behind] + HP[tied]/2)
    NPot ← HP[ahead][behind] + HP[tied][behind]/2 + HP[ahead][tied]/2
    NPot ← NPot/(HPTotal[ahead] + HP[tied]/2)
    return (PPot, NPot)
end function
```

However, as we shall see in Section 1.1, an efficient opponent modeling system can provide us better predictions about our opponent's hand. Thus, opponent modeling is also very beneficial for increasing the quality of basic metrics of the game.

Finally, we can compute the winning probability at showdown, based on hand strength and hand potential as[16]:

$$P(win) = HS \times (1 - NPot) + (1 - HS) \times PPot \tag{2.1}$$

2.3 Abstractions and Game Theoretic Strategies

Game Theoretic Strategies aim to calculate the Nash equilibrium of the game. If none of the players in a game can benefit from changing its strategy, it is called Nash equilibrium. In this equilibrium point, the strategy of the players are optimal. For example, for the even-odd game described in section 1.1 optimal strategy is selecting odd or even randomly with equal probabilities.

In simple games, Nash equilibrium can be computed by finding best responses of each player from the payoff matrix. However, since state space of poker is very large, exact computation of optimal play is computationally infeasible. Nevertheless, researchers achieved to compute this for simplified versions of the game [9]. A way of decreasing the computational cost is to abstract and generalize some aspects of the game.

The game of poker has above 10^{17} possible game states. Because of this size, our current computational capacity is not powerful enough to solve the game, i.e.: compute the Nash equilibrium. Abstractions are conceived as a remedy for this problem. The most basic one is the card abstraction in which cards with certain similarity are grouped and treated equally during the calculations. Similarity metric can be expected hand strength (E[HS]) or expected hand strength squared (E[HS^2]) [5]. Another abstraction is called round abstraction in which betting rounds are reduced to decrease the number of possible game states. For example, PsOpti, which is the first artificial player that employed near-equilibrium solutions of the game, had used betting round abstractions[17].

Another approach is to calculate ε-Nash equilibrium. In ε-Nash equilibrium, the strategy is

not immune to losing but it is only exploitable at maximum by a certain amount ε. State-of-the-art method for calculating ε-Nash is Counterfactual Regret Minimization [18]. In this method, regret (opportunity cost), which is the difference between the highest possible payoff and the payoff of the taken action, is minimized at each information set. This method is proved to be ε-Nash in heads up game and shown to be successful in multiplayer games.

These approaches create agents that play near optimally so that they will not lose in the long run even against the worst case opponent. However, they also assume their opponent plays near optimally. While this creates robust agents, they can not exploit weak players that are far away from optimal play. Rubin states that in order to exploit the weaknesses of an opponent one must have an opponent modeling system[9]. In the next section, we will review an adaptive playing technique which harnesses the power of opponent modeling to take advantage of weak players.

2.4 Adaptive Game Tree Search

For the game tree of the limit poker, the branching factor for each node is three; each branch corresponds to one of the possible actions: fold, call or raise. As previously explained, fold means the player withdraws from betting. Call means the player puts an equal amount of money. Raise means the player puts an additional amount and challenges his opponent to call. An example game tree for limit poker is shown in Figure 2.2, adapted from [19]. In this example, number of raises is limited to two, limiting the depth of the tree for sake of simplicity. This game tree is constructed for the case our agent bets. Leafs of the tree shows expected values, assuming our opponent has the winning hand and previously there is $35 in the pot. Here $35 is given as an example and does not have a significance. The probabilities for branches leading to opponent nodes in the figure come from opponent modeling module.

Figure 2.2 also shows the application of miximax algorithm and probability distribution of opponent cards. Darse Billings proposed miximax algorithm for evaluation of game trees of imperfect information [15]. In this algorithm, in order to determine the expected value of an agent node, values coming from each branch is averaged, each branch is weighted with assigned probability. The formula for the expected utility of a particular node is:

$$EV(O) = \sum_{i \in \{f,c,r\}} Pr(O_i) \times EV(O_i) \tag{2.2}$$

Here O_i is the ith possible action of the opponent. For each possible action we multiply the probability of that action ($Pr(O_i)$) with expected utility of that action $EV(O_i)$. All possible opponent decisions are mixed and the agent is assumed to choose always the branch with the maximum expected value (EV). This is why it is called miximax algorithm. If the agent uses a chance factor to determine its action, then the algorithm is called miximix algorithm[15].

Calculating expected values at fold leaf nodes are straightforward because the agent loses all the money. In order to determine expected values at call leaf nodes, opponent hand probability distribution is used. This distribution gives us an idea about what cards our opponent may be holding in his hands. With this information, we can calculate winning probability (P_{win}) using the formula at Section 2.2. Formula for expected utility of a leaf node is given below:

$$EV(Leaf) = (P_{win} \times PotValue) - (0.5 \times PotValue) \tag{2.3}$$

In the formula, half of the pot value is subtracted because it is the cost of reaching the leaf node. While expanding the game tree, after each of the hypothetical opponent action, the probability distribution of the opponent cards is updated and assigned to the subsequent agent node. Creation and updating of this probability distribution is discussed at the next section.

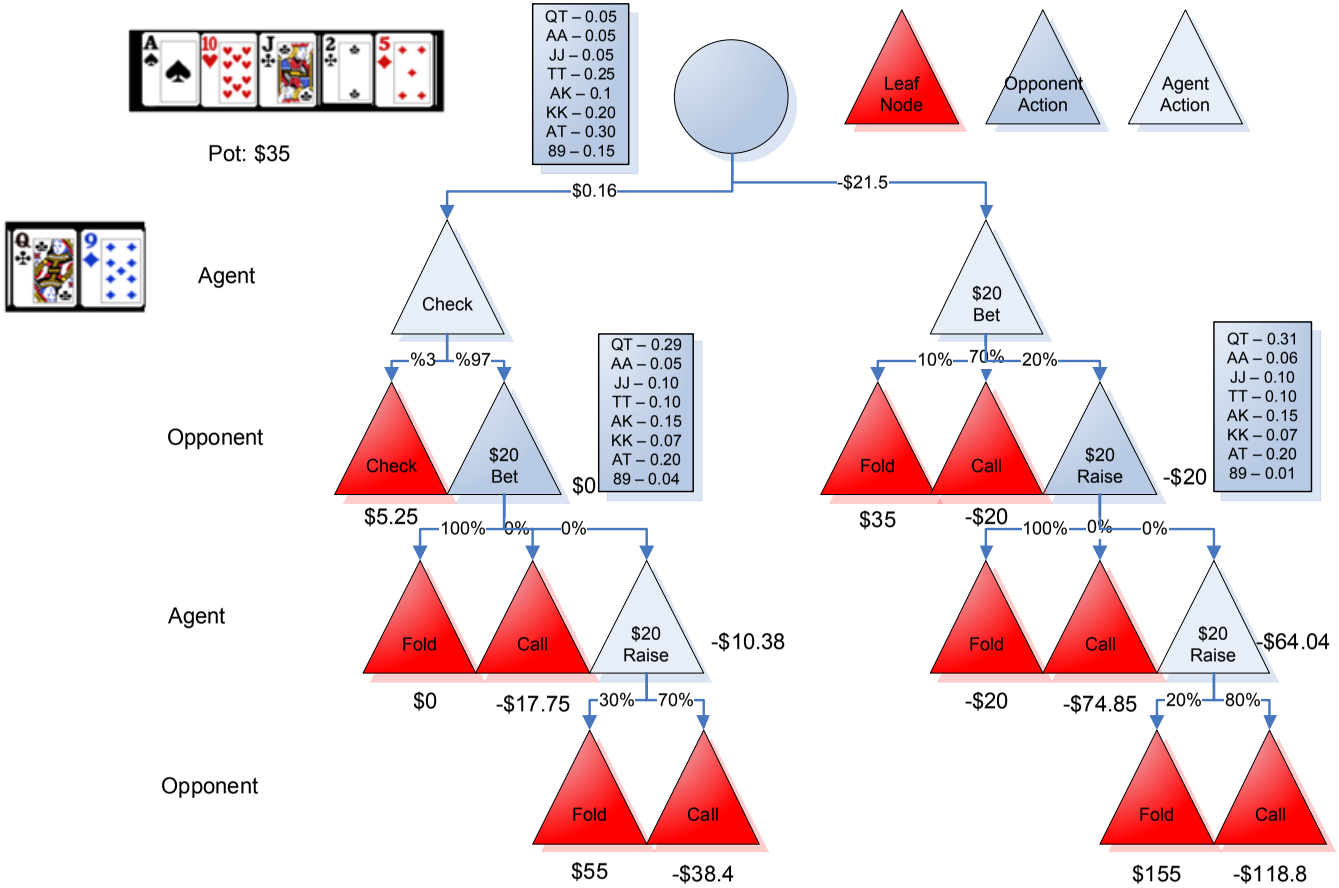

Figure 2.2: Adaptive game tree

2.5 Opponent Modeling

Opponent Modeling algorithms predict how the opponent may respond to our particular action. In 2001 book of *Machines that learn to play games*[20], Fürnkranz states that opponent modeling is important, but somewhat neglected research area in game playing. Opponent Modeling information is critical for decision making systems as can be seen from the previous section. However, research on opponent modeling in the domain of games of imperfect information has been somewhat limited. Billings et al.[21] builds an opponent model by computing rank histogram of opponent's hand for different betting sequences. Hands are categorized to 5 groups according to their strength. To put it another way, they use a hand abstraction as discussed in Section 2.3. After showdown we learn the hand of the opponent and increase the corresponding bin of the all matching betting sequences. This way, they get a relationship between a betting sequence and the hand of an opponent. Unfortunately, they do not give a performance metric for their opponent modeling module, which is part of a larger system. We observe that their method is similar to popular KNN method of machine learning which will be introduced at Section 3.7.3.

Davidson uses Neural Networks to assign each game state a probability triple[22]. Many features such as number of active players, relative betting position, size of the pot, and characteristics of the community board cards are used to represent the game state. They report a typical prediction accuracy around %80.

A lot of times, expert human players have been able to guess their opponent's hand by observing their actions. If we had some idea about what our opponents may hold in their hands, determining the optimal action is straight-forward as shown in the previous section. Darse Billings, in his PhD thesis shows a way to predict the secret cards of the opponent[15]. He keep predictions about opponent's hand in a probability distribution table. At the beginning of the game each possible hand has equal probability. As the game progresses and new actions of our opponent are observed, the probability distribution is updated accordingly. The Opponent Modeler gives probability of a particular opponent action given its hand and the table state. We can update the probability distributions using Bayes' Law as follows:

$$P(OH|\ OA,\ TS) = \alpha \times P(\ OA\ |OH,\ TS) \times P(OH,\ TS) \tag{2.4}$$

Here, OH denotes Opponent Hand, OA denotes Opponent Action and TS denotes Table State

which represents all the other information (features) that we have about the current state of the game such as previous action history and relative positions of players. α is the normalization coefficient for making the sum of probabilities one. In Figure 2.2, evolution of the probability distribution table can be seen in the context of a game tree. This formulation shows that if we have an idea about how a certain players would play with a particular hand, we can predict its hand. In order to gain that information, we have to build a model about that player, because each player tends to play a particular hand differently. In the following chapter of this thesis, we will propose a system that is capable of building such models for different players.

2.6 Chapter Conclusion

In this chapter, major approaches and concepts in the problem domain are reviewed. Previous research can be grouped into two camps. First one is trying to build an artificial player that plays optimally, i.e.: robust to the worst case opponent. On the other hand, second group of researchers are trying to build an adaptive player that are able to exploit non-optimal players. We note that most of the poker players are playing non-optimal and one must exploit this in order to make profit. We see that opponent modeling is essential for building dynamic agents that are able to adapt itself to different opponents. We also note the lack of a comprehensive machine learning study in the domain of opponent modeling, which motivates us to provide one.

CHAPTER 3

A LEARNING MODEL FOR OPPONENT MODELING

In this chapter, our main aim is to build an opponent modeling system that can be used in an artificial player architecture. For this purpose, we will introduce and apply several machine learning techniques. First, we will describe our feature set and analyze the data. Then, several feature selection algorithms are discussed. After selecting the feature set, we analyzed the performance of three classifiers: Neural Networks, KNN and SVM. Results for these classifiers are reported and compared. Finally, we examine generalization performance of our system for different and unknown players.

3.1 Opponent Modeling as a Classification Problem

As explained in Section 1.5, at each decision point, there are three actions that a poker player can take: fold, call and raise. In order to use machine learning techniques, we interpret this decision problem as a classification problem. Classification is the act of assigning a class label to an object by using the feature vector corresponding to that object [23, p. 9-16]. Feature vector is composed of individual features, each of which describes a different aspect of the object at hand. In our case, the object is the state of the game. At each decision point, we extract several features about current state of the game and produce a feature vector. This feature vector is fed to the classifier for the purpose of training or testing.

A supervised pattern recognition procedure includes two main tasks: training and testing. Supervision means that during training, we supply correct labels of the samples to the classifier. In training phase, the classifier learns to differentiate between different classes of objects. In the testing phase, we supply a different set of data to the classifier and measure how well it

labels the objects.

3.2 System Overview

The block diagram of the system can be seen at Figure 3.1. Game logs are the files describing each game, listing mutual actions of the players. We take game logs and extract features at each decision point. This procedure generates feature files. At this point, some of data is taken for training and some of data is spared for testing. We take 20% of the data for testing. Feature files contain lots of redundant information. In the feature selection process, we drop some of the features and obtain a smaller feature set. After that, we feed the resulting data to classifiers. Each classifier has a different set of parameters to tune. Thus, parameter selection is an important phase of the training process. After selecting parameters and training the classifiers we test the system with the remaining data. We experimented with three classifiers: Neural Network, KNN and SVM. The flow of Figure 3.1 is executed for each of these classifiers. Then, their results are compared.

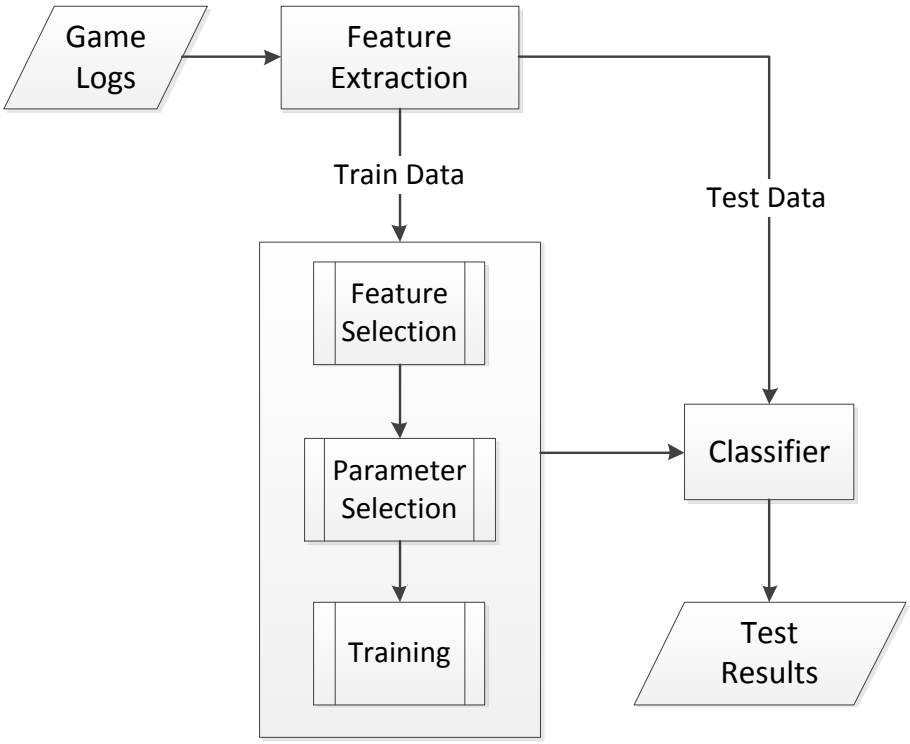

Figure 3.1: System Overview

As explained in Section 1.2, the game is consisted of four phases: pre-flop, flop, turn and river. Previous similar approaches contain one learning system for all the phases of the game. This has some advantages because phases are not inherently independent. Players remember previous phases of the game during the match. But, their playing styles also change as the game progresses. Because of this, we propose different learners for each phase of the game. As will be shown, this gives better results than a single learner for all of the game. Several features are selected to propagate information between phases to overcome the disadvantages of this approach. Thus, the flow of Figure 3.1 is executed for each phase of the game and for each of the different players we trying to model.

Classifiers are tested with several players and for all phases. Since tabulating all the details of the experiments are not feasible, we generally present detailed results for an example player or an example phase and give summaries for the others in the following sections.

3.3 Data Analysis

Game logs contains actions of players and the results. We treated each decision point for the player we are trying to model, as a sample. At each of these points, we extracted several features. Following will describe our initial feature set and its analysis.

3.3.1 Candidate Features

Before selecting candidate features, we consider all the elements affecting a decision of the player. The obvious element is the cards: cards the player holds and cards on the board. There are $C(52, 2) = 1326$ possibilities for hole cards and there are $C(52, 5) = 2598960$ board configurations (the cards that are opened to the table). Exposing card sets to a machine learner as a feature is doomed to fail because of this very high dimensionality. Therefore, researchers usually convert card information to a couple of numerical values. Hand Strength is used to describe current winner of the game, and hand potential is used to describe the future possibilities. These metrics are explained in Section 2.2.

After careful consideration, we listed the features in Table 3.1 as candidates. Most of the features are either obvious or described before so we will explain here only the new ones. Basic

Table 3.1: Candidate features

Feature Id	**Explanation**
1	Hand Strength, See Section 2.2.2
2	PPot, See Section 2.2.2
3	NPot, See Section 2.2.2
4	Whether the player is dealer or not.
5	Last action of the opponent (null, call or raise)
6	Last action of the opponent in context (null, check, call, bet or raise)
7	Stack (money) committed by the player in this phase
8	Stack committed by the opponent in this phase
9	Number of Raises by the the player in this phase
10	Number of Raises by the opponent in this phase
11	Hand Rank
12	Winning Probability, See Section 2.2.2
13	Hand Outs
14	Number of Raises by the player in previous phases
15	Number of Raises by the opponent in previous phases
16	Highest valued card on the board
17	Number of Queens on the board
18	Number of Kings on the board
19	Number of Aces on the board

concepts of the game is explained in Section 1.2 and Section 2.2 explains hand evaluation metrics. 11th feature, Hand Rank is the relative ranking of a particular five card hand by the rules of the game at showdown. That is, the player with the greater Hand Rank wins the game. 13th feature, Hand Outs are the number of possible cards that may be opened to the table and improve the rank of the hand.

All the features listed in Table 3.1 is applicable to flop and turn phases. A number of features are not applicable for river and pre-flop phases. For example, hand potential is not meaningful for the river phase because all the board cards are already dealt. Number of features may seem to be small, but note that first three features embodies lots of information about the present and future of the game. Furthermore, as it explained in the introduction of this chapter, we build four independent learning machines for four different phases of the game. Hence, phase information is also inherent in a learning machine. That is, there is no need to list it as a separate feature.

3.3.2 Class Distributions

Before feeding all the features to a classifier, one must analyze the data and determine the suitability of it to the classification procedure. A good practice in Machine Learning is to look at the class distributions before performing any classification task. Table 3.2 shows mean and standard deviations of features in flop phase. It can be seen from the table that fold class is quite separable from the others, since its mean is a couple of standard deviations away from them for several features. For example, for the features with id 1, 3, 5, 6, 8, 10, 11, 12, 13, we see that the distance between fold mean and the call mean is greater than the fold standard deviation. Call and Raise classes are rather close. We do not have a feature whose means are more than a standard deviation away between these classes. But still there is some distance in between.

Another tool for data inspection is cross-correlation matrix of the individual features. The formula for calculating each entry of the cross-correlation matrix is given below:

$$\begin{aligned} R(x_i, x_j) &= \frac{Cov(x_i, x_j)}{\sqrt{Cov(x_i, x_i)Cov(x_j, x_j)}} \\ \text{where} \quad Cov(x_i, x_j) &= E\left\{(x_i - \mu_i)(x_j - x_j)\right\} \end{aligned} \tag{3.1}$$

Table 3.2: Mean and standard deviations of features in flop phase

Feature Id	Fold Mean	Fold Std	Call Mean	Call Std	Raise Mean	Raise Std
1	0.21	0.12	0.49	0.26	0.59	0.28
2	0.13	0.03	0.10	0.08	0.09	0.09
3	0.06	0.03	0.11	0.05	0.11	0.05
4	0.25	0.43	0.37	0.48	0.49	0.49
5	1	0	0.49	0.47	0.43	0.36
6	0.77	0.06	0.37	0.37	0.27	0.28
7	0.04	0.13	0.03	0.14	0.007	0.06
8	0.54	0.13	0.25	0.31	0.11	0.22
9	0.02	0.06	0.01	0.06	0.003	0.03
10	0.25	0	0.10	0.12	0.05	0.10
11	0.003	0.002	0.009	0.02	0.01	0.03
12	0.30	0.07	0.50	0.16	0.57	0.18
13	0.28	0.02	0.31	0.13	0.31	0.15
14	0.06	0.10	0.09	0.12	0.19	0.10
15	0.22	0.06	0.20	0.09	0.13	0.12
16	0.84	0.15	0.80	0.17	0.80	0.17
17	0.08	0.15	0.08	0.15	0.07	0.15
18	0.08	0.16	0.07	0.15	0.07	0.15
19	0.09	0.16	0.07	0.14	0.07	0.15

In this formulation, x_i represents i^{th} feature, μ_i is its mean and E is the expectation which is calculated by averaging over all of the samples. Cross-correlation matrix can be used to detect linear dependencies between the features[24].

Cross correlation matrix is shown as Table 3.3. Only the upper triangle is shown because it is symmetric. We observe several strong correlations between features. Some significant ones are listed below:

- Since each commit is a result of a call or raise action, we see a strong positive correlation between PhaseRaise (9) and PhaseCommit (7). A similar correlation exists between (6) and (8).
- HandStrength (1) is negatively correlated with PositiveHandPotential (2) because a strong hand is harder to improve.
- OppLastAct (5) is correlated with OppPhaseCommited (8) and OppPhaseRaises (10). We observe that previous commitments encourages the player to raise more.
- Between LastAction (6) and OppPhaseRaises (10). This is also expected because last action being a raise increases the committed amount.
- An interesting positive correlation can be seen between PlayerDealer (4) and PlayerGameRaises (14). In addition, there is a negative correlation between PlayerDealer (4) and OppGameRaises (15). This shows that being in the dealer position, strongly encourages making raises, and vice versa.
- Between WinningProb (12) and the Hand Strength (1). This comes from the definition of the winning probability as seen in Section 2.2.2.

These observations suggest that we can reduce the size of our feature set without loss of much information. We will return this subject in Section 3.6 and determine our feature set.

Table 3.3: Cross-correlation matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	-0.81	0.68	0.08	-0.03	-0.03	0.03	-0.03	0.03	-0.05	0.26	0.95	-0.36	0.17	-0.06	0.03	0.03	0.03	0.01
2		1	-0.67	-0.05	0.01	0.01	-0.02	0.01	-0.02	0.02	-0.25	-0.64	0.74	-0.09	0.04	-0.11	-0.05	-0.07	-0.05
3			1	0.06	-0.02	-0.03	0.01	-0.03	0.01	-0.04	0.19	0.48	-0.38	0.12	-0.05	-0.13	-0.03	-0.07	-0.07
4				1	0.37	0.28	0.24	0.13	0.25	0.05	0.07	0.09	-0.01	0.76	-0.75	-0.01	-0.01	-0.01	-0.01
5					1	0.97	0.23	0.83	0.24	0.89	-0.05	-0.04	0.01	0.07	-0.09	0.02	0.01	0.01	0.01
6						1	0.38	0.93	0.39	0.95	-0.05	-0.04	-0.01	0.02	0.01	0.02	0.01	0.02	0.01
7							1	0.61	0.98	0.30	0.01	0.04	-0.01	0.18	-0.29	0.03	0.02	0.02	0.01
8								1	0.61	0.94	-0.05	-0.03	0.01	-0.06	0.14	0.03	0.01	0.02	0.01
9									1	0.29	0.01	0.03	-0.01	0.19	-0.32	0.02	0.02	0.02	0.01

Table 3.3 (continued)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
10										1	-0.07	-0.06	0.01	-0.15	0.31	0.02	0.01	0.02	0.01
11											1	0.24	-0.29	0.20	-0.05	0.00	0.01	0.01	-0.01
12												1	-0.18	0.17	-0.07	0.03	0.03	0.03	0.01
13													1	-0.02	0.01	-0.09	-0.02	-0.04	-0.04
14														1	-0.57	-0.03	-0.01	-0.01	-0.01
15															1	-0.01	-0.01	-0.01	-0.01
16																1	0.23	0.38	0.56
17																	1	-0.07	-0.09
18																		1	-0.09
19																			1

3.3.3 K-Means Clustering

Before applying classification algorithms, we investigated whether or not data naturally separates to three classes. For this purpose we employ the K-Means clustering algorithm. Clustering is the unsupervised classification of data points into clusters[25]. K- Means is the most basic clustering algorithm introduced by James MacQueen in his 1967 paper[26]. An intuitive description of the algorithm is given below.

Algorithm 3 K-Means Algorithm

```
Assign k random means
repeat
    Assign each data item to the closest mean
    Update means by computing the centroid of all data points in each group
until centers converge
```

The weakness of the algorithm is that you should somehow know the number of clusters in the data (the k parameter) beforehand. However, in our case, number of classes is fixed and that makes K-Means a suitable tool for our purposes.

Results of the algorithm, for the data of Calamari flop phase, is shown at Table 3.11. Results are validated with several runs of the algorithm. As it is seen from the confusion matrix at Table 3.4, all of the fold samples are contained in the first cluster. Second cluster is mostly composed of call samples and third cluster is mostly composed of raise samples. We see that fold class overlaps with call class significantly, but it is contained in a smaller cluster. There is also some overlap between call and raise classes. We see that the results of this experiment are compatible with previous analysis regarding means and variances of the three classes.

Table 3.4: Confusion matrix for k-means clustering

Classes	Cluster-1	Cluster-2	Cluster-3
Fold	3599	0	0
Call	9173	8525	202
Raise	2187	4385	8132

Table 3.5: An example confusion matrix

	Predicted		
	Fold	Call	Raise
Fold	ff	cf	rf
Call	fc	cc	rc
Raise	fr	cr	rr

3.4 Parameter Selection and Cross Validation

As we will see in the next sections, each learning algorithm has a different set of parameters. These parameters should be tuned to achieve maximum test performance. Since parameter selection is a part of training process, we can not use test results for this purpose. Instead, we use validation results, which are obtained by treating some of the training data for testing purposes.

K-fold cross validation is a commonly used technique for validating our parameter set. In k-fold cross validation, first, training data is divided into k subsets. The learner is trained tested k times. Each time one subset (fold) is spared for testing[27]. Then, all k results are combined to determine the success of a selected parameter. This way, a validation accuracy is obtained for a parameter set. Next, we should select the best parameter set. *Grid Search* is a basic method for finding optimum parameter set. For each parameter, we define a set of values to test. Thus, our search space is not continuous, it forms a grid. In grid search procedure, we exhaustively compute the validation accuracy for all possible parameter sets and take the one giving maximum score[28].

3.5 Metrics

When assessing the performance of a particular parameter set or a classifier, we use several metrics. All metrics are based on the confusion matrix which shows the classification performance of an algorithm. Table 3.5 shows an example confusion matrix.

In the confusion matrix, columns represent our predictions and rows represent the true labels. During validation or testing phases, we get predictions from the classifier with a certain pa-

rameter set and compare it with the true labels. In the example table, first character of an entry denotes the prediction and the second character denotes the true label. For example, ff means classifier predicts fold and the true label was fold whereas fc means classifier predicts fold but true label was call. Then, we can define precision and recall as below:

$$Precision_{fold} = \frac{ff}{ff + fc + fr} \quad Precision_{call} = \frac{cc}{cf + cc + cr} \quad Precision_{raise} = \frac{rr}{rf + rc + rr}$$

Hence, precision metric reflects what percentage of our predictions for a particular class is correct. Whereas, recall metric reflects what percentage of a given class is correctly labeled:

$$Recall_{fold} = \frac{ff}{ff + cf + rf} \qquad Recall_{call} = \frac{cc}{fc + cc + rc} \qquad Recall_{raise} = \frac{rr}{fr + cr + rr}$$

These are the metrics for individual classes. For an overall metric, we look at the total number of correct classifications in the test data:

$$Accuracy = \frac{ff + cc + rr}{ff + cf + rf + fc + cc + rc + fr + cr + rr}$$

We could have test accuracy or validation accuracy. As explained in the previous section, validation is an important stage of the training process in which a part of the training data is used for parameter testing. For example, for the case of 4-fold cross validation, we spare 25% of the training data for validation. We get confusion matrix and accuracy for this 25% percent and calculate the accuracy. Doing this for all four quarters of the data and taking average of these accuracies give us the cross validation accuracy which can be used to measure the performance of a parameter set.

3.6 Feature Selection

Before the classification step, one must try to select an optimal subset of candidate features that leads to the greatest performance according to a predefined optimality criteria. For this purpose, we employ a feature selection algorithm. In our problem, feature selection is the process of selecting a particular subset of features for each player and game phase.

As explained in [29] a feature selection algorithm consists of four parts: 1- Subset Generation, 2- Subset Evaluation, 3- Stopping Criterion, 4-Result Validation. Backward elimination and forward selection are the basic subset generation algorithms which iteratively find the optimal feature set.

In backward elimination technique, we start with full feature set. In our case, this is the set with 19 features listed in Table 3.1. In each iteration, we remove the least useful feature. Least useful feature is the one whose absence improves the performance most. Whereas, in forward selection procedure, new features are progressively added to the current set. As explained in [24], backward elimination has a certain advantage over forward selection method. A feature may be more useful in the presence of another certain feature so if we start with a small feature set we might miss a good combination. Hence, backward elimination can give better results. However, it can also be computationally more expensive because it works on larger sets than the forward selection algorithm.

We experimented with sequential backward elimination and forward election techniques for subset generation. Cross-validation accuracy is used as the evaluation criteria for each subset. We terminate the algorithms when addition or subtraction of a feature can not generate better performance then the previous iteration.

In forward selection method, we start from empty set and select one feature at each step. We get cross validation scores for each of the candidate expansion. Then, we go on with the most promising one, i.e.: the one with the greatest validation accuracy. Table 3.6 shows the steps for the player LittleRock in flop phase. As seen from the table, expansion stops after 11th step because we can not find a set with 12 features better than our current 11-sized set.

Table 3.6: Steps during forward selection for the data of LittleRock flop phase.

Feature Set	Validation Accuracy
{14}	61.5%
{6, 14}	77.5%
{6, 12, 14}	83.2%
{1, 6, 12, 14}	86.4%
{1, 6, 12, 14, 16}	87.3%
{1, 6, 12, 14, 15, 16}	87.9%
{1, 2, 6, 12, 14, 15, 16}	88.1%
{1, 2, 6, 12, 13, 14, 15, 16}	88.3%
{1, 2, 6, 9, 12, 13, 14, 15, 16}	88.4%
{1, 2, 3, 6, 9, 12, 13, 14, 15, 16}	88.5%
{1, 2, 3, 4, 6, 9, 12, 13, 14, 15, 16}	88.5%

In sequential backward elimination, at each iteration we generate all possible N-1 subsets of the N-sized feature set. We get cross-validation scores for each of them. We continue in the next iteration with the one with the highest accuracy. Table 3.7 shows the steps during reduction for the player LittleRock in flop phase of the game. As seen from the table, first two exclusions increase the performance significantly but later ones are insignificant. After 9th step, we can not find a better subset and stop. Please observe that the final feature set also agrees with suggestions of the Section 3.3.2. Feature sets produced by two methods are similar but not exactly the same. Features with id 1, 6, 9 and 16 are excluded by backward reduction but included in forward selection. We see that backward elimination gives slightly better performance throughout the entire data set.

In Table 3.8, selected feature sets for each player produced by the backward selection algorithm is presented. They are similar but not exactly the same. This is expected because the style of each player is different. Some features may work in some players but not in others.

Sequential Backward Elimination method is computationally very expensive because it analyses each N-1 subset of N-sized feature set at each step. It is very difficult to perform this analysis for all the players and phases. Therefore, we need an algorithm that can provide us the relative importance of the features. If we would have such information, instead of trying

Table 3.7: Steps during backward elimination for the data of LittleRock flop phase

Feature Set	Validation Accuracy
All Features	88.2%
All Features - {16}	88.6%
All Features - {1, 16}	88.8%
All Features - {1, 7, 16}	88.8%
All Features - {1, 5, 7, 16}	88.8%
All Features - {1, 5, 6, 7, 16}	88.8%
All Features - {1, 5, 6, 7, 9, 16}	88.8%
All Features - {1, 5, 6, 7, 9, 10, 16}	88.8%
All Features - {1, 5, 6, 7, 9, 10, 11, 16}	88.8%

Table 3.8: Selected feature sets for different players

Player	Feature Set
2Bot	{2, 3, 4, 9, 10, 11, 12, 13, 14, 15 }
LittleRock	{2, 3, 4, 8, 12, 13, 14, 15, 17, 18, 19}
Slumbot	{1, 2, 3, 4, 8, 11, 12, 14, 15}
ZBot	{1, 3, 5, 8, 12, 13, 14, 15, 16, 18}
Feste	{1, 3, 4, 8, 11, 12, 13, 14, 15}
Hyperborean	{2, 3, 6, 10, 12, 13, 14, 16}
Sartre	{2, 6, 8, 12, 13, 14, 15}
Calamari	{1, 2, 3, 4, 6, 9, 12, 13, 14, 15, 16}

each subset, we could just drop the least important one. Relief-f is a popular algorithm for obtaining that information.

Relief-f is an algorithm introduced by Kononenko[30], which is a multiclass extension to the original Relief algorithm by Kira and Rendell[31]. It is used for estimating the quality of individual features with respect to classification. This algorithm works by how well a feature distinguish among instances that are close to each other. The algorithm first selects random samples. Then, for each sample, it looks for the nearest neighbor belonging to the same class and belonging to the other class. An attribute which has different values between different classes and which has similar values in the same class gets more weight. In this way, Relief-f algorithm computes relative importance of individual features.

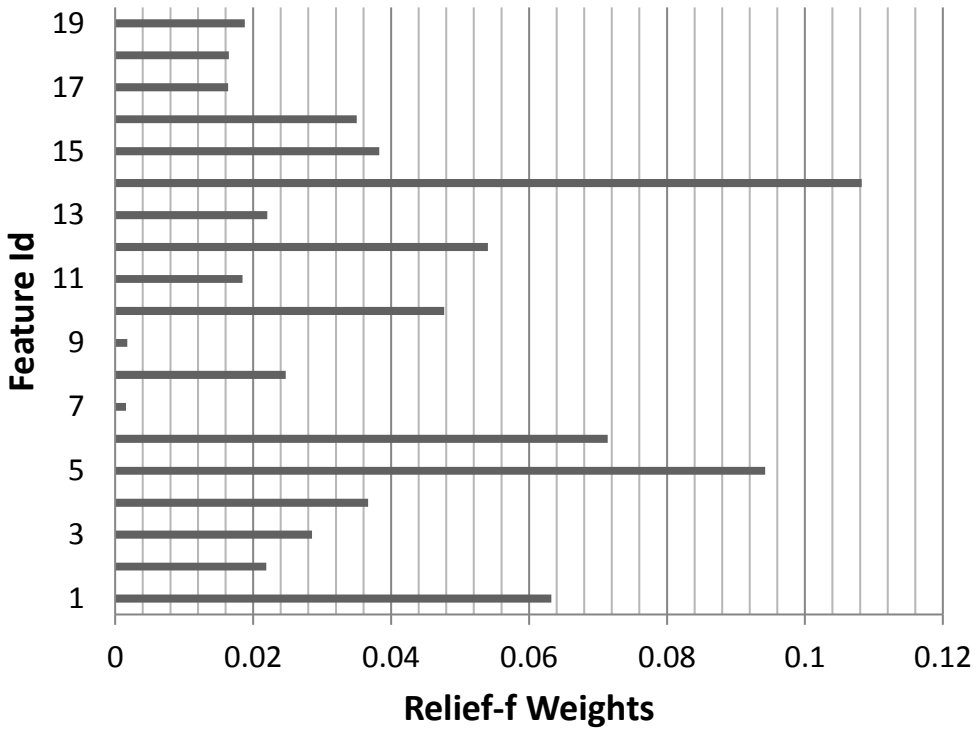

Figure 3.2: Weights assigned to features by Relief-f for the data of LittleRock flop phase

Figure 3.2 shows the weights assigned by the algorithm. After getting weights, the features are sorted in ascending order. Then, at each iteration, we drop the feature with the lowest weight. Finally, we select the feature set with maximum validation accuracy. In Table 3.9 results with this approach is shown. We see that dropping the features suggested by relief-f does not improve performance. Although we present here results for a particular player as an example, we observe this manner of results for the entire data set. Because of this, we did not

Table 3.9: Steps during reduction of feature set guided with relief-f

Feature Set	Validation Accuracy
All Features	88.2%
All Features - {7}	82.0%
All Features - {7, 9}	82.4%
All Features - {7, 9, 17}	82.7%
All Features - {7, 9, 17, 18}	82.7%
All Features - {7, 9, 11, 17, 18}	82.6%
All Features - {7, 9, 11, 17, 18, 19}	82.5%
All Features - {2, 7, 9, 11, 17, 18, 19}	77.6%
All Features - {2, 7, 9, 11, 13, 17, 18, 19}	77.4%
All Features - {2, 7, 8, 9, 11, 13, 17, 18, 19}	76.3%
{1, 4, 5, 6, 10, 12, 14, 15, 16}	76.2%
{1, 4, 5, 6, 10, 12, 14, 15}	76.2%
{1, 5, 6, 10, 12, 14, 15}	75.9%
{1, 5, 6, 10, 12, 14}	75.0%
{1, 5, 6, 12, 14}	70.0%
{1, 5, 6, 14}	69.3%
{5, 6, 14}	68.3%
{5, 14}	52.0%
{14}	49.5%

include relief-f in the final system and go with the complete backward reduction algorithm.

3.7 Learning the Opponent

After the feature selection procedure, we have optimal features for all of the players and phases. Now it is time to apply classification algorithms. The game of poker has a very large game space and we have a limited amount of data. Because of this, we should use classifiers that are compatible with sparse data. We employ three popular learning algorithms suitable to sparse data: Neural Networks, Support Vector Machines and K Nearest Neighbors.

3.7.1 Neural Networks

An Artificial Neural Network (ANN) is a classifier inspired by biological neural networks. Gerald Tesauro had used ANNs to create a backgammon playing agent called TD-Gammon[32]. TD-Gammon achieved a high level of competency by playing with itself and analyzing the results. In the poker domain, Davidson was the first one to use ANNs for opponent modeling [22].

In a standard multilayer feedforward network there are three layers. Figure 3.3 shows an example Neural Network with these there layers. In this example we have three output nodes each corresponding to an action in the poker game, 6 input nodes and 4 hidden layer nodes. For each connection to a node in hidden layer or output layer there is an associated weight w_i. In the feedforward operation, these nodes compute the weighted sum of its inputs as the net activation. The activation function of the jth node with the input vector x is given below:

$$A_j = \sum_{\forall i} x_i w_i \tag{3.2}$$

We see that the activation function only depends on inputs and the corresponding weight. To introduce non-linearity, each node feeds this value to an output function. The most common output function is the sigmoid function. Output of the jth node is given below as:

$$O_j = \frac{1}{1 + e^{A_j}} \tag{3.3}$$

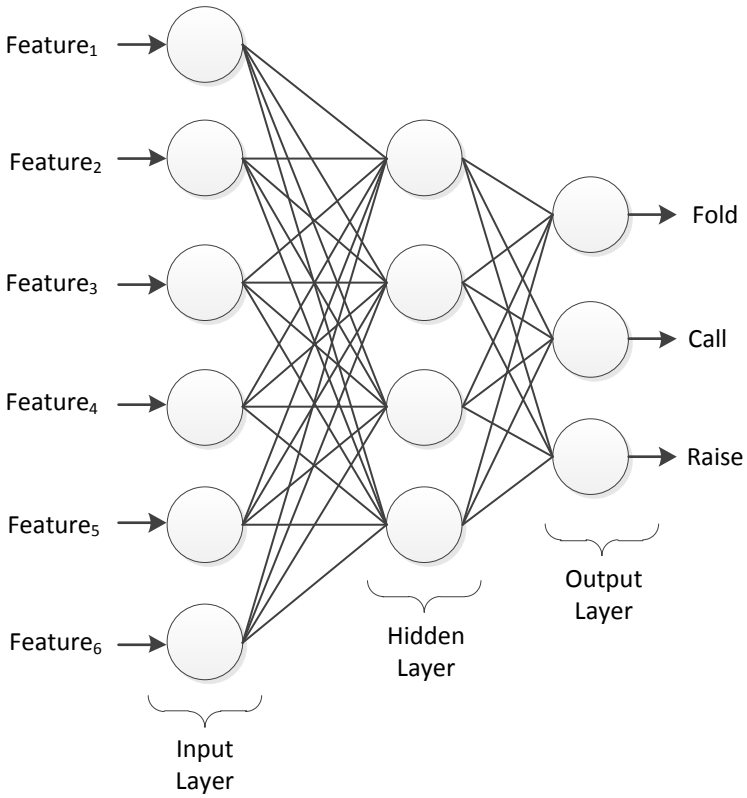

Figure 3.3: ANN example for poker

It has been proven that Neural Networks with sufficient number of hidden units are able to approximate any function [23, p. 286]. In order to approximate a specific function, we should find the "correct" weights for all the connections so that the network generates desired outputs. We do this in the training phase. The most commonly used learning algorithm in the training phase is called *backpropagation*. In the backpropagation algorithm we start with assigning random values to each weight. Then, we iteratively find better weights to minimize our error at the output as we go over the training set. The number of times we go over the training set is called the *epoch number*. The error of the network is defined as:

$$E = \sum_{\forall j} (O_j - d_j)^2 \tag{3.4}$$

where d_j is the desired output. Now we can adjust the weights using the method of *gradient descendent*:

$$\begin{aligned} w_{ji} &\leftarrow w_{ji} + \Delta w_{ji} \\ \text{where} \quad \Delta w_{ji} &= -\eta \frac{\partial E}{\partial w_{ji}} \end{aligned} \tag{3.5}$$

Here η is the learning rate, an important parameter of the Neural Network training. If we update the Δw_{ji} with its value in the previous iteration by factor of α, this is called momentum. Below is the new calculation of the error:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E}{\partial w_{ji}} + \alpha \Delta w_{ji}(n-1) \tag{3.6}$$

Momentum is used to keep the direction of the gradient descent consistent. For the details of the backpropagation learning, please refer to [23, p. 282-350].

3.7.1.1 Experiments

Since we have three possible actions in each decision point, there are 3 output nodes in the network each corresponding to one of fold, call or raise actions. During training, we labeled fold data as $\{1, 0, 0\}$, call as $\{0, 1, 0\}$ and raise as $\{0, 0, 1\}$. So, in the testing phase we expect the correct output node to produce values near 1 and the others near 0. The output node with the maximum value is interpreted as the decision of the network during testing. However, the output values can also be treated as action probabilities after normalization.

Number of hidden nodes, epoch number, learning rate and momentum investigated. 20% of the training data is used for validation in the experiments. We observed that introducing momentum to the Neural Network training process does not improve performance. Figure 3.5 shows this for the case of Hyperborean flop phase. Number of hidden layer nodes has been searched around the number of selected features. We have observed that Neural Network performance is generally stable around this number as shown in Figure 3.4. Figure 3.6 shows the validation accuracies for different number of epochs for the same data with $learningRate = 0.2$. Figure 3.7 shows the effect of varying learning rate for the same data with $epochNumber = 1000$. In general, we have observed that Neural Network performance is not sensitive to the varying parameters for this data.

Table 3.10 to 3.12 shows the detailed results for the player Hyperborean. Please note that, for these results, Hyperborean is modeled and resulting Neural Network is tested with the hand history of Hyperborean. If we use this Neural Network to predict the actions of another player, say Slumbot, prediction accuracy would be lower than the case of predicting the same player. This issue will be addressed at Section 3.8.

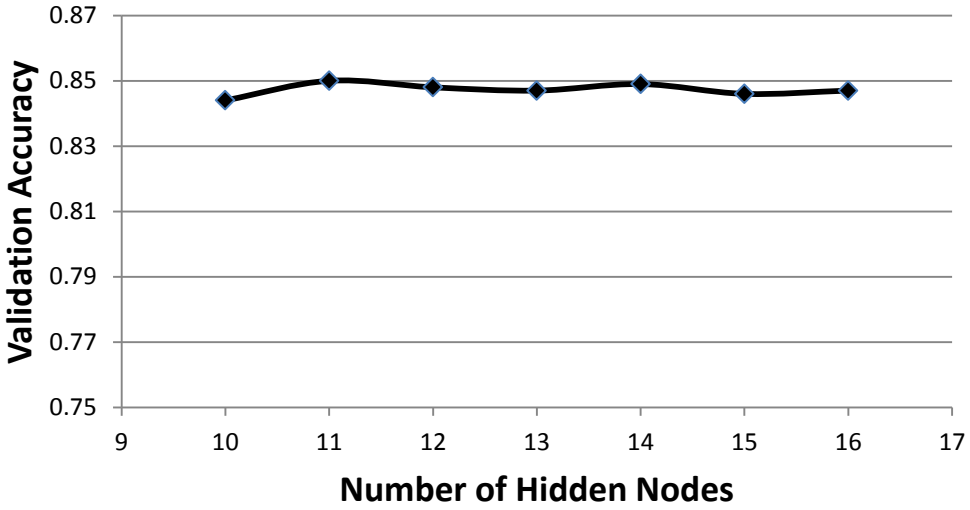

Figure 3.4: Number of hidden nodes vs. Neural Network performance

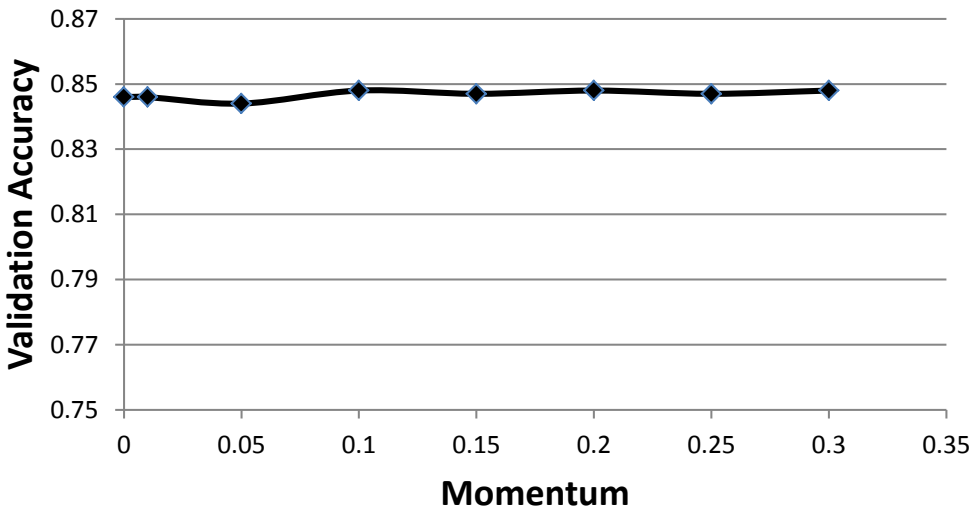

Figure 3.5: Effect of momentum parameter in Neural Network performance

Table 3.10: Train error and prediction accuracy in different phases for Hyperborean flop phase

Phase	Train Error	Test Accuracy
Pre-Flop	9%	87%
Flop	11%	84%
Turn	13%	81%
River	13%	82%
Average	12%	84%

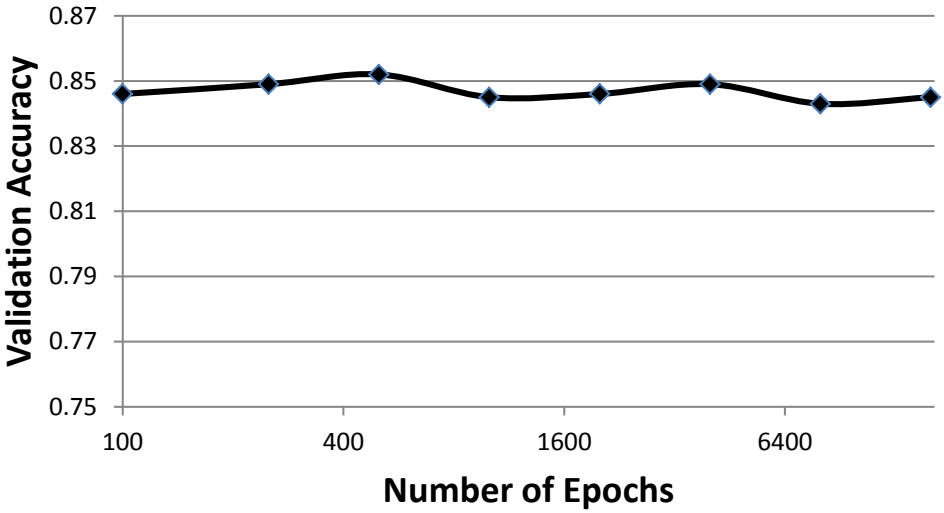

Figure 3.6: Number of epochs vs. Neural Network performance

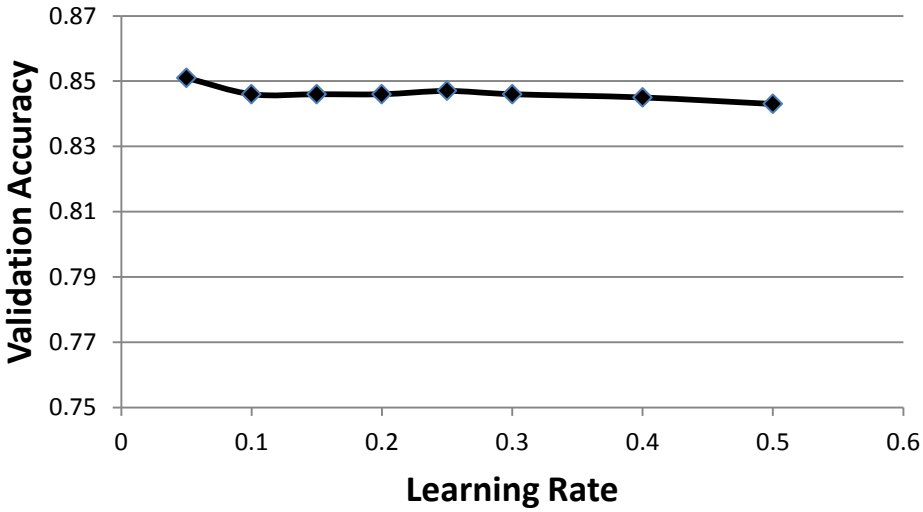

Figure 3.7: Learning rate vs. Neural Network performance

Table 3.11: Confusion matrix for Neural Network testing with Hyperborean flop phase

	Predicted		
	Fold	Call	Raise
Fold	513	80	13
Call	52	2725	436
Raise	12	381	2071

Table 3.12: Precision and recall for Neural Network testing with Hyperborean flop phase

	Fold	Call	Raise
Precision	89%	86%	82%
Recall	85%	85%	84%

Results for all the players can be seen from Table 3.13 with an average of 85%. We observe that pre-flop and flop phases, in which we have more data, have better performance than the others.

As previously noted, feature sets are identical for flop and turn phases, because all the features listed in 3.1 are applicable to both phases. Because of this, it is very easy to investigate the effect of our approach providing different learning machines for both phases. We simply combine the data for these phases, train and test it with a single Neural Network. For the data of Hyperborean, we get an overall accuracy of 81% with this setting, which is slightly less than our original arrangement. The results for other players are also similar.

An interesting pattern stands out in Table 3.13. According to total bankroll results, Calamari is the first and Sartre is the second of the tournament. So, the most successful a bots in the competition turn out to be most predictable by our system.

Table 3.13: Prediction performance for different players with Neural Networks

	Pre-Flop	Flop	Turn	River	Average
Sartre	90%	89%	94%	90%	91%
Hyperborean	87%	84%	81%	82%	84%
Feste	89%	87%	82%	81%	85%
ZBot	79%	76%	79%	82%	79%
Slumbot	88%	80%	79%	77%	81%
LittleRock	87%	87%	81%	81%	84%
2Bot	92%	85%	78%	77%	83%
Calamari	88%	95%	93%	93%	92%
Average	88%	85%	83%	83%	**85%**

3.7.2 Support Vector Machines

Support Vector Machine, or SVM in short, is a very popular pattern recognition algorithm created by Cortes & Vapnik [33]. In the binary classification problem, one class is labeled as 1 and the other class is labeled -1. During the training phase, SVM builds a model to differentiate between two classes. Model is constructed so that a separating hyperplane lying between the classes is determined. There can be an infinite number of hyperplanes satisfying this condition. Thus, we find the one with the maximum margin to the closest data points from each class. These data points are called support vectors. Figure 3.8 illustrates the idea.

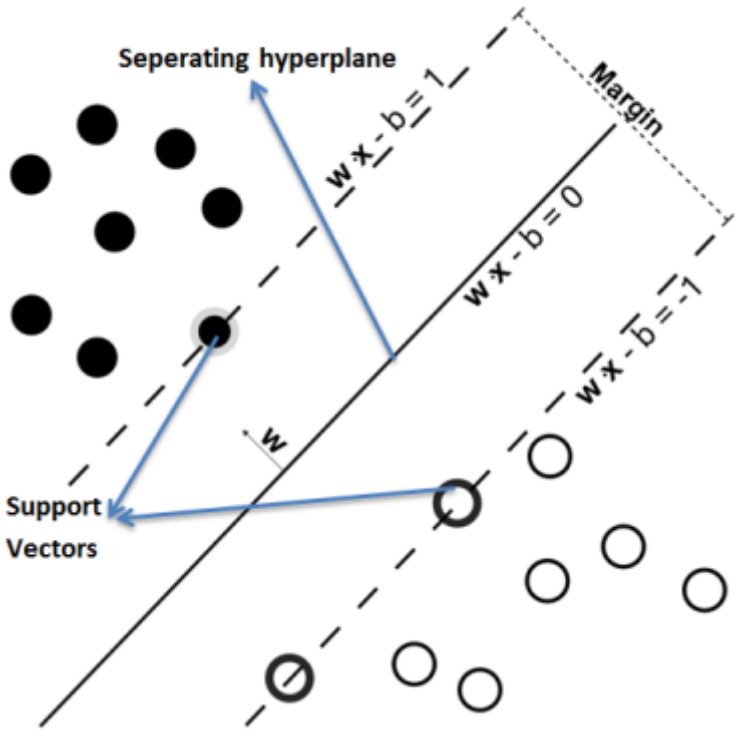

Figure 3.8: SVM classification with separating hyperplane

Given a training data, $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ where x_i is the data point is and y_i is the class label, a separating optimal hyperplane should satisfy:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} w^T w \\ \text{subject to} \quad & w \cdot x_i + b > y_i \end{aligned} \tag{3.7}$$

It is observed that this is a constrained optimization problem. For the solution of this problem, the Lagrangian function is used. For the details of the solution, please see [34]. We provide

the decision function of the SVM below:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i K(x, x_i) + b \tag{3.8}$$

Here, $K(x, x_i)$ term is called the *kernel function*. Linear, polynomial, radial basis function (RBF) and sigmoid are some of the kernel functions often used. Linear kernel is the simple inner product: $K(x, x_i) = x^T x_i$ whereas radial basis function kernel resembles to a Gaussian function: $K(x, x_i) = exp(-\gamma \parallel x - x_i \parallel^2)$.

In many problems, there is no hyperplane separating all of the samples. There may be some outliers, and so some mislabeled samples. Because of this, soft margin method is introduced. In this method, we try to find an optimal hyperplane even if we can not separate all the instances perfectly, by penalizing mislabeled samples by a a c parameter. We measure the contribution of each misclassified sample to the penalty with so called *slack variables* ξ_i. Generally, original data projected to a higher dimension to achieve better separability by a function ϕ. Incorporating these, equation 3.7 becomes:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi_i, \\ \text{subject to} \quad & y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0. \end{aligned} \tag{3.9}$$

Although SVM is initially developed for binary classification, they can be also used for multiclass classification problems. In order to handle k-class classification problem, we can construct k separate SVMs so that each machine performs one-versus-the-rest classification[35].

3.7.2.1 Experiments

In poker context, the hyperplane separating the classes can be thought as the the line determining the actions of a player. For example, a player folds when the situation lies in one side of this line but he calls when in the other side. Of course, it is a large approximation. In poker, a player may not be consistent with his playing. In fact, it is a desirable feature to prevent predictability. Since we observed overlap between the classes, we experimented with cost SVM. We investigated the effect of varying the c parameter with 5-fold cross validation. We see that

results are stable for $c \geq 4$. Figure 3.9 shows the accuracies for different c values in the case of Calamari turn phase. Linear and radial basis function has been used as kernel. Table 3.14 and Table 3.15 shows testing accuracies for different players. We see that rbf kernel is more successful for this data. They are also a little (2%) better than that of Neural Networks.

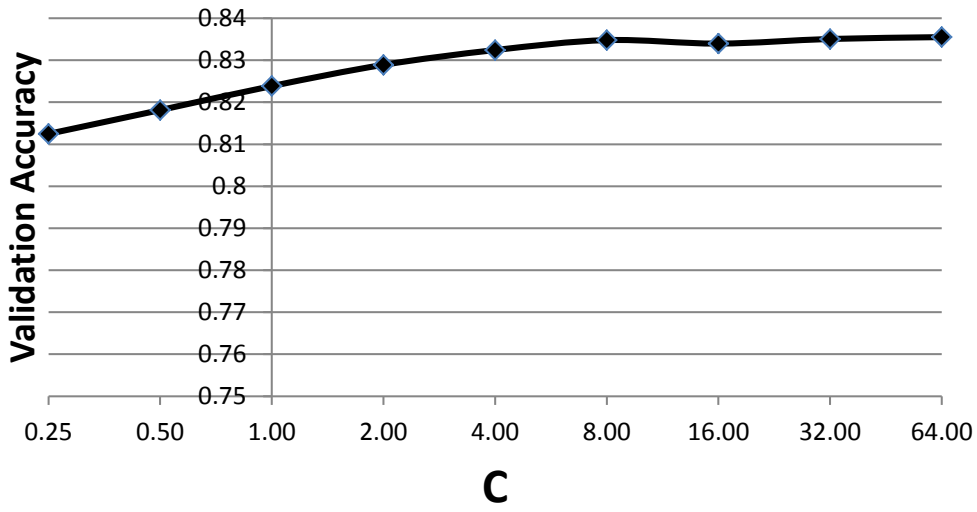

Figure 3.9: Effect of c parameter in cost SVM

Table 3.14: Testing accuracies for different players with linear kernel based SVM

	Pre-Flop	Flop	Turn	River	Average
Sartre	89%	93%	86%	85%	88%
Hyperborean	87%	81%	77%	76%	80%
Feste	90%	84%	75%	78%	82%
ZBot	77%	73%	75%	77%	76%
Slumbot	88%	77%	76%	74%	79%
LittleRock	84%	84%	75%	76%	80%
2Bot	92%	82%	73%	74%	80%
Calamari	87%	95%	84%	85%	88%
Average	87%	84%	78%	78%	**82%**

Table 3.15: Testing accuracies for different players with rbf kernel based SVM

	Pre-Flop	Flop	Turn	River	Average
Sartre	90%	96%	96%	94%	94%
Hyperborean	88%	85%	83%	83%	85%
Feste	90%	87%	82%	84%	86%
ZBot	80%	81%	80%	83%	81%
Slumbot	88%	80%	81%	77%	82%
LittleRock	86%	87%	83%	83%	85%
2Bot	92%	84%	78%	78%	83%
Calamari	89%	97%	96%	93%	94%
Average	88%	87%	85%	84%	**86%**

Same observations with the Neural Networks also apply to the results of the SVM. First two phases (pre-flop and flop) and most successful two players (Calamari and Sartre) have the highest performance.

3.7.3 K-Nearest Neighbors

The K-Nearest Neighbor (KNN) is one of the most popular and basic classification algorithms. It is a type of lazy learning that computations are deferred until classification[36]. KNN classifies objects according to the closest training samples in the feature space. Classification rule is very simple: an object is classified according to voting among its closest k data points. That is, if its close neighbors are generally labeled as a particular class, the test object at hand is also classified as that class. In the illustration of Figure 3.10, $k = 4$ and unknown square object is classified with circle objects. As can be seen, k is the parameter to tune in KNN classification.

Generally, we give more weight to the closer instances in voting[37]. In order to determine which objects are closest, we need a distance metric. Most basic one is the Euclidean metric. Many times, the ranges of indivudial features are not the same. Because of this, we should normalize them so that each of them contributes to the Euclidean distance equally.

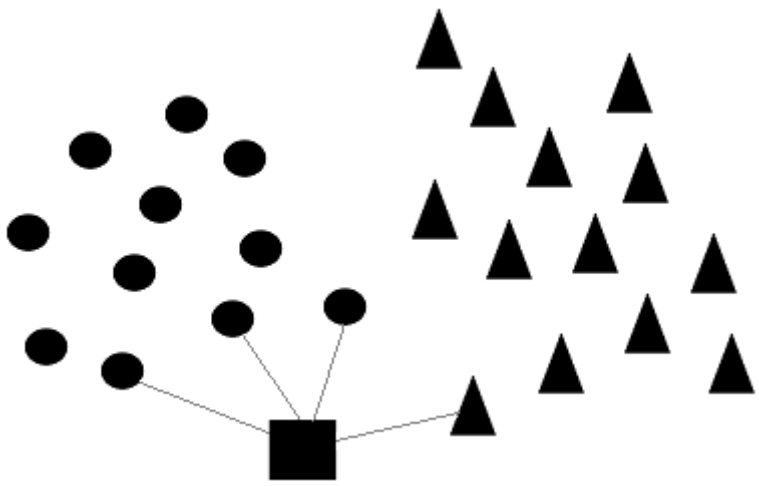

Figure 3.10: KNN classification

3.7.3.1 Experiments

In poker context, applying KNN corresponds to finding the most similar situations to the current situation observed in the past and predicting the same behaviour. Game playing data of the eight players are trained and tested with KNN. Before applying KNN algorithm, data is normalized. If the range of a feature is significantly larger than the others, this feature can dominate the distance calculation. Because of this, all features are mapped to 0.0 - 1.0 range so that each of them contributes equally to the Euclidean distance metric.

We investigated the effect of varying the k parameter with 10-fold cross validation. We see that results are generally stable after $k = 7$. Figure 3.11 shows the case for the data of the Hyperborean flop phase. Table 3.16 shows the confusion matrix and Table 3.17 shows precision and recall performance for the same data. We see that precision and recall rates are similar among the classes. Table 3.18 shows the test results for all the players. KNN performance seem to be a little better than that of Neural Networks and SVM. KNN results are not an exception to the previous observations that first two phases (pre-flop and flop) and most successful two players (Calamari and Sartre) has the highest performance.

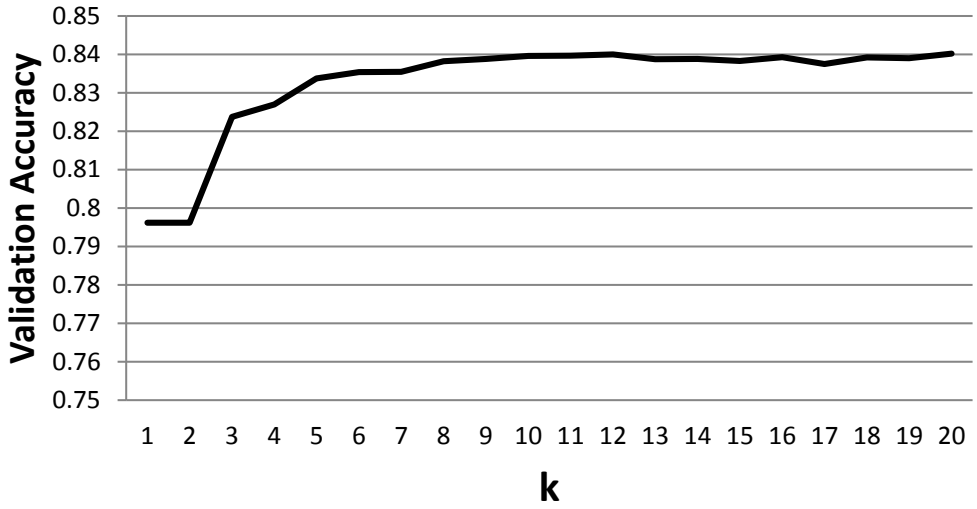

Figure 3.11: Effect of k parameter on KNN performance

Table 3.16: Confusion matrix for KNN testing for Hyperborean flop phase

	Predicted		
	Fold	Call	Raise
Fold	548	52	6
Call	78	2739	396
Raise	24	435	2005

Table 3.17: Precision and recall rates for KNN testing for Hyperborean flop phase

	Fold	Call	Raise
Precision	84%	85%	83%
Recall	90%	85%	81%

Table 3.18: Testing accuracies for different player and phases with KNN

	Pre-Flop	Flop	Turn	River	Average
Sartre	100%	97%	96%	95%	97%
Hyperborean	96%	84%	82%	83%	86%
Feste	96%	89%	85%	86%	89%
ZBot	93%	80%	79%	82%	84%
Slumbot	95%	79%	78%	76%	82%
LittleRock	98%	89%	83%	86%	89%
2Bot	95%	86%	78%	78%	84%
Calamari	99%	97%	94%	94%	96%
Average	97%	88%	84%	85%	**88%**

3.8 Learning Different Styles

Until now, we trained and tested the learning system with the same player. What would happen if we tested the suggested learning system with a different scenario? In fact, this is a very common situation with human players. They observe some players and later they can benefit from their previous observation with other players. Table 3.19 shows what happens when we test the Neural Network trained with particular players tested with others. In Table 3.19, left column shows the training set and upper row shows the test set respectively.

Table 3.19: Testing with different players.

	Testing Against			
	Calamari	Sartre	LittleRock	Slumbot
Calamari	93%	83%	81%	73%
Sartre	81%	90%	75%	71%

We see that performance generally drops when we use a particular model to predict another. For example, when we use the Neural Network trained with Calamari to predict Sartre, prediction accuracy falls to 83% from 93%. This can be a problem for an artificial agent when it confronts a particular opponent for the first time. Since the learner trained with Calamari best performs when testing against Calamari, we call this learner Calamari-Expert.

3.8.1 Ensemble Learning

We look at the literature of ensemble learning to solve the problem of generalizing the opponent modeler across players. Ensemble learning refers to the idea of combining multiple classifiers into a more powerful system with a better prediction performance than the individual classifiers[38]. Alpaydin covers seven methods of ensemble learning: voting, error-correcting output codes, bagging, boosting, mixtures of experts, stacked generalization and cascading in his book[39, p. 419-445]. In simple *voting* scheme, all learners are given an equal vote. In *error-correcting output codes*[40], multiclass classification task is divided into simpler tasks which are handled by the individual learners. *Bagging* is a voting scheme in which each base learner is made different by exposing it to a different subclass of the data set. In *boosting*[41], each base learner is trained on the subset of the data where the previous learners are erroneous, to make them complementary. In stacked generalization[42], we have first level of classifiers and a meta-classifier trained on the outputs of these first level classifiers. *Mixtures of experts* is similar, but in the second-level, we have a gating network, which is generally a Neural Network, and a combining system such as a weighted majority[43]. Finally, in cascading[44], multiple classifiers with increasing complexity is combined sequentially that the next one is only used if the previous classifiers are not confident.

In our case, we already have several experts each trained on a particular player. Because of this, we see that the ensemble scheme of stacked generalization best fits to our purposes. To overcome the problem of generalizing an opponent model to other players, we set up the system which is illustrated at Figure 3.12. We select the expert modelers for the top performers of the annual poker competition, namely: Calamari-Expert, Sartre-Expert, Hyperborean-Expert and LittleRock-Expert. In this system, the output of each expert learner is fed to the meta classifier as a feature. Each Neural Network has three output nodes, so we concatenate them to make a 12-sized feature vector and feed this to the meta-classifier, which is also a Neural Network.

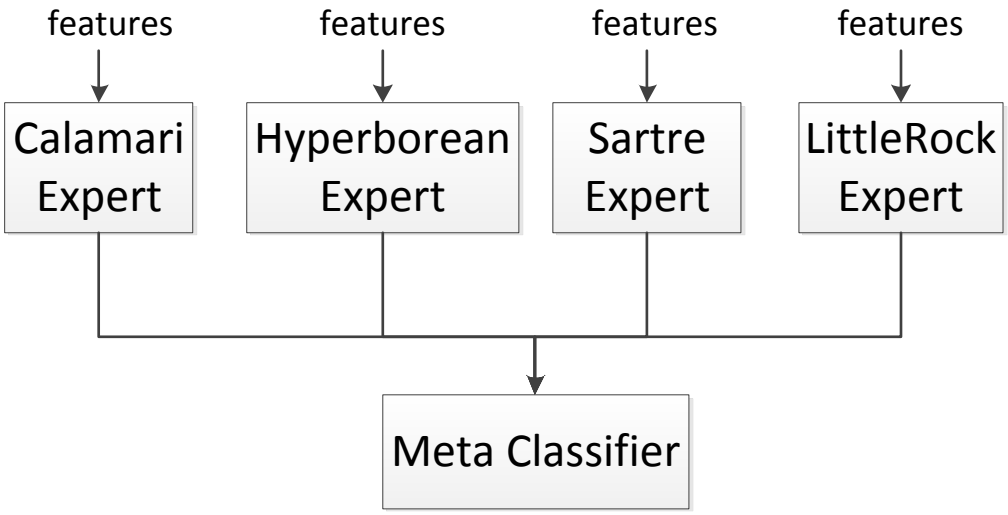

Figure 3.12: Ensemble of experts

For this experiment, we split our data into 3 parts. First part is used to train experts. Each expert is trained with the data of a particular player. Second part is used to train the meta classifier. That is: experts see this data newly. Data of four different players are fed into each expert and their predictions are obtained. After that, their outcomes are concatenated and fed into the meta-classifier as input. Finally, third part is used to test the entire system. Table 3.20 shows the results for this system.

In Table 3.20, we see that when testing against a certain player, the expert of that player gives the best performance. This can be seen from the first four diagonal entries. Ensemble usually comes the second after the expert when predicting a particular player. We see that ensemble has managed to model different players with fairly stable performance. The average performance of the ensemble is greater than the individual experts. In Table 3.20, both expert learners and the ensemble see Slumbot and 2Bot for the first time. We see that ensemble is able to cope with unknown opponents better than the experts. Thus, in the case of encountering with an opponent first time, ensemble of previous models should be the choice for the opponent modeler.

Table 3.20: Ensemble accuracy compared with experts

	Testing Against						
	Hyperborean	Calamari	Sartre	LittleRock	Slumbot	2Bot	Avg.
Hyperborean	81%	90%	80%	79%	71%	75%	79%
Calamari	80%	93%	83%	81%	73%	75%	81%
Sartre	73%	81%	90%	75%	71%	71%	77%
LittleRock	80%	90%	86%	82%	75%	75%	81%
Ensemble	82%	92%	89%	82%	76%	77%	**83%**

3.9 Chapter Conclusion

Through this chapter, we tried to build an opponent modeling system. Since there are not many studies on this specific area, ours can be considered as the first comprehensive study, analyzing different learners, providing feature and parameter selection details. Our initial data analysis suggested that there are redundant information in the full feature set. We reduced the feature set employing backward reduction algorithm.

There are two previous studies using Neural Networks for opponent modeling in poker. First one is presented in the master thesis of Davidson[22] in which he uses a single Neural Network for all phases, reporting 81% prediction accuracy. Another similar work by McCurley[19], again using a single network for all phases, reports performances around %70. We obtained an average of 85% accuracy with Neural Networks using four different learners for each phase. We boosted the performance with the feature and parameter selection process.

We are not aware of any studies involving SVM or KNN for opponent modeling problem in poker. However, we see that they are better than Neural Networks with SVM giving an average 86% accuracy and KNN giving a 88% accuracy.

This is also the first study applying stacked generalization for modeling different and unknown opponents. Ensemble system proved to be improving performance in these cases.

CHAPTER 4

HANDLING THE COMPUTATIONAL COMPLEXITY

As we previously seen, Hand Potential is a very important feature for the success of the learning system. As explained in Section 2.2, we need above a million hand rankings for the hand potential calculation. Calculation of this metric depends on evaluating hands and comparing them. So, here the parameter to optimize is number of hand evaluations calculated per second.

As previously noted, hand potential calculation is essentially a million times hand evaluation for each possible future situation and taking average of them. Next section describes our first attempts to port poker hand evaluations to GPU. CUDA, which is a general purpose GPU programming solution provided by NVIDIA, is used as the platform for our parallelization efforts. CUDA is very suitable for data-parallel problems in which each piece of data is subject to the same calculations. The threads performing these calculations, each of which runs on a core of the GPU, is called *kernel*.

Hand evaluation is basically assigning each poker hand a number so that if that hand is superior to another according to rules of the game, that hand has a greater value. If they are tied, both hands get the same value. In Poker, each hand belongs to one of 9 major hand classes. They are in decreasing value: Straight Flush, Four of a Kind, Full Houses, Flush, Straight, Three of a Kind, Two Pair, One Pair, and High Card. If two hands belong to the same major class, individual values of cards came into play to determine the winner.

4.1 Hand Evaluation Using Prime Numbers

First, we try to port Cactus Kev's Poker Hand Evaluator[45] to GPU since it's simple and efficient. Cactus Kev tried to find a way to quickly transform each of the two and a half million unique five-card poker hands into its rank value. For example, if the sequence of cards Kd Qs Jc Th 9s generates the value 1601; any different order, for example Th Kd Qs Jc 9s should also generate the same value. Considering this, he came up with the idea of prime numbers. He assigned each card a prime number so that when you multiply the prime values of the each card in your hand; you get a unique product, regardless of the order of the five cards. His card representation is in the order of 27 bits storing the prime number, rank and suit of the card. Each card is mapped to a 32 bit integer and each hand is represented by 5 integers. You can find the details of the algorithm, in which he uses lookup tables, bitwise operations and binary search, in his web page[45].

There are $C(52, 5) = 2598960$ five card hand combinations possible in a 52 car deck. Evaluating all of these are used as a benchmark. In the GPU implementation, we first generate all possible 5-card hands and store it in a big vector. Then, it is passed to both CPU and GPU implementations. We also take host to device and device to host copying times into consideration. In the CUDA implementation, each hand is assigned to a different thread. Basically, we unroll the loops in the CPU algorithm and evaluate all the hands in parallel. Each thread reads its input from global memory and writes its output to global memory.

Cactus Kev's own CPU implementation took 288ms in a Phenom 2 AMD machine using a single core to evaluate all the hands. Our first GPU implementation took only 51 seconds with a GTX 460 card, which is a speed up about 5.65. Then, we analyzed the program with the profiler shipped with CUDA toolkit. We see that GPU occupancy is around 67%. This indicates that we were not able to utilize all the streaming multiprocessors, which are the blocks of computing cores in the GPU. In order to deal with it, we decreased block size, which indicates number of threads assigned to a single streaming multiprocessor, to 256 from 1024. This fixed the occupancy to 100%.

In the original algorithm, binary search is used to map multiplication of prime numbers to their hand rank values. Binary search introduces a lot of branching to the kernel, and this decreases the performance. Paul Senzee replaces the binary search in the original algorithm

with a perfect hash function which works in $O(1)$ time[46]. This improved the performance of the kernel significantly. Finally, we used constant memory of the device for the lookup tables which increased the speed-up. Table 4.2 shows timings, speed up and GPU cycles for kernel as measured by CUDA profiler of the four different implementations.

Table 4.1: Timings for different implementations using prime numbers

	evals / ms	speedup	Kernel cycles
CPU	9024		
GPU - simple	50960	5,65	9794
GPU - occupancy fixed	53040	5,88	7640
GPU - hash table	56499	6,26	3495
GPU - const tables	59067	6,55	3052

We also investigated the speed of the GPU implementation as the input vector size changed. Figure 4.1 is the data size vs. speed plot. This graph shows that in order to fully utilize the GPU, we must supply large chunks of data.

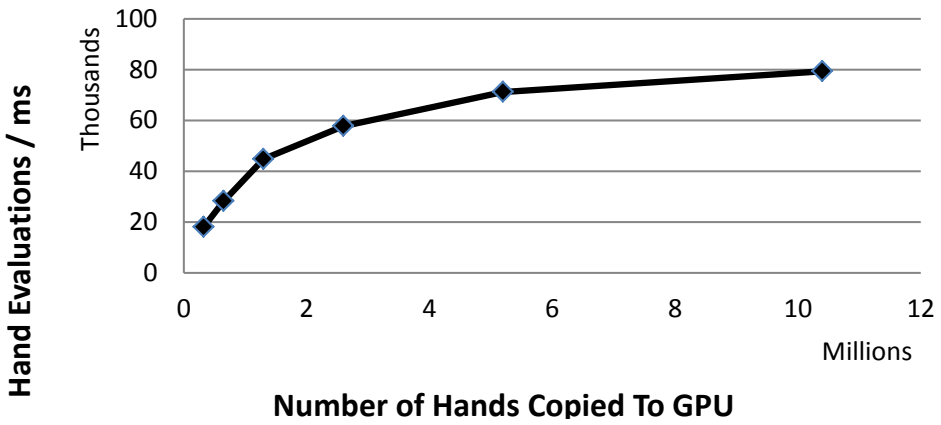

Figure 4.1: Hand evaluation speed for different sizes using prime numbers

All these optimizations reduced the kernel cycles but still speed-up is not much. This is because of memory copy operations. CUDA profiler shows that when kernel took 3052 cycles, copying to device from host took 19086 cycles and copying from device to host took 2010 cycles. Hence, only about %12 of the total time is devoted to actual work. Considering this, we looked for a more compact representation of hands. In next section, we will describe a three times efficient hand representation and related results.

4.2 Hand Evaluation Using Bitwise Representation

Poker-Eval is another library for hand evaluation which uses a more efficient hand representation. We ported Keith Rule's C# implementation[47] of the library to CUDA. In this library, each hand is represented with 52 bits. Each bit corresponds to a different card of the deck. If the hand includes a card, corresponding bit is turned on. Because of this compact representation, hand evaluation is more complex than the previous one: requires more lookup tables so that constant memory of the device was not enough, more bitwise operations and more branching. Therefore, an increase in the kernel cycles is expected but we expect that reduction in the memory transfer time covers up for that.

Table 4.2 shows timings for this algorithm. In this implementation, kernel took 1694 cycles to finish. It's 25% of the total time. We have decreased the memory transfers and increased the ratio of actual work. This increased our speed-up. To further decrease the memory transfer time, pinned memory is used instead of pageable memory. By declaring the memory pinned, we force the operating system to hold this data in physical RAM, which makes memory transfers faster.

Table 4.2: Timings for different implementations using bitwise representation

	evals / ms	speedup	HtoD cycle	DtoH cycle
CPU	13720			
GPU-pageable	125905	9,18	3024	2040
GPU-pinned	194580	14,19	1554	791

We also investigated the speed of the GPU implementation as the input vector size changed. Figure 4.2 is the data size vs. speed plot. This graph shows that in order to fully utilize the GPU, we must supply large chunks of data.

4.3 Chapter Conclusion

Since hand evaluation calculations are independent, it is easy to port them to GPU. However, this also make it necessary to transfer data to and from GPU. We have seen that memory transfers are the major overhead of GPU usage. Thus, in order to get a reasonable performance,

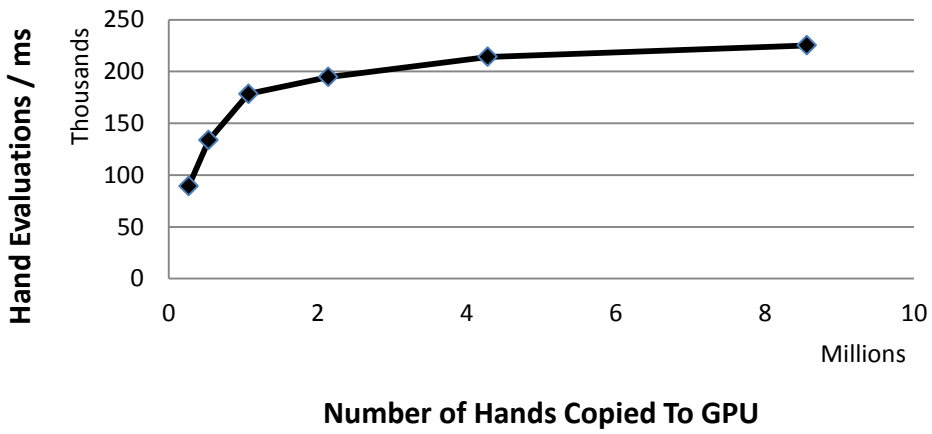

Figure 4.2: Hand evaluation speed for different sizes using bitwise representation

we should minimize the time spent for data transfers between host machine and the GPU. We observe that pinned memory reduces memory transfer times. We see that in order to fully utilize GPU, branching in the calculations should be minimized.

After making some improvements in the algorithm and calculation, we are able to get a speed-up of 14 times over CPU implementation. With this rate of reduction in calculation time, exact calculations of the hand evaluation metrics introduced in Section 2.2 becomes feasible.

CHAPTER 5

CONCLUSION

Opponent modeling is critical for the success of exploitive artificial agents in games of imperfect information. It is found to be an essential ingredient in adaptive player architectures. However, research up to now on this domain has been limited. This work can be considered as a preliminary study regarding the application of machine learning techniques to the problem.

In this study, we suggest a new opponent modeling system for predicting the behaviour of opponent players by using machine learning techniques. We observe that machine learning techniques are able to capture the essence of a poker player's style. This task is achieved by careful selection of features and parameters. For this purpose, we first define a feature space for representing the situations in the game. After analyzing the feature space by examining means, variances and cross-correlations of the features, we employed several feature selection algorithms to reach a final feature set. We proposed to employ different classifiers for each phase of the game and propagate information between phases by selecting appropriate features. Parameters of the classifiers are optimized with respect to validation performance. This thesis is one of the first studies experimenting with SVM and KNN in this domain. We have provided a comparative analysis between them and Neural Networks. KNN is found to have the best performance in this domain but results are not wildly different. Neural Networks give an average accuracy of 85%, SVM gives 86% and KNN gives 88%. Therefore, dependency of the system to a particular classification algorithm is rather low. We also showed that state-of-the-art artificial players are highly predictable. Our expert classifiers are able to predict their actions up to 95% accuracy.

It is marked that each player has a different playing style so that observing a player may not be sufficient to predict another player's behavior. In order to tackle with this, an ensemble

learning scheme has been proposed and shown to be successful to generalize to different players. Ensemble system has the advantage of combining different experts and different styles, giving a more stable predictor. This can be very beneficial when the artificial player confronts an opponent for the first time. In this case, the ensemble of previous models can provide a baseline prediction.

The chapter about computational complexity proves that GPU programming may be a useful tool for poker simulation. GPU programming is different from CPU programming and to achieve performance gains, programming style should be consistent with the underlying hardware.

5.1 Future Directions

We have come up with an opponent modeling system that is able to model most successful artificial poker players created up to date with a good accuracy. As previous research shows, opponent modeling is a critical component for decision systems in artificial players in games of imperfect information. Thus, our system should be integrated into a complete artificial player architecture and its contribution should be observed.

The system needs previous game histories of players in order to generate models of them. This can be considered as a weakness if the artificial player did not have any previous data. To overcome this, our system can be extended with capability of online learning.

We have applied three popular classification algorithms, namely Neural Networks, SVM and KNN. It may be beneficial to try the system with other algorithms and compare the results.

REFERENCES

[1] A. M. Turing. Digital computers applied to games. In *Faster than Thought*, pages 286–310. Pitman, London, 1953.

[2] T.A. Marsland and J. Schaefer. *Computers, chess, and cognition*. Springer-Verlag, 1990.

[3] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):210–229, 1959.

[4] D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Poker as a testbed for ai research. *Advances in Artificial Intelligence*, pages 228–238, 1998.

[5] M.B. Johanson. Robust strategies and counter-strategies: Building a champion level computer poker player. In *Masters Abstracts International*, volume 46, 2007.

[6] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[7] E.A. Poe. *The purloined letter*. University of Virginia Library, 1940.

[8] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge Univ Pr, 2009.

[9] J. Rubin and I. Watson. Computer poker: A review. *Artificial Intelligence*, 175(5-6):958–987, 2011.

[10] AAAI Poker Competition. `http://www.aaai.org/Conferences/AAAI/2012/aaai12poker.php`. Last visited on 20 August 2012.

[11] Annual Poker Competition. `http://www.computerpokercompetition.org/`. Last visited on 20 August 2012.

[12] R.I. Follek. *SoarBot: A rule-based system for playing poker*. Master's thesis, Pace University, 2003.

[13] J.E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

[14] D. Billings, D. Papp, L. Peña, J. Schaeffer, and D. Szafron. Using selective-sampling simulations in poker. In *American Association of Artificial Intelligence Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, pages 13–18, 1999.

[15] D. Billings. *Algorithms and assessment in computer poker*. PhD thesis, University of Alberta, 2006.

[16] D. Felix and L. Reis. An experimental approach to online opponent modeling in texas hold'em poker. *Advances in Artificial Intelligence-SBIA 2008*, pages 83–92, 2008.

[17] 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 *International Joint Conference on Artificial Intelligence*, volume 18, pages 661–668, 2003.

[18] N.A. Risk and D. Szafron. Using counterfactual regret minimization to create competitive multiplayer poker agents. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1*, pages 159–166. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[19] P. McCurley. *An Artificial Intelligence Agent for Texas Hold'em Poker*. Undergraduate dissertation, University of Newcastle Upon Tyne, 2009.

[20] J. Fürnkranz. Machine learning in games: A survey. *Machines that Learn to Play Games*, pages 11–59, 2001.

[21] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron. Game-tree search with adaptation in stochastic imperfect-information games. *Computers and Games*, pages 21–34, 2006.

[22] A. Davidson, D. Billings, J. Schaeffer, and D. Szafron. Improved opponent modeling in poker. In *International Conference on Artificial Intelligence, ICAI'00*, pages 1467–1473, 2000.

[23] H. Duda, P. Hart, and D.G. Stork. *Pattern Classification*. John Wiley & Sons, 2001.

[24] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[25] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[26] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[27] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145, 1995.

[28] C.W. Hsu, C.C. Chang, and C.J. Lin. *A Practical Guide to Support Vector Classification*, 2003. Techincal Report, Department of Computer Science, National Taiwan University.

[29] H. Liu and L. Yu. Toward integrating feature selection algorithms for classification and clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 17(4):491–502, 2005.

[30] I. Kononenko. Estimating attributes: analysis and extensions of relief. In *Machine Learning: ECML-94*, pages 171–182. Springer, 1994.

[31] K. Kira and L.A. Rendell. A practical approach to feature selection. In *Proceedings of the ninth international workshop on Machine learning*, pages 249–256. Morgan Kaufmann Publishers Inc., 1992.

[32] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1):181–199, 2002.

[33] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[34] J.P. Vert. *Introduction to support vector machines and applications to computational biology*, 2001. Seminar Report, Kyoto University, Japan.

[35] V. Vapnik. *Statistical learning theory*. Wiley, New York, 1998.

[36] D. Wettschereck, D.W. Aha, and T. Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11(1):273–314, 1997.

[37] M. Awad, B. Thuraisingham, and L. Khan. *Design and implementation of data mining tools*. Auerbach Publications, 2009.

[38] L. Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1):1–39, 2010.

[39] E. Alpaydin. *Introduction to machine learning*. The MIT Press, 2010.

[40] T.G. Dietterich and G. Bakiri. Error-correcting output codes: A general method for improving multiclass inductive learning programs. In *Proceedings of AAAI-91*, pages 572–577, 1991.

[41] R.E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[42] D.H. Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[43] R. Polikar. Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45, 2006.

[44] C. Kaynak and E. Alpaydin. Multistage cascading of multiple classifiers: One man's noise is another man's data. In *Proceedings of the 17th International Conference on Machine Learning (ICML-2000)*, pages 455–462, 2000.

[45] C. Kev. Poker hand evaluator library. `http://www.suffecool.net/poker/evaluator.html`. Last visited on 20 August 2012.

[46] P. Senzee. Some perfect hash. `http://www.paulsenzee.com/2006/06/some-perfect-hash.html`. Last visited on 20 August 2012.

[47] K. Rule. C-Sharp port of the poker-eval library. `http://www.codeproject.com/KB/game/MoreTexasHoldemAnalysis1.aspx`. Last visited on 20 August 2012.