Applying first order logic decision tree induction to opponent modelling in No-limit Texas Hold'em Poker

R.J. Prinse
ICA-3019942
r.j.prinse@students.uu.nl

Supervised by:
dr. ir. J.M. Broersen
j.m.broersen@uu.nl

Department of Information and Computing Sciences
Faculty of Science, Utrecht University

June 7, 2012

Abstract

In this thesis methods will be discussed to attempt to improve the decision making for agents involved in playing Texas Hold'em poker. Poker is a card game which features uncertainty, hidden information, deception and an environment in which multiple agents compete with each other to win the game. Opponent modelling will play a key role in improving the decision making of the agent(s).

The opponent models will help the agents to adapt more quickly to the playing style of their opponents and help them to make a better prediction of their next actions. In order for the agents to cope better with changes in playing style or the environment, it is important that the models are dynamic and keep evolving during the game.

Tilde, a first-order logic decision tree induction algorithm, will be used to construct models from a dataset. In turn, Monte-Carlo tree search will yield the action with the highest expected value after simulation. The opponent models will guide the simulation.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

First of all, I would like to thank my thesis supervisor dr. ir. Jan M. Broersen. His enthousiasm, friendliness, open attitude, support and encouragement made me feel motivated to write and finish my thesis, which hasn't always been that easy. Thank you for all your help Jan!

Secondly, but not any less important, I would like to thank both of my parents Antonie and Jack and my sister Stéphanie, for without them, I wouldn't have been where I am today! Thank you for your continued support, motivation and never ending love.

Lastly, I want to thank my girlfriend Ana. She is and always has been there to help me, support me and keeping me motivated. Thank you for all your love and everything you do for me Ana, because you help me to be the person I want to be!

"Choose the right opponents. If you don't see a sucker at the table, you're it."

Amarillo Slim, 2005, *"Play Poker to Win"*

1 Introduction

Artificial intelligent players have been able to play games such as chess or checkers for quite some time now. They have been able to do so quite competitively too. That is, on a high level of play. The main reason why they are able to play those games so well is due to the fact that they provide *perfect* information. In other words, all players have access to all information at all times during the course of the game. This is in complete contrast with games such as poker. Poker is a game of *imperfect* information i.e. usually your cards are hidden (and should be) from all the other players. This should hold for all the other players as well, obviously.

The fact that not all information is observable at any time during the course of the game, makes it a lot harder for AI players to play this game. It is really hard to make well founded decisions without knowing everything about the current state of the game. The more information that is hidden, the more complex it will be to calculate a move or even solve the game. Thats because of the fact that a lot more alternatives have to be considered during the calculations. Up to the point of having to consider so much alternatives, that the computations simply become intractable. Therefore it is imperative that smarter and quicker methods are being devised. It should be obvious by now that cutting on the possible number of alternatives as early as possible is one way to improve upon this.

1.1 Artificial intelligence

Every human being is unique. Even more so, when you look at the way people think, reason and act. Artificial intelligence[36] is the field of science which aims to *simulate* or *recreate* those aspects.

Notice the very subtle difference between simulate or recreate here. The former implies that mechanical or software entities or will never spring a form of self-awareness and/or intelligence. The latter of course does. This very distinction spawned two main branches[38] in A.I., namely:

- Strong A.I.; at some point in the future we will be able to create a self-aware entity with an intelligence that matches or exceeds that of humans.
- Weak A.I.; we will only be able to approximate or simulate intelligence, in other words, we cannot match or exceed the capabilities of the human intelligence.

Some subfields of A.I. zoom in on one of those aforementioned aspects of human intelligence. But in general A.I. could be abstractly viewed as a *black-box* function, that takes a certain *input* and produces a certain *output*. Besides those two, it could also possibly have 1 or more *side-effects* on the *environment* it exists in.

The term black-box emphasises the fact that an *external* observer doesn't know how the A.I. function is implemented. It could even be a very simple and trivial implementation, that tricks the observer into believing the entity is actually showing self-awareness. This of course creates more (philosophical) questions, as to what intelligence and/or self-awareness actually means.

In order to actually implement that function, A.I. researchers have over time created a great array of tools[36]. Some of those tools are:

- Math;
- Logics (temporal, modal, first-order, propositional);
- Statistics and probability;
- Simulation (of natural processes);
- Algorithms and datastructures.

From all this follows that A.I. is very tightly connected with other sciences such as math, computer science, philosophy, psychology, and many others.

In a society where a lot of (automated) processes are becoming more demanding and complex and where we, us (simple) humans, just cannot keep track of everything anymore, A.I. becomes more and more important. Not the mention the entertainment industry, the games industry, etc.

1.2 Games

Games, in all its current forms, are a perfect test environment for A.I. The interactions between all the players and the game often are really complex.

Decision making, prediction, simulation, reasoning are one of the key aspects needed in playing a game (well). The major drawback here is that all of those things are far from trivial to implement. And most of those are also heavily dependent on added domain knowledge. This necessitates that almost every game needs to be *solved* in a different way.

Games often have a clear definition of the rules and the winconditions. This makes them perfectly suitable to test the performance of an A.I. player, against another human player or maybe another A.I. player.

Games can be classified along a few criteria[46]:

- cooperative vs. non-cooperative; a game is cooperative if it allows for the players to work together, committing to the same shared goal,
- symmetric vs. asymmetric; a game is symmetric if the identity of a player can be changed without changing the *payoff* to the strategies,
- zero-sum vs. non-zero-sum; a game is zero-sum if the gain of a certain player is equal to the loss of the rest (i.e. -1 + 1 = 0),
- simultaneous vs. sequential; games are simultaneous if all players make their move simultaneously (i.e. at the exact same moment in time),
- perfect information vs. imperfect information; a game of perfect information is a game where all actions by all players can be observed,
- discrete vs. continuous; in discrete games, most things are finite, whereas in continuous games players can pick from an infinitively large strategy set,
- deterministic vs. stochastic; deterministic games are games where each action a player takes, will always have the same effect, whereas stochastic games incorporate outcomes dependent on chance.

1.3 Poker

1.3.1 Introduction

Poker[47] is a family of card games that is played between 2 to 10 players. Playing the game dates back to the early 1900s. Each player gets dealt a certain number of cards, some or all of them being hidden to the other players.

During multiple betting rounds, each player can bet a certain amount of *chips*. During these betting rounds a shared card will also be revealed to all players. At the end of all betting rounds, the winner is usually determined by the player who still is in the game and also has the "best" combination of his own cards and the shared *community* cards.

Each poker variant has its own set of rules governing the structure of betting. Also, the definition of "best" combination of cards varies among these different poker game variants. The most popular poker variant being *Texas Hold'em*.

Following now is a brief and probably incomplete introduction to the common structure of a poker game. In Texas Hold'em poker, there are 4 betting rounds. Namely; *Pre-flop*, *Flop*, *Turn* and *River*. Each round usually starts with one or more shared cards getting revealed after which a certain player starts by making a predetermined forced bet (called the *small* or *big-blind*). After that, the action continues clock-wise, with every player having 3 options:

call: match the amount bet by the previous player;

raise: match and raise the amount bet by the previous player.

fold: throw your cards away, meaning you're out of the current game.

After all the active players have the same amount of chips in the *pot*, the current betting round ends. After all betting rounds have finished, the *showdown* phase starts. Every active player folds, or shows his cards. The winner is the person who has the best combination of cards, and wins all the chips currently in the pot. In case of a tie, the pot is split equally.

1.3.2 Relevance to A.I.

Where biology and physics can draw emperical knowledge directly from nature herself, computer science draws from algorithms, datastructures and their implementations.

Poker lends itself so well to being a proper test environment for computer science but most importantly for artificial intelligence. This is because of the complex nature of the game, the intricate interactions between players, and the inherent uncertainty of the cards.

All of this means that older and newer algorithms, learning and search techniques need to be combined and improved upon, to discover new and exciting methods to produce better poker playing agents during each such iteration.

Human poker playing experts and general playing strategies are also a big inspiration to keep improving the level of performance of the currently, most well known and best performing *pokerbots*.

1.4 Structure

This thesis is structured as follows: in section 2 on page 10, the background research on which this thesis is based is introduced and discussed. Section 3 on page 20 discusses the approach to the research presented in this thesis. The underlying concepts of this research are dicussed in section 4 on page 24. All the algorithms, implementation details and the formal framework are presented in section 5 on page 26. The results of this thesis are presented in section 6 on page 46. Finally, the conclusion and the future work discussion can be found in section 7 on page 48 and section 8 on page 50, respectively.

2 Background

2.1 Machine learning

Machine learning[36, 28] is a certain branch of A.I. that focusses on giving intelligent software the ability to evolve a certain behaviour based on empirical data. This data can stem from multiple sources such as sensors or databases. Data could be viewed as examples that illustrate or exhibit certain (hidden) relationships between the observed variables. Patterns need to be extracted from these examples to help make the intelligent software make better decisions.

Learning is a concept that is studied by a lot of branches in science. Biology being a prime example of this. It does pose the question however, if machines will be able to learn as well? The answers to this question will in fact draw a lot of inspiration from those other branches in science.

But how does a machine learn exactly? In a very broad, generic way it could be stated that a machine or piece of intelligent software learns when its internal structure changes, that is, its programming, its data (based on internal datastructures or external data inputs) with the expectation that its performance improves over time.

Tim M. Mitchell[28] states this as: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"

2.1.1 Taxonomy

Algorithms used in machine learning could be classified into different categories based on their desired outcome:

supervised learning: algorithms in this category learn from *labeled* (class labels) examples, for instance, making new predictions based on known data.

unsupervised learning: in this category algorithms operate on *unlabeled* examples, modelling their inputs, as for instance in *clustering* and *data mining*.

reinforcement learning: algorithms learn based on (positive or negative) feedback provided by the environment they operate in, which is based on the action that was performed.

2.1.2 Approaches

The actual "learning" can be approached in a lot of ways, some (but not all) example approaches are:

decision tree learning: learning based on constructing a predictive model using decision trees.

artificial neural networks: learning based on biological neural networks naturally found in brains, computations are structured in terms of interconnected neurons which only fire off when their activation value is higher than some threshold, feedback is usually propagated back through the network, adjusting several weights assigned to neurons.

bayesian networks: learning based on Bayes' rule, in terms of conditional independencies expressed in a directed acyclic graphs, with interconnected random variables, on which algorithms perform probabilistic inference.

clustering: large sets of unlabeled examples subdivided into smaller subsets based on a certain measure of similarity, where examples being more similar to each other end up together in the same cluster.

2.2 Game theory

As stated before games involve complex interaction between players[46, 49]. What makes those interactions even more difficult in terms of decision making, is that each player has their own goals, desires and preferences about what the world (gamestate) should look like.

2.2.1 Utility

Utilty is a way to formally express desirability for a certain outcome of the world. Or a game, in this matter. Utility could be compared to real world money, although not being exactly the same. For example, a big international company could be in debt for millions of dollars. Obviously this state is really undesirable and consequently has a really low utility value.

Where 100 million dollars or 99 million dollars aren't really a big difference, relatively speaking, every company would pick being in debt for 99 million dollars over 100 million dollars.

Both numbers are still really big, but they are both (almost) equally undesirable, however. Therefore, it is not a linear relationship (i.e. the utility value for both numbers is about the same).

This works in the opposite way too, having a lot of money on your bank-account generally is believed to have a high utility value. But again, while the difference between having 100 million dollars or 99 million dollars is 1 million dollars, it both still is a lot of money. Therefore, the increase in utility isn't as big as the increase in money.

Def. 2.2.1. Let Ω be the set of all the n possible (abstract) outcomes of the world, namely ω_1 to ω_n, be defined as:

$$\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}.$$

□

Def. 2.2.2. Given any two players i and j and a set of outcomes Ω, the utility functions u_i and u_j of both players, map every possible world outcome to a real value. Formally expressing a preference relationship over all the world outcomes:

$$u_i : \Omega \to \mathbb{R},$$
$$u_j : \Omega \to \mathbb{R}.$$

Note that there are no further restrictions on those particular values, and they are supposed to be modelled on a case by case basis.

□

Defining utility functions like that, naturally leads to the notion of an ordering on the preferences on outcomes of a player. It also helps to formally express that a player prefers outcome ω over ω'.

Def. 2.2.3. Given any player i and any two possible world outcomes $\omega, \omega' \in \Omega$, let $\succeq_i$ be defined as the utility ordering relationship for player i:

$$\forall i \forall \omega, \omega' \in \Omega : \omega \succeq_i \omega' \Rightarrow u_i(\omega) \geq u_i(\omega').$$

Or when ω is *strictly* prefered over ω':

$$\forall i \forall \omega, \omega' \in \Omega : \omega \succ_i \omega' \Rightarrow u_i(\omega) > u_i(\omega').$$

□

In most games, but not all, the actions of the players are performed in a sequential manner. Although a famous counter-example to this is the game *rock-paper-scissors*. However, in both situations, the actions of all the individual players taken together will determine the outcome of the world or game. This doesn't necessarily imply that all the players will affect the current outcome though. The *environment transition function* defined below maps the actions from all the players to a new outcome.

Def. 2.2.4. Let A_i denote the set of the possible n actions a_1^i to a_n^i player i can take. It's called the *action set* and is defined as follows:

$$A_i = \left\{a_1^i, a_2^i, \ldots, a_n^i\right\}.$$

□

Def. 2.2.5. Given n players, let A be the set, which defines for every player i, a possible action $a_j^i \in A_i$. It's called the *action profile* and is defined as follows:

$$A = A_1 \times A_2 \times \ldots \times A_n.$$

□

Def. 2.2.6. Let A denote the action profile as defined in Def. 2.2.5 on page 12. Given any two players i and j and a set of outcomes Ω, the outcome of both their actions is defined in the *environment transition function* called τ:

$$\tau : A \to \Omega.$$

Informally, it maps any action profile A to a possible outcome defined in Ω. Note that this implies that each player will have to perform an action. They don't have to necessarily be aware of the action performed by the other player though.

□

Def. 2.2.7. Given any two players i and j, outcomes ω_1 to ω_4 from Ω, let the environment transition function be defined as:

$$\tau(\langle a_1^i, a_1^j \rangle) = \omega_1, \quad \tau(\langle a_2^i, a_1^j \rangle) = \omega_2, \quad \tau(\langle a_1^i, a_2^j \rangle) = \omega_3, \quad \tau(\langle a_2^i, a_2^j \rangle) = \omega_4.$$

In order to neatly represent this two player interaction given their utility functions, they are put in matrix form as shown in table 2.2.1 on page 13 and is called the *payoff matrix*. The payoff matrix for a game represents the *normal* or *strategic form* of a game.

□

j \ i	a_1^i	a_2^i
a_1^j	$u_i(\omega_1)$ $u_j(\omega_1)$	$u_i(\omega_2)$ $u_j(\omega_2)$
a_2^j	$u_i(\omega_3)$ $u_j(\omega_3)$	$u_i(\omega_4)$ $u_j(\omega_4)$

Table 1: Payoff matrix

2.2.2 Dominance

Now that the basics are covered, the real question is; how do players pick an action to perform? While this is a seemingly very simple question, it deeply touches upon all the things that will be covered in this thesis. It would take way to much time and space to cover everything, but a nice start is the concept of a *dominant* strategy.

Def. 2.2.8. Given any player i, the corresponding preference ordering relationship $\succ_i$ covered in Def. 2.2.3 on page 12 and finally a set of outcomes Ω. Let the following be the case:

$$\Omega_1 \subset \Omega, \quad \Omega_2 \subset \Omega, \quad \Omega_1 \cap \Omega_2 = \varnothing.$$

If the following condition is true then Ω_1 is said to be *strongly dominating* Ω_2:

$$\forall \omega_1 \in \Omega_1, \forall \omega_2 \in \Omega_2 : \omega_1 \succ_i \omega_2.$$

Intuitionally, this makes sense because every outcome ω_1 is strictly prefered, by player i, over outcome ω_2.

□

Def. 2.2.9. Given any strategy s_1 and s_2, let Ω_{s_1} and Ω_{s_2} be the sets of possible outcomes after playing s_1 or s_2, respectively. Then if Ω_{s_1} strongly dominates Ω_{s_2} as defined in Def. 2.2.8 on page 13 it is said that s_1 strictly dominates s_2.

□

The notion of dominance is very important. It makes sense for a rational player i to pick a strategy (i.e. a sequence of actions from the set A_i), which leads to *dominating* outcomes.

On the contrary, if there exist *dominated* strategies, it makes sense for the player to eliminate them from consideration. Because it can do better by choosing to play the strategy that is dominating the others.

2.2.3 Nash equilibrium

Nash equilibrium[31, 30, 49] is an important concept in game theory. It is a state in a game where no player can do better by deviating from their current strategy, assuming the other players will stick to their current strategy. In other words; it mutually locks each player in a certain action.

For most european countries, driving on the right is a nice example of a Nash equilibrium. Assuming other drivers will be driving on the right as well, there is no incentive for the driver to deviate to driving to the left (for obvious collision-avoiding reasons).

More formally it could be described as:

Def. 2.2.10. Given any two players i and j and their respective strategies s_i and s_j, then a Nash equilibrium is:

- assuming player j plays s_j, player i can do no better than to play s_i; and
- assuming player i plays s_i, player j can do no better than to play s_j.

□

Or in context of all the previous defined notions:

Def. 2.2.11. Given a player i and a corresponding action set A_i, let A_{-i} denote an action profile for all the players except player i. Then an action profile A^* is a Nash equilibrium if and only if:

$$\forall i, \forall a^i \in A_i : u_i(A^*) \geq u_i(A^*_{-i} \cup \{a^i_j\}).$$

where $a^i \neq a^i_*$ and $a^i_* \in A^*$. Informally, this states that all players can do no better by taking another action except the action from the nash equilibrium A^*.

□

Note that in a Nash equilibrium every player is assumed to be rational, i.e. only taking the best possible actions. It is important to note however, that it has been proven that there are games with multiple Nash equilibria, or even worse, none at all.

Game theory comprises much more, this was only a brief introduction into some basic concepts, which will be refered to later.

2.2.4 Nash equilibria in tree search

The notion of Nash equilibria in games is relevant, because section 2.3 on page 15 discusses the Monte Carlo Tree Search technique. Using the standard implementation presented in that section, i.e. without the addition of additional domain knowledge to guide the search process, will most likely produce an (approximated) Nash equilibrium strategy. Section 2.4 on page 18 discusses the need for additional domain knowledge.

2.3 Monte-Carlo tree search

2.3.1 Introduction

Monte Carlo Tree Search[8, 7] (MCTS) is a best-first search algorithm. Which means that instead of exploring and trying everything, it will try to explore the most promising part of the gametree first. So that more time could be spent on doing a better analysis of the game in that particular part of the tree.

Monte Carlo methods rely on sampling. Sampling of the enormous state-space of a poker game in our case. Basically it will simulate a lot of games, starting from the current game state. It keeps track of each unique gamestate it comes across during the simulation. Gamestates which are seen more often are apparently more significant. The following simulations will be drawn to those parts in the tree. In effect satisfying our premise, that MCTS will explore the most promising parts in the gametree first. MCTS algorithm consists of 4 steps which will be explained in the following subsections.

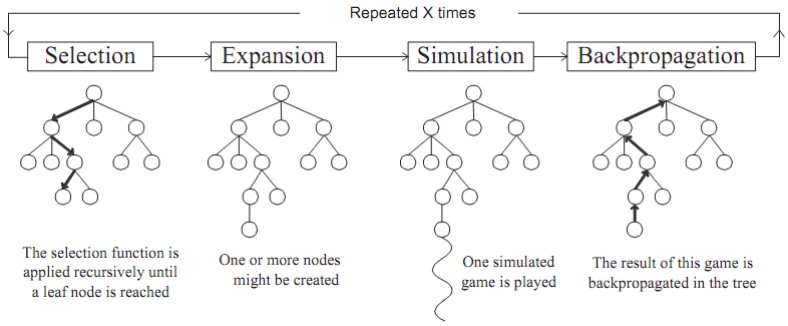

Figure 1: MCTS algorithm flow

2.3.2 Selection

Selection balances between exploration and exploitation. Too much exploration and its all over the place. Not finding a strategy that exploits the weakness(es) of an opponent. Too much exploitation and it will never find interesting and possibly even better strategies against said opponent. In other words, does the search has to broaden or deepen? Selection, as the name implies, selects the gamestate which is next to be further explored.

Selection is usually guided by a strategy that balances between *exploration* and *exploitation*. One the one hand, the task is to select the action that leads to

the highest expected value, (exploitation). On the other hand, there is a chance that more promising game states are hiding behind actions with a smaller expected value. This necessitates the need for the consideration of those actions as well (exploration).

Def. 2.3.1. Given a game-tree node T_i, the parent node T_p of T_i and a coefficient C which determines the balance between exploration and exploitation (usually $C = 2$), let v be the expected value for node T_i and let n_p and n_i be the visit count of nodes T_p and T_i respectively. Then the formula for Upper Confidence Bound applied to trees[22] is defined as:

$$UCT(T_i) = v + C * \sqrt{\frac{\ln n_p}{n_i}}$$

Note that in an actual implementation, care has to be taken for the fact that a division by zero error could occur. This is usually solved by introducing a very small constant $\epsilon = 1 * 10^{-9}$.

□

Algorithm 1 MCTS Selection[1]

Precondition: T, a MCTS game tree node,
Postcondition: a descendent node T_i with the highest UCT value.

1: **procedure** SELECT(T)
2: **return** $\arg\max_{\text{descendants } T_i \text{ of } T} UCT(T_i)$

This doesn't however imply that UCT has to be used. Other alternatives for selection methods are *greedy*, *ϵ-greedy* or *softmax* selection.

2.3.3 Expansion

Expansion, adds some or all alternatives which are reachable from the current state. I.e. it expands the current node in the game tree.

Algorithm 2 MCTS Expansion[1]

Precondition: T, a MCTS game tree node,
Postcondition: expanded node T.

1: **procedure** EXPAND(T)
2: **for all** possible actions a_i in node T **do**
3: add T_i as descendent of T with edge labeled as a_i

2.3.4 Simulation

Simulation, plays out the game till the end, from a formerly expanded gamestate using random moves. Although if more informed decisions are being

made, the simulation will be more effective. The downside of course being that more informed decisions cost more computation time.

Note that simulation does not expand nodes. No algorithm is provided for simulation as this is entirely dependent on the current game and/or domain.

2.3.5 Backpropagation

The result from the simulation, winning or losing for instance, will be propagated back up the tree along the path from which it came. This will update the expected value of the gamestates, making them more or maybe less promising than the rest in the gametree.

Algorithm 3 MCTS Backpropagation[1]

Precondition: T, a MCTS game tree node,
Precondition: v, the resulting simulation value,
Postcondition: updated values in MCTS game tree.

```
1: procedure BACKPROPAGATE(T, v)
2:     T.num_visits ← T.num_visits + 1
3:     T.value ← T.value + v                ▷ expected value of node T
```

2.3.6 MCTS algorithm

The complete algorithm composed of Alg. 1 on page 16, Alg. 2 on page 16 and Alg. 3 on page 17 is shown in Alg. 4 on page 17. Note that *simulation* does *not* expand nodes. The algorithm is usually repeated for as long as there is computation time, or for as long as some user defined counter is less then or equal to some predefined maximum number of iterations.

Algorithm 4 MCTS (single iteration)[1]

Precondition: T, a MCTS game tree root node,
Postcondition: new MCTS game tree rooted at T.

```
1: procedure MCTS(T)
2:     V ← {T}                              ▷ add T to visited nodes set
3:     P ← T                                ▷ make P the current pointer
4:     while P is not a leaf node do
5:         P*_i ← SELECT(P)
6:         V ← V ∪ {P*_i}                   ▷ add P*_i to visited nodes set
7:         P ← P*_i
8:     EXPAND(P)
9:     T*_j ← SELECT(P)                     ▷ T is ancestor of T*_j
10:    v ← SIMULATE(T*_j)                   ▷ v is the resulting value
11:    for all V_i ∈ V do                   ▷ backpropagate over all visited nodes
12:        BACKPROPAGATE(V_i, v)
```

2.4 Opponent modelling

Pure implementations of the MCTS algorithm have been used before, primarily in the game *Go*. This pure form of MCTS produces an approximated Nash equilibrium strategy (i.e. the best possible strategy against itself, see Def. 2.2.11 on page 14). In other words, the resulting strategy will be completely oblivious to opponent mistakes and treats them as rational players. In this thesis it has already been established that poker is inherently far more complex.

Instead of producing an approximated Nash equilibrium strategy, MCTS should be adapted to take advantage of the predictability, mistakes and general patterns in any opponents' playstyle. Which will most likely yield a larger net profit than a purely rational strategy.

In order for MCTS to take into account the specific opponents, Ponsen et al. propose the some adaptations[33]. Note however that they assume a two player *heads-up* Texas Hold'em Limit poker game. This places considerable constraints on the allowed bets a player can make. It reduces the number of possible actions a player can take and as such dramatically reduces the complexity of the state space.

Moreover, they opt to train their opponent models specifically to each opponent. Alternatively one could chose to learn models specific to player *types*. They collected 5000 games for each of the two pokerbots, *ACE1* and *Poki*, they were trying to model. Note that those are implemented to behave according to (simple) predictable rules.

2.4.1 Learning the opponent model

In order to train their model they introduce some notation first:

Def. 2.4.1. Let a_i be the action player p took at timestep i, moreover let c_p be the private cards of player p at timestep i and let S_{i-1} be the complete history (i.e. community cards and the action history of all the other players) of the game state up until timestep i. Then the *example* is defined as:

$$\langle i, p, a_i, c_p, S_{i-1} \rangle.$$

□

Given those examples, they can be used to train a classifier as follows:

Def. 2.4.2. Given a set E of examples as defined in Def. 2.4.1 on page 18, the learning tasks can be formulated as follows; predicting the private cards c_p of player p using the examples being labeled with c_p, given the complete game state history S_{i-1}:

$$\Pr(c_p \mid S_{i-1}).$$

or predicting the action a_i of player p using the examples being labeled with a_i, given the private cards c_p and the complete game state history S_{i-1}:

$$\Pr(a_i \mid S_{i-1}, c_p).$$

□

2.4.2 Approach

Ponsen et al. propose a two fold learning approach:

1. settle on a certain *prior* distribution, which an be achieved in many ways (learning functions over general poker players, player types or rational players).

2. learning a *differentiating function*, by adapting the *prior* distribution to the observed distribution of the particular player.

Consider two distributions D_* and D_p, the prior distribution of examples drawn randomly from the prior distribution and the distribution of examples drawn randomly from the observed player p distribution respectively. The distribution D_{*+p} is then a *mixture* of D_* and D_p. The learning problem could then be viewed as:

Def. 2.4.3. Given a randomly drawn example x from the mixture D_{*+p}, predict whether x belongs to D_* or D_p. Learning the function $\Pr(D_p \mid x)$, which gives for each example x the probability that it belongs to D_p, can be achieved by generating examples from D_* and D_p and labeling them with $*$ or p. Learning the function $\Pr(x \mid D_p)$ in terms of $\Pr(D_p \mid x)$ is possible using Bayes' rule:

$$\Pr(x \mid D_p) = \frac{\Pr(D_p \mid x)\Pr(x)}{\Pr(D_p)}.$$

Since $|D_*| = |D_p|$, by generating an equal amount of examples for both:

$$\Pr(D_*) = \Pr(D_p) = \frac{1}{2},$$

and

$$\Pr(x) = \Pr(D_*)\Pr(x \mid D_*) + \Pr(D_p)\Pr(x \mid D_p).$$

Substituting the latter two in the first:

$$\begin{aligned}\Pr(x \mid D_p) &= \frac{\Pr(D_p \mid x)(\frac{1}{2}\Pr(x \mid D_p) + \frac{1}{2}\Pr(x \mid D_*))}{2} \\ &= \Pr(D_p \mid x)\Pr(x \mid D_p) + \Pr(D_p \mid x)\Pr(x \mid D_*) \\ &= \frac{\Pr(x \mid D_*)\Pr(D_p \mid x)}{1 - \Pr(D_p \mid x)}.\end{aligned}$$

Where $\Pr(x \mid D_*)$ is the prior probability of x, $\Pr(D_p \mid x)$ is the learned *differentiating function* and finally $\Pr(x \mid D_p)$ is the posterior probability for example x belonging to player p. Which can be used in MCTS.

□

The big advantage of this approach over others is that this is an elegant method to learning a multi-class classifier (e.g. predicting over $\frac{52*51}{2}$ possible card combinations). Secondly, assuming the prior distribution D_* is reasonable, accurate predictions of player p are possible with only a few examples.

2.4.3 Integration with opponent model

The opponent modelling is integrated into the MCTS algorithm. Using the opponent model the card probilities can be sampled much more accurately. Instead of sampling at random during each MCTS iteration, the opponent model might state that after having observed a bet from a certain opponent, the likelihood of higher ranked private cards is much higher. Attaching a higher probability the these cards, will lead to MCTS executing more iterations with them.

At the start of each game state where the MCTS player has to act, a probability distribution of the opponent's private cards will be computed (according to $\Pr(c_p \mid S_{i-1})$). During the current iteration the cards will be sampled from this distribution. Whenever an opponent has to take an action the action probabilities are queried from the opponent model (according to $\Pr(a_i \mid S_{i-1}, c_p)$).

During *simulation*, roll-out simulations are employed. Which means that each player will only check or call for the rest of the game. Then all community and private cards are dealt to determine the winner(s). The value returned by the simulation step is the amount of chips lost or gained.

Note that the simulation is performed twice. Once from the view MCTS player itself and once from the view of all the opponents. For the MCTS player its private cards are known. For the opponents the private cards have to sampled (as described above).

Algorithm 5 MCTS Selection with opponent model integration[33]

Precondition: T, a MCTS game tree node,
Precondition: c_p, the sampled private card combination for this iteration,
Postcondition: a node with the highest value

1: **procedure** SELECT(T, c_p)
2: **if** T is not opponent node **then**
3: **return** $\arg\max_{\text{descendants } T_i \text{ of } T} UCT(T_i)$
4: **else**
5: **return** $\arg\max_{\text{descendants } T_i \text{ of } T} \Pr(a_i \mid S_{i-1}, c_p)$ ▷ for every action a_i leading to T_i

2.4.4 Results

Ponsen et al. report that using an opponent model improves the performance a lot. The ACE1 bot is beaten quite easily as it's based on simple rules. Even though the number of iterations of the MCTS algorithm are quite small, the Poki bot gets beaten by a small margin. Without the opponent model (i.e. pure MCTS), the performance is relatively quite bad.

3 Research

3.1 Motivation

Improving the performance of intelligent computer players has been a very significant part of research since the inception of artificial intelligence. Mostly

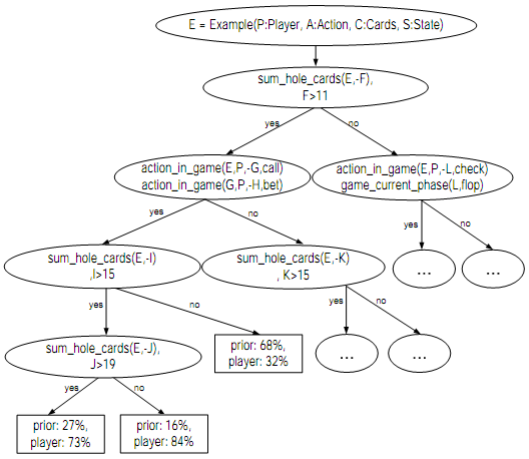

Figure 2: Example Tilde decision tree encoding the differentation function for predicting cards[33]

because games offer such a well defined environment to test the performance of any intelligent implementation.

Games like *Tic-tac-toe*, *Chess* and *Checkers*, have been completely analysed in the past. Which wouldn't be as easy or even possible if those games had any elements of uncertainty or complexity induced by a very large action set. Not to mention more than two players.

This is were Poker differs greatly from those games. Poker is still largely misunderstood by a large group of people, even professionals, who still learn about new game mechanics every day. The rules of Poker are relatively simple but they provide for a very rich set of possible strategies.

Research in Poker, or uncertainty in games in general, will hopefully provide more insight into the complex decision making process necessary to play those games on a reasonable level.

3.2 Research question

The main goal for this thesis is testing whether or not the approach[33] proposed by Ponsen et al. could be extended in such a way that it would also be viable in a *no-limit* Texas Hold'em poker game with more than two players. No-limit Texas Hold'em poker has an even larger state space, because of the less constrained betting structures.

The big challenge is partly how to deal with such a large state space. The other part is how to incorporate the opponent modelling into all of this. The working hypothesis guiding this thesis is formulated as:

- Using real tournament no-limit Texas Hold'em pokerdata to learn a reasonable *prior* opponent model of a generic poker player in combination with the Tilde and Monte Carlo tree search algorithms, applied in a no-limit Texas Hold'em poker game environment will result in a performance conform the results reported by Ponsen et al.

3.3 Assumptions

During this research the following assumptions will be made:

- players will be able to switch between strategies, they can both exhibit a static or dynamic play style,
- players' beliefs and goals could conflict between each other,
- the game state is not influenced by a single individual player only (i.e. no trivial folding),
- each player is self-interested (i.e. no cooperation),
- the opponents can consist of real human or computer A.I. players,
- the number of players active in a game of poker can vary from 2 to 10.

3.4 Theoretical approach

Game tree search is not a new concept. It's a very useful technique, which has been applied on many occasions in the past. Especially in the field of *perfect* information games. The most well known example is the *minimax* algorithm.

Game tree search algorithms can be applied to fully expanded game trees, or the search can be guided by an *heuristic* function. This function returns the best estimate of the objective value for a particular node in the game tree (with respect to the current player).

This works well in "simple" perfect information games such as *tic-tac-toe* or *checkers*, because of the fact that, relatively speaking, these games don't have a very large state space. In other words, the entire game tree fits into modern day computers' memories. Game tree search algorithms are applied to these game trees and are able to solve the game to return the best move available for the current player.

As mentioned before, problems arise in *imperfect* information games. Because inherently not all information is known during play time, game trees cannot be structurally analyzed as before. The best option is to design a really good heuristic function, where care has to be taken to find the best balance between *accuracy* and *computational costs*.

In this thesis, the newer Monte Carlo tree search algorithm, as explained in section 2.3 on page 15 will be used. This algorithm has proven to work very well for perfect information games with extremely large state spaces such as the game of *Go*. It iteratively expands and samples the state space so that approximately the most available computational time is spend on expanding the most "promising" nodes in the game tree.

As explained before, MCTS, in its pure form, uses the very generic UCT *heuristic* to guide the search. In this form the search is mostly oblivious to exploiting the opponents' weaknesses. Which is a crucial part of playing poker *well*.

In order to enable the game tree search algorithm to find exploitative game states, opponent modelling will be used in place of the heuristic function. In this thesis the opponent models will be provided by the *Tilde* algorithm, which is explained in detail in section 2.4 on page 18 and section 5.4 on page 36.

Tilde, is a top-down decision tree induction algorithm, which uses first order logic. This is a major difference with respect to the more common decision tree induction algorithms such as C4.5. These algorithms use propositional logic to calculate the "best" splits (see section 5.3 on page 28). Because of this fact Tilde is able to find and *express* more complex relationships between players and/or gamestates.

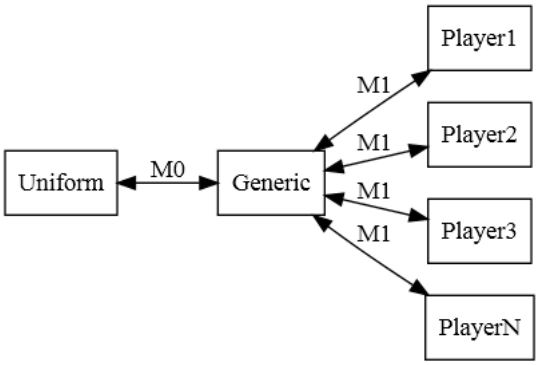

Figure 3: How models discriminate between players based on a single model.

In figure 3 on page 23 is depicted how N players can be discriminated by only using a single opponent model. Optionally one could even start from a uniform poker player (i.e. all actions have an equal probability of being performed during tree expansion). The generic poker player will serve as a *prior* distribution, and the opponent model will serve as the *differentiating function*, as is all explained in section 2.4.2 on page 19.

The main motivation of this approach is that, learning the difference between two distributions is a very elegent way of learning a *multi-class* classifier, by generalizing over many one-against-all learning tasks (i.e. not N player specific classifiers). The other one is that, assuming the prior distribution is reasonable, already accurate predictions are possible.

3.5 Practical approach

In order to test the validity of the hypothesis the approach taken in this thesis consists of:

1. collecting the necessary high quality poker data,
2. implementing the algorithms discussed in this thesis,
3. learning a generic poker player model from the aforementioned poker data,
4. collecting enough data from playing against players and/or pokerbots,
5. applying the prior generic poker playing model in combination with the collected data from the players who are to be modelled,
6. comparing the resulting performance of this implementation, with the results reported by Ponsen et al.

3.6 Previous work

As an introduction into computer poker[2], Billings extensively writes about the relevance of computer poker in artificial intelligence. Answering why poker is such a great game to study in the context of computer science.

In his following PhD thesis[3], he describes all the current techniques, algorithms and approaches to computer poker. This is probably the most extensive guide in the field of computer poker.

Billings et al. describe their implementation of the poker bot called *Loki*[4]. Each opponent gets assigned a weights table indexed by starting hands. Depending on the actions of those opponents some card combinations get a higher weighting, where others get a lower one.

Neural networks[13, 14] are applied by Davidson et al. to poker as well. A neural network is trained on certain input variables which describe the game state, to predict if an opponent would fold, call or raise. An additional advantage is that the resulting weights of the internal network connections are a great indication of the most predictive input attributes.

Bayesian networks are also a popular approach to opponent modelling in computer poker. Southy et al. reduce the complexity of poker to a more simple variant called *Leduc Hold'em*[39]. A bayesian network is trained using observations from opponents in combination with a prior distribution, to calculate a best response to a certain move. Korb et al. introduced bayesian networks to model poker opponents[25].

4 Concepts

4.1 Inductive reasoning

Inductive reasoning[36, 26] is the complete opposite of deductive learning. Deductive logic arrives at a conclusion by applying the rules of logic to a set of *premises* or *hypotheses*. When those are *true* and *valid*, the process of deduction, shows that this newly gained knowledge *necessarily* follows from them.

If deduction were to be labeled as a bottom-up approach to gaining new knowledge, induction should be labeled as the top-down approach to gaining knowledge. Induction uses a set of, possibly abstract, examples or observations, from the environment where those were extracted from. It then uses those examples or observations, to possibly discover a more general rule or hypothesis that describes them.

The problem with inductive reasoning is that, there will never be concrete absolute proof for the existance of those induced rules or hypotheses. Take for instance this classic statement: "I have never seen a black swan before, so by induction I state that all swans must be white." When a black swan would however be observed, then the whole reasoning would collapse and its meaning would be void. This is more generally known as the *induction problem*[26].

4.2 Inductive learning

Inductive learning[36] is a form of *supervised* learning. It involves learning from examples, which are already labeled with their corresponding output value.

The problem of inductive learning is then to induce an hypothesis, an approximation to the *true* output function, which is often *unknown*. More formally the problem could be expressed as:

Def. 4.2.1. Let an *example* be an ordered pair $\langle \vec{x}, f(\vec{x}) \rangle$, where $\vec{x}$ is the input value and $f(\vec{x})$ the output value. Then the task of induction is:

Given a set of examples of f, return a function h that approximates f.

□

This function h is then called the hypothesis. When the true function f is unknown, which is almost always the case, it's impossible to know if h is a good approximation of f. One property of a good hypothesis is, that it *generalizes* well.

To break ties between multiple equally performing hypotheses, *Ockham's razor* is used. In other words; the most simple hypothesis consistent with the data, is the prefered one. Intuitively this makes sense because a really complex model is more likely to suffer from *overfitting* and most probably doesn't generalize well, that is, being really bad at predicting the outcome for new examples.

4.3 Inductive logic programming

Inductive logic programming[36, 35, 29] (ILP) takes the inductive methods and combines them with the power of first-order logic. Specifically the ability to represent the learning problem in terms of background theories using logic programming.

A widely known and popular implementation of such a system is the *Prolog* programming language. Which uses facts and rules, represented as predicates and *Horn* clauses respectively, to express its logic theories. This enables extensions of the core Prolog programming language which allow for inductive logic programming. Or more formally:

Def. 4.3.1. Let B be any background theory, H be any hypothesis, E be the set of examples and C the set of corresponding classifications, all expressed in the same logic programming language, then the entailment constraint to be solved is defined as:

$$B \wedge H \wedge E \models C.$$

where H is unknown and needs to be induced.

□

One of the main reasons why ILP techniques gained so much popularity over existing supervised learning methods is that, first-order logic enables the ability to capture underlying relationships between attributes, whereas other methods wouldn't be able to do so.

This makes ILP more suitable to more complex induction problems, especially those with attributes being heavily biased to being described by their inherent relationships that are existing between them.

4.4 Decision tree learning

Decision trees are an algorithmically perfect fit to inductive learning. They are so succesful because of their generally great performance with respect to the amount of input data and the quality of the resulting models.

The models resulting from running those algorithms are very easy to interpret because of their rule-based nature, which is a huge advantage in comparison to more abstract models resulting from other learning algorithms.

Decision tree theory will be discussed more in depth in section 5.3 on page 28.

4.5 Opponent modelling

One way to reduce uncertainty in any imperfect information game is to make sure you have at least an idea about your opponent(s)[33, 4, 3, 2, 14, 13, 39, 25, 40]. An idea about their style of playing. Initially this could be having no idea at all. Or you could assume your opponent to play exactly as you would.

In most games, in order to achieve the "best" performance of an intelligent computer player, it is safely assumed that the opponent is 100% rational. That is, given that the opponent is assumed to take the best possible action at any given time during a game, the game tree is expanded to find the best possible response to beat your opponent.

The major caveat here is that, this only works in perfect information games. Secondly, the state space of the game has to be of manageable size (i.e. it has to be possible to somehow fit it all into the current memory). In other words, this is only feasable for a game of a highly analytical nature.

Then there are games with an amazingly large state space, imperfect imformation, uncertainty, and a relatively huge variety of actions to choose from. Add to this complex and involved player interaction dynamics. Poker is one of those games.

In poker, the assumption that every player is 100% rational, is not a safe one to make. Players often switch from one strategy to another. The state space is just so large that even human players cannot possibly make a 100% rational, informed decision.

On which basis do they, or should they, make their decisions then? This is where opponent modelling plays a really important part. It helps players to make decisions on the basis of what they *believe* the gamestate looks like for their opponents.

In this thesis, two techniques will be discussed to render the immense state space of poker into a more managable one, using monte carlo sampling discussed in section 2.3 on page 15 and logical decision trees discussed in section 5.4 on page 36.

5 Implementation

5.1 Poker Academy Pro

Poker Academy Pro[19], produced by Biotools Inc., is essentially a poker simulation and analysis tool. It also creates profiles of previously encountered players during online play, which includes the storage of all their observed hands

in their corresponding profile. It supports limit hold'em and no-limit hold'em poker.

Besides its extensive poker game playing and analysis capabilities, it also features the creation and interaction with *pokerbots*. A great tool for poker researchers. It also features the inclusion of several popular pokerbots created by the University of Alberta Computer Poker Research Group, such as: *Vezbot*[37], *Loki*[32] and *Poki*[15].

Although it has been mentioned before that it's considered dangerous to draw *generic* conclusions about the performance of your own pokerbot with respect to those existing implementations. Especially about the extrapolation of those results into the domain of real online play for actual real money!

5.1.1 Meerkat API

The Meerkat API[18] is a small *Java* library that comes with Poker Academy Pro. The API is event driven and therefore the pokerbots have access to all the information which a normal human player could observe as well.

Implementing your own pokerpot is very straightforward and relatively easy to do. As a programmer of pokerbots, you don't want to be concerned about implementing a poker engine, instead Poker Academy Pro already takes care of all those details.

5.2 Prolog

A substantial part of this project uses the Prolog programming language. Because of the fact that learning from data, requires a lot of it, a performant implementation of the language is needed. The faster, the better.

5.2.1 JIProlog

Java Internet Prolog or JIProlog is implemented in 100% pure Java, developed by dr. Ugo Chirico[10]. It's also fully compliant to the ISO Prolog standard[20, 21]. Some mobile devices are supported as well. It's fairly backwards compatible with respect to older Java versions, dating back to JRE version 1.1.

JIProlog is still actively used in 2APL[12] (pronounced as double-a-p-l) the agent platform developed here at the University of Utrecht, as the internal Prolog based plan and goal based reasoning engine.

The main advantage of JIProlog is the full two-way integration of Java and Prolog. It also provides the ability to write your own custom Prolog predicates in Java. Providing you with the tools to create your own personal Prolog suite for custom implementation needs.

The downside however is that the code is getting older. Java and Prolog are both moving ahead, while JIProlog hasn't received any update in years.

5.2.2 SWI-Prolog

SWI-Prolog[43, 44, 45] is a fully featured Prolog, portable, free as in speach, opensource implementation, licenced under LGPL/GPL. It's being developed by Jan Wielemaker and it has a lot of other active contributors in its community.

Some, but absolutely not all, of the features are; a large collection of libraries, integration with C and C++ languages, Java interface, multi-threading support, socket support, etc.

The big advantage of SWI-Prolog is how mature the codebase is. With a wide range of builtin predicates. Features which JIProlog clearly lacked.

5.3 Decision trees

5.3.1 Introduction

Classification[36, 41, 34] is the act of assigning a class to a certain object. The meaning of class and or object can be very loosely interpreted here. Most usually the object is a record, stemming from a certain database.

Why do decision trees matter and why are they so popular? One of the major reasons is the fact, that a fully induced decision tree is very easy to interpret and understand by a human. Which again allows for possible further analysis by a human domain-expert. Decision trees produce so-called *white-box* models.

Other advantages over other data-mining techniques are:

- The input data requires little to no *preprocessing* or *data normalisation*;
- The ability to handle multiple types of data (numerical or categorical);
- The resulting models can be easily *validated* by statistical tests;
- The ability to cope with large amounts of data in a fairly straightforward manner;
- The core techniques are relatively easy to implement.

Of course, decision tree induction has its own limitations and problems too:

- Learning an *optimal* model is known to be NP-complete. (i.e. only approximations are tractible);
- The resulting tree models are often overly complicated and do therefore not generalize well, most algorithms have to deal with *over-fitting*;

Note that optimal wasn't defined. The most obvious definition would of course be in terms of the complexity of the resulting tree. Under most definitions of optimality, the induction of such models is still NP-complete.

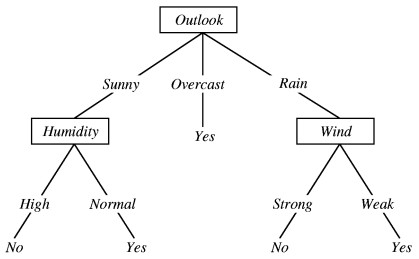

Figure 4: Example decision tree predicting whether or not to play tennis based on the current weather conditions

Def. 5.3.1. Let A_i be any attribute. A_i is called *continuous* or *discrete* if its possible values are numerical or nominal respectively.

□

Def. 5.3.2. A single record or example from a certain dataset, which will be used for machine learning purposes is called *labeled* if it includes the class label. Otherwise it is called *unlabeled*.

□

5.3.2 Structure

A decision tree is a *rooted* tree, which stores a set of *nodes* such that the nodes have a *parent-child* relationship.

Or more formally recursively defined as:

Def. 5.3.3. Let T be a *rooted* tree such that T is:

- a *leaf*-node l, storing an element e.
- a *root*-node r and a set of trees whose roots are the *children* of r.

□

5.3.3 Framework

Imagine a simple database table with historical data about loan applicants. Applications are often accepted or rejected depending on certain characteristics of the applicant. Those could be name, age, gender, income, marital status, etc. These are considered to be the *attributes*. Applications are therefore classified as either accepter or rejected and thus are the *class labels* in this particular example. Or more abstractly:

Def. 5.3.4. Let $\mathcal{D} = \langle C, A, T, E \rangle$ denote a generic framework to express any decision tree induction problem in. Then the individual components of that framework are defined as:

- Let C be the set of *discrete* classlabels then it is defined as:

$$C = \{c_1, c_2, \ldots, c_n\} \text{ where } n \geq 1. \tag{5.3.1}$$

- Let A be the set of attributes, which characterizes E, then it is defined as:

$$A = \{A_1, A_2, \ldots, A_n\} \text{ where } n \geq 1. \tag{5.3.2}$$

- Let T be the set of tests, defined as:

$$T = \{t_1, t_2, \ldots, t_n\} \text{ where } n \geq 1. \tag{5.3.3}$$

  where each individual test $t_i \in T$ is defined as a function:

$$t_i : E \to V_i^t. \tag{5.3.4}$$

  and where each V_i^t is the set of values, function t_i can take on:

$$V_i^t = \left\{v_1^t, v_2^t, \ldots, v_n^t\right\} \text{ where } n \geq 1. \tag{5.3.5}$$

- Let E be the set of *labeled* examples then it is defined as:

$$E = \{e_1, e_2, \ldots, e_n\} \text{ where } n \geq 1. \tag{5.3.6}$$

where each individual *labeled* example $e_i \in E$ is constructed as a tuple of length $n+1$, as follows:

$$e_i = \langle a_1^i, a_2^i, \ldots, a_n^i, c_j^i \rangle. \tag{5.3.7}$$

under the following restrictions that $\forall 1 \leq k \leq n : a_k^i \in A_k, A_k \in A$ and $c_j^i \in C$. In other words, an example e_i takes on a value a_k^i for every attribute A_k and labels that combination of values with a class label c_j^i. For convenience, the function which returns the label for example e_i is defined as:

$$\text{lbl}(e_i) = c_j^i. \tag{5.3.8}$$

□

One important thing to note is, that the tests are not usually pre-defined but calculated on the fly, during the execution of the tree induction algorithm. That doesn't necessarily restrict our abstract notion of such a generic framework however.

Given this framework, the objective of any tree induction algorithm is to find a function that predicts the class label for any new unseen examples. This implies of course that those are *unlabeled* examples.

Def. 5.3.5. Given $\mathcal{D} = \langle C, A, T, E \rangle$ as defined in Def. 5.3.4, let the goal function g, which predicts the classlabel for a given *unlabeled* example, be defined as:

$$g : \times_{i=1}^{|A|} A_i \to C.$$

□

5.3.4 Candidate tests

As discussed before there are multiple kinds of attributes, *continuous* or *discrete* resp. Each of those kinds impose a different kind of condition to test on.

Def. 5.3.6. Given $\mathcal{D} = \langle C, A, T, E \rangle$ as defined in Def. 5.3.4, let $A_k \in A$ be any attribute and let $t \in T$ be any test condition:

- A_k is *discrete* with n different values:
  - $t(\langle \ldots, a_k^i, \ldots, c_j \rangle) := [a_k^i = x]$: a test condition that tests on a single value $x \in A_k$.
  - $t(\langle \ldots, a_k^i, \ldots, c_j \rangle) := [a_k^i \in X_p]$: a test condition that tests if a single value a_k^i is contained in a proper subset of all values in A_k.

  where $a_k^i \in A_k$, $X_p \subset A_k$, $X_p \in X$ and $\bigcap_{X_p \in X} = \varnothing$.

Algorithm 6 Generating all test conditions for attribute A_k[24]

Precondition: S, the set of examples,
Precondition: A_k, the set of attribute values for attribute k,
Postcondition: set of generated tests T for attribute k.

1: **procedure** GENERATETESTCONDITIONS(S, A_k)
2: $T \leftarrow \varnothing$
3: **if** A_k is discrete **then**
4: **for all** $x \in A_k$ **do**
5: $T \leftarrow T \cup \{t(e_i) := [a_k^i = x]\}$
6: **else if** A_k is continuous **then**
7: $S^* \leftarrow$ sort S in ascending order according to the value of a_k
8: **for** $i \leftarrow 1$ **to** $|S^*| - 1$ **do**
9: **if** $\mathrm{lbl}(e_i^*) \neq \mathrm{lbl}(e_{i+1}^*)$ **then**
10: $\tau \leftarrow (a_k^i + a_k^{i+1})/2$
11: $T \leftarrow T \cup \{t(e_i) := [a_k^i \leq \tau]\}$
12: **return** T

- A_k is *continuous*:
$$t(\langle \ldots, a_k^i, \ldots, c_j \rangle) := [a_k^i \leq \tau].$$
a test condition that tests if a single value a_k^i is smaller than or equal to some threshold τ.

□

Def. 5.3.7. Given $\mathcal{D} = \langle C, A, T, E \rangle$ as defined in Def. 5.3.4, let $S \subseteq E$ be any training set of examples:

- Given any function $t \in T$, let $i_t : E \to \mathbb{N}$ be the function that assigns a number to any example $e_j \in E$ such that, that number equals the index of the corresponding index of the value v_i^t in the codomain V_i^t of function t:
$$i_t(e_j) = i \text{ where } t(e_j) = v_i^t.$$
- Given the function i_t the splits pertaining to any function $t \in T$ can be elegantly defined as: (note that applying the function t is implied by applying i_t)
$$\Pi(S, t) = \{S_1^t, S_2^t, \ldots, S_n^t\} \text{ where } S_i^t = \{e_j \in S \mid i_t(e_j) = i\}.$$
where Π is called the partition function, which partitions a set of examples into subsets according to some test or condition.

□

5.3.5 Impurity measures

Now that some methods of splitting any set S of examples is defined, the next problem to solve is; how is the quality of a certain split with regards to a test condition t determined? Before that can be answered, the following piece of information is required:

Def. 5.3.8. Given $\mathcal{D} = \langle C, A, T, E \rangle$ as defined in Def. 5.3.4, let $S \subseteq E$ be any training set of examples and let the relative frequency of classlabel $c_j \in C$ in S be defined as:

$$f_r(c_j, S) = p(c_j \mid L).$$

where $L = \{c_i \in C \mid e_i \in S \wedge c_i = \mathrm{lbl}(e_i)\}$ is the *sequence* of all classlabels in S.

□

Def. 5.3.9. Given $\mathcal{D} = \langle C, A, T, E \rangle$ as defined in Def. 5.3.4, let $S \subseteq E$ be any training set of examples, then a few well-known and/or widely used impurity measures are defined as:

Resubstitution error: Measures the fraction of incorrectly classified examples, when every example is assigned to the majority class in S:

$$I_R(S) = 1 - \max_{c_j \in C} f_r(c_j, S).$$

Gini-index: Intuition behind this index is, that it captures the notion of labeling every example in S randomly according to the statistical distribution of said labels in S. It then measures how often a random example from S would be incorrectly classified:

$$\begin{aligned} I_G(S) &= \sum_{c_j \in C} f_r(c_j, S)(1 - f_r(c_j, S)) \\ &= \sum_{c_j \in C} f_r(c_j, S) - f_r(c_j, S)^2 \\ &= \sum_{c_j \in C} f_r(c_j, S) - \sum_{c_j \in C} f_r(c_j, S)^2 \\ &= 1 - \sum_{c_j \in C} f_r(c_j, S)^2. \end{aligned}$$

Entropy: Measures the chaos, or rather, the *unpredictability*. Shannon denoted the entropy H of a discrete stochastic variable X with n outcomes $\{x_i\}_{i=1}^n$ as:

$$H(X) = \mathrm{E}(I(X)).$$

where E is the expected value operator and I is the *information content* of a stochastic variable. $I(X)$ is in itself another stochastic variable. $I(x_i)$ is called the *self-information* of an outcome of a stochastic variable X. The *lower* the probability $p(x_i)$ of outcome x_i the *higher* the self-information is of that outcome. Intuitionally; when an event with low probability

occurs, inherently, it carries much more information than an event which occurs very commonly. It is defined as:

$$\begin{aligned} I(x_i) &= \log\left(\frac{1}{p(x_i)}\right) \\ &= -\log_b\left(p(x_i)\right). \end{aligned}$$

The expected value of a stochastic variable is of course defined as:

$$E(X) = \sum_{i=1}^{n} p(x_i)x_i.$$

plugging the above all into the first one, the derivation of the formula for entropy follows quite trivially:

$$\begin{aligned} H(X) &= \sum_{i=0}^{n} p(x_i)I(x_i) \\ &= -\sum_{i=0}^{n} p(x_i)\log_b(p(x_i)). \end{aligned}$$

Plugging in that $p(c_j) = f_r(c_j, S)$ to derive the final formula:

$$I_E(S) = -\sum_{c_j \in C} f_r(c_j, S)\log_b(f_r(c_j, S)).$$

Note that $\log_b(0)$ is taken to be 0, which is consistent with $\lim_{x\to 0^+} x\log_b x = 0$.

□

Now that some impurity measures are defined upon a single set of examples, the last part that's missing is measuring the quality of a certain split. This enables the decision tree construction algorithm to find the split that, at that stage maximizes some information gain. From this point on the *entropy* function I_E will be used.

Def. 5.3.10. Given $\mathcal{D} = \langle C, A, T, E\rangle$ as defined in Def. 5.3.4, let $t \in T$ be any test condition, let $S \subseteq E$ be any training set of examples and let Π be the partition function defined in Def. 5.3.7. Then:

Gain measures the information gained in relation to the set S from which the splits were generated:

$$\Delta(S, t) = I_E(S) - \sum_{i=1}^{|\Pi(S,t)|} \frac{|S_i^t|}{|S|} I_E(S_i^t).$$

where $S_i^t \in \Pi(S, t)$.

Potential adjusts for the fact that, if each example gets split into a singleton set, it maximizes the gain measure. A trivial split like that is not a very generalizing split in nature and therefore unwanted:

$$P(S, t) = -\sum_{i=1}^{|\Pi(S,t)|} \frac{|S_i^t|}{|S|}\log_b\left(\frac{|S_i^t|}{|S|}\right).$$

where $S_i^t \in \Pi(S, t)$.

GainRatio measures the information gained in relation to the set S, but also takes into account the potential information from the partition itself:

$$\Delta_r(S,t) = \frac{\Delta(S,t)}{P(S,t)}.$$

Note that $\log_b(0)$ is taken to be 0, which is consistent with $\lim_{x\to 0^+} x \log_b x = 0$.

□

5.3.6 Tree induction

Almost all of the decision tree induction algorithms follow a very similar pattern. They differ mostly in the way the impurity of a set of examples is calculated, which kind of attributes are supported and how they handle *overfitting*.

The most well known decision tree induction algorithms are: CART, ID3 and C4.5. The most modern algorithm of those is C4.5 which is the succesor to ID3. Both are developed by Ross Quinlan[34].

Algorithm 7 Generic tree induction[24, 41, 36]

Precondition: S, the set of examples,
Precondition: A, the set of attributes,
Precondition: $k \geq 1$, the minimum amount of examples in a leaf node,
Postcondition: a decision tree model $\mathcal{M}$.

1: **procedure** CONSTRUCTTREE(S, A, k)
2: **if** $I_E(S) = 0$ **or** $|S| \leq k$ **or** $A = \varnothing$ **then** ▷ check stopping condition
3: create external node $leaf$
4: $leaf.label \leftarrow \arg\max_{c_j \in C} f_r(c_j, S)$ ▷ majority vote
5: **return** $leaf$
6: **else**
7: $T \leftarrow \varnothing$
8: **for all** $A_k \in A$ **do**
9: $T \leftarrow T \cup \{\text{GENERATETESTCONDITIONS}(S, A_k)\}$
10: $t_i^* \leftarrow \arg\max_{t_i \in T} \Delta_r(S, t_i)$ ▷ find best split condition
11: create internal node $root$
12: $root.cond \leftarrow t_i^*$
13: **for all** $S_j \in \Pi(S, t_i^*)$ **do**
14: $child_j \leftarrow \text{CONSTRUCTTREE}(S_j, A - \{A_k\}, k)$
15: add $child_j$ as descendant of $root$ with edge labeled as $v_j^{t_i^*} \in V_i^{t_i^*}$
16: **return** $root$

5.3.7 Classification

Once a model has been constructed, or induced of you will, classifying a new example is trivial. The example is just propagated along the correct edge, (i.e. the edge with the correct value of the test condition). Whenever a leaf node is reached, the example is classified as the class stored in that leaf node.

Algorithm 8 Classification[24, 41, 36]

Precondition: e_i, a fresh *unlabeled* example,
Precondition: T, the decision tree root node,
Postcondition: the classification label for example e_i.

```
1: procedure CLASSIFY(e_i, T)
2:     if T is a leaf node then
3:         return T.label
4:     else
5:         T_j ← descendant of T with edge labeled as the value of T.cond(e_i)
6:         return CLASSIFY(e_i, T_j)
```

5.3.8 Tree pruning

Because of the greedy nature of decision tree induction (trying to maximize the information gain at each step in the algorithm), *overfitting* needs to be accounted for.

An easy analogy for the phenomenon of overfitting is; telling, a little child, who is looking at a red ball, that the object is, in fact, a ball. By which the child now has learned that all red spherical objects are balls. Obviously the color of an object is not relevant to classifying a spherical object as a ball.

In a more formal way, overfitting could be explained as the resulting model being adapted too much to all the random errors or noise in the input data, instead of the underlying relationships.

There are several methods to counter overfitting during and after model construction. The former being; using statistics, during execution of the algorithm, to decide if a split is statistically significant. The problem with this however is, what if a future split, being a descendant of the current one, turns out to be really good? The algorithm will never discover the relationship because it has been *cut-off* too early.

Because of this reason, it is often better to decide upon removal of a certain split, after the model has been produced. Which is why the latter method is usually prefered, where the produced model is *pruned* from superfluous splits.

When the number of errors made by each leaf node is assumed to follow a binomial distribution, an upper bound to the expected training error can be calculated[41].

Def. 5.3.11. Given the number of examples N, the error rate e and confidence level α, then the statistical approximation of a binomial distribution with a normal distribution could be used to derive an upper bound to the error rate e as follows:

$$e_{upper}(N, e, \alpha) = \frac{e + \frac{Z^2_{\alpha/2}}{2N} + Z_{\alpha/2}\sqrt{\frac{e(1-e)}{N} + \frac{Z^2_{\alpha/2}}{4N^2}}}{1 + \frac{Z^2_{\alpha/2}}{N}}.$$

where $Z_{\alpha/2}$ is the quantile of the standard normal distribution.

□

α	0.5	0.3	0.25	0.2	0.1	0.05	0.98	0.01
$1-\alpha$	0.5	0.7	0.75	0.8	0.9	0.95	0.98	0.99
$Z_{\alpha/2}$	0.67	1.04	1.15	1.28	1.64	1.96	2.33	2.58

Table 2: Commonly used confidence level α to standardized Z-value mappings

The actual implementation of algorithm 9 on page 36 is a bit different than it is presented over here. This is because the fact that, in the actual implementation, the trees are represented with Prolog facts. It was impossible to remember certain parts of the tree. The memory usage would explode. Instead, every split is checked to see if it needs to be pruned. Then the tree is reconstructed, keeping the order of the original splits, using only those splits (which was much, much quicker in Prolog).

Algorithm 9 Tree pruning[24]

Precondition: T the decision tree root node,
Precondition: α the CF value for the appropriate confidence interval,
Postcondition: a new pruned decision tree rooted at T.

1: **procedure** PRUNETREE(T, α)
2: **if** T is a leaf node **then**
3: **return** $\mathrm{e}_{\mathrm{upper}}(T.num\_cases, T.num\_errors, \alpha)$
4: **else**
5: $T_j^* \leftarrow \arg\max_{\text{descendants } T_j \text{ of } T} T_j.num\_cases$
▷ the subtree with the highest proportion of cases
6: $E_{T_j^*} \leftarrow \mathrm{e}_{\mathrm{upper}}(T_j^*.num\_cases, T_j^*.num\_errors, \alpha)$
7: $E_L \leftarrow 0$
8: **for all** descendants T_j of T **do**
9: $E_L \leftarrow E_L$ + PRUNETREE(T_j, α)
10: $E_T \leftarrow \mathrm{e}_{\mathrm{upper}}(T.num\_cases, T.num\_errors, \alpha)$
11: **if** $\min\left\{E_L, E_T, E_{T_j^*}\right\} = E_L$ **then**
12: replace T with newly created external node L
13: **else if** $\min\left\{E_L, E_T, E_{T_j^*}\right\} = E_{T_j^*}$ **then**
14: replace T with subtree T_j^*

5.4 Tilde

Tilde[6, 5], invented by Blockeel and de Raedt, is a specific implementation of an ILP system, as discussed in section 4.3 on page 25. Tilde, is a top-down decision tree induction (TDIDT) algorithm, which will be discussed in further detail in this section.

5.4.1 Introduction

As was stated before, decision tree induction techniques are really popular, well known and succesfully applied to a lot of problems. Decision trees employ

a *divide-and-conquer* strategy. This is in sharp constrast to its competitors, rule-based systems, which are based on *covering* strategies.

Within the attribute value learning domain (such as described in section 5.3 on page 28), decision trees have been the far more popular approach however. Yet, in inductive learning and inductive logic programming settings, they've only been used in a couple of instances. The main reason for this is, that the representation of clausal formulas used in ILP systems don't necessarily adapt themselves properly to the underlying structures of decision trees.

Given the power and expressiveness of a well-established programming language like *Prolog*, it is easier to take those clauses and lift them into a factual representation in that language (e.g. rules in the form of Horn clauses).

Instead of the regular *propositional* tests on attribute values, the power of Prolog's (sub)goal oriented problem solving and backtracking, can both be used to their full potential.

Note that in the original paper, the authors propose to model an individual example as a collection of seperate Prolog facts. In this thesis, a single example will be modelled as a single Prolog fact. It makes coping with a large set of examples much clearer and easier. Furthermore, it corresponds much better to the framework that was laid out in section 5.3.3 on page 29.

The way examples will be specified instead is:

Def. 5.4.1. Given framework $\mathcal{T}$ as defined in Def. 5.4.4 on page 38, let $e_i \in E$ be any example and let $c_i \in C$ be the classlabel, then example e_i in Tilde will be specified as:

$$e_i = \mathrm{ex}(i, Case, c_i).$$

Where i is the identifier for lookup purposes, $c_i = \mathrm{lbl}(e_i)$ is the corresponding class label for e_i and $Case$ is any (first-order logic) literal modelling this particular example.

□

5.4.2 Logical decision trees

Def. 5.4.2. The structure of a binary decision tree (BDT) T is an extension to Def. 5.3.3 on page 29, such that T is either:

- a *leaf* node contains a class label c, referenced by $T.label$;
- or a $root$ node, which contains a test t, with two possible outcomes, referenced by $T.test$ and a subtree T_j for both outcomes of t, referenced by $T.left$ and $T.right$ respectively.

□

But, in order to turn a binary decision tree into a logical decision tree, some constraints need to be enforced:

Def. 5.4.3. A logical decision tree (LDT) is a binary decision tree (BDT), such that:

- every test t is a (first-order logic) conjunction of literals,

- a variable v that is introduced in a node n, cannot appear in in its right subtree.

□

Assuming the right subtree signifies failure of the test t, the last condition is necessary because every variable v is implicitly existentially quantified. It wouldn't make sense to refer to a newly introduced variable of a failed test. In other words, suppose X was the newly introduced variable. It wouldn't make sense to express something like "there is no such X that...". Variable X doesn't have a meaning further down the right subtree (i.e. a failed test can never result in a valid binding of all variables).

The Tilde algorithm shares a lot of functionality with the more generic version of decision tree induction discussion in section 5.3 on page 28. It could be viewed as a specific instance of those generic algorithms, with some minor but key differences.

In order to deal with those differences, the framework that was discussed in section 5.3.3 on page 29 needs to be expanded upon like so:

Def. 5.4.4. Given $\mathcal{D} = \langle C, A, T, E \rangle$ as defined in Def. 5.3.4, let the framework for first order logic $\mathcal{T} = \langle C, A, T, L, E \rangle$.

- Let L be any first order logic background theory.

where the test conditions $t \in T$ are implemented as:

$$t(e_i) = [e_i \wedge L \models \top].$$

□

With this framework now defined, the entailment constraint defined in Def. 4.3.1 on page 25, could be made more specific and defined as:

Def. 5.4.5. Given $\mathcal{T}$ as defined in Def. 5.4.4 on page 38, find a hypothesis H such that:

$$\forall e_i \in E : [e_i \wedge H \wedge L \models c_i \in C] \text{ and } [e_i \wedge H \wedge L \not\models c_k \in C - \{c_i\}]$$

where $\text{lbl}(e_i) = c_i$.

□

5.4.3 Refinement operator

One point where the Tilde algorithm differs from the original methods is the computation of the tests, that are to be put in a certain node. Tilde employs a so-called classical refinement operator ρ, under θ-subsumption.

Def. 5.4.6. Given two clauses c_1 and c_2, then c_1 θ-subsumes c_2 if and only if there is a substitution θ such that:

$$c_1\theta \subseteq c_2.$$

□

Def. 5.4.7. Given a refinement operator ρ under θ-subsumption, then it maps clauses onto sets of clauses, such that, for all clauses c and c', c θ-subsumes c', or more formally:

$$\forall c, \forall c' \in \rho(c) : c\theta \subseteq c'.$$

□

For the Tilde algorithm, a refinement operator ρ will be used that adds one or more literals to a test t. It is important to note that ρ is an input parameter to the algorithm. In effect the definition of the operator ρ determines the language bias, which will be discussed in the next section.

5.4.4 Language specification

As mentioned before, the specification of the refinement operator ρ and the language bias coincide. The language bias basically tells the Tilde algorithm which literals can be added to expand a certain test t. The original authors implemented mode and type specifications of which type conformity is not strictly necessary.

Mode specifications determine the restrictions on the variables in the newly added literal. They can be on of the following three;

- (+) or *in* means that the variable must be unified with an already existing variable;
- (*) or *in/out* means that the variable can, but needs not be unified with an existing variable;
- (-) or *out* forces a variable to be freshly introduced.

Def. 5.4.8. Given any ternary literal $p_i/3$, let n be the maximum number of times $p_i/3$ can be introduced along one single path in the tree. Then the specification of $p_i/3$ is defined as:

$$\mathrm{rmode}(n : p_i(+, *, -)).$$

Note that $n = -1$ means no limit on the maximal number of occurances of p_i. Also, this doesn't imply that the modes for all variables need to be like in this example. Finally, the arity of p_i in this example was randomly chosen to be 3, but it can be anything.

□

5.4.5 Discretization

As most decision tree induction algorithms, Tilde also supports *continuous* attributes. Most of those algorithms, dynamically calculate the appropriate values for the thresholds during the execution however (as shown in Alg. 6 on page 31). In Tilde this would imply, that for every refinement also all the appropriate thresholds need to be dynamically calculated.

Algorithm 10 Tilde tree induction[6, 5]

Precondition: S, the set of examples,
Precondition: Q, the current conjunction of literals,
Precondition: L, the language bias,
Precondition: $k \geq 1$, the minimum amount of examples in a leaf node,
Postcondition: a decision tree model $\mathcal{M}$.

1: **procedure** TILDE(S, Q, L, k)
2: **if** $I_E(S) = 0$ **or** $|S| \leq k$ **or** $L = \varnothing$ **then** ▷ check stopping condition
3: create external node $leaf$
4: $leaf.label \leftarrow \arg\max_{c_j \in C} f_r(c_j, S)$ ▷ majority vote
5: **return** $leaf$
6: **else**
7: $T \leftarrow \rho(L)$ ▷ generate all possible refinements from L
8: $t_i^* \leftarrow \arg\max_{t_i \in T} \Delta_r(S, Q \cup \{t_i\})$ ▷ find best split condition
9: create internal node $root$
10: $root.cond \leftarrow t_i^*$
11: $child_\top \leftarrow$ TILDE($S_\top, Q \cup \{t_i^*\}, L - \{t_i^*\}, k$)
12: $child_\bot \leftarrow$ TILDE($S_\bot, Q, L, k$)
13: add $child_{\{\top,\bot\}}$ as descendants of $root$ with edge labeled as $v_j^{t_i^*} \in V_i^{t_i^*}$
14: **return** $root$

Obviously this would lead to a large explosion of time-consuming repeated iteration over all the examples. In order to cut back on the *branching factor* of the tree and any redudant calculations to improve the performance, the thresholds values are pre-calculated in a single pass.

The discretization method implemented in Tilde is a simple modefication of an algorithm proposed by Fayyad and Irani[16]. The only difference applied in this implementation is that; the stopping criterion in their original method is very strict. Resulting in almost no calculated threshold values. In this implementation Tilde expects the number of required threshold values as an input value.

Def. 5.4.9. Given the specification of any literal p_i^a with arity a, as defined in Def. 5.4.8 on page 39, let this template indicate the need to discretize literal p_i^a:

$$\text{to\_be\_disc}(n : p_i^a).$$

Finally, let this template indicate which term t_j needs to be discretized:

$$\text{to\_be\_disc\_val}(p_i^a(t_1, t_2, \ldots, t_a), t_j).$$

Where $1 \leq j \leq a$. Note that this means that only one single term per literal can be discretized. If more terms in literal p_i^a would need to be discretized, they can be split up into multiple literals.

□

5.5 Model evaluation

After a model has been constructed, it's often desirable to evaluate its performance with regards to the world it's been modelled after. This is to make sure

that the model behaves correctly even in cases when the data fed into the model hasn't been observed before.

It's also useful to compare the resulting model to another model, which resulted from a similar but popular algorithm of which the correctness has already been proven (empirically). That way, a nice baseline can be established to analyze and contrast the performance of both models.

A model can be evaluated on the basis of a lot of criteria. It would take too much time and space to discuss them all. So, instead the now following criteria are used to evaluate the model.

5.5.1 Accuracy

The accuracy of a model expresses how well the model performs in terms of the prediction of the class labels of certain set of examples.

Def. 5.5.1. Given a set of examples E, let n_c be the number of examples that were classified correctly by model $\mathcal{M}$ and let n_t be the total number of examples in E. Then the accuracy A of model $\mathcal{M}$ is defined as:

$$A_{\mathcal{M}} = n_c/n_t.$$

□

To determine the confidence interval for the accuracy, a probability distribution needs to be established that governs the accuracy measure. One of the possibilities is modelling the classification task as a binomial experiment[41].

Def. 5.5.2. Given a set of examples E, a model $\mathcal{M}$ and its accuracy $A_{\mathcal{M}}$, let X be a binomially distributed stochastic variable that is the number of correctly predicted examples by $\mathcal{M}$, finally let $N = |E|$. Then the mean and variance of X are defined as:

$$\bar{X} = NA_{\mathcal{M}},$$
$$\sigma_X^2 = NA_{\mathcal{M}}(1 - A_{\mathcal{M}}).$$

□

Def. 5.5.3. Given the number of examples N, a model $\mathcal{M}$ and its accuracy $A_{\mathcal{M}}$ and confidence level α, the binomial distribution can be approximated by a normal distribution when N is sufficiently large, by using:

$$\mathrm{CI}_{A_{\mathcal{M}}}(N, A_{\mathcal{M}}, \alpha) = \frac{2NA_{\mathcal{M}} + Z_{\alpha/2}^2 \pm Z_{\alpha/2}\sqrt{Z_{\alpha/2}^2 + 4NA_{\mathcal{M}} - 4NA_{\mathcal{M}}^2}}{2(N + Z_{\alpha/2}^2)}.$$

where $Z_{\alpha/2}$ is the quantile of the standard normal distribution.

□

5.5.2 Kappa-statistic

Cohen's kappa-statistic[11] κ, is a coefficient to evaluate the agreement among *two* raters. In this case the agreement between the predictions of two seperate models or the agreement between the prediction of a model and the corresponding true observations. Note that the two raters are assumed to rate or rather predict statistically independently from one another. The kappa-statistic is supposed to correct for the hypothetical probability of agreement due to (random) chance.

In the past, others have expressed concerns regarding this statement. In reality this would suggest that if a rater isn't completely certain, the rater would just guess randomly. A more accurate measure should explicitly model the raters' decisionmaking process. In this case of comparing predictions and observations, it's not a relevant issue though. A classifier usually doesn't just guess randomly, it's guided by certain rules.

		Rater 2				
		$\Gamma_2 = 0$	$\Gamma_2 = 1$	$\cdots$	$\Gamma_2 = k$	
Rater 1	$\Gamma_1 = 0$	n_{00}	n_{01}	$\cdots$	n_{0k}	$n_{0\cdot}$
	$\Gamma_1 = 1$	n_{10}	n_{11}	$\cdots$	n_{1k}	$n_{1\cdot}$
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
	$\Gamma_1 = k$	n_{k0}	n_{k1}	$\cdots$	n_{kk}	$n_{k\cdot}$
		$n_{\cdot 0}$	$n_{\cdot 1}$	$\cdots$	$n_{\cdot k}$	n

Table 3: Confusion matrix

Def. 5.5.4. Given table 5.5.2 on page 42, let k be the number of different classes in C. Let $n_{c\cdot}$ and $n_{\cdot c}$ be the marginal frequencies for class $c \in C$, for the observations and predictions resp. Finally let n_{ij} denote the frequency of agreement or disagreement for observation i and prediction j. The marginal frequencies are the sum of the frequencies per row or column resp. Or more formally:

$$
\begin{aligned}
n_{c\cdot} &= \sum_{i=0}^{k} n_{ci}, \\
n_{\cdot c} &= \sum_{i=0}^{k} n_{ic}, \\
n &= \sum_{i=0}^{k} n_{i\cdot} = \sum_{i=0}^{k} n_{\cdot i} = \sum_{i=0}^{k}\sum_{j=0}^{k} n_{ij}.
\end{aligned}
$$

□

Def. 5.5.5. Given table 5.5.2 on page 42, let $\widehat{p_{ij}} = P(\Gamma_1 = i, \Gamma_2 = j)$ be the point-estimation of the probability of raters Γ_1 and Γ_2 being equal to outcome i and j respectively.

$$
\widehat{p_{ij}} = \frac{n_{ij}}{n}.
$$

□

Def. 5.5.6. Given table 5.5.2 on page 42, let $\widehat{p_o}$ denote the point-estimation of the probability of the observed agreement where the classification between two raters is equal. In the context of observations and predictions from a model, this is also often being described as *accuracy*.

$$\widehat{p_o} = \frac{\sum_{i=0}^{k} n_{ii}}{n}.$$

□

Def. 5.5.7. Given table 5.5.2 on page 42, let $\widehat{p_e}$ denote the point-estimation of the probability of agreement expected by chance. Which is calculated in terms of the marginal frequencies $n_{i\cdot}$ and $n_{\cdot i}$ like so:

$$\widehat{p_e} = \sum_{i=0}^{k} \left(\frac{n_{i\cdot}}{n}\right)\left(\frac{n_{\cdot i}}{n}\right) = \frac{1}{n^2}\sum_{i=0}^{k}(n_{i\cdot})(n_{\cdot i}).$$

□

Def. 5.5.8. Given $\widehat{p_o}$ and $\widehat{p_e}$ from Def. 5.5.6 and 5.5.7, the kappa coefficient is defined as follows:

$$\hat{\kappa} = \frac{\widehat{p_o} - \widehat{p_e}}{1 - \widehat{p_e}} \in [0, 1].$$

Note that $\hat{\kappa} = 0$ means no agreement and likewise $\hat{\kappa} = 1$ means total agreement.

□

Def. 5.5.9. Given table 5.5.2 on page 42, $\widehat{p_o}$ and $\widehat{p_e}$ from Def. 5.5.6 and 5.5.7, the standard devation of the kappa coefficient will approximate a normal distribution, according to Cohen, when n is large enough. Expressed in terms of $\widehat{p_o}$, $\widehat{p_e}$ and n the standard devation is given to be:

$$\sigma_\kappa = \sqrt{\mathrm{var}(\kappa)} = \sqrt{\frac{p_o(1-p_o)}{n(1-p_e)^2}}.$$

The corresponding confidence interval can be estimated using:

$$\mathrm{CI}_\kappa = [\kappa - \sigma_\kappa Z_{\alpha/2},\ \kappa + \sigma_\kappa Z_{\alpha/2}].$$

where $Z_{\alpha/2}$ is the quantile of the standard normal distribution.

□

5.5.3 Kappa-statistic interpretation

In table 5.5.3 on page 44 a classification of the κ-statistic, as proposed by Landis and Koch[27] is shown. Note however that their classification is chosen semi-arbitrarily.

κ	agreement
(0.00;0.20]	slight
(0.20;0.40]	fair
(0.40;0.60]	moderate
(0.60;0.80]	substantial
(0.80;1.00]	(almost) perfect

Table 4: Unofficial kappa-statistic classification by Landis and Koch

5.5.4 K-fold cross-validation

Cross-validation[23] is most commonly used as a method to assess the generalization capabilities of a particular model with regards to an independent data set. In a setting where the goal is to predict, cross-validation can be used to estimate how accurately a predictive model will perform in practice.

Cross-validation essentially partitions the data set into multiple complementary subsets, doing the model construction on all but one of the subsets (training-set) and the evaluation on the remaining last subset (test-set). To reduce the variance, this process is repeated multiple times (called the folds) using different subsets, where the individual evaluation results per fold are averaged or otherwise combined.

Algorithm 11 k-fold cross-validation[41, 36]

Precondition: E, set of examples
Precondition: $k \geq 1$, the number of folds,
Postcondition: the average accuracy over all folds.

```
1: procedure KFOLDCROSSVALIDATION(E, k)
2:     P ← random permutation of set E                ▷ shuffles examples
3:     divide P in k parts P_i as uniformly as possible
4:     A ← ∅
5:     for i ← 0 to k do
6:         M ← CONSTRUCTMODEL(P − P_i)                ▷ black box
7:         Acc_i ← EVALUATEMODEL(M, P_i)
8:         A ← A ∪ {Acc_i}
9:     return (∑ A)/k          ▷ average accuracy over all the folds
```

Commonly used types of cross-validation are:

k-fold cross-validation: Partitions the dataset into k subsets. Using $k-1$ subsets as training-set and the last remaining subset as the test-set for validation/evaluation purposes. This process is then repeated k times. The advantage of this method is that each example is used for both training and validation. Each example is used for validation exactly once. The evaluation results are usually averaged, although can they can be combined otherwise. See algorithm 11 on page 44 for the actual implementation.

2-fold cross-validation: In this case each example is used once for training and once for validation. The training-set and test-set are about

equal size.

10-fold cross-validation: The most common value for k is 10. This represents a good trade-off between bias, variance, training-set size (90%) and test-set size (10%).

n-fold cross-validation: Otherwise called *leave-one-out* cross-validation. Here k is equal to n, the number of examples in the dataset. The size of the test-set is always 1, using the rest of the examples as a training-set (hence the name). Obviously, this is really expensive because of the large number of folds. Even more so when the model construction algorithm is really expensive to execute.

stratified k-fold cross-validation: Only different in the sense that the training- and test-set both contain an approximately equal proportion of all class labels.

repeated random sub-sampling validation: Instead of splitting the examples into a training- and test-set according to some value k, they are split randomly. Essentially removing the dependence for the proportions on the value of k. The disadvantage though, is that there is no guarantee that each example is used once during the validation. Some examples may even be used more than once.

Other uses of cross-validation besides model evaluation, are:

model selection: Selection of a model with the most optimal evaluation score. Note that no assumptions are mode about the origins of a model.

feature selection: Starting with all attributes or none at all, respectively. Then, in recursion removing or introducing attributes as long as the performance of the resulting model goes up. Usually this is called *backward* or *forward* feature selection, respectively.

5.6 Practical problems

During the implementation of the whole project, I ran into so many issues. Many of them unforseen. In this section I will try to explain all the difficulties I had to face.

The first problem was (and still is) getting a top notch quality poker data set. Preferably one with data collected from a real money tournament. When a large amount of money is at stake, players tend to play more seriously and therefore will the data of the players be much more reliable (i.e. no constant blind all-ins).

Two major problems will present itself when looking for a poker dataset:

1. A good quality poker dataset costs quite a lot of money;
2. Almost all of them have missing values, these often stem from not having recorded players folding their cards, which makes it nearly impossible to create a reliable model. These examples cannot be trown away however, as doing so will introduce a major bias in the data. Remember that often only good hands reach the showdown phase. A sophisticated algorithm would need to be designed to fill in those missing values according to

some model (which I didn't have time for, it requires a lot of computational power and time as well).

On *pokerftp.com*, people offered a large amount of poker data, which was purposefully scrambled to prevent player modelling and identification, the one thing a lot of poker researchers are trying to achieve.

In order to get the pokerdata, a lot of personal data had to be submitted to them in order for them to verify that the poker data would only be used for scientific ends. Soon after discovering this website, I found out that those people completely stopped maintaining this dataset and their corresponding website, leaving me with nothing.

Through several contacts, we were able to recover the dataset, but at this point in time it was too late to do anything meaningful with the data. It was also stored in a custom fileformat, which needed a library to open and read. This library was also hosted on *pokerftp.com* which was already taken down by that time.

The second big issue was that, the software I was going to use (Poker Academy Pro), which our university has a valid license for, stopped working. It seemed their license validation server was taken down. After inspection I found out that the website of Poker Academy Pro was completely taken down as well. Leaving me with nothing again.

At the start of my thesis project everything seemed to be working and in order. I didn't assume that everything I would need in the near future would be completely taken down behind my back. This was all a very frustrating experience.

Finally, I had decided to use JIProlog as the prolog engine to implement the Tilde algorithm. The big issue during the implementation of the Tilde algorithm in JIProlog was, that a lot of necessary features had to be implemented by myself.

After doing some custom tests, near the end of the implementation in JIProlog, it turned out, that JIProlog just couldn't handle the massive amount of input data.

During execution, it started allocating and leaking huge amounts of memory. Eventually crashing by raising an out-of-memory exception. This forced me to translate the complete implementation to SWI-Prolog, which luckely turned out to be much more efficient and much faster.

6 Results

6.1 Dataset

6.1.1 Introduction

The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by Sir Ronald Aylmer Fisher[17]. This data set became really popular testing case for classification algorithms in machine learning.

The dataset contains 150 examples of 3 different species of the *Iris* flowers. Each of those 3 different species contains 50 examples, so every species is equally well represented. The task is to discriminate between the 3 species on

the bases of 4 attributes namely; *petal length*, *petal width*, *sepal length* and *sepal width*. All of those are expressed in centimeters.

Figure 5: Iris-setosa, Iris-versicolor and Iris-virginica

6.1.2 Statistical analysis

In this section the statistical analysis of the Iris dataset will be presented. The scatter plots show all the 3 species plotted against each combination of 2 attributes. Notice the very high class correlation of petal length and width indicating a high predictability.

species	**num**
Iris Setosa	50
Iris Versicolour	50
Iris Virginica	50
total	150

Table 5: Iris class distribution

(in cm)	**min**	**max**	**mean**	**std.dev.**	**class correlation**
sepal length	4.3	7.9	5.84	0.83	0.7826
sepal width	2.0	4.4	3.05	0.43	-0.4194
petal length	1.0	6.9	3.76	1.76	0.9490
petal width	0.1	2.5	1.20	0.76	0.9565

Table 6: Iris data statistics

6.2 Evaluation

Te results of running the Tilde algorithm will be compared against the C4.5[34] implementation in Weka[48] called J48. The same settings were used for both algorithms. The minimum amount of examples in a leaf was set to 4. The confidence interval for tree pruning was set to 0.25, which is the default value.

Both algorithms ran with 10-fold cross-validation enabled. The best model from any of the 10 folds was selected and was applied to the whole dataset.

From the results it's easy to see that this dataset is not one of the "hardest" around. But it does prove that both algorithms perform about equally well.

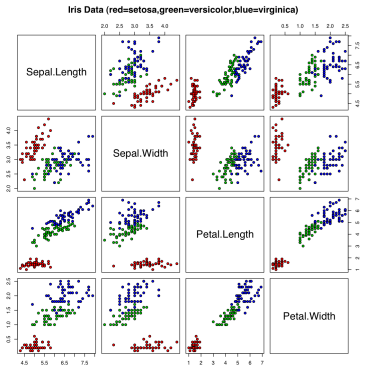

Figure 6: Iris data scatterplots

From the resulting decision trees, it's easy to see that both algorithms almost picked the same set of splits and thresholds.

The scatterplots confirm that the Setosa species is linearly seperable from the other two species. Only the Virginica and Versicolor species show overlapping features. Therefore those two are the hardest to discriminate and every model will make some prediction errors because of that.

```
petal_length > 2.6:  Iris-setosa
petal_length <= 2.6
|   petal_width > 1.75
|   |   petal_length > 4.9: Iris-virginica
|   |   petal_length <= 4.9: Iris-versicolor
|   petal_width <= 1.75
|   |   petal_length > 4.9: Iris-virginica
|   |   petal_length <= 4.9: Iris-versicolor
```

Verbatim 1: Best model from running Tilde

```
petal_width <= 0.6: Iris-setosa
petal_width > 0.6
|   petal_width <= 1.7
|   |   petal_length <= 4.9: Iris-versicolor
|   |   petal_length > 4.9: Iris-virginica
|   petal_width > 1.7: Iris-virginica
```

Verbatim 2: Best model from running J48

7 Conclusion

In this thesis a view was presented on opponent modelling using inductive logic programming and decision trees in the form of the Tilde algorithm, im-

	num	percent
correctly predicted	144	96.32%
incorrectly predicted	6	3.68%
accuracy	0.9632	-
accuracy CI (95%)	[0.9197;0.9836]	-
kappa	0.9448	-
kappa std.dev.	0.0230	-
kappa CI (95%)	[0.918;1.008]	-

Table 7: Statistics for the resulting model from J48

		Prediction			
		$\Gamma_2 = a$	$\Gamma_2 = b$	$\Gamma_2 = c$	
Observation	$\Gamma_1 = a$	49	1	0	50
	$\Gamma_1 = b$	0	47	3	50
	$\Gamma_1 = c$	0	2	48	50
		49	50	51	150

Table 8: Confusion matrix for the resulting model from J48

	num	percent
correctly predicted	146	97.3%
incorrectly predicted	4	2.7%
accuracy	0.973	-
accuracy CI (95%)	[0.9334;0.9896]	-
kappa	0.96	-
kappa std.dev.	0.0197	-
kappa CI (95%)	[0.921; 0.999]	-

Table 9: Statistics for the resulting model from Tilde

		Prediction			
		$\Gamma_2 = a$	$\Gamma_2 = b$	$\Gamma_2 = c$	
Observation	$\Gamma_1 = a$	50	0	0	50
	$\Gamma_1 = b$	0	49	1	50
	$\Gamma_1 = c$	0	3	47	50
		50	52	48	150

Table 10: Confusion matrix for the resulting model from Tilde

plemented using the prolog programming language. A lot is still open to improvement.

During the implementation of this project I stumbled upon so many problems which prevented my progress or my ability to present or generate the results I was hoping to be able to present. Because of that reason I (sadly)

didn't succeed in reaching my goals that I set in section 3.2 on page 21.

However, I still strongly feel that the approach taken by Ponsen et al. is a very elegant one and definately seems promising! Research still has to show how "viable" the approach would be in a No-limit poker setting with multiple players, though. Nevertheless, in my opinion, this approach definately is worth looking into more.

Also, it seems there is a conflict of interests in the poker playing community and the poker researching community. Poker players, playing for real money in tournaments or casinos want to keep their money making secrets, for obvious reasons. Whereas poker research would benefit so much from professionals acting as domain experts, providing crucial knowledge about the (meta)game. Poker players hate to be idea of being analysed through collected data, while in fact the poker research community really needs the data. It is sad to see that websites providing such data or analysis tools are taken down left and right.

From the results it is clear that there are a lot of possibilities still left to be explored. And from other results it shows that the Tilde algorithm could be a viable approach to modelling opponents in any imperfect information game.

Poker is an amazing game and it provides such a rich universe of possibilities to playing it. It is a great testing environment for artificial intelligence and therefore I sincerely hope that somebody takes this as an inspiration to further improve the state of computer poker and don't let themselves be discouraged by the enormous complexity of the game.

8 Future work

Research on Poker data is far from done. There is so much left to explore (and exploit). I think that the most important thing for the evolution of opponent modelling in poker is having a top quality poker data set (with no missing values) open to public use. So many machine learning algorithms are dependent on having existing data to learn from.

The MCTS algorithm is essentially guided by a very generic principle, which balances between exploration and exploitation in a certain way. Even though many more action selection functions exist, it would be interesting to see if domain knowledge could be added into the equation. It would also be interesting to see if that results in a more broad or more focused game tree expansion. Also, there's been research done on UCT combined with progressive bias[9]. Which adds domain knowledge into UCT.

Instead of creating one single model for a generic poker player, clustering[42] could be applied to the poker data to discover meaningful clusters of poker playing strategies. Then those could be used as prior distributions to adapt to new players, prividing the new player can easily be classified as belonging to one of those clusters. It would be interesting to see if the initial clustering does make a significant difference in learning and or prediction.

The Tilde algorithm relies on a user supplied language bias (i.e. the definition of the refinement operator). Maybe the language bias could be automatically generated based on further analysis of the input data. Additionally, the Tilde algorithm could be adjusted to handle missing values as the C4.5 algorithm already does. Weighted examples could also be introduced to create an explicit bias in the input data.

9 Bibliography

[1] MCTS Java Implementation, 2012. `http://mcts.ai/?q=code/simple_java`.

[2] Darse Billings. Computer poker. *Artificial Intelligence*, 134:2002, 1995.

[3] Darse Billings and Darse Billings. Algorithms and assessment in computer poker. Technical report, 2006.

[4] Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Opponent modeling in poker. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 493–499, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[5] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[6] Hendrik Blockeel and Luc De Raedt. Top-down induction of logical decision trees. In *Artificial Intelligence*, 1997.

[7] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[8] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[9] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap Van Den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation (NMNC)*, 4(03):343–357, 2008.

[10] Ugo Chirico. JIProlog v3.0.3-1 SP1, 2007. `http://www.ugosweb.com/jiprolog`.

[11] J Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[12] Mehdi Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[13] Aaron Davidson. Using artificial neural networks to model opponents in texas hold'em. 1999.

[14] Aaron Davidson, Darse Billings, Jonathan Schaeffer, and Duane Szafron. Improved opponent modeling in poker. pages 493–499. AAAI Press, 2000.

[15] J.A. Davidson and University of Alberta. Dept. of Computing Science. *Opponent modeling in poker: learning and acting in a hostile and uncertain environment*. University of Alberta, 2002.

[16] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pages 1022–1029, 1993.

[17] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(7):179–188, 1936.

[18] BioTools Inc. Meerkat API, 2003. `http://www.poker-academy.com/community.php`.

[19] BioTools Inc. Poker Academy Pro v2.5, 2003. `http://www.poker-academy.com`.

[20] ISO. *ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core*. International Organization for Standardization, Geneva, Switzerland, 1995.

[21] ISO. *ISO/IEC 13211-2:2000: Information technology — Programming languages — Prolog — Part 2: Modules*. International Organization for Standardization, Geneva, Switzerland, 2000.

[22] Levente Kocsis and Csaba Szepesvri. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[23] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. pages 1137–1143. Morgan Kaufmann, 1995.

[24] Ron Kohavi and Ross Quinlan. Decision tree discovery. In *IN HANDBOOK OF DATA MINING AND KNOWLEDGE DISCOVERY*, pages 267–276. University Press, 1999.

[25] Kevin B. Korb, Ann E. Nicholson, and Nathalie Jitnah. Bayesian poker. In *UAI*, pages 343–350, 1999.

[26] J. Ladyman. *Understanding philosophy of science*. Routledge, 2002.

[27] J R Landis and G G Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[28] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[29] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[30] J.F. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

[31] John F. Nash. Equilibrium points in n-person games. *Proc. of the National Academy of Sciences*, 36:48–49, 1950.

[32] D.R. Papp and University of Alberta. Dept. of Computing Science. *Dealing with imperfect information in poker*. University of Alberta, 1998.

[33] Marc Ponsen, Geert Gerritsen, and Guillaume Chaslot. Integrating opponent models with monte-carlo tree search in poker. In *Proceedings of Interactive Decision Theory and Game Theory Workshop at the Twenty-Fourth Conference on Artificial Intelligence (AAAI-10)*. AAAI press, 2010.

[34] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[35] Luc De Raedt, editor. *Inductive Logic Programming, 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers*, volume 5989 of *Lecture Notes in Computer Science*. Springer, 2010.

[36] Stuart J. Russell and Peter Norvig. *Artificial intelligence - a modern approach: the intelligent agent book*. Prentice Hall series in artificial intelligence. Prentice Hall, 1995.

[37] T.C. Schauenberg and University of Alberta. Dept. of Computing Science. *Opponent modelling and search in poker*. University of Alberta, 2006.

[38] John R. Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3:417–424, 1980.

[39] Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. Bayes bluff: Opponent modelling in poker. In *In Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI*, pages 550–558, 2005.

[40] J.A. Stankiewicz and M.P.D. Schadd. Opponent modelling in stratego. Technical report, Department of Knowledge Engineering, Maastricht University, 2009.

[41] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[42] A.A.J. van der Kleij. Monte-carlo tree search and opponent modeling through player clustering in no-limit texas hold'em poker. 2010.

[43] Jan Wielemaker. SWI-Prolog v5.11.x, 1990. `http://www.swi-prolog.org`.

[44] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[45] Jan Wielemaker. *Logic programming for knowledge-intensive interactive applications*. PhD thesis, University of Amsterdam, 2009.

[46] Wikipedia. Game theory — wikipedia, the free encyclopedia, 2012. `http://en.wikipedia.org/w/index.php?title=Game_theory`.

[47] Wikipedia. Poker — wikipedia, the free encyclopedia, 2012. `http://en.wikipedia.org/w/index.php?title=Poker`.

[48] Ian H. Witten, Eibe Frank, Len Trigg, Mark Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical machine learning tools and techniques with java implementations, 1999.

[49] Michael J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.)*. Wiley, 2009.