Predicting Player Strategies in Real Time Strategy Games

Frederik Frandsen (ffrand09@student.aau.dk)
Mikkel Hansen (mfha09@student.aau.dk)
Henrik Sørensen (hsoere01@student.aau.dk)
Peder Sørensen (psoere09@student.aau.dk)
Johannes Garm Nielsen, (jniels06@student.aau.dk)
Jakob Svane Knudsen, (jknuds05@student.aau.dk)
Aalborg University, Denmark

December 20, 2010

Aalborg University
Department of Computer Science

TITLE:
Predicting Player Strategies in Real Time Strategy Games

PROJECT PERIOD:
September 1^{st}, 2010
December 21^{st}, 2010

PROJECT GROUP:
d516a

GROUP MEMBERS:
Frederik Frandsen
Mikkel Hansen
Henrik Sørensen
Peder Sørensen
Johannes Nielsen
Jakob Knudsen

SUPERVISORS:
Yifeng Zeng

NUMBER OF COPIES: 8

TOTAL PAGES: 53

SYNOPSIS:

This paper examines opponent modeling in the real-time strategy game StarCraft. Actual game replays are used to identify similar player strategies via unsupervised QT clustering as an alternative to relying on expert knowledge for identifying strategies.

We then predict the strategy of a human player using two well-known classifiers, artificial neural networks and Bayesian networks, in addition to our own novel approach called Action-Trees. Finally we look at the classifiers' ability to accurately predict player strategies given both complete and incomplete training data, and also when the training set is reduced in size.

CONTENTS

CHAPTER

ONE

INTRODUCTION

To win in a multiplayer game a player should not only know the rules of the game, but knowledge about the behaviour and strategies of opponents is also important. This is a natural thing for human players, but in commercial video game AI agents this is only an emerging trend. Most video game AI agents only use rules to achieve victory. To improve the competitiveness of AI agents in games one approach uses opponent modelling to create models of the opponent that describe their behaviour in order to predict their next move.

Opponent modelling can be described intuitively as figuring out how an opponent thinks or behaves. This could for instance be getting into the mindset of an enemy to predict where on a road they would ambush a military convoy[6] or figuring out an opponent's hand while playing poker based on his behaviour. A Texas Hold 'em Poker program called Poki[4] that makes use of opponent modelling has been playing online Poker since 1997, and has consistently yielded small winnings against weak to intermediate human opponents.

Perhaps somewhat surprisingly, the game RoShamBo, also called Rock-Paper-Scissors, has also been shown to benefit from opponent modelling. Programs playing this game have been submitted annually to the International RoShamBo Programming Competition, where the programs using opponent modelling placed consistently higher in the competitions than those using rule-based decision making[4]. In fact RoShamBo provides a classic example of the motivation of opponent modelling.

The optimal strategy for both players is randomly selecting rock, paper or scissors, resulting in wins, losses and draws that each make up a third of players' game outcomes. However, if one player knows that his opponent has a tendency to pick rock more than the other two options, the player can use this knowledge and pick paper more often, which would make that player win more often.

1.1 Opponent Modelling in Video Games

Video game environments are often more complex with more variation in possible states than classical game environments. Thus, creating an entertaining video game AI opponent becomes difficult as it must respond to an unpredictable players behaviour in a large environment of many possible state variations[26].

Even so, some commercial games do use opponent modelling. A recent example is the First-Person-Shooter Left 4 Dead. In this game four players fight through various types and amounts of zombies to reach their goal. An "AI director" is responsible for placing these zombies in the game by observing each player's stress level to determine the best type and amount of zombies to create. Here opponent modelling is used not by each individual zombie opponent, but by the AI director as a way of providing players with an enjoyable and challenging game experience[5].

Use of opponent modelling in Real Time Strategy (RTS) video games has been the subject of previous study[12, 3, 1, 20, 16, 19] and Herik et al.[26] stipulates that, as video games become even more complex in the near future and as the AI opponents' inability to account for unpredictable player behaviour will become more and more apparent, the need for adaptive AI opponents and

opponent modelling will increase accordingly.

Modern RTS games are prone to changes in the metagame, i.e.: the trends in the strategies used by players. The complexity of RTS games and patches released after the initial release makes for an unstable, evolving metagame for most RTS games. As an example, the developers of an RTS might determine that a certain strategy is too dominant and is harmful to the gameplay experience, and then decide to tweak certain aspects of the gameplay to alleviate this situation. A recent example of this is Blizzard Entertainment's StarCraft II, which underwent a balancing patch[7] that resulted in new strategies and major changes to existing player strategies. In response, competitive players may come up with other ways of playing as the previously dominant strategy may no longer be viable, hence changing the metagame.

This example shows that relying on predetermined opponent models is arguably undesirable. A method for generating opponent models from previously observed player behaviour could accommodate the mentioned changes in the metagame, thus maintaining an appropriate level of challenge for the players.

Alexander[1] and Mehta et al.[20] explores the use of manually annotated traces from human-played games to teach an AI opponent how to adapt to changing player behaviours as an alternative to scripting the AI and Sander et al.[12, 3], Houlette [16] suggests a method to create a player model to assist an AI or the player in making improved strategic decisions. Weber et al.[19] have studied methods for utilizing datamining approaches to replay traces of games involving human players to help adaptive AIs recognize patterns in the opponents. However, Sander et al.[12] and Weber et al.[19] use supervised labelling of replays, thus assuming that the number of possible strategies is known beforehand. Spronck et al.[3] uses datamined replays to improve their previously developed novel approach to Case-based Adaptive AI.

1.2 Project Goals

In this project we aim to implement and test three classification methods, naive Bayes classifiers, multi-layer perceptrons and our own ActionTrees, to predict the strategy of a player playing a real-time strategy game. We use the game StarCraft as a basis for our research and base our predictions on data extracted from replays of real players playing StarCraft.

We use QT clustering as opposed to expert knowledge when identifying and labeling these player strategies from replays, where a player strategy is a certain unit and building composition for a player at a given time in the game.

QT clustering only groups strategies according to their unit and building compositions at one time in the game, but we want to predict at an early time in the game which of these strategies a player is following. For this we use the three classification methods, trained on the previously labelled strategies.

Finally we evaluate the prediction accuracy of each of these three classifiers to determine in which situations each classifier is used best.

1.3 The Process Pipeline

In order to predict strategies in games of StarCraft, we need to do a series of different tasks from extracting the raw data from StarCraft replays to encoding them in our formats described in chapter 2. In Figure 1.1 the pipeline of the entire process is shown.

In the figure rounded boxes are software processes, and rectangular boxes are the outputs of a prior process. The pipeline consists of the following steps:

1. While replaying a game we dump the build orders and gamestate sequences as raw text to be able to cluster the data in the later steps.

2. The raw text build orders and gamestate sequences are then parsed into a suitable class structure, yielding unlabelled build orders and gamestates.

3. The QT clustering algorithm is then run on these, where similar strategies are grouped together and assigned a label.

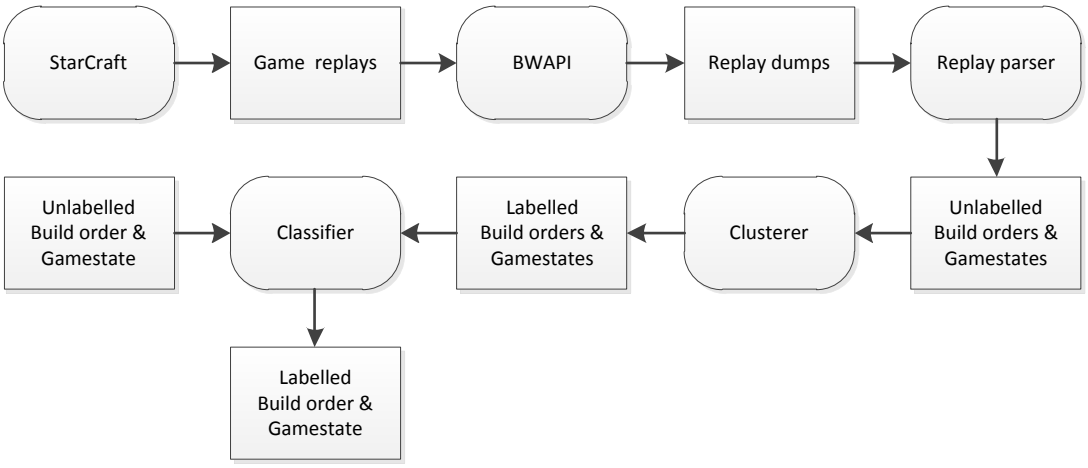

Figure 1.1: The intended pipeline for the process of predicting StarCraft strategies.

4. The three classifiers, naive Bayes, multi-layer perceptron and ActionTree, are then trained from the labelled strategies.

5. Finally each classifier is tested on a new gamestate or build order, where the classifier outputs the most likely strategy for this gamestate or build order.

We later describe this process pipeline in more detail, along with defining gamestates and build orders more formally, and explaining the QT clustering method and the three classifiers listed above. An overview of where these are described can be found in the following section.

1.4 Report overview

We now give an overview of the remainder of this paper and explain the contents of each chapter.

In chapter 2, we present the relevant aspects of StarCraft as a game and define what a strategy in StarCraft is. Additionally we define an abstraction of the game that will be used as a basis for our later work.

Chapter 3 describes how we identify player strategies using the clustering technique Quality Threshold clustering. We also briefly look at how to decide which factors we need to look at to evaluate a clustering to see whether it is better or worse than others.

In chapter 4 we present our three methods to predict player strategies, Bayesian networks, artificial neural networks and ActionTrees. Each of these methods are briefly motivated and how they work described.

Chapter 5 contains the evaluation of our prediction methods. The results are discussed for each of the methods and finally they are compared to each other, and a final result overview on their effectiveness will be given.

Finally in chapter 6 we conclude on the project as a whole and discuss the results of our investigations into opponent modelling and prediction of player strategies using clustering and classifiers as predictors.

CHAPTER

TWO

STARCRAFT AS A PROBLEM DOMAIN

For opponent modelling we need a game with a large user base, a lot of viable strategies, the ability to extract information from recorded games, and a large pool of available recorded games. StarCraft fits all these criteria, and as such is perfect platform for opponent modeling.

StarCraft is a popular real time strategy game (RTS) created by Blizzard Entertainment released in 1998, it ended up becoming the best selling PC game of the year with more than 1.5 million copies sold[24], reaching over 11 million copies by February 2009[13]. Later in 1998 an expansion pack called Brood War was released by Blizzard, which added three new single player campaigns, new units and game balance changes for multiplayer games. Blizzard has continually supported StarCraft with new balance and bug fix updates to keep competitive play alive.

StarCraft is an economically focused RTS game, with three different resources, minerals and gas that are required for the production of units and buildings, and supply which limits the size of the players army. The game has 3 unique playable races, previous RTS games either used virtually identical races or 2 reasonably distinct races. StarCraft is a game of imperfect information this is represented by using "Fog of War", that is, players can only observe the parts of the game's map where they have a unit or building granting vision. The player has to concurrently balance multiple tasks in order to achieve victory such as resource gathering, allocation, army composition, map and terrain analysis, controlling units individually to maximise damage given and minimise damage received, deciding when and where to engage the enemy, scouting to determine what the opponent is doing and reacting to that information. All of this in a real time environment, where a couple of seconds inattention can decide the match while the opponent is doing his best to do the exact same thing to you. Academic AI researchers are showing increasing interest in RTS game AI development, as it presents many challenges that have to be solved for a AI player to be competitive with a skilled human player. In connection with the AI and Interactive Digital Entertainment (AIIDE) 2010 Conference hosted a StarCraft AI competition[2] with 4 different competition levels for AI players. There have been some previous work on opponent modelling in StarCraft such as [19], where the authors datamine game replays, use an expert defined rule set to assign strategy labels, and then test different classification methods to predict player strategies in replays. Hsieh and Sun [17] use StarCraft replays to predict a single player's strategies using Case Based Reasoning.

StarCraft has the ability to record so called replay files that contain every action performed by the players in a multiplayer game. The recorded replay can then be used to rewatch the game later on, or shared with others. It quickly became popular to watch replays from tournaments and games played by high level players, both has entertainment and as a learning tool. In order to create more advanced AI players, the community has created an API to hook into the game called BWAPI[25] (Broodwar API). This API also enables users to extract any desired data from the game engine.

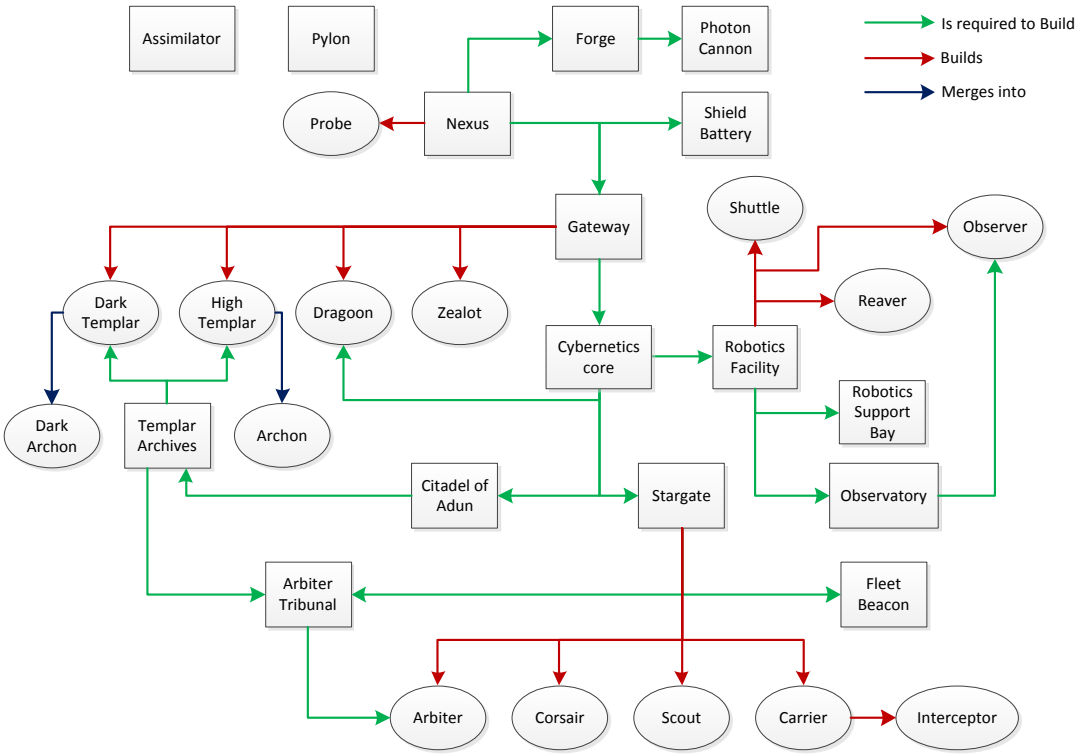

Figure 2.1: The tech tree for the Protoss race in StarCraft

2.1 Strategies in StarCraft

In StarCraft a player's strategy is a combination of a chosen game opening and reacting to the opponent's actions. The openings are a frequently discussed topic in the StarCraft community and has a significant impact on the success in a game. In the StarCraft community these openings are called build orders, which simply describes the order of actions a player should take in the beginning of the game.

The order in which units and buildings can be built in StarCraft is well-defined as different "tech trees" for each race. In order to gain access to more advanced units a player has to construct a building that grants access to that particular unit.

For example, before a player playing the race Protoss can build a basic melee unit called a Zealot, the player needs to have a building called a Gateway completely built. If that player instead wants to build a more advanced ranged combat unit called a Dragoon, that player will also need another building called a Cybernetics Core completely built.

The tech tree can be expressed as a directed acyclic graph. The tech tree for the Protoss race can be seen in Figure 2.1. The decision to focus on building a lot of basic units, expanding your base to improve your resource gathering rate, or focusing on exploring paths through the tech tree to build more advanced units, is the linchpin of StarCraft strategies.

It is these strategies that we wish to predict for an opponent; if a player can learn his opponent's strategy he will be more likely to find a counter-strategy and win the game. But to predict these strategies we need a formal model we can use to algorithmically determine an opponent's strategy.

2.2 Defining StarCraft Formally

To reason about the game state in StarCraft we need to formalize the game in a manner based on these game states. One such definition can be the following complete definition of the game as a transition model:

Definition 1 (StarCraft as a transition model)**.** *We define a game as a tuple* $(P, \{A_i\}, S, T, \{G_i\})$*, where*

- P *is the set of participating players.*
- A_i *is the set of actions available to player* $i \in P$*.*
- S *is the set of gamestates.*
- T *is a transition model which gives us, for any state* $s \in S$*, any player* $i \in P$ *and any action* $a \in A_i$ *a new state* $s' = T(s, i, a) \in S$*.*
- G_i *is the set of goal states for player* $i \in P$*. Player* i *wins the game if it reaches a state in* G_i*.*

However, the enormous amount of actions and the complexity of the transition function makes this definition very difficult to work with. Essentially the state space is too large to be reasoned about. Thus it is necessary to make an abstraction over the gamestate space.

Since the full state of a StarCraft game is very complex, we treat the transition model as a black box and concern ourselves with observations about a subset of the game; that is, army composition without regard for the combat between and positioning of units. Thus we define a more simple formalization of the game as a sequence of gamestates:

Definition 2 (State Sequence)**.** *A state sequence is a sequence of observations on the form* (t, p_1, p_2) *where*

- t *is the elapsed time since the game started.*
- *each* p_i *contains the current amount of minerals, gas, supply used and the amount of units of each unit type and buildings of each building type player* i *has in his possession.*

As such, an observation describes the state of the game at a given time in terms of the units and buildings each player has. If we have a game of StarCraft where both players play Protoss and the game is at the 25 second mark the observation will be

$$(25, \{Minerals = 64, Gas = 0, Supply = 5, Probe = 5, Nexus = 1\},\\ \{Minerals = 72, Gas = 0, Supply = 5, Probe = 5, Nexus = 1\})$$

with entries of 0 units or buildings omitted here for brevity.

A related form of observation is the build order, which records the build actions of a player. In this form of observation we capture the order in which units and buildings are constructed, which is not shown explicitly in state sequences.

Definition 3 (Build Order)**.** *A build order for player* i *is a sequence of tuples* (t, a) *where*

- t *is the elapsed time since the game started.*
- a *is an action from the set of available build actions, which contains a build action for every unit type and building type.*

Note that the build orders described here differ slightly from the build orders used in the StarCraft community that use supply rather than time to indicate build action ordering. An example build order using our definition could be

$$\{(0, BuildProbe), (30, BuildPylon)\}$$

The definitions of state sequence and build order form the basis for the information we will extract from StarCraft, to be used in predicting player strategies. The process of extracting these state sequences and build orders from the game are described in the following.

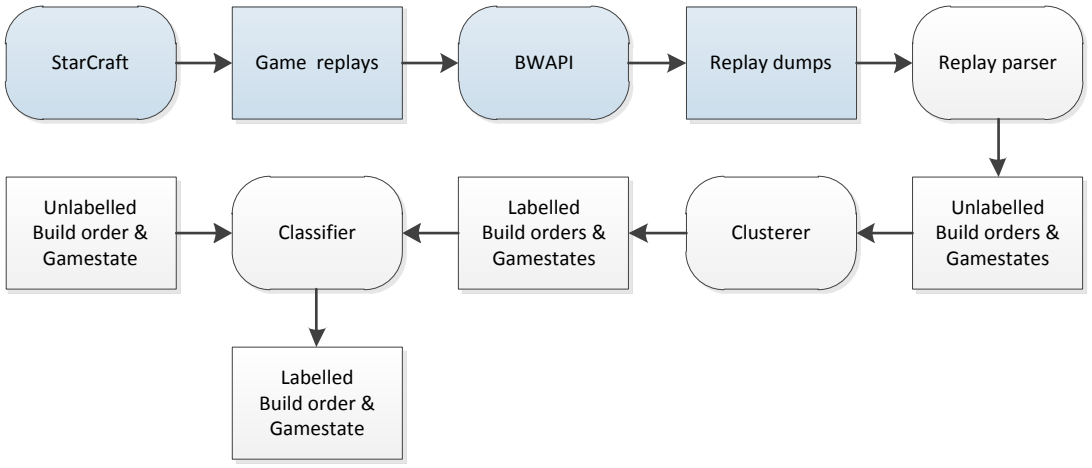

Figure 2.2: The process pipeline (also seen in Figure 1.1) with the steps involving gathering usable data from StarCraft replays highlighted.

2.3 Gathering Data From StarCraft

To gather data from StarCraft, we use replays from competitive play between real players. We extract state sequences and build orders from these game replays using the Brood War API. This information is then stored in readable text files. This is also shown highlighted in our process pipeline in Figure 2.2.

The process of extracting both build orders and state sequences takes a large amount of time, as the replays cannot be read directly and instead must be simulated through StarCraft. This made an intermediate data format necessary, in order to facilitate the next steps in the pipeline.

We have used BWAPI[25], an API to hook into StarCraft mentioned in section 2. Using this API a data extractor was written to output game state sequences and build orders from StarCraft replays.

We have chosen to only record replays with a single race from the game, the race Protoss. The reasoning is that this simplifies the later implementations, as the number of different units and buildings is cut down significantly in addition to not having to take into account unique features of three different races. We postulate that if it is possible to predict strategies for a single race, it is possible for the others as well.

In the next chapter we continue the process by parsing these replay dumps to our clusterer to identify possible similar player strategies.

CHAPTER

THREE

IDENTIFICATION OF PLAYER STRATEGIES

In this chapter we identify different player strategies. We previously described how to extract replay data from StarCraft and interpreting it as unlabelled build orders and game state sequences. The next step is to identify groupings of these and label them as part of a player strategy using the clusterer, as seen highlighted in Figure 3.1.

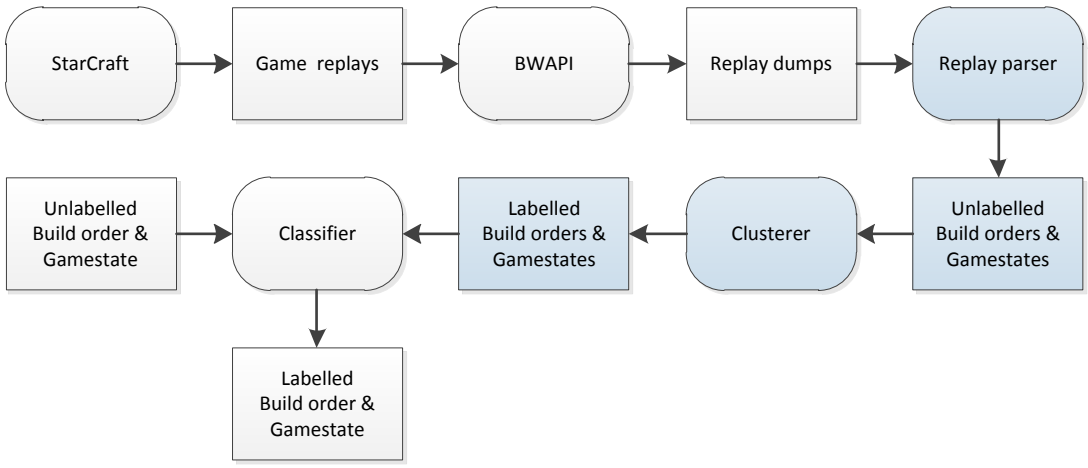

Figure 3.1: The process pipeline (also seen in Figure 1.1 and Figure 2.2 with the steps involving clustering highlighted.

A commonly used approach to this strategy identification is to manually define the recognizable strategies such as in [1, 20]. Another approach is to construct a set of rules that define possible strategies, and criteria for when a player is using a given strategy according to these rules.

However, doing so would necessitate the reinvolvement of experts should the set of known strategies need to be changed, such as if a new strategy surfaces or the game is altered in some fashion. When manually identifying strategies large amounts of data can however quickly become a hindrance, and when using rulesets experts may not always be able to translate their knowledge directly into clearly defined rules.

If an automated system could be made to identify the strategies currently employed by players it might be an alternative to continuous expert involvement. We propose to use Quality Threshold clustering with a game-specific distance metric to find similar strategies in StarCraft instead of using an expert to identify them. This section covers the reasoning in choosing QT clustering for the purpose of finding player strategies, the proposed distance metric used by the QT clustering as well as the quality metrics we employ and an evaluation of the performance of the methods.

3.1 Classification of Strategies

The goal of the classification of player strategies is to find the different strategies that is found normally in StarCraft. It is however not possible to say exactly how many of these strategies exist, or how much variance there can be between two different strategies, as strategies are continually evolving.

As discussed in Chapter 2.1 there is a large focus on the first few minutes of the game and the strategies emerging from this. We wish to classify player strategies into groups that reflect this. There are several arguments for only looking at the first part of a game instead of the entire game.

- In the StarCraft community discussions on build orders and strategies are often focused on the initial part of the game.
- Many games never get past the initial part of the game, because players use strategies focusing on rapidly ending the game.
- Any mid or late game strategy will depend on the opening chosen by the player.
- The player may change his strategy radically depending on his opponents strategy later in the game.
- Beyond the initial part of the game, the player begins to lose units and buildings making it more difficult to reason about the game state. Our intuition is that when a player starts losing units or buildings in the game, he is likely to divert from his initial opening strategy and instead start counter strategies against his opponent.

We wish to find strategies that are as long as possible without the factors listed above factoring in too much. The most notable point is the last - the amount of units lost. We will examine how far into the replays measure before most players start to lose units or buildings.

We use the data previously extracted from the StarCraft replays, specifically the game state sequences, to divide them into different strategies. We wish to label the game state sequences and build orders from earlier and therefore we need to expand their definitions to include a label for each build order and game state sequence.

Definition 4 (Labelled build orders and state sequences)**.**

- *A labelled build order is a build order annotated with a label identifying its strategy.*
- *A labelled state sequence is a state sequence annotated with a label identifying its strategy.*

To create and identify the labels of the data we have decided to use clustering, specifically QT clustering described in the next section. At a certain time into all recorded StarCraft games we recorded the game state of the player, and according to a distance metric on these amounts determine how different two players' strategies are from each other.

We will use a candidate game state from each replay. The information in these candidates is assumed to be the player's goal using his strategy. As mentioned earlier we wish to find this candidate as far into the game as possible to have more diverse compositions of units and buildings in the candidates. However it is not enough to be able to identify a strategy at only one time in the game, therefore we need to be able to reason over all the game states leading to the candidate game state.

The further we push the candidate game state, the longer the interval we can predict during. This will make it possible to predict as the game progresses and inform the player of the opponent's most probable strategy as the accuracy of the predictions increase.

The exact description of a strategy is not a primary concern in this project, as our focus is to predict what units and buildings a player will have at a point in the future and not on analysing how that players strategy works. We will however compare the unit compositions in each cluster in an effort to evaluate their quality or usefulness in this purpose.

Function $QT\_Clust(G, d)$
Input: A set of data points G to be clustered, and a maximum diameter of the clusters d.
Output: A set of clusters.
if $|G| \leq 1$ **then**
  return G
end
foreach $i \in G$ **do**
  $flag \leftarrow true$
  $A_i \leftarrow \{i\}$ // A_i is the cluster started by i
  while $flag = true$ *and* $A_i \neq G$ **do**
    find $j \in (G - A_i$ such that $diameter(A_i \cup \{j\})$ is minimum
    if $diameter(A_i \cup \{j\}) > d$ **then**
      $flag \leftarrow false$
    else
      $A_i \leftarrow A_i \cup \{j\}$ // add j to cluster A_i
    end
  end
end
identify set $C \in \{A_1, A_2, ..., A_{|G|}\}$ with maximum cardinality
return $C \cup$ QT_Clust$(G - C, d)$

Algorithm 1: Quality Threshold Clustering [15]

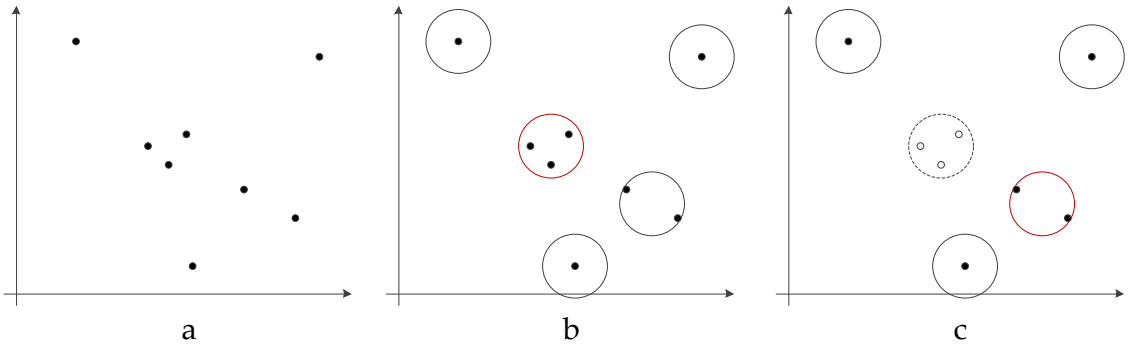

Figure 3.2: QT clustering example: a) Unclustered data. b) The red cluster has the highest cardinality. c) QT_Clust is called again with the previous cluster removed from G.

3.2 Quality Threshold Clustering

Quality Threshold clustering, uses a distance metric and a diameter value to determine clustering. In Algorithm 1 the pseudo code for QT clustering can be seen, and how this algorithm works is shown in Figure 3.2.

In Figure 3.2a the data is unlabelled, and in Figure 3.2b the algorithm determines the distance from every data point to every other data point, as dictated by the distance metric, and saves it as a set if the distance metric finds it to be within a certain diameter. In Figure 3.2c the set with the highest number of other data points within its diameter is then stored as a cluster and removed from the dataset, and the remainder of the data points are then treated in the same manner.

QT clustering has a number of attractive properties when applied to the problem domain of RTS games. Namely, it does not require an injection of expert knowledge dictating the number of opening books to be identified. Clustering methods that do require the number of clusters to be specified essentially require expert knowledge regarding the number of different opening strategies a player could employ. However a suitable diameter value needs to be determined in order for QT clustering to distinguish strategies.

3.3 Distance Metric

The distance metric needs to capture the difference between two opening strategies. We have selected to use the following definition of weighted composition distance between two gamestates in StarCraft:

Definition 5 (Weighted Composition Distance)**.** *Let U be the set of all unit and structure types in StarCraft. Given a player's gamestate S and $u \in U$, then u_S is the amount of units of type u the player owns in gamestate S.*

Now, given a weight function $w : U \to \mathbb{R}$ and player gamestates a and b, the weighted composition distance $d(a, b)$ between a and b is given by

$$d(a,b) = \sum_{u \in U} w(u)|u_a - u_b|$$

For our identification of player strategies, the weight function $w(u)$ used for weighted composition distance is defined as

$$w(u) = m_u + 2g_u$$

where m_u is the mineral cost of unit type u and g_u is the gas cost of that unit type. The reasoning for using unit costs as weighting is that the strategy a player employs will be visible through the units and buildings that player has allocated resources to produce.

To illustrate the distance between two game states a and b an example of the calculation can be seen below.
$A = (150, Minerals = 64, Gas = 0, Supply = 8, Probe = 8, Nexus = 1)$
$B = (150, Minerals = 72, Gas = 0, Supply = 7, Probe = 7, Nexus = 1, Pylon = 1)$

As stated in the definition we only look at the units and buildings of the particular game states. As each unit and building is not of equal importance and cost in the game we multiply with its cost in the game. We then have a vector containing the weighted values of each unit and building. The distance is then calculated using Manhattan distance of this. The distance between game state A and B is then:

$$\begin{aligned} d(a,b) = (|Probe_B - Probe_A|, |Nexus_B - Nexus_A|, |Pylon_B - Pylon_A|) &\cdot (Probe_{cost}, Nexus_{cost}, Pylon_{cost}) = \\ (|7-8|, |1-1|, |1-0|) &\cdot (50, 400, 100) = \\ (1, 0, 1) &\cdot (50, 400, 100) = 150 \end{aligned}$$

3.4 Determining the Time when Candidate Game States are Chosen

We want to make predictions about the furthest future possible. As stated previously the problem is that players do not fully dictate their own unit composition as they suffer losses from enemy attacks and may be forced to counter specific occurrences in the game, making them deviate from their original plan.

We found that less than 16 percent of the sampled players have lost at least 3 units 360 seconds into the game. 70 percent of all sampled players had managed to build a Cybernetics Core 270 seconds into the game. The cybernetics core is important as it unlocks many branches of the tech tree, and shortly after completion the player will very likely decide on a specific strategy.

Thus we choose to classify strategies based on the unit composition 360 seconds into the game, deeming that at this time most players have not suffered significant loses, and have had time enough to build any of the tech buildings beyond cybernetics core which is a significant tell as to which strategy the player will choose.

3.5 Measuring QT Clustering Parameter Quality

To determine a good diameter to use in QT clustering we use two different metrics to measure the goodness of a clustering.

Variable Error

As mentioned earlier, we will later use the clusters to predict the unit composition of a player by associating him with a cluster and assuming that he is aiming for a unit composition that is representative of that cluster.

This approach is only sound if the clustering method produces a clustering where the points in the same cluster more or less agrees on the values in the individual variables. What we want to see is that the value of each variable does not deviate much from the mean in the cluster. One way to measure this is using an Variable error calculation shown below.

Definition 6 (Variable error)**.** *Given a point i belonging to a cluster with the centroid j, then the variable error e_{i_a} for point i on variable a is defined as*

$$e_{i_a} = abs(i_a - i_a)$$

Variable Error Test Results

Figure 3.3 shows the sum of squared error values for each cluster on each variable. We can observe that the large errors occur on the variables with high mean values, which are to be expected. We note however that there is a low average error on buildings that advance the player through the tech tree and implicitly gives the strategy that the player is using.

Silhouette Coefficient

Players in any cluster should also have a unit composition that differs from the unit compositions of players outside the cluster and is similar to that inside the cluster. The silhouette coefficient is a metric to calculate the average goodness of a cluster[23].

Definition 7 (Silhouette coefficient)**.** *Given a point i and a set of clusters CS Then the silhouette coefficient for i is*

$$s_i = \frac{b_i - a_i}{max(a_i, b_i)}$$

Where a_i is the average distance to every other point contained in its cluster:

$$a_i = \frac{1}{|C_i \setminus i|} \sum_{j \in C_i \setminus i} dist(i, j)$$

and b_i is average distance to the closest cluster.

$$b_i = min_{C \in CS \setminus C_i} \frac{1}{|C|} \sum_{j \in C} dist(i, j)$$

where $C_i \in CS$ is the cluster containing i.

The silhouette coefficient ranges between -1 and 1, approaching 1 as the difference between b_i and a_i increases when $b_i \geq a_i$ and reaching a negative value when $a_i \geq b_i$. A high positive value of the mean silhouette coefficient in a cluster indicates that the cluster is more dense than the area around it. In essence a positive value is good, and a negative value is bad.

Silhouette Test Results

We have tested the average silhouette coefficient for each cluster, for diameter 4000 these silhouettes are as shown in Figure 3.4.

The silhouettes are acceptable for almost all clusters except the largest one which contains over half the data points, but we ignore this fact and continue regardless. Other settings have shown themselves to have a worse silhouette value.

Figure 3.3 shows the mean values for each cluster on each variable. We can observe that the large errors occur on the variables with high mean values, which are to be expected. We note however that there is a low average error on buildings that advance the player through the tech tree and implicitly gives the strategy that the player is using.

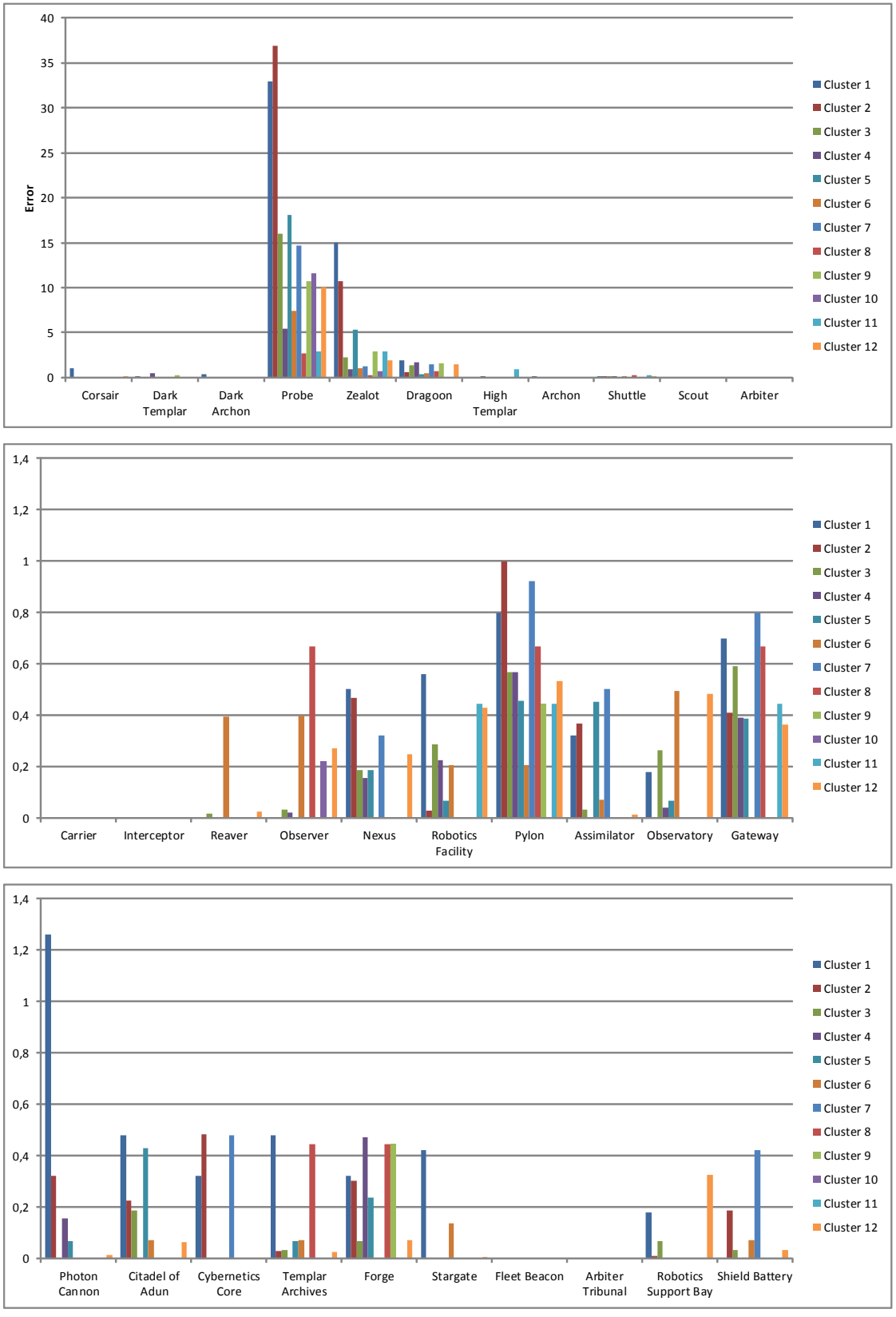

Figure 3.3: Sum of Squared Error on each variable

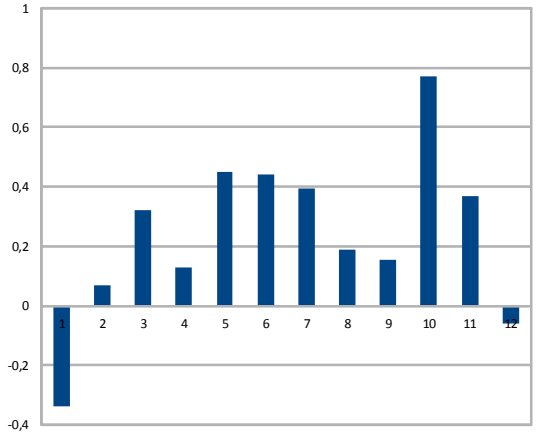

Figure 3.4: Average silhouette coefficient for each cluster.

The choice of diameter 4000 was a compromise between the number of clusters QT clustering produces and the variable error and silhouette coefficient for each cluster produced. In the case of smaller diameters QT clustering produced a very large amount of different clusters which we determined to be very unlikely. We chose to limit the number of labels and 4000 proved itself to have an acceptable silhouette and variable error.

3.6 The Strategies Identified by Clustering

QT clustering, using a diameter of 4000 and game states at 360 seconds into the game from each state sequence, produced 12 clusters. One of these, cluster 1, is a special cluster that contains points from small clusters that would otherwise only contain two points. We combined these small clusters due to the reasoning that two points do not describe a tendency, which has reduced the number of clusters from 17 to 12.

The 12 clusters contain a varied amount of data points, and their number of contained points can be seen in Figure 3.5.

The distribution seen suggests that there are two very popular strategies; cluster 12 which contains roughly half of the data points, and the second most popular cluster 2 that contains a quarter of the data points. There is a disproportionately large amount of data points in two of the clusters, which may be explained by the fact that players have had a very long time to gain a good understanding of the game, resulting in a small number of very strong strategies being very popular.

The remaining clusters contain a very small amount of data points; for example four clusters contain only three data points each. These must represent certain uncommon strategies that have been tried in the games, and this could be players experimenting with new strategies.

Now the strategies found in the StarCraft replays have been extracted and clustered together, with labels assigned to these clusterings. Our work is far from done, however; identification of player strategies only grouped strategies together according to players' unit and building composition at time 360 seconds.

To make actual predictions instead of simply grouping strategies together we need more advanced methods of classifying new players' strategies as one of the 12 labels we have just identified. These predictions will allow us, for any given time in the game up to 360 seconds, to select the strategy label most closely resembling a new player's strategy.

Once this strategy label is known we can use one of the identified strategies under that label as a model of our new opponent. And with a model of our opponent we can, if we so choose, exploit any weaknesses in his strategy to better challenge or defeat him in that game of StarCraft.

In the next chapter we look at three methods for making these predictions using the already labelled gamestates and build orders, combined with a new player's build order and gamestates.

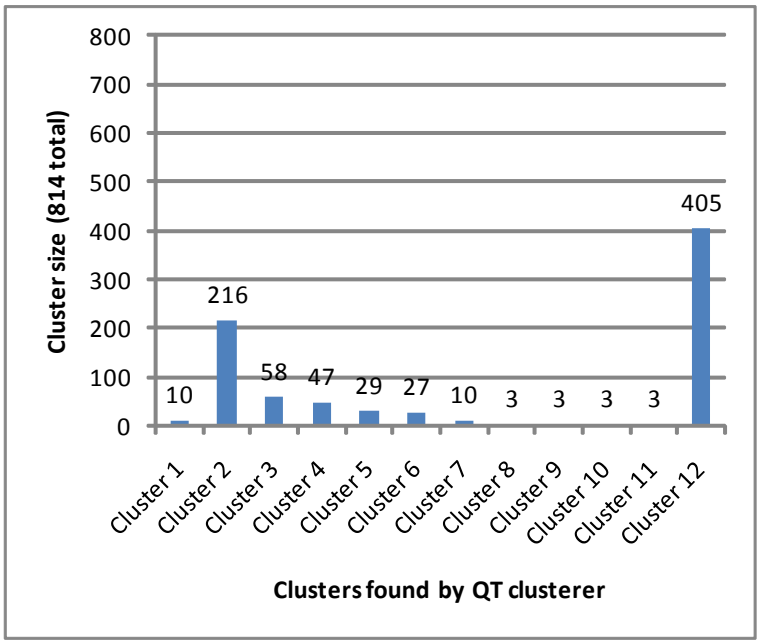

Figure 3.5: The distribution of data points in the different strategies.

CHAPTER
FOUR

PREDICTING PLAYER STRATEGIES

In this chapter we describe several classification approaches and how they can be used to predict player strategies in StarCraft based on their build orders and gamestate sequences. We have already identified and labelled existing build orders and state sequences from collected replays as described in the last chapter, and are now at the final stage of the pipeline seen in Figure 4.1.

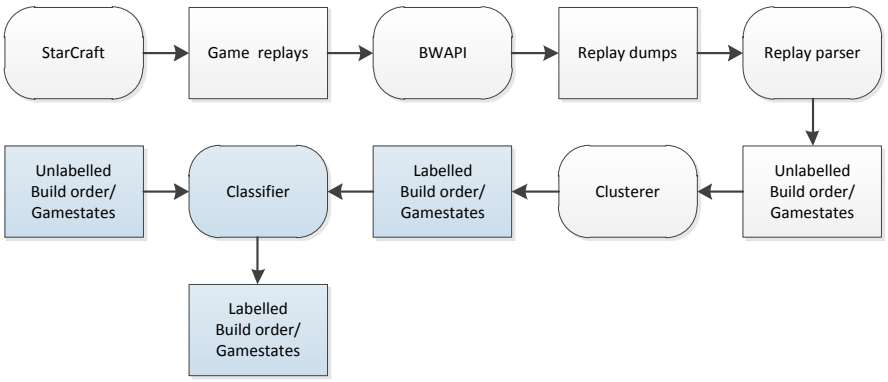

Figure 4.1: Process pipeline with the strategy prediction steps highlighted in blue.

We explain the three distinct classification methods we can use for player strategy prediction: Bayesian network classification, artificial neural network classification, and the novel ActionTree classification method.

All of these three methods follow a common approach to classification; first the classifier is trained with a training set of labelled build orders or gamestates, then it is tested by giving it an unlabelled build order or gamestate and having it attempt to label the build order or gamestate by using the model it has learned from the training set. The purpose of opponent modelling is in our case to predict the strategy an opponent player is following. Once this prediction has been found the AI player will be able to follow a counter-strategy to better defeat or challenge the opponent.

Artificial neural networks and Bayesian networks are two existing classifiers that have been used extensively in AI research already, and we will attempt to apply these to the domain of StarCraft by classifying its gamestates.

Additionally we propose our own ActionTree classification as an alternative for player strategy prediction that classifies a sequence of a player's actions rather than single gamestates. We believe the order of players' actions in StarCraft has a substantial impact on their strategy and hope that this method captures this better than the prior two methods.

4.1 Bayesian Networks

The first classification method we examine is based on Bayesian networks. A Bayesian network is a model that is particularly well-suited to illustrating dependencies in domains with inherent

uncertainty.

Bayesian networks reason over the probability of seeing one thing given a number of related features. A small example is the following to discern the number of units an opponent might have constructed. In Figure 4.2 the relationship between the number of soldiers an RTS video game player owns given their amount of barracks, armouries and quality of weapons can be seen as a Bayesian network.

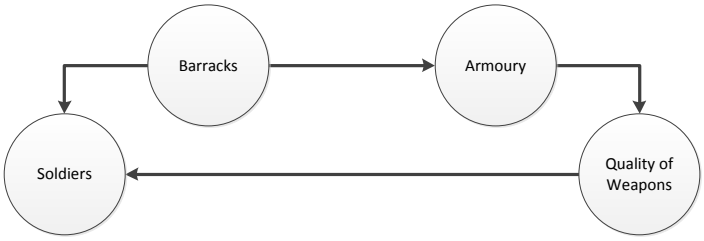

Figure 4.2: An example Bayesian network.

In figure 4.2, the number of soldiers a player has depends on how many barracks are present, but also on the player's quality of soldier weapons that in turn depends on how many armouries the player owns. The structure of the network thus shows a relationship between the different features of the network. We now give a more formal definition of Bayesian networks[18].

Definition 8 (Bayesian Network)**.** *A Bayesian Network consists of the following:*

- *A set of variables and a set of directed edges between variables.*
- *Each variable has a finite set of mutually exclusive states.*
- *The variables together with the directed edges form an acyclic directed graph (DAG).*
- *To each variable A with parents B_1,...,B_n a conditional probability table $P(A|B_1, ..., B_n)$ is attached.*

Often Bayesian networks are used as causal models assisting decision problems in domains of uncertainty, but they can also be used as models to support reasoning in classification problems. In this case the Bayesian network has a set of feature variables, one for each feature inherent in the object to be classified, and a class variable with a state for each class the object can be classified as.

However, to reason about anything an important step must be taken to calculate the parameters of the network, called parameter learning. For very large networks with a large number of states this can easily become intractable. When Bayesian networks are used for classification this is usually the case.

If the Bayesian network is constructed such that a lot of nodes are connected to the classification node, the state space of the classification node will become undesirably large. Instead a very simple Bayesian network structure can be used in place of more complex networks that allows for much simpler parameter learning calculations, called a naive Bayes classifier.

4.1.1 Naive Bayes Classifier

As stated above, in many cases Bayesian networks can become intractably large, for example when a class variable is dependent on a large number of feature variables. In the game StarCraft we look at a very large number of feature variables with a large number of states (31 variables, each with many states). To avoid very large probability tables and calculations we can instead use naive Bayesian classifiers. Naive Bayes classifiers are a special type of simple structure Bayesian network. In naive Bayes classifiers each feature variable has the class variable as their only parent.

The naive Bayes classifier is therefore easy to implement and is often the method preferred by machine learning developers according to Millington & Funge[21], since most other classifiers have not been shown to perform much better than Naive Bayes.

In the case where the class variable is the parent of all feature variables and no other connections are allowed, the assumption that all feature variables are completely independent from each other, would intuitively mean an information loss from the modelled problem domain. However, the method usually performs well despite this intuition[18]. As these restrictions already give the complete structure of naive Bayes classifiers, the only thing that needs to be learned is the parameters of the features given each class.

Parameter Learning

The parameter learning of a naive Bayes classifier is quite simple. Since each feature variable only has one parent the probability calculations will all be in the form

$$P(\text{FeatureVariable} \mid \text{ClassVariable})$$

which gives the probability of each state in a feature variable conditioned on the states in the class variable.

To clarify this we will give a small example of the learning of a naive Bayes classifier Consider the following set of feature variables

$$F = \{\text{Barracks, Armoury, Quality Weapons}\},$$

where each variable in F has a binary state of $\{yes, no\}$, and the class variable

$$C = \{Soldiers\}$$

that also has a binary state $\{few, many\}$.

As stated earlier naive Bayes classifiers are simplified models, as not all feature variables are likely to be completely unrelated; Quality Weapons is likely to depend on the player actually having an armoury to produce quality weapons, but this is not modelled in the naive Bayes classifier The structure of this example classifier can be seen in Figure 4.3.

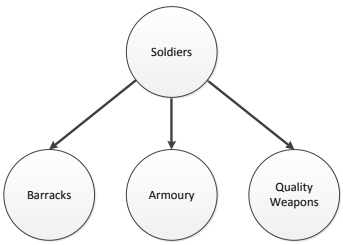

Figure 4.3: A simple naive Bayes classifier

The probability calculations are quite simple; the probability tables will always have only two dimensions in naive Bayes classifiers. The probability table for the feature variable Armoury can be seen in Figure 4.1.

		Soldiers	
		few	*many*
Armoury	*yes*	0,7	0,3
	no	0,3	0,7

Table 4.1: The probability table of the Armoury feature variable.

Following the principle of maximum likelyhood estimation each value is calculated from statistics, i.e. the number of cases in the training data in which the variable Barracks is *yes* and the label for this case is *many*. This gives the probability that if the player owns a barracks it is like that he has many soldiers. An example of such training data can be seen in table 4.2.

Barracks	Armoury	QualityWeapons	Soldier
yes	no	no	many
yes	yes	no	many
yes	yes	yes	few
...	...	...	...

Table 4.2: A training set for the naive Bayes classifier example.

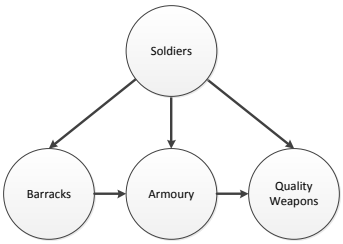

Figure 4.4: An example of a possible TAN structure.

4.1.2 Tree Augmented Naive Bayes Classifier

The Tree Augmented Naive Bayes classifier (TAN) is an extension to naive Bayes classifiers. The extension is in the structure of the network, which now permits a tree structure among the feature variables where each feature variable can have one parent feature variable beside the class variable. This should have no impact when using complete information, however since the TAN structure can reason about the likely state of feature variables if no evidence is given based on the state of other feature variables, it could potentially perform better under partial information.

The downside to this extension is that the tree structure is not given, so one must use expert knowledge or a structure learning algorithm, like the Chow-Liu algorithm described by Jensen and Nielsen[18], to create this structure before classification is possible. Probability calculations are fortunately still relatively simple, as probability tables in each variable only have up to three dimensions.

An example of a TAN for the RTS example can be seen in Figure 4.4. Here we can see the TAN structure captures the RTS example's conditional dependence between Armoury and Barracks, as well as Quality Weapons and Armoury, as was not the case for the naive Bayes classifier.

4.1.3 Bayesian Networks and StarCraft

In this section we apply the naive Bayes and TAN classifier concepts to the StarCraft problem domain and show the resulting models used by these two classifiers. The naive Bayes classifier uses the model shown in Figure 4.5, where each of the 31 feature variables depends only on the class variable. The class variable *Label* is coloured red, all buildings in StarCraft are coloured green, and units are coloured blue. Additionally, if not shown in the figure, all feature variables have an implicit dependency on the class variable *Label*.

The model used by the TAN classifier is shown in Figure 4.6. This model has been built using the structure of the tech trees in StarCraft, rather than using automatic structure learning. These tech trees impose dependencies between certain units and buildings inherent in StarCraft which the naive Bayes classifier model fails to encapsulate.

These two Bayesian network classifiers will be evaluated in the next chapter, along with the two other classification methods to be described in the following sections.

4.2 Artificial Neural Networks

ANNs are useful for classification when the domain is complex, as ANNs approximate a model of the domain over several iterations without requiring expert supervision. An ANN is a mathematical model inspired by biological neural networks and is often simply referred to as a neural network.

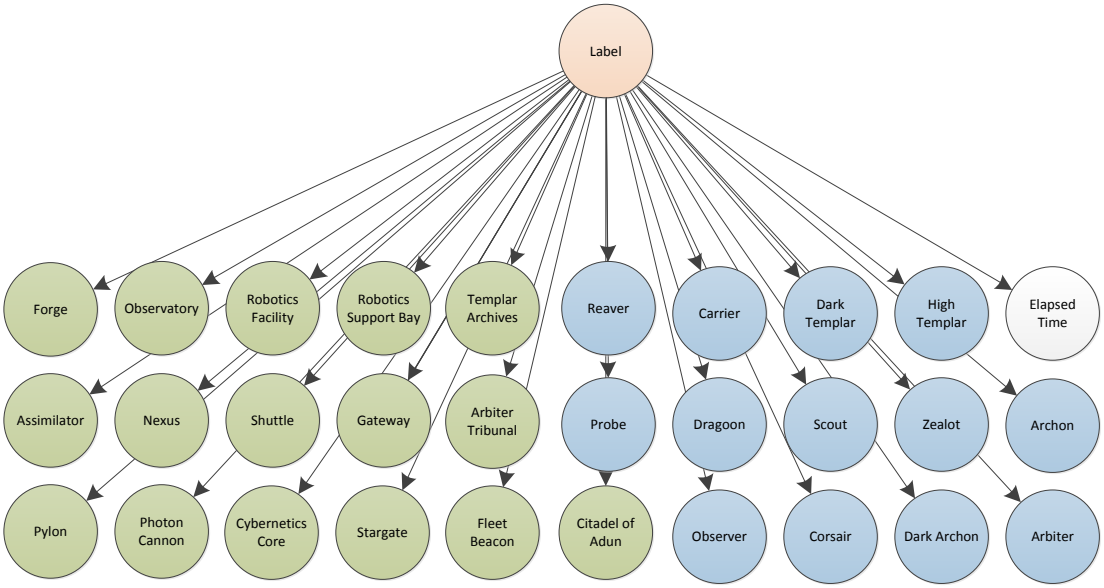

Figure 4.5: A Naive Bayes classifier for StarCraft.

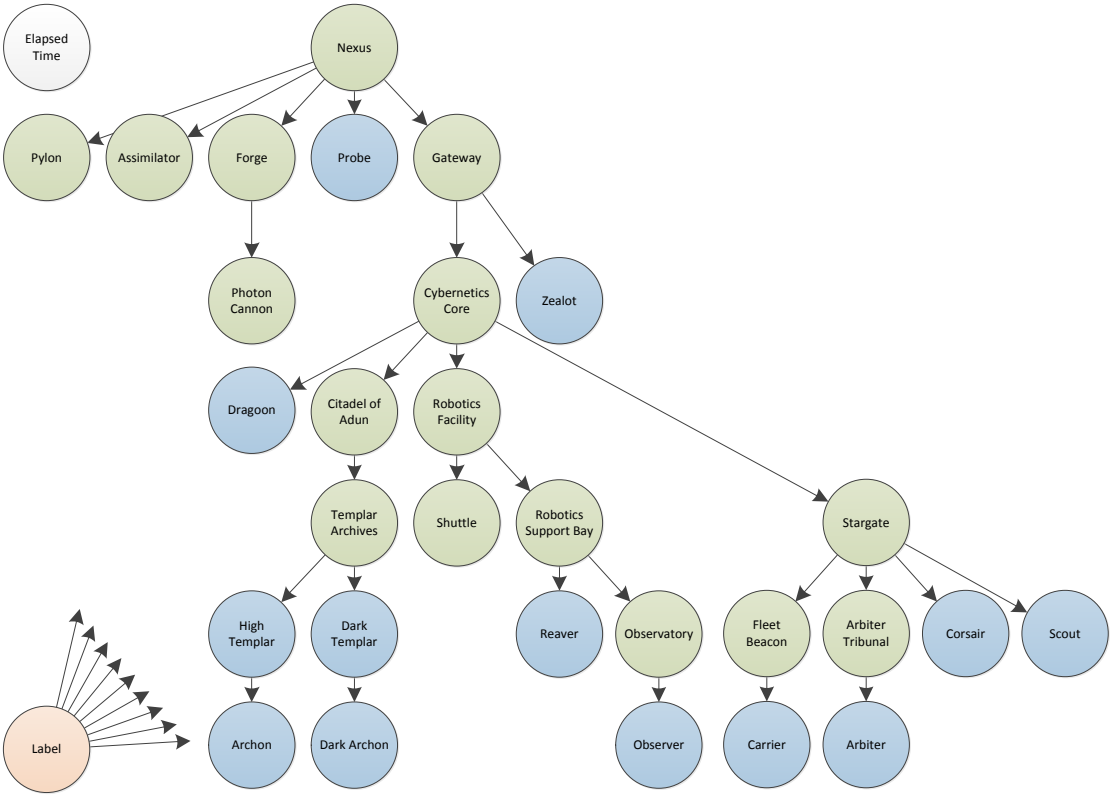

Figure 4.6: A possible TAN classifier for StarCraft.

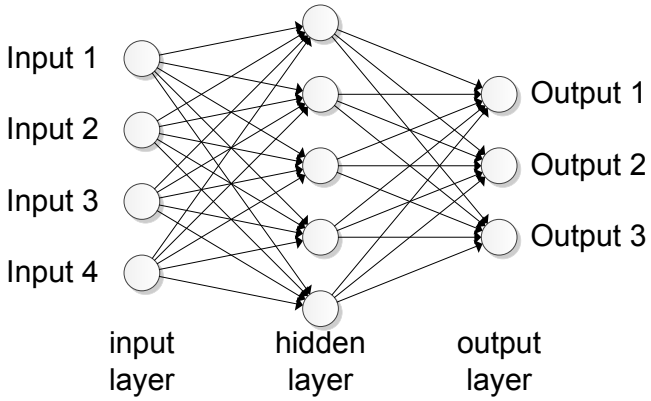

Figure 4.7: Three-layer artificial neural network

ANNs, similar to their biological cousins, consist of simple neurons that are interconnected to perform computations.

ANNs have so far been used to solve tasks including function approximation and regression analysis, classification via pattern or sequence recognition, data filtering and clustering, as well as robotics and decision making[10].

Numerous different types of ANNs exist, and most have been evolved for specialized use. Some of these are feed-forward ANNs, radial basis function ANNs, recurrent ANNs and modular ANNs. Because of this diversity in ANNs it can be difficult to state generally what an ANN is.

In the following we focus exclusively on feed-forward ANNs, specifically the Multi-Layer Perceptron (MLP) for classification. Whether it is the technique most suited for classifying for video games, however, is still an open question.

4.2.1 Feed-forward Networks

A feed-forward ANN consists of one or more layers of neurons interconnected in a feed-forward way. Each neuron in one layer has directed connections to all the neurons of the subsequent layer. Figure 4.7 shows an example of a three-layer feed-forward artificial neural network.

A neuron in a feed-forward ANN takes input from all neurons in the preceding layer and sends its single output value to all the neurons in the next layer. This is the reason it is a feed-forward network, since information can not travel backwards in the network.

Feed-forward ANNs are frequently used for classification. Here each output node in the ANN corresponds to one of the different labels to assign to one input instance. Through running a learning algorithm on several training instances the ANN adjusts weights on the inputs to each node. When learning has finished the ANN should be able to activate one of its output nodes when receiving a new input instance, hopefully the output node that labels the input correctly.

4.2.2 Computational units

As mentioned ANNs consist of interconnected neurons, or computational units. Each of these units j in the network, except for the input nodes in the first layer, is a function on inputs x_{ji} with weights w_{ji}. This function then gives a real number output depending on these inputs and weights.

In the following we explore two computational units, the perceptron and the sigmoid unit, as described by Mitchell[22].

Perceptron

A perceptron, as seen in Figure 4.8, receives input with a weight associated. All inputs are then summed up and compared to a threshold, with the perceptron finally outputting 1 if the summed weighted input is above the threshold and 0 otherwise.

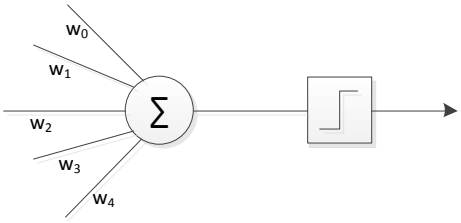

Figure 4.8: Perceptron algorithm

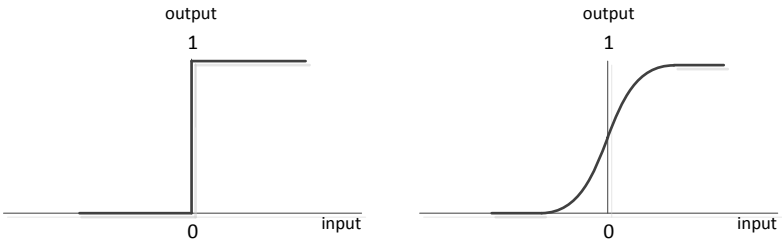

Figure 4.9: Step (left) and sigmoid (right) threshold functions

More formally, the perceptron output $o_j(\vec{x})$ for node j on input vector $\vec{x}$ is given by

$$o_j(\vec{x}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_{ji} x_{ji} > 0 \\ 0 & \text{otherwise} \end{cases},$$

where the inputs $x_{j1}...x_{jn}$ to unit j are given by the outputs of units in the previous layer, and input $x_{j0} = 1$ to unit j is a constant. Since $x_{j0} = 1$ the associated weight w_{j0} can be seen as the threshold the perceptron must exceed to output a 1.

Since the perceptron output is either 1 or 0, the perceptron corresponds to an undifferentiable step function. We next examine the sigmoid unit, an alternative computational unit to the perceptron based on a differentiable function.

Sigmoid unit

The sigmoid computational unit is similar to the perceptron, since it also sums over weighted inputs and compares this sum to a threshold. The difference lies in the threshold being defined by a differentiable sigmoid function instead of the perceptron's undifferentiable step function (see Figure 4.9).

In greater detail, the sigmoid unit's output $o_j(net_j)$ on the summed weighted input net_j is given by

$$o_j(net_j) = \frac{1}{1 + e^{-net_j}},$$

where

$$net_j = \sum_{i=0}^{n} w_{ji} x_{ji}$$

and e is the base of the natural logarithm. Somewhat counter-intuitive, the multi-layer perceptron is based on sigmoid units, and not perceptrons. We now proceed to describe the multi-layer perceptron and its backpropagation algorithm for learning the weights given to each input.

4.2.3 Multi-Layer Perceptrons

The MLP is the artificial neural network we will be using later on when comparing different classification methods on StarCraft training data. As previously stated it is a feed-forward network where information travels from input nodes toward output nodes, and where each hidden and

output node is based on the sigmoid computational unit. In the following we briefly explain how the MLP can be trained to use for classification:

1. Create network of input features and output labels.
2. Initially set all weights to small values.
3. Perform learning algorithm iterations:
   - Feed training input forward through network.
   - Calculate error between target and MLP output.
   - Propagate errors back through network.
   - Update node weights based on errors.
4. Stop when MLP is sufficiently accurate.
5. The output with the highest value on the input instance is the label of the instance.

The classification approach above makes use of the backpropagation learning algorithm to make the MLP approximate target outputs over several iterations, and we will describe this in detail in the following. As we saw above, the algorithm computes the output for a known input instance and then propagates errors back through the network to update the input weights on each node.

Backpropagation

Backpropagation learning over several iterations updates the weights on each input to each node in the network according to the delta rule

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j x_{ji}$$

which is comprised of the following elements:

- w_{ji} is the weight from node i to node j.
- η is the the learning rate for the backpropagation algorithm.
- δ_j is the the error term propagated back to node j.
- x_{ji} is the the input from node i to node j.

The full backpropagation for three-layer networks can be seen in Algorithm 2. In the algorithm each *training_ example* is a pair of the form $\langle \vec{x}, \vec{t} \rangle$ where $\vec{x}$ is the vector of input values and $\vec{t}$ is the vector of target output values. n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

Backpropagation Example

We now go through a simple example showing one iteration of the backpropagation algorithm. Backpropagation is called with one single *training_ example* consisting of only one input with value $\vec{x}_1 = 5$ and one target network output with value $\vec{t}_1 = 1$. Learning rate η is set to 0.05 and the number of input, hidden and output units n_{in}, n_{hidden} and n_{out} are all set to 1. The network for this can be seen in Figure 4.10.

First the feed-forward network of one unit in each layer is created. All weights are then set to a small random number, in this example 0.01. After initialization we go through one iteration as our termination condition has not yet been reached. The termination condition, deciding the accuracy of the model, is not important in this example.

Now, since we have only one *training_ example* the following for-loop has only one iteration. We input the value $\vec{x}_1 = 5$ and compute o_h and o_k, respectively the outputs of the singular hidden

*BackPropagation(*training_examples, η, n_{in}, n_{out}, n_{hidden}*)*

$Network \leftarrow$ CreateNetwork(n_{in}, n_{hidden}, n_{out})
Set all Network weights to small random numbers.

while *not sufficiently accurate* **do**
  foreach $\langle \vec{x}, \vec{t} \rangle$ *in* training_examples **do**
    Input instance $\vec{x}$ to the network.
    foreach *unit* $u \in Network$ **do**
      Compute output o_u.
    end
    foreach *output unit* $k \in outputs$ **do**
      $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
    end
    foreach *hidden unit* $h \in hidden$ **do**
      $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh}\delta_k$
    end
    foreach *weight* w_{ji} **do**
      $w_{ji} \leftarrow w_{ji} + \eta\delta_j x_{ji}$
    end
  end
end

Algorithm 2: BackPropagation(*training_ examples*, η, n_{in}, n_{out}, n_{hidden})

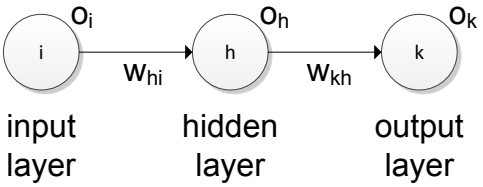

Figure 4.10: Backpropagation example network.

and output nodes in our network. We recall that the output of the singular input node is simply $\vec{x}_1 = 5$. The other two outputs are calculated as

$$\begin{aligned} o_h &= w_{hi}x_{hi} \\ &= 0.01 \times 5 \\ &= 0.05 \\ o_k &= w_{kh}x_{kh} \\ &= 0.01 \times 0.05 \\ &= 0.0005 \end{aligned}$$

and will be used in the following. We have now propagated the input forward through the network. The next two for-loops propagate errors back through the network. Since we have only one output unit k and one hidden unit h we can calculate the errors δ_k and δ_h as

$$\begin{aligned} \delta_k &= o_k(1 - o_k)(t_k - o_k) \\ &= 0.0005(1 - 0.0005)(1 - 0.0005) \\ &= 0.000099980001 \\ \delta_h &= o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh}\delta_k \\ &= 0.05(1 - 0.05)(0.01 \times 0.000099980001) \\ &= 0.000000047490500475 \end{aligned}$$

and use them in the final part of the iteration. We have now propagated the errors back through the network and all that remains is updating the weights w_{hi} and w_{kh} on the input to the hidden and output nodes h and k. This is done using the delta rule for each case as

$$\begin{aligned} w_{kh} &= w_{kh} + \eta\delta_k x_{kh} \\ &= 0.01 + 0.05 \times 0.000099980001 \times 0.05 \\ &= 0.0100002499500025 \\ w_{hi} &= w_{hi} + \eta\delta_h x_{hi} \\ &= 0.01 + 0.05 \times 0.000000047490500475 \times 5 \\ &= 0.01000001187262511875 \end{aligned}$$

which results in both weights increasing slightly.

This iteration is then repeated several times until the termination condition is met, where after the weights on each node better approximate the optimal weights. This example network of only one node in each layer has very limited use, however; it cannot be used properly for classification, as its singular output node will classify all input as the same output. Still, the example helps illustrate how several iterations of backpropagation allow an MLP to approximate a target output via delta rule weight updating.

Backpropagation Momentum

As an optional addition to backpropagation, momentum can be used to alter the delta rule for weight updating. Here the weight update at iteration n depends in part on the weight at the previous iteration $(n - 1)$. The altered momentum delta rule $w_{ji}(n)$, along with the original delta rule w_{ji} for comparison, are given by

$$\begin{aligned} w_{ji} &= w_{ji} + \eta\delta_j x_{ji} \\ w_{ji}(n) &= w_{ji(n-1)} + \eta\delta_{j(n)}x_{ji(n)} + \alpha\eta\delta_{j(n-1)}x_{ji(n-1)} \end{aligned}$$

where the the first two terms are given by the original delta rule and the final term relies on the previous weight $n - 1$ and the momentum α.

The momentum addition to weight updating causes backpropagation to converge faster, as weights change more when the error term $\delta_{j(n)}$ and input $x_{ji(n)}$ have not changed from the previous $\delta_{j(n-1)}$ and $x_{ji(n-1)}$. This can be compared to momentum speeding up or slowing down a ball when i rolls down- or uphill respectively.

4.2.4 MLP Classification for StarCraft

Using the Weka Toolkit[14] we applied artificial neural networks to the StarCraft domain using unit counts and elapsed time as input nodes, which results in the MLP seen in Figure 4.11. Edges between nodes have been omitted for clarity, but all nodes are implicitly connected in a feed-forward manner.

This MLP of has 32 input nodes, one for each Protoss unit and building type in StarCraft, and 12 output nodes corresponding to the twelve strategies identified by QT clustering. The 22 hidden nodes has in our evaluations been determined by Weka's default setting, which creates a number of hidden nodes given by

$$n_{hidden} = \frac{inputs + outputs}{2} = \frac{32 + 12}{2} = 22$$

The next method to be described is ActionTrees, our novel approach to strategy prediction. This method is different from the naive Bayes and MLP classifiers as it looks at player histories instead of gamestates when predicting player strategies.

4.3 ActionTrees

As an alternative to already developed classifiers we developed our own novel approach called ActionTrees. In this section we will introduce this new method and discuss the motivation for its development.

In the previous classification methods we only looked at a single gamestate, whereas with ActionTrees we look at a history of actions to predict a player's next action. It is our assumption that the order of actions might better determine the strategy a player is using, rather than an independent snapshot of a gamestate.

Thus a technique which looks at these sequences, we postulate, will be able to predict StarCraft strategies more accurately. We will examine to which extent this affects prediction accuracy in the next chapter, but first an explanation of ActionTrees.

4.3.1 ActionTree Structure

The tree structure used in ActionTrees includes associated probabilities on outgoing edges. The leaf nodes represent complete observed instances and as such contain the appropriate label of that instance. We formally define an ActionTree as in Definition 9.

Definition 9 (ActionTree)**.** *An ActionTree consists of the following.*

- *A finite set of nodes.*
- *A finite set of edges representing actions and the probability of these actions.*
- *Edges connect nodes forming a tree structure.*
- *Leaf nodes are labelled with a strategy given by the path.*

Recall that we in section 3.1 defined a build order as a sequence of actions and timestamps followed by a strategy given by this sequence. Using this as a basis we make an abstraction from time and only focus on the sequence of actions with the corresponding strategy. More precisely we define a *history* as in definition 10.

Definition 10 (History)**.** *A history h is a sequence of actions a_n and a label s, $h = \{a_0, a_1, .., a_n, s\}$.*

A path from the root node to a leaf node in an ActionTree is a history with a sequence of actions equivalent to the edges in the path. We call a history complete with regards to an ActionTree if the history sequence length equals a path in the tree. Moreover if the strategy of a history is unknown we call it an unlabelled history.

Figure 4.12 shows a simple example of an ActionTree learned from the histories $\{A, Strategy1\}$, $\{B, C, Strategy2\}$ and $\{B, B, Strategy3\}$, later we will discuss how we precisely learn an ActionTree from an arbitrary number of histories.

As can be seen the tree ends at a labelled node containing the strategy which the history defines. This implies that an ActionTree is a collection of all possible build orders in the game. As such ActionTrees can in theory be learned to contain all possible finite build orders in a game without lessening the accuracy of the prediction of individual build order, however, the resulting tree would be very large.

4.3.2 ActionTree Learning

The first step is to build a tree from the labelled build order given by the clusterer as illustrated by the pipeline in figure 4.1. Learning the ActionTree is described in algorithm 3

```
ActionTreeLearning() Input: Output: A constructed ActionTree.

Create empty rootNode
foreach labeled build order b in replays do
    currentNode ← rootNode
    foreach action a in b do
        edge ← currentNode.GetOrCreateEdge(a)
        edge.visited ← edge.visited + 1
        currentNode.visited ← currentNode.visited + 1
        currentNode ← edge.childNode
    end
end
initialize openSet
initialize newSet
openSet ←(add) rootNode
while openSet != empty do
    foreach node n in openSet do
        foreach edge e in n do
            e.probability ← e.visited / n.visited
            newSet ←(add) e.childNode
        end
    end
    openSet ← newSet
    empty newSet
end
```

Algorithm 3: ActionTree Learning Algorithm

An example of this learning method can be seen in figure 4.13.

In the figure we already have a simple ActionTree, and we wish to insert a new history $\{B, B, Strategy3\}$. To do this the root is examined: If this already has an edge for B we go along it, else we create one. In this case the root does have a B edge. This procedure is done again for the node lead to by the B edge. In this case a B edge does not exist and additionally, because it is the last action in the sequence, we know that the following node should be a label. An edge B is therefore created which leads to the leaf node for strategy 3. Finally we need to update the probabilities for each action. Initially we had probability of 50% associated with edges of the edges leading from the root node however, as we have inserted a new action sequence for the B edge, these will become 33% for action A, and 66% for action B. Similarly the second action C and B will get a probability of 50%.

4.3.3 Prediction using ActionTrees

Given a sequence of actions we want to predict the strategy that is most likely to be taken by the player. We formulate our prediction algorithm as seen in algorithm 4.

```
ActionTreePrediction(partialHistory)
Input: A unlabelled partial history
Output: A strategy
foreach action a ∈ partialHistory do
    if currentNode is LeafNode then
        return currentNode.Strategy
    end
    foreach edge e ∈ currentNode do
        if e.Action = a then
            edgeToTake ← e
        end
    end
    if edgeToTake not found then
        edgeToTake ← Take edge leadning to most probable strategy from currentNode
    end
    currentNode ← edgeToTake.Child
end
if currentNode is not LeafNode then
    strategy ← Most probable strategy from currentNode
else
    strategy ← currentNode.Strategy
end
return strategy
```

Algorithm 4: ActionTree Prediction Algorithm given a partial history

As an example assume we have built a tree as seen in figure 4.14 and we are given the unlabelled history $\{B, C, A\}$. To make the prediction we simply follow each edge in the tree and lookup the strategy in the leaf node and see that this gives strategy 2. So given a complete unlabelled history we see that actions trees are able to correctly classify a strategy that has been encountered during training assumed that the data is noise-free.

Given only a partial history we use the probabilities given by the tree to classify the most probable strategy. So for example given the unlabelled history $\{B, B\}$ we first take the edge B followed by edge B again. Two choices can now be taken, C or A, but as C leads to the most probable strategy we choose this resulting in strategy 3.

The two previous examples assume that the unlabelled history given was also seen or at least partially seen when training the tree. Given a unlabelled history $\{D, B, A\}$ we are faced with the player doing an action never seen while the tree was constructed, namely D. Like before where we faced with not knowing what future actions the player would take we simply counter this be taking action leading to the most probable strategy instead. This results in the sequence $\{B, B, A\}$ and a classification as strategy 4.

4.3.4 Using Lookahead to Improve Predictions

To improve the prediction rate of ActionTrees we make use of a lookahead method to more precisely capture what the player is doing. Instead of only looking at one action at a time we instead look at multiple. To illustrate how this works assume we are given an unlabelled history $\{C, C, B\}$ and we want to classify this with the ActionTree in Figure 4.14.

We see that C is the first action given by the history but no such action is available in the tree. Without lookahead we would then choose B as the most probable alternative, followed by $\{C, A\}$, and finally end up classifying the history as strategy 2.

If we instead were to use a lookahead of 2 actions beyond the current we would see that if we instead choose A as our first action we will end up being able to choose action C and B afterwards, resulting in a total of 2 actions matching up instead of only 1 if we choose B first, thereby resulting in a different classification as strategy 1.

Under the assumption that making more matches results in a better prediction of the player strategy we will with use of a lookahead be able to maximize the number of matches to a much

larger degree than without. We can take the lookahead concept further by looking at multiple ways of how to define a match.

- Precise Match - an action is only a match if it occurs at the same history length or tree depth and the action at this depth is the same as the action given. In other words correct ordering of actions in the lookahead is required for precise matching.
- Jumbled Match - for a jumbled match it does not matter at what depth a match is at, as long as it is within the lookahead and has not been matched already. Thus correct ordering in the lookahead is not necessary for jumbled matching.
- Hybrid Match - the combined number of precise and jumbled matches for one action is compared to that of the other actions and the one scoring highest is selected. Here correct ordering of actions is important, but not required.

If we were to relate what precise and jumbled matches correspond to in a game context we see that by using precise matches we prefer predictions with a precise ordering whereas when using jumbled matches the order of how actions in a build order are executed is less important. Lastly, we combine the two approaches into a hybrid approach, which attempts both a precise and jumbled match. We can finally combine the two in a hybrid approach where we for each path look at both precise and jumbled matches and take the path that results in the most combined jumbled and precise matches.

Using the assumption that the number of matched actions is critical for making correct predictions, if we have two histories then no matter how many jumbled or precise matches a given history evaluates to, if there are more matches in one path then this history is chosen. But what if we have two histories that evaluate to the same number of matches? We then have two different cases:

1. If the two histories results in exactly the same number precise and jumbled matches we then go by the most probable path.
2. If we still find the total number of matches are equal but we have more precise matches than jumbled we will take the history with the larger number of precise matches.

To give an example of all the different match methods, consider the ActionTree given in Figure 4.14. We want to classify an unlabelled history $h = \{C, B, A\}$.

- Precise matching prefers strategy 4 as it has the most precise matches, namely B and A.
- Jumbled matching finds that both strategy 1 and 2 have a total of three jumbled matches, but as strategy 2 is the most probable it is the preferred strategy.
- Using the hybrid approach we again see that strategies 1 and 2 have three matches, but strategy 2 is preferred as it has more precise matches than strategy 1, namely the last action A.

Note that the hybrid approach here prefers the same strategy as the jumbled approach, but the grounds for choosing the strategy is different.

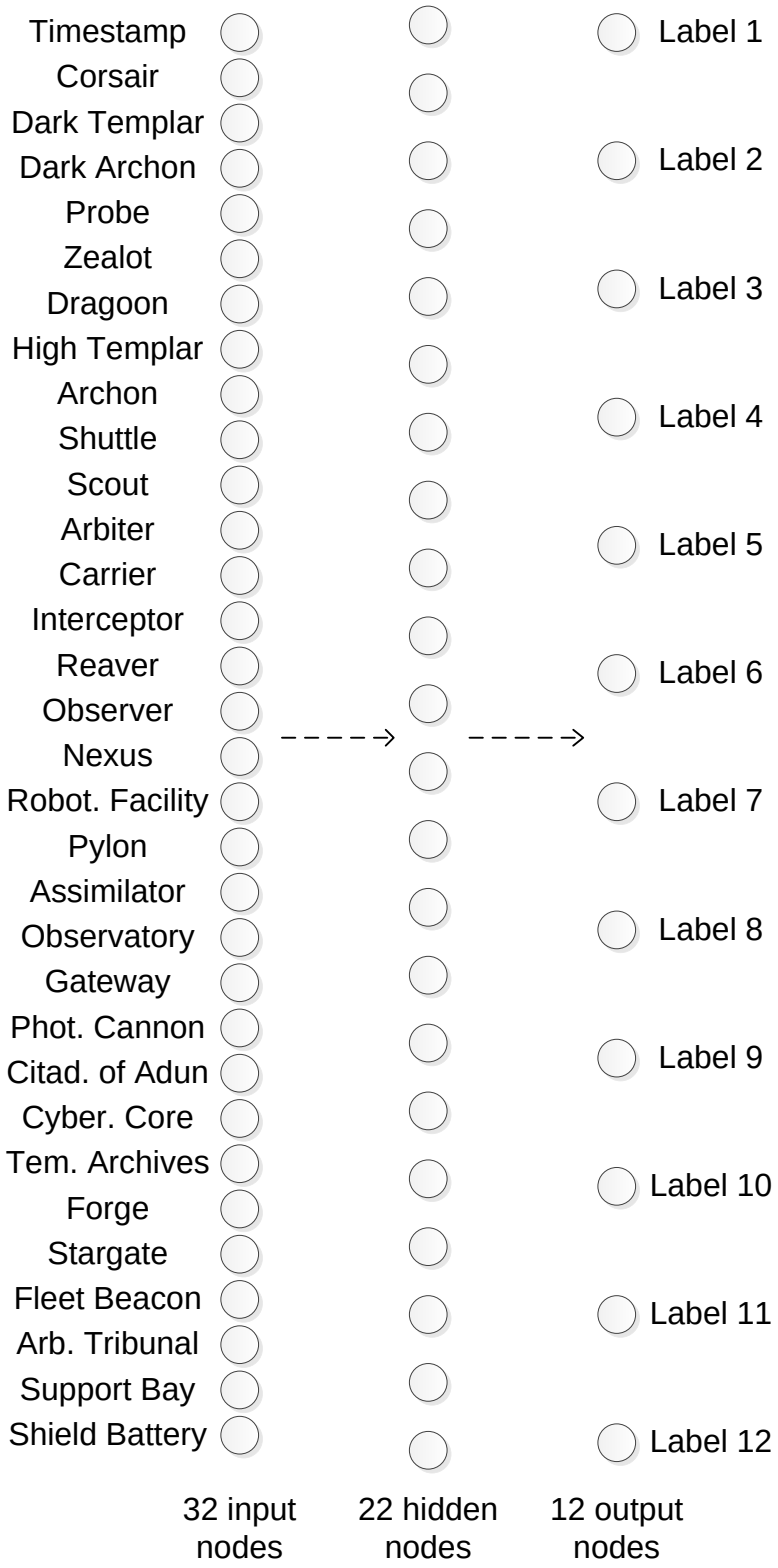

Figure 4.11: MLP for classifying StarCraft player build.

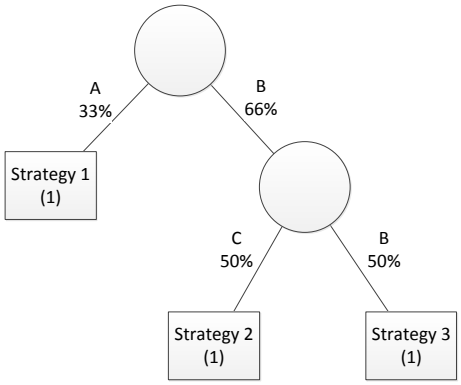

Figure 4.12: Example of a simple ActionTree build from the histories $\{A, Strategy1\}$, $\{B, C, Strategy2\}$ and $\{B, B, Strategy3\}$

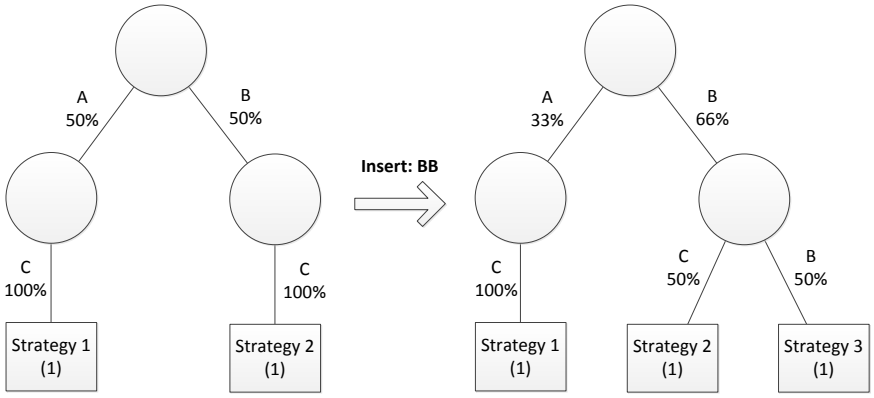

Figure 4.13: Inserting the history $\{B, B, Strategy3\}$ into the ActionTree

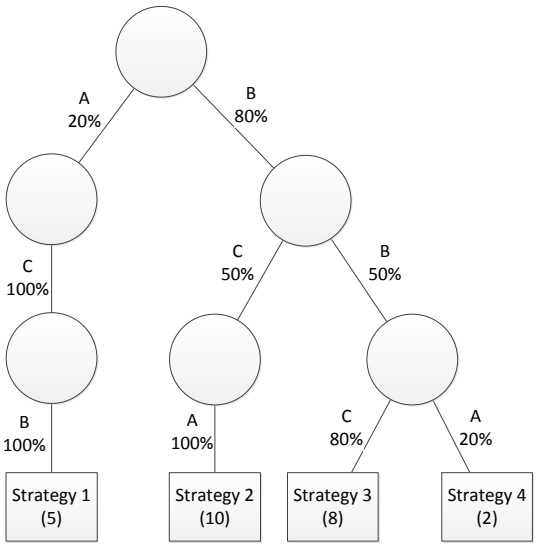

Figure 4.14: ActionTree Example

CHAPTER

FIVE

EVALUATION OF PREDICTION MODELS

In this chapter we will evaluate how well the three methods, naive Bayes classifiers, multi-layer perceptrons and ActionTrees, can predict which strategy a player in StarCraft is using. Each method is first evaluated individually, after which a comparison of the three methods is presented.

We expect the naive Bayes classifier to perform well on complete data, and the TAN classifier to perform better than naive Bayes on missing data due to its use of tech trees in StarCraft. We expect the MLP to have accuracies similar to the naive Bayes classifier, as they both see time as a variable on line with unit and building composition.

Finally we expect the ActionTree method to give good accuracy results early on, as different orderings in players' action histories allow the classifier to eliminate wrong strategies earlier than the other methods. And overall, if missing values are introduced or the training data for each method is reduced, all methods are expected to perform worse than if given complete data.

5.1 Test Methodology

To evaluate the predictive capabilities of our proposed methods, we have used a dataset consisting of 814 gamestate sequences and 814 build orders extracted from 407 replays. The replays were downloaded from Team Liquid[9] and GosuGamers[8].

We conducted our prediction accuracy tests on five different sets of this data:

- Complete dataset training data - The full dataset used as both training and test set.
- Complete dataset cross validation - 5-fold cross validation was run using the full dataset.
- Imperfect dataset training data - Full data as training set and 20% of the variables from removed from the test data.
- Imperfect dataset cross validation - 5-fold cross validation using the full dataset, where the test partition has 20% missing variables.
- Data increase - A variable size of the training data tested against the complete dataset as a test set.

The *Full dataset* test is a straightforward test to see whether the method can classify correctly on previously learned data. It is essential that our methods can predict already seen strategies. To detect whether our methods has a tendency to overfit on our training data we also run a five-fold cross validation test. To simulate the environment which the models were designed for, i.e.: predicting in a game of imperfect information, we tested the models accuracy when withholding evidence from 20% of the variables on each observation from the testset, with both the full dataset as test data and using five-fold cross validation. To test how sensitive our methods are to the size of the training set, we perform a test where the methods are given 20%, 40%, 60%, 80% and 100% of the data as training set and the full data set as test set.

5.2 Bayesian Networks Evaluation

To evaluate the ability of Bayesian classifiers to predict strategies, we have implemented two models: a naive Bayes classifier, figure 4.5, and a tree augmented naive Bayes classifier (TAN), figure 4.6. The TAN is constructed by augmenting the Naive Bayes classifier with the relations given in the tech tree described in section 2.1. Both models were created by defining the structure in the Hugin[11] GUI. We did a maximum likelihood estimation to decide the probability tables and tested the accuracy of each model on different sets of test and training data using the Hugin API.

Each unit and building type has a corresponding node in the classifiers, each of which contain a number of labels which denote quantities of that unit in a game state. The state space of each node has been learned from the training data, such that all values for of the corresponding unit or building type observed in the training data translates to a state in the state space. To predict a strategy, all available evidence is inserted into the network and the most probable state in the label node is read predicted strategy.

As can be seen in figure 5.1, when given the full dataset of all 814 samples containing all data, the Naive Bayesian classifier begins to distinguish strategies after the 120 second mark. Its highest precision level is around 75% at time 30. After time 300 the accuracy starts to decline. The TAN shows comparable results, with a slightly better performance after 270 second mark.

It appears that during the first 90-120 seconds the two Naive models are inaccurate, but then they drastically improve on the 150 second mark. We postulate that this is due to the fact that the players options concerning what to construct in the game is extremely limited during the first couple of minutes of gameplay. Thus the observed gamestates will be very similar.

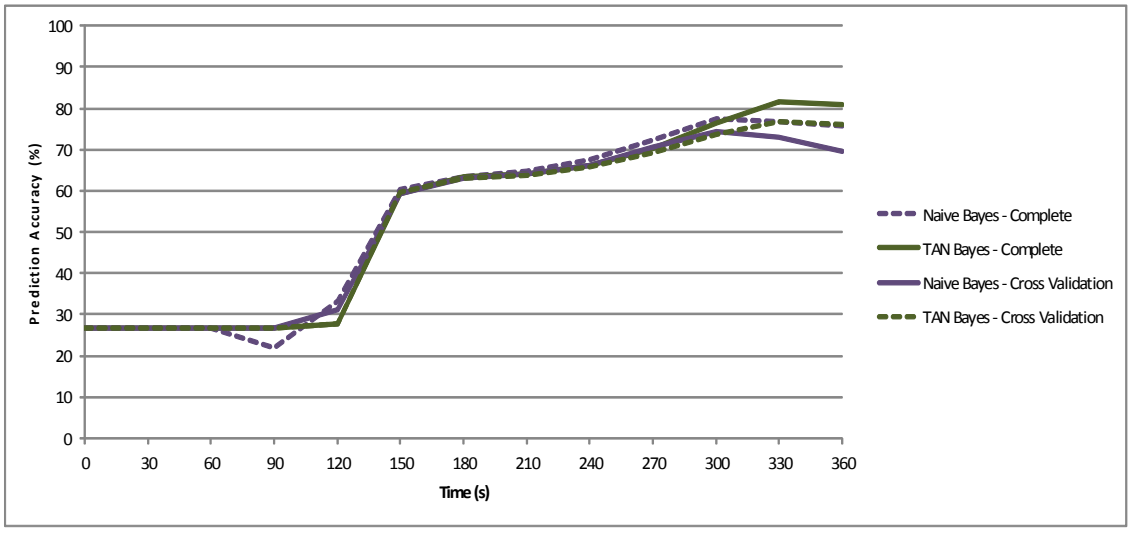

Figure 5.1: Accuracy of Naive and TAN classifiers over time on the full dataset

Figure 5.1 also shows the result when using 5-fold crossvalidation on the full dataset. It tops around 180 to 270 seconds, after which the the precision deteriorates. The Naive classifiers' precision in the cross validation test are almost equal to the test on the full dataset.

The results of training the model using varying percentages of the full test set can be seen in figure 5.2. The accuracy of both the TAN and the Naive classifier are comparable to the results seen earlier, for both time 210 and 360. Even though the TAN performs marginally better at 360 seconds, it seems that the size of the dataset does not have much impact on the classification accuracy for these classifiers.

The results, which are depicted in figure 5.3, are comparable to those seen when given full data and thus we can conclude that the accuracy of the models do not deteriorate when given only 80% of the available information.

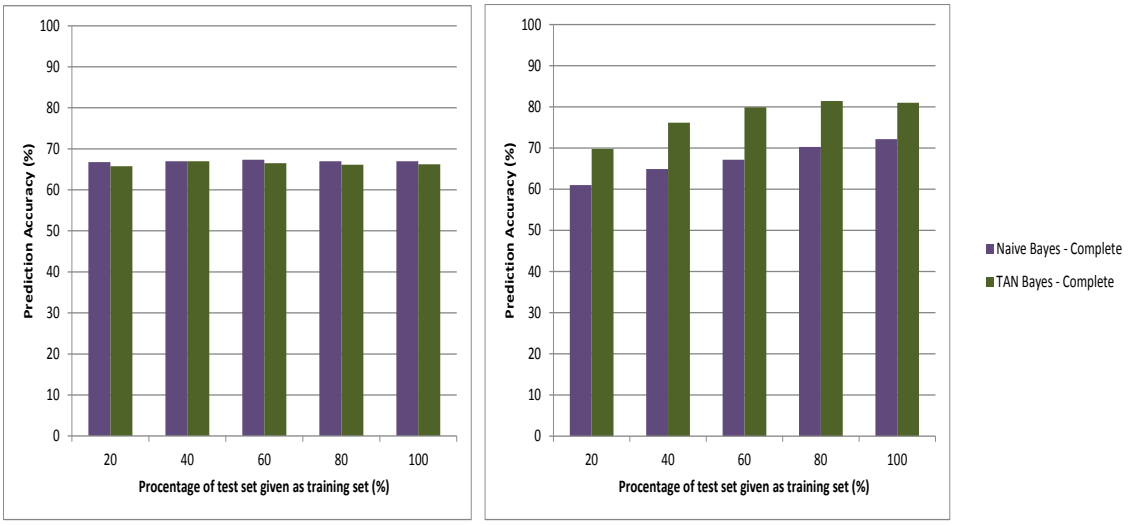

Figure 5.2: Accuracy on data increase at times 210 and 360, left and right graphs respectively.

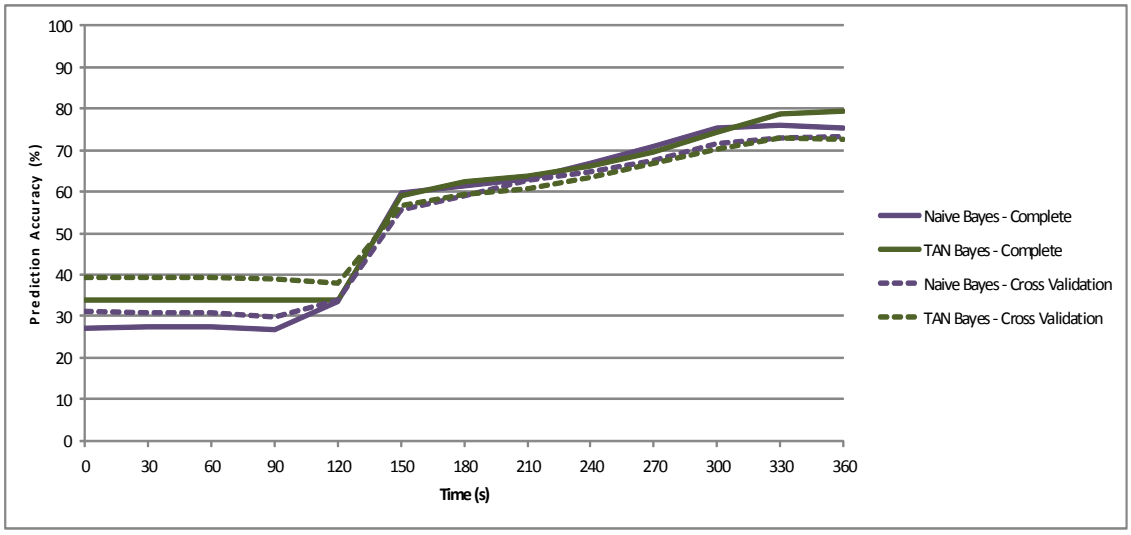

Figure 5.3: Accuracy on incomplete data

5.3 Multi-Layer Perceptron Evaluation

The ANN-based Multi-Layer Perceptron we have used for classification will be evaluated on in the following. Being based on the Weka implementation of MLPs there is little in the way of implementation details to discuss.

The MLP is as stated earlier a feed-forward ANN based on sigmoid computational units; at any given time in an ongoing game, the current timestamp and a player's current units and buildings can be input to the MLP and it will be able to output the strategy the player is most likely to follow, according to the clusters provided by the QT clustering algorithm.

Before the main experiment, a series of tests have been run to determine the best learning rate η, momentum α and number of iterations n for the Weka MLP to use. All parameter tests have been conducted by five-fold cross-validation on complete information training data containing gamestates for all available timestamps.

5.3.1 Determining Iterations

The number of iterations n for the MLP to run has been determined by keeping η and α at Weka's default values of 0.3 and 0.2 respectively and then varying n while comparing MLP training time and classification accuracy.

	$n = 5$	$n = 10$	$n = 50$	$n = 100$	$n = 200$
Accuracy	62.46%	63.13%	**64.28%**	64.06%	64.54%
Time	6.5s	11.9s	**58.9s**	116.4s	239.8s

Table 5.1: MLP accuracy at different numbers of iterations n.

In the tests MLP training time was seen to increase linearly with increases in the number of iterations n. However, the classification accuracy did not, as can be seen in Table 5.1. Thus $n = 50$ has been selected as the best number of iterations for the remainder of MLP evaluation, as higher n start to introduce diminishing returns to classification accuracy.

5.3.2 Learning Rate and Momentum

What remains is determining the learning rate η and momentum α. These have been found via nine tests either raising or lowering η and/or α from Weka's default values and observing the MLP accuracy afterwards. The test results can be seen in Table 5.2 with learning rate horizontally and momentum vertically.

	$\eta = 0.1$	$\eta = 0.3$	$\eta = 0.5$
$\alpha = 0.1$	64.05%	**64.35%**	64.18%
$\alpha = 0.2$	64.12%	64.28%	63.79%
$\alpha = 0.5$	64.20%	63.84%	62.57%

Table 5.2: MLP accuracy with a given learn rate η and momentum α.

Here having learning rate η and momentum α at 0.3 and 0.1 respectively was shown to give the best accuracy, and thus these values have been selected for the remainder of MLP evaluation.

5.3.3 MLP Prediction Results

The parameters for the MLP have now been determined and analysis of its prediction accuracy can begin. The results of the complete data and its cross validation experiments on MLP accuracy over time can be seen in Figure 5.4.

Accuracies on complete data start out at 50% and do not increase until around 120 seconds into recorded games. A small quick increase in accuracy occurs around 180 seconds, and for the remainder of the experiments the accuracies increase steadily, with a slight steepening of accuracy increase starting at 240 seconds.

As expected the experiments with incomplete data in Figure 5.5, where 20% of input values are missing, gave lower accuracies than the prior experiments on complete data. It is not until 240 seconds that the MLP classifies with accuracies over 50%.

Looking at MLP prediction accuracy when training data increases from 20% to 100% in Figure 5.6, having a smaller training set does not reduce prediction accuracy much. A small reduction in accuracy can be seen at 360 seconds, but this reduction is minor at best.

Overall the MLP provides stable predictions, starting at 50% and increasing to 92% at 360 seconds. When tested on increasing sizes of test data from 20% to 100% of the complete training set, the method's accuracy is also high. Its reduced accuracy on incomplete data where only 20% of values are missing is however somewhat alarming. With even more missing values this accuracy may likely deteriorate even further.

We now proceed to look at evaluation results for the ActionTree classifier, which instead of gamestates uses the current build order of a player to predict their current strategy.

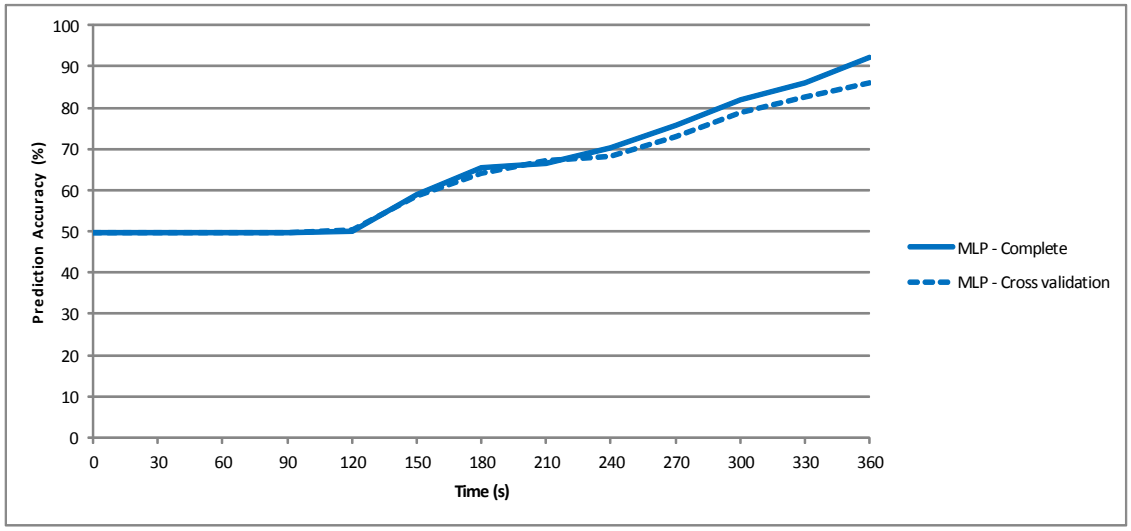

Figure 5.4: MLP prediction accuracy over time with complete and cross validated training data

5.4 ActionTree Evaluation

In this section we look more closely at the novel ActionTree model and evaluate it to see how it performs with different parameters. This evaluation should indicate which parameters will achieve the best prediction accuracy. Once these parameters are found, the model can be compared to Bayesian networks and multi-layer perceptrons.

5.4.1 Lookahead Runtime Analysis

Recall that we in Section 4.3 discussed the use of lookahead to make better predictions of the strategies learned in an ActionTree. Clearly the larger the lookahead we use, the more accurate the model will make its predictions, however, the time complexity grows exponentially as the action lookahead is raised. The theoretical worst case time complexity of a prediction using ActionTrees is $O(m^n)$ where m is the number of possible actions and n is the lookahead used. It is however unlikely that all possible orderings of actions are encountered during training, in part due to the limitations imposed by the tech tree on players and in part because certain sequences of actions do not make for effective strategies.

In Figure 5.7 we can see the actual running time of a prediction on an ActionTree constructed with our dataset with zero to ten lookahead steps. We have chosen only to test ActionTrees with a lookahead of a maximum of 10 and a lookahead of 0 to see how they perform in comparison to each other. Next we evaluate the prediction accuracy of the different lookahead matching methods.

5.4.2 Matching Approach Accuracy Comparison

As discussed in Section 4.3 we have suggested 3 different matching approaches to traverse the tree: precise, jumbled and hybrid.

The precise matching approach demands that the order of the actions taken are very closely related to previously learned actions in the tree. This should result in an especially accurate prediction when the history given is very closely related to an existing learned history. However it will be unable to relate two strategies that have the same outcome but use different orderings to achieve that strategy. As a result of there being no matches it falls back on using the most probable actions.

Jumbled takes the opposite approach - its strength lies in ignoring the ordering of actions. Jumbled looks at what actions are performed in the future and disregards the order they are performed in. This should lead to more accurate predictions in the cases where players use different orderings to reach the same strategy, but could result in false positives in cases where the precise order of actions matter for distinguishing strategies from each other.

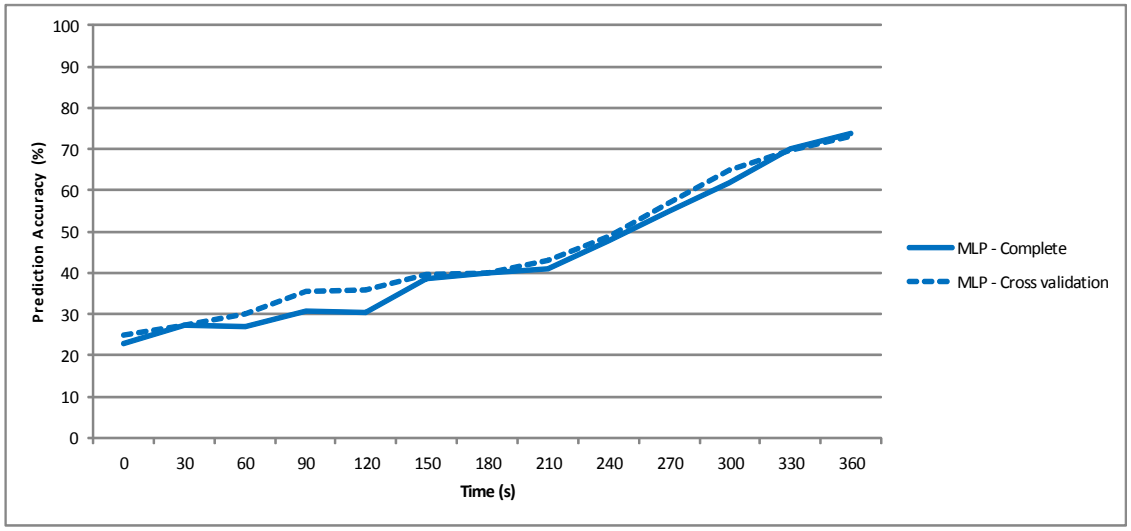

Figure 5.5: MLP prediction accuracy over time with incomplete and incomplete cross validated training data

The hybrid approach tries to compromise between the two methods and preserve the strengths of both methods. If the exact history given has been observed before, the method is the same as the precise matching approach, while if that is not the case it should act more like jumbled and be more tolerant to ordering differences in strategies.

Figure 5.8 shows the three matching approaches as well as one without using any lookahead with all replays used as both training and test set. If we look only at the complete data results we see that all methods are able to correctly predict the strategy 50% of the time just as the game starts. This matches with the results of the clustering, which showed that the largest cluster found in the training data contains 50% of cases, and with ActionTree theory, as without evidence excluding any strategies, the most frequently observed strategy is selected.

In the context of the game this is symptomatic of there being very few logical actions to take for players in the beginning of a game, resulting in nothing to differentiate games from each other at the onset. As time progresses, the intention of the players becomes clearer, as the actions they take differentiate them from each other and the model is then able to more reliably predict the strategy of the player.

When we compare the four matching approaches we see that precise, hybrid and no lookahead perform exactly the same, whereas jumbled matching is all-round worse in making an accurate prediction. This is to be expected, as all histories in the test set were also in the training set which is ideal for the matching approaches that attempt a precise match, since a precise match will be present. It is worth noting that we at 210 seconds are able to almost 100% correctly predicting a player strategy correctly with three of the methods, and already after 180 seconds we can predict with 90% accuracy.

Now, looking at the cross validation results in Figure 5.8, we see that all methods produce less reliable predictions compared to the previous test. This illustrates that the ActionTree model is great at identifying previously known strategies but that abstracting these to unknown histories is not as successful. Still, the quick rise in prediction accuracy on complete data shown earlier is assuring.

When we compare the methods, we observe that hybrid makes the best predictions over the entire time interval. Precise and no lookahead are very similar with only a small margin in the precise method's favour. Jumbled starts off with worse prediction than the rest but ends up doing just as well as the hybrid approach during cross validation.

This is an indication that many path evaluations have the same number of matches and that jumbled in these cases would end up taking the most probable where it would have been more prudent to take the order of the actions into account as can be seen by the higher overall prediction accuracy of the hybrid approach. It is also very interesting to note that there is a better chance of

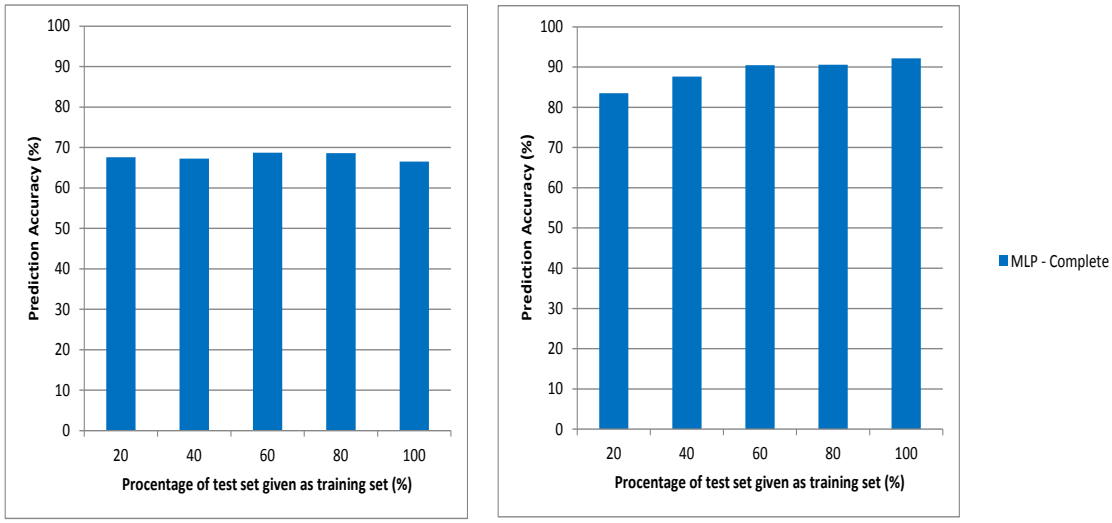

Figure 5.6: MLP prediction accuracy on data increase at 210 seconds (left) and 360 seconds (right)

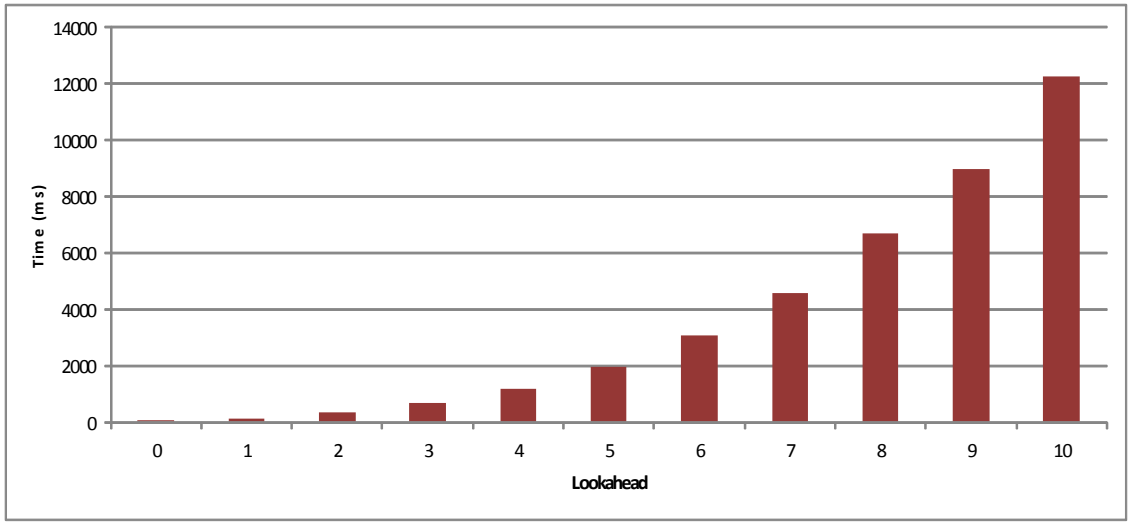

Figure 5.7: ActionTree running time of complete history prediction as lookahead increases

predicting at the 120 seconds mark for all matching approaches for cross validation.

The test using the entirety of the available data as both training and test set with incomplete information can be seen in Figure 5.9. This test yielded much the same result as the equivalent test with complete information. Again, we see that the approaches seeking a precise match perform better than the jumbled approach. Additionally the version without lookahead performs worse when compared to the precise and hybrid approaches.

We next examine the cross validation results on incomplete data also in Figure 5.9. These results and those for cross validation on complete information in Figure 5.8 yield many similarities; all four matching methods performed a little less reliably under incomplete data than they did under complete data and the jumbled and hybrid matching approaches outperform the no lookahead and precise matching approaches consistently.

We also again see a good chance of predicting the strategy correctly at the 120 second mark for all methods during cross validation. This trend might indicate that the sequence of actions a player performs at that point is revealing of that players overall strategy.

Looking at prediction accuracy as training data increases in Figure 5.10, all matching methods have increases in accuracy as more and more training data is given to the ActionTree classifier.

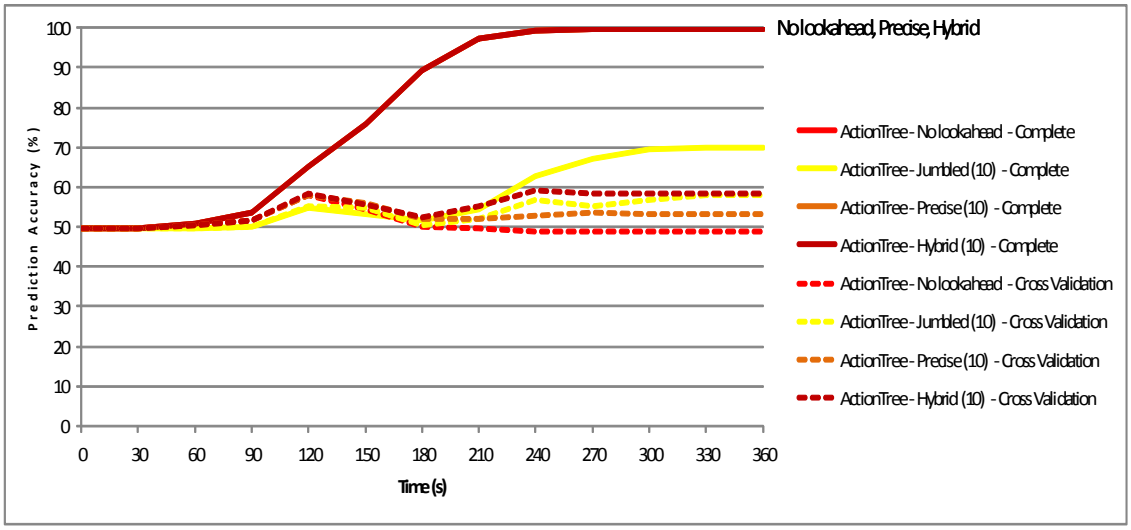

Figure 5.8: ActionTree prediction accuracy over time with complete and cross validated training data

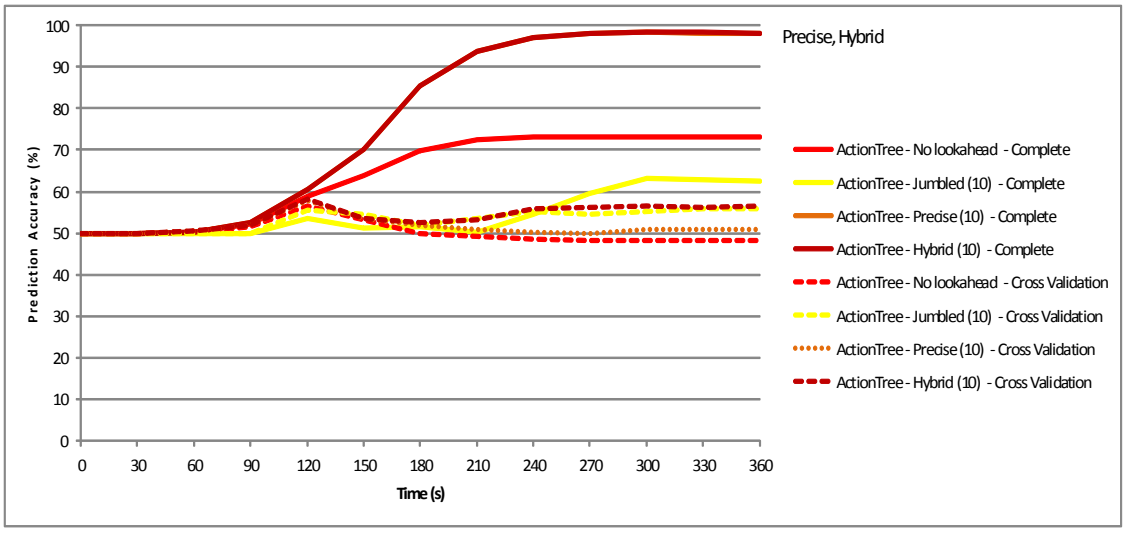

Figure 5.9: ActionTree prediction accuracy over time with incomplete and incomplete cross validated training data

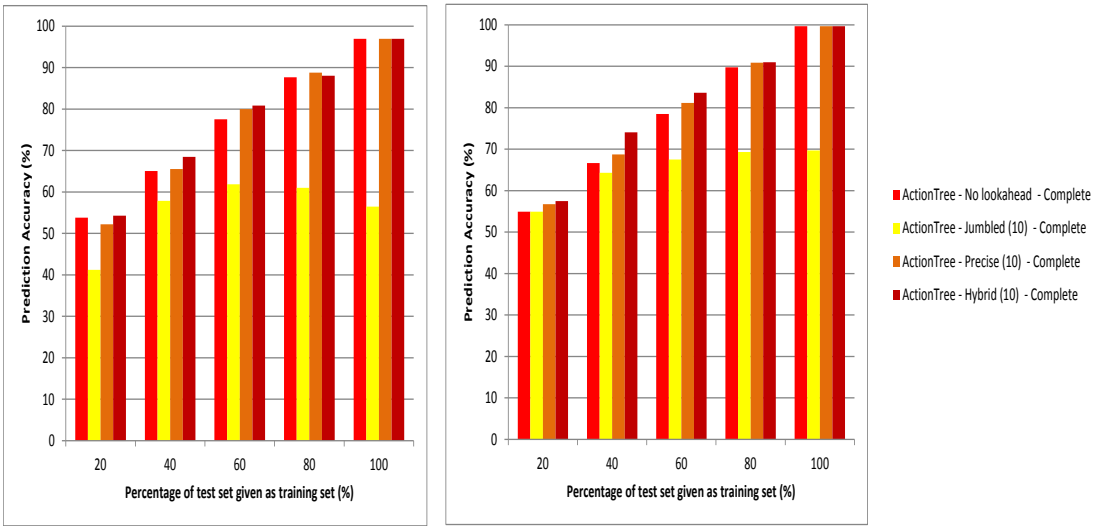

Figure 5.10: ActionTree prediction accuracy on data increase at 210 seconds (left) and 360 seconds (right)

One exception is the jumbled matching method, however; its accuracy even drops as training data size approaches 100%. This could be caused by the larger training data giving the classifier more jumbled match paths to take even though some of these do not lead to the correct strategy.

Overall the ActionTree method provides good results, especially by being able to predict with high accuracies early in the game compared to the other methods, but only as long as all test instances are known in advance. These results along with those of the prior two prediction methods will be compared in the final evaluation results below.

5.5 Evaluation Results

The ActionTree method's accuracy rises earlier than the other methods, indicating that classifying on histories in StarCraft rather than single game states is indeed beneficial. After 240 seconds the classifier reaches accuracies well over 90% for all but the jumbled lookahead 10 matching method. However, the method's accuracy never rises much above 60% for crossvalidation where it is tested on histories not in the training set.

Thus for ActionTrees to be beneficial a large and diverse training set is required more than for the other two methods. This reduced accuracy for cross validation can however potentially be improved as the ActionTree method is a new classification approach with plenty of room for tweaking and optimization. Even so, the classifier stays at a stable 50% minimum accuracy at all times with missing training data.

By comparison the Multi-Layer Perceptron method's accuracy rises more slowly as time increases, but its cross validation accuracy after 270 seconds starts to exceed that of the ActionTree method. This indicates that the MLP method performs better than ActionTrees when the training set is small. With missing training data the MLP however does not exceed ActionTree accuracies before the 270 second mark is reached.

The naive Bayes classifier compared to the MLP starts out at much lower accuracy, but quickly rises after 90 seconds and peaks at 77% after 300 seconds. The classifier also performs similarly for cross validation, where the highest accuracy at time 300 seconds is 74%. On missing data the is not reduced considerably, with a peak at 75% for testing on the full training set and 73% when testing with cross validation. Therefore the naive Bayes classifier appears more robust to missing training data than the two other methods we have tested.

Overall, the ActionTree method classifying on histories is best suited to classification on known strategies. The gamestate sequence based naive Bayes and MLP methods in comparison are better suited for classification of strategies not already in the training set. Of these two, the MLP performs best on complete test data, but the naive Bayes classifier is preferable for classifying on missing

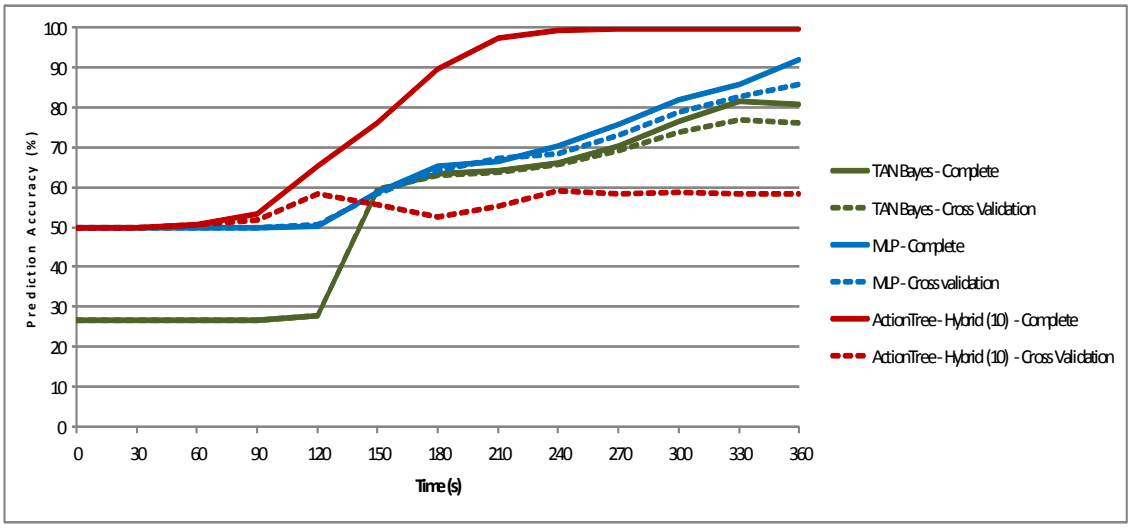

Figure 5.11: TAN, MLP and ActionTree (Hybrid lookahead) prediction accuracy with a complete training set and using 5-fold cross validation

test data. The three methods are all suitable to predict strategies in StarCraft, how to use them during a game in real time is still an issue.

As can be seen in Figure 5.11, predictions using naive Bayes classifiers and multilayer perceptrons on game state sequences can be quite accurate, with a maximum precision of up to 80% for the TAN and 94% for multi-layer perceptrons on already learned strategies. In predicting strategies that were not already learned the TAN achieved 75% as its highest accuracy and the multi-layer perceptron had a maximum accuracy of 85%.

Predicting using ActionTrees, where we use a history of player actions rather than just a single gamestate, showed itself to be able to predict accurately as well. In the case of predicting already known strategies it was possible to identify strategies nearly 100% prediction accuracy at 240 seconds into the game. The results suggest that predicting player strategies using the observed sequence of actions from a player, can yield better results, than trying to predict using only the current state of the game. In the case where we try to identify strategies that were not learned the accuracy never rises above 60%.

In the case of missing data, as seen in 5.12, the TAN achieves almost identical results, a maximum of 80% accuracy. In the case of the TAN we expected that it would be better using missing data as the internal structure would give hints to the state of related variables if they were missing, however, the results from section 5.2 show that we gained little compared to the naive Bayes classifier. The multi-layer perceptron however does not do as well, the best result is around 75% accuracy. The ActionTree can still predict very accurately, up to 99% using hybrid approach, however, it reaches this accuracy at 270 seconds into the game, 30 seconds later than if all data was known.

The two traditional machine learning approaches are quite feasible in predicting player strategies, the two models were very slow to learn. But the prediction of player strategies from new observations afterwards were fast enough for use in an RTS game. The long learning time is not a bit issue as it can be done offline, the online predicting process is quite important as the method should not be used on the expense of not being able to run the game. Learning of the structure of the ActionTrees was very fast, but the prediction process showed itself to be slow when using a large lookahead.

This can however be alleviated somewhat by only predicting with certain intervals and/or distributing the calculation over a period of time. Another issue that may present itself with ActionTrees is the high memory usage of the tree. Many of these issues can be optimized on as further work.

We found that it was not feasible to identify strategies before around 210 seconds into game for most of the methods with around 65% to 70% accuracy. The states before this are possibly

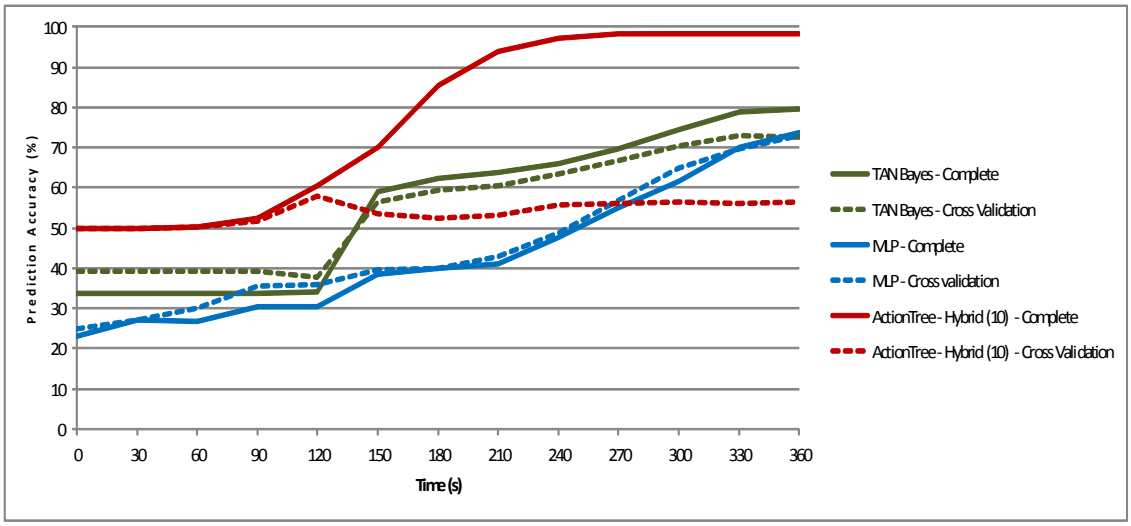

Figure 5.12: TAN, MLP and ActionTree (Hybrid lookahead) prediction accuracy with a complete training set and using 5-fold cross validation with an incomplete test set

so similar that it is not possible to make a good distinction between gamestates. ActionTrees, however, showed that looking at the order of a players actions it is possible to identify a players strategy much earlier, at around 150 seconds with roughly 75% accuracy.

CHAPTER

SIX

CONCLUSION

During this project we have used QT clustering to divide a set of replays into sets of labelled strategies. Using these labels we attempted to predict the strategy of a player at different times during a game, using three different classifiers; naive Bayes classifiers, multi-layer perceptrons and ActionTrees.

Using the labelled strategies we trained the three classifiers, using the labelled game state sequences for naive Bayes and multi-layer perceptrons and using labelled build orders for ActionTrees, to be able to reason about all actions and states leading up to the candidate game state used in the clustering.

Unsupervised strategy classification using QT clustering did not give us the expected results. One of the issues with the clustering was a single cluster containing half the data points, which we expect not to be in accord with reality. With more study into different clustering methods and a more sophisticated distance measurement clustering may be a viable method to identify strategies, but at the moment we believe that using a supervised classification method is the best option to retrieve accurate information about the different strategies used in different games. Still, the results of our evaluation show that prediction of player strategies using models learned from unsupervised game replay classification is viable.

It was possible to predict with a high level of precision the correct strategy of a player. In the beginning of a game the predictions were, as expected, not very accurate, but when predicting at points of time further into the game the accuracy improved.

Our research has shown that traditional approaches can be used effectively to predict a players strategy. The traditional approaches achieve good results in both identifying known and unknown strategies. However, we found that using build orders rather than single game states can improve the prediction accuracy, using ActionTrees. Both of the approaches are valid in predicting strategies, and have different uses. Most RTS games are games of imperfect information. It is easy to extract a game state from a game, as it is what the player can observe at a single point in time. To observe an opponents build order, a player needs to observe the actions of the opponent at several points in time. With missing information there is a large chance that both the game state and build order contains errors, but our three approaches show that with 20% missing variables performance is still comparable.

6.1 Future Work

While the results presented in this report clearly indicate that predicting player strategies is possible using a variety of opponent modeling techniques, there is a lot of areas that could benefit from refinement.

The ActionTree approach is interesting to look at, as classification on a sequence of actions in games seems not to be thoroughly researched. How to improve the prediction accuracy on data which has not been seen before and optimizing the performance of the method in terms of execution time and memory usage are especially interesting topics of research.

Multi-layer perceptrons is unlikely to be improved much on, but their use in video games, and

whether other artificial neural networks could improve prediction, may be an interesting topic as multi-layer perceptrons have proven themselves to be acceptable in predicting player strategies. We do however believe that we cannot improve much on the results of the basic multi-layer perceptron.

Different structures of Bayesian classifiers could also be an interesting venue of research, specifically how to overcome the problem of the often intractable size of the state space in the models to represent video games.

It would also be interesting to investigate whether it is possible to, given two different game states, to infer a possible build order to reach the second game state from the first. This would make it possible to create build orders for ActionTree classification without the need to constantly observe the opponent.

BIBLIOGRAPHY

[1] Thor Alexander. Gocap: Game observation capture. *AI Game Programming Wisdom*, 2002.

[2] Expressive Intelligence Studio at UC Santa Cruz. Starcraft ai competition. `http://eis.ucsc.edu/StarCraftAICompetition`, 2010.

[3] Sander C. J. Bakkes, Pieter H. M. Spronck, and H. Jaap van den Herik. Opponent modelling for case-based adaptive game ai. *Entertainment Computing*, 2009.

[4] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 2002.

[5] Michael Booth. The ai systems of left 4 dead. `http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf`, 2008.

[6] Brett Jason Borghetti. The environment value of an opponent model. *IEEE Transactions on Systems, Man, and Cybernetics*, 2010.

[7] Dustin Browder. Developer's corner: 1v1 game balance. `http://eu.battle.net/sc2/en/blog/761331`, 2010.

[8] The GosuGamers Community. Gosugamers replay archive. `http://www.gosugamers.net/starcraft/replays/`, [Online; accessed December 3. 2010].

[9] The Team Liquid Community. Team liquid replay archive. `http://www.teamliquid.net/replay/`.

[10] Wikipedia contributors. Artificial neural networks — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=399864820`, 2010. [].

[11] Hugin Expert. Hugin expert. `http://www.hugin.com/`.

[12] Sander Bakkes Frederik Schadd and Pieter Spronck. Opponent modeling in realtime-strategy games. 2007.

[13] Kris Graft. Blizzard confirms one "frontline release" for '09, 2009.

[14] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[15] Kruglyak S Heyer LJ and Yooseph S. Exploring expression data: identification and analysis of coexpressed genes. *Genome Research*, 1999.

[16] Ryan Houlette. Player modeling for adaptive games. *AI Game Programming Wisdom 2*, 2004.

[17] Ji-Lung Hsieh and Chuen-Tsai Sun. Building a player strategy model by analyzing replays of real-time strategy games. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 3106 –3111, june 2008.

[18] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs, Second Edition.* Springer Science+Business Media, LLC, 2007.

[19] Ben G. Weber & Micheal Mateas. A data mining approach to strategy prediction. 2009.

[20] Manish Mehta, Santi Otanon, and Ashwin Ram. Adaptive computer games: Easing the authorial burden. *AI Game Programming Wisdom 4*, 2008.

[21] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition.* Morgan Kaufmann, 2009.

[22] Tom M. Mitchell. *Machine Learning, International Edition.* McGraw-Hill, 1997.

[23] Vipin Kumar Pang-Ning Tan, Michael Steinbach. *Introduction to Data Mining.* Pearson Education, 2005.

[24] IGN Staff. Starcraft named #1 seller in 1998. `http://pc.ign.com/articles/066/066492p1.html`, 1999.

[25] The BWAPI Team. Bwapi - an api for interacting with starcraft: Broodwar (1.16.1). `http://code.google.com/p/bwapi/`.

[26] H. Jaap van den Herik, Hieronymus Hendrikus Lambertus Maria Donkers, and Pieter H. M. Spronck. Opponent modelling and commercial games. *IEEE Symposium on Computational Intelligence and Games*, 2005.