A Process and Tool-Set for the Development of an Interface Agent for use in the RoboCup Environment

By

Paul Marlow, B.Sc.H., Queen's University

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Masters of Engineering (M.Eng.)

in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario K1S 5B6, Canada

December 13, 2004

Your file Votre référence
ISBN: 0-494-00767-2
Our file Notre référence
ISBN: 0-494-00767-2

Abstract

A tool-set and methodology was created in order to improve a basic interface that was developed to allow a human to act as a player in the RoboCup soccer environment. These tools allow play recognitions to be learned in a layered approach from previous teams and incorporated into an advanced interface that increases the reaction time of a human player. The end result is an iterative process that can be used to incorporate further learning capabilities into such an interface. With the addition of more layered skills and recognitions, it is believed that a team of humans should be able to compete effectively against a team of computer agents. At this point, the interface can be used to learn from human players instead of their computer counterparts.

Acknowledgements

There are two people largely responsible for the progress of this approach. Dr. Esfandiari acted as a guide and a valuable research resource, especially in the areas of machine learning.

Tarek Hassan also deserves special recognition of his assistance with generating the basic human interface, as well for his effort in running some of the testing trials.

Also deserving recognition are the numerous students who donated their time and efforts in testing each of our interfaces, without whom no results would have been obtained.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

In this chapter we will give a general introduction to RoboCup. In addition, we will provide insight into our motivations to our approach, as well as an explanation as to the contributions made to research within this domain.

1.1 RoboCup

RoboCup, standing for Robotic Soccer Cup, was originally developed for use of designing, and analyzing artificial intelligence techniques involving communication, coordination and competition. Since the game of soccer provides a countless number of situations, it is the perfect domain to test such techniques. It has grown into a world-wide competition where researchers from universities and other research organizations gather to promote and test their work. The goal stated by the RoboCup community is "By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champions." Of course, this is a long way off, however great strides are being made in the development of the related techniques. Depending on the research interest of those involved, there are currently five *leagues* in which one can participate: simulation, small-sized robot, middle-sized robot, 4-legged robot, and the humanoid league. Our current interest involves the simulation league.

1.2 Motivation

Multi-agent Systems (MAS) is a branch of artificial intelligence (AI) where multiple agents work collectively to accomplish a specific task. Recently, such systems have become a hot topic in the realm of Artificial Intelligence because of their attractive nature of operating in an open and distributed environment, such as the Internet [35]. Because of such a dynamically changing environment, Multiagent Systems can be inherently complex, and as such there is a great deal of interest in applying Machine Learning (ML) techniques in order to combat this complexity. In order to apply such learning techniques, a good test bed must be used, hence the reason for using a simulated soccer server.

The approach used by many, is to use Machine Learning techniques as a tool to develop an increasingly complex agent which learns on its previous learned skills (or layers), hence the term *layered learning* [28]. Because of the soccer domain, it makes little sense to attempt to learn complex and intelligent behaviors from only those primitives provided by the soccer server. Therefore, there is a set of low-level skills that must be learned before higher-level skills can even be attempted – this will be described in detail later on.

Our motivation was to create an interface agent in order to provide an interface that a human could use to play as one of the players on the field. The idea is that a team comprised of human players should be able to outperform the previous computerized teams created, and that the logs of the human players could be used to train artificial players. Unfortunately, the nature of the soccer server does not lend itself well to human interaction, and as such a human player is significantly slower at responding to the

stimuli than a computer player. Hence, the interface needed to be refined to incorporate *smart buttons* that provide a means of quickly executing more complicated skills, such as passing, using only the basic server commands available in the background. When the interface achieves an appropriate level of performance, it will then be used to learn complex scenarios such as positioning and tactics, but this is beyond the scope of this thesis.

1.3 Contributions

The contributions to research supplied within this thesis are comprised of the following:

- Development of a human interface for a human to compete as one of the soccer players.
- Identification of a set of tools and a methodology to learn and incorporate basic soccer skills to improve the human player's performance, which in turn can then be mined for higher-level learning (of tactics and strategies), at which point one should be able to remove the interface altogether.
- Low-level skills (shooting) and play recognition (can pass, can shoot) have been learned through the development and use of several utilities: Classifier, LogServer, LogExtractor, and the use of WEKA [36], a library of common machine learning algorithms.
- Play recognition options have been learned using other recognition skills, creating a layered learning approach. As an example, the *can pass* recognition, makes use of the following recognition patterns: *ball kickable*, *is teammate*, *is reachable*, *is goalie* and *opponent between* (See Chapter 5 and 6 for more details).

- Learning actions and play recognitions allow these to be incorporated into the human interface as features, improving a human player's performance. The *can pass* recognition for example, indicates to the human user which teammates a pass should be able to be completed successfully.
- The act of learning an action or a play option recognition and incorporating this into the human interface produces a process, which should allow easy incorporation of more features.
- When the human interface allows a group of human players to compete effectively against computer players, it can be extended into an interface agent in order to learn tactics and scenarios, as it is our belief that with an effective interface a team of humans should be able to defeat a team of computer agents.

1.4 Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 deals with the background information of the RoboCup Simulation League and the various machine learning algorithms we make use of; Chapter 3 summarizes the state of the art of agent-based approaches used for RoboCup; Chapter 4 is a detailed examination of our approach; Chapter 5 provides our experimental results; finally Chapter 6 provides a conclusion and a discussion of possible future work.

Chapter 2

Background

As previously mentioned, there are a number of various leagues in which research teams participate. There are four robotic leagues, in which physical robots play: *Small-sized*, *Middle-sized*, *Sony Four Legged*, and as of RoboCup 2002, *Humanoid.* In addition, and most importantly to us, is the *Simulation League*, where everything is simulated by software, allowing us the ability to create an interface for human competition.

2.1 RoboCup Simulation League

The simulation league focuses on developing the software-oriented AI techniques that go into creating an autonomous soccer player – represented as individual software agents. Of course, for an agent to be completely autonomous, it must be able to adapt to its environment. Because the game of soccer represents a dynamic environment, it is the perfect venue to pursue research in this area. In addition, soccer is played as a *team*, which implies that an agent must be able to at least coordinate in some fashion with other agents (players) in order to achieve the desired objective.

Therefore, RoboCup offers a wide variety research areas such as: machine learning and data mining techniques to learn from past experiences, as well as multi-agent cooperation and communication to be able to accomplish the end goal. Ultimately, in

competition this goal is to win, however there are many sub-goals that the process can be applied to: defending (ball clearing, blocking), attacking (shooting, ball interception).

2.1.1 Soccer Server and Monitor

A soccer server, and soccer monitor, *Figure 2.1*, has been developed [18] to simulate the game. The server actually simulates the game, while the monitor is used as a graphical display so the designers can view the games in real-time. Each player (agent) connects to the soccer server using a UDP socket connection over a local area network.

2.1.2 Communication Protocol

All communication must be carried out through the server, which is restricted so that only one teammate can be heard at one time, as well as limiting the range of communication. As in a real soccer game, it's unlikely that a player at one end of the field could hear a teammate from the other end in an actual game. In addition, each player may only send one command, for the most part, every time cycle. There are certain commands that can be executed more often, but for the main action commands such as kick and dash, only one command is possible (see *Table 2.2* for more details) The player is assumed to be idle if no command is sent. However, if more than one command is sent, then one is executed at random – and therefore may not have the intended effect.

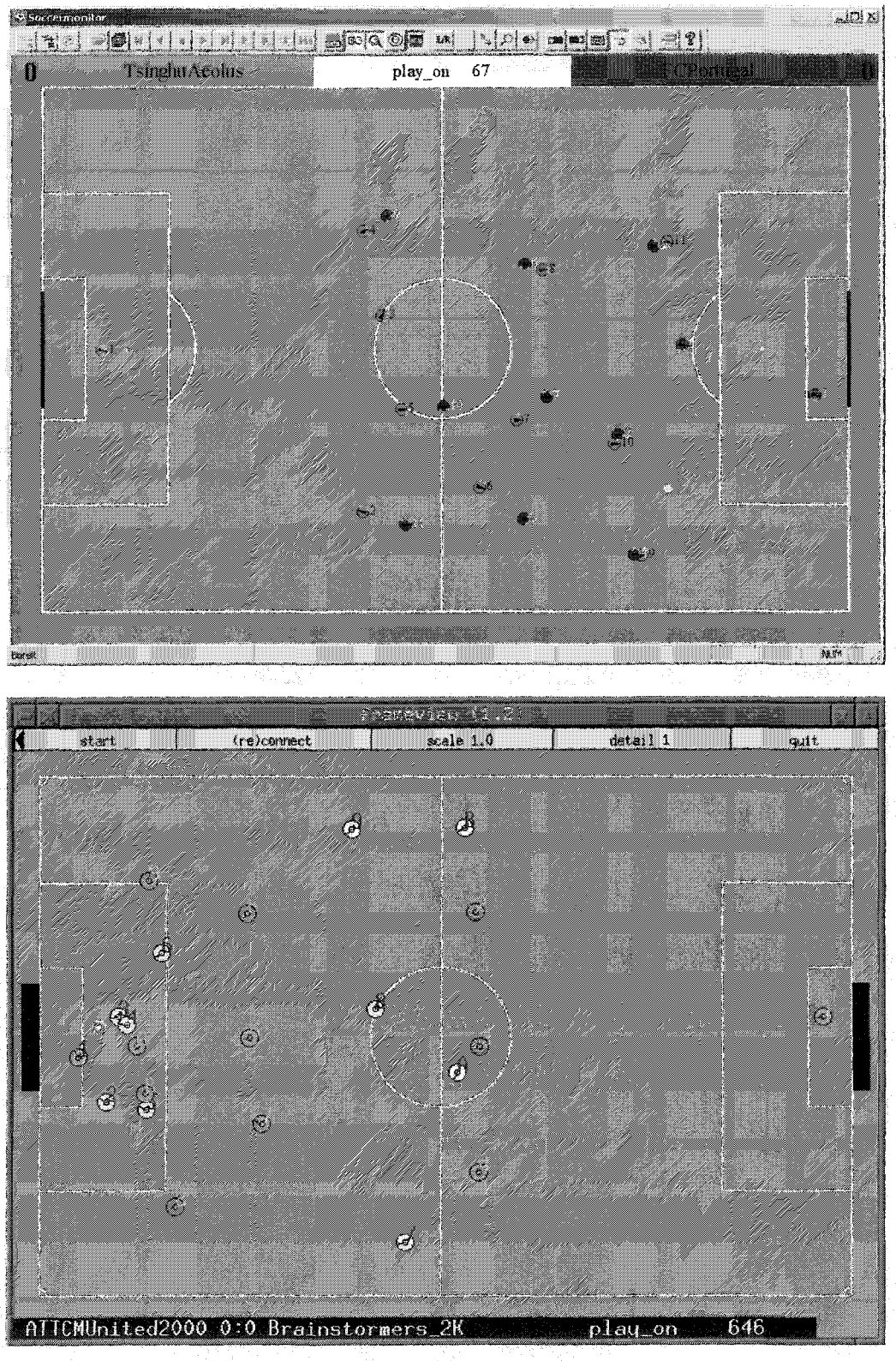

Figure 2.1: The soccer monitor
Official (top), and a third party monitor – FrameView [19] (bottom), both used for graphical representation of a soccer game.

2.1.2.1 Client Command Protocol

Before any communication between the server and the client can take place, the client must initialize itself with the server. The *init* command is used to initiate the first connection. During half-time, the players can be disconnected to allow for modifications or even substitutions. The *reconnect* command is used to re-establish a connection for a specific player with the server. Finally the *bye* command is used to indicate to the server that the player is leaving and will no longer receive commands. The specific protocols for each are detailed in *Table 2.1*.

From Client to Server	From Server to Client
(init TeamName [(version VerNum)] [(goalie)])	(init Side Unum PlayMode) Side ::= l|r Unum ::= 1~11 PlayMode ::= one of play modes
(reconnect TeamName Unum)	(reconnect Side PlayMode)
(bye)	

Table 2.1: Connecting, reconnecting and disconnecting a client. [3]

Once connected, the client can make use of a variety of commands. Some commands are time-dependent, meaning that only one command can be sent during a single cycle of 100 ms. In other words, if a command is deemed as time-dependent, no other commands sent during that cycle will be acknowledged by the server. The full list of client commands and their protocols are detailed in *Table 2.2*.

From Client to Server	Once per Cycle (100 ms)
(catch Direction) - only for goalie Direction ::= *minmoment ~ maxmoment*	Yes
(change_view Width Quality) Width ::= narrow | normal | wide Quality ::= high | low	No
(dash Power) Power ::= *minpower ~ maxpower*	Yes
(kick Power Direction) Power ::= minpower ~ maxpower Direction ::= *minmoment ~ maxmoment*	Yes
(move X Y) X ::= -52.5 ~ 52.5 Y ::= -34 ~ 34	Yes
(say Message) Message ::= a message	No
(sense_body) The server responds with: (sense_body *Time* (view_mode {high | low} {narrow | normal | wide}) (stamina *Stamina Effort*) (speed *AmountOfSpeed DirectionOfSpeed*) (head_angle *HeadAngle*) (kick *KickCount*) (dash *DashCount*) (turn *TurnCount*) (say *SayCount*) (turn_neck *TurnNeckCount*) (catch *CatchCount*) (move *MoveCount*) (change_view *ChangeViewCount*))	No
(score) The server responds with: (score *Time OurScore TheirScore*)	No
(turn Moment) Moment ::= *minmoment ~ maxmoment* degrees	Yes
(turn_neck Angle) Angle ::= *minneckmoment ~ maxneckmoment*	Yes

Table 2.2: Client control commands to the server. [3]

2.1.2.2 Client Sensory Protocol

The server is responsible for the visual (*see*) and audio (hear) information that is relayed to the players, which is sent in the form of textual strings. Without this information, it would be impossible to have a collaborative team of agents working

towards a common goal. *Table 2.3* examines these visual and audio protocols in greater detail.

From Server to Client
(hear *Time Sender "Message"*) *Time* ::= simulation cycle number of the soccer server *Sender* ::= online_coach_right | online_coach_left | coach | referee | self *Direction* *Direction* := -180 ~ 180 degrees *Message* ::= string

(see *Time ObjInfo**)
Time ::= simulation cycle number of the soccer server
ObjInfo ::= (*ObjName Distance Direction DistChange DirChange BodyFaceDir HeadFaceDir*)
| (*ObjName Distance Direction DistChange DirChange*)
| (*ObjName Distance Direction*)
| (*ObjName Direction*)
ObjName ::= (p ["TeamName" [UniformNumber [goalie]]]) // player
| (b) // ball
| (g [l | r]) // goal
| (f c) // flags - used for positioning
| (f [l | c | r] [t | b])
| (f p [l | r] [t | c | b])
| (f g [l | r] [t | b])
| (f [l | r | t | b] 0)
| (f [t | b] [l | r] [10 | 20 | 30 | 40 | 50])
| (f [l | r] [t | b] [10 | 20 | 30])
| (l [l | r | t | b]) // lines on the field (left, right, top, bottom)
| (B) // unknown ball
| (F) // unknown flag
| (G) // unknown goal
| (P) // unknown player
Distance ::= positive real number
Direction := -180 ~ 180 degrees
DistChange ::= real number
DirChange ::= real number
HeadFacingDir ::= -180 ~ 180 degrees
BodyFacingDir ::= -180 ~ 180 degrees
TeamName ::= string
UniformNumber ::= 1 ~ 11

Table 2.3: Client Sensory Protocol from the server. [3]

Noda's Soccer Server [21] is quite realistic in that:

- "Vision of the players is limited to 45° on either side, and therefore not all objects are visible during a particular sensory step.

- Players communicate by posting to a *blackboard*, which can be read by all players.
- Each player is controlled by its own separate process.
- Each player has 10 teammates and 11 opponents.
- Players have a limited amount of stamina, and hence can tire.
- Actions and sensory information is noisy, meaning information may not be exactly as perceived (i.e. player numbers and team is not visible from a distance).
- Play occurs during real-time." [31]

In this section we have examined the Simulation League in greater detail, though for updated information the Soccer Manual [3] should be consulted, as it is a work in progress. We have examined how the soccer server and the monitor operate, and looked at the main command protocols, though there are many more actions and other sensory information available. As mentioned above, RoboCup provides a wide variety of research opportunities. The simulation league especially allows researchers to concentrate on the development and use of machine learning and cooperation techniques. The following sections provide brief background information on some of the machine learning algorithms we used.

2.2 Machine Learning

2.2.1 Decision Trees

Decision tree learning is a learning technique that is used to approximate the result of a discrete-valued target function [20]. In other words, the result of a learned decision tree, will be an answer, or classification, from a finite list in response to a specific

question – the most simplest answer being a boolean response of True or False (Yes or No). What makes decision trees so widely used is that they are easy to implement, visualize, and are easily ported to *if-then* structures.

Each *node* of the decision tree corresponds to an attribute relating to the instance, where each branch descending from the attribute node indicates a specific value for the attribute. Each attribute is compared as such, until a leaf node is reached, where all leaf nodes represents a classification for the instance. *Figure 2.2* is an example of a typical decision tree [20].

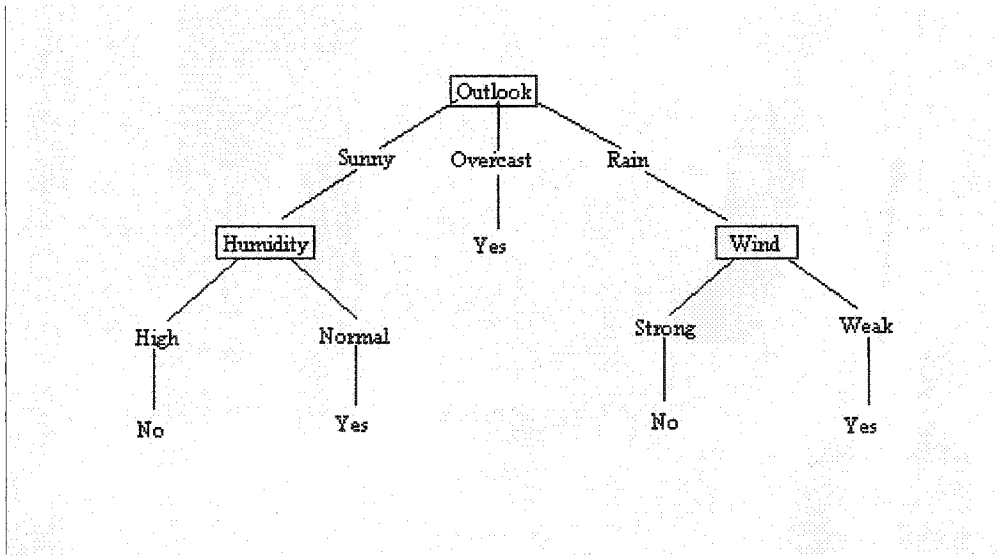

Figure 2.2: Play Tennis decision tree [20]

Even though decision trees are widely used, there are problems with certain characteristics. Decision trees are more appropriate to use in the following situations, according to [20].

- Instances are represented by attribute-value pairs.
- The target function has discrete output values.
- Disjunctive descriptions may be required.

- The training data may contain errors.
- The training data may contain missing attribute values.

However, there are instances where decision trees are not appropriate, specifically when the classification could be one of a large set of finite, or infinite classifiers. In cases like this, a learning algorithm to handle *regression* is more feasible.

2.2.2 Instance-Based Learning

Many learning methods operate by constructing a general, explicit description of the target function, however instance based learning just stores the training examples instead of extrapolating the target function. Hence, any generalization beyond the examples provided is not accomplished until a new instance needs to be classified.

The most basic implementation of this method is termed the *k-Nearest Neighbor* algorithm. Essentially, it assumes that all instances correspond to points in the *n*-dimensional space, where the classification of a new instance will correspond to the majority of it's *k* nearest neighbours, calculated by comparing the Euclidean distance. *Figure 2.3* shows *x* as the instance to classify as either negative or positive. It can be seen, that with *k=1*, the resulting value is positive, whereas with *k=5*, the value is negative.

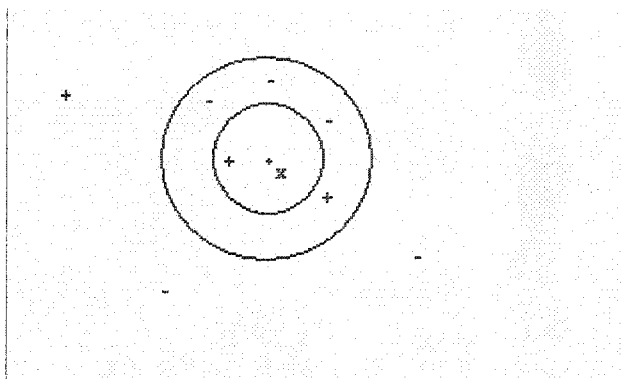

Figure 2.3: k-Nearest Neighbor, where k=1 and k=5

The strengths of instance based learning is that they work well for approximating both real-valued as well as discrete-valued target functions. In addition, it is also robust against noisy training data, and is also efficient when training on a substantially large number of training examples.

However, such approaches as *k-Nearest Neighbor* can have problems when dealing with a large number of attributes. The reason for this is the calculated distance between instances are based on all attributes of the instance, and can be misleading, especially if there are irrelevant attributes present. In cases like these, "the distance between neighbours will be dominated by the large number of irrelevant attributes" [20]. Furthermore, with the processing delay until a new query, the necessary computations needed may be significant and memory intensive.

Hence, instance based learning, and in particular *k-Nearest Neighbor*, is best suited for tasks consisting of:

- A discrete-valued or real-valued classification.
- A small, yet relevant set of attributes.
- With training data that may be noisy.
- Computation time for classification is not a concern.

Now that we've examined the details of the RoboCup environment, as well as the machine learning techniques we will use, next we shall give an overview of the approaches other teams have taken and the specific problems they address.

Chapter 3

State of the Art: Agent-Based Approaches for RoboCup

There are three types of agents: reactive, deliberative and hybrid (which combine aspects of both). According to [38], "Many researchers have suggested that neither completely deliberative nor completely reactive approach is suitable for building agents."

Purely reactive agents are inappropriate for RoboCup because the players need to have both global strategy as well as local tactics, and purely deliberative agents may be too slow to deal with the dynamically changing environment because of the complex planning and reasoning it may entail. Therefore, many teams tend to focus on a more hybrid approach.

Since its conception in 1996, RoboCup has provided researchers with an avenue to express and improve upon their work in multi-agent machine learning systems. There has been a wide range of approaches used, ranging from decision trees, to neural networks, and logic programming using Prolog. In particular, many teams tend to use some form of a layered learning approach.

3.1 Layered Learning

Layered learning is the process by which skills and behaviours of an agent are learned using previously learned techniques. The actual learning process of these lower skills differs from team to team, which algorithms are used on what kind of data (if any),

however the overall structure is the same, so that the more complex behaviours are learned from the simpler ones. We will use CMUnited's [32] layered learning implementation as an explanation.

The thought behind this is that there are two levels of learned behaviour. The client (or player) must first learn the low-level skill allowing them to control the ball individually. It's only then that higher learning can take place; learning that is deemed as a more *social* skill thereby involving multiple agents, see *Table 3.1.*

Layered Strategic Level	Examples
Player-ball	intercept
One-to-one player	pass, aim
One-to-many player	pass to a teammate
Action selection	pass, dribble, or shoot
Team collaboration	strategic positioning

Table 3.1: Strategic level decomposition.

3.1.1 Learning a Low-Level Skill

The most essential low-level skill according to [32], is the ability for an agent to intercept a moving ball. However, this is much more difficult to achieve than simply moving to a stationary ball, because of the ball's unpredictable movement (due to the sensory noise), and because the player may have to move and turn in such a way that it will lose sight of the ball, see *Figure 3.1.*

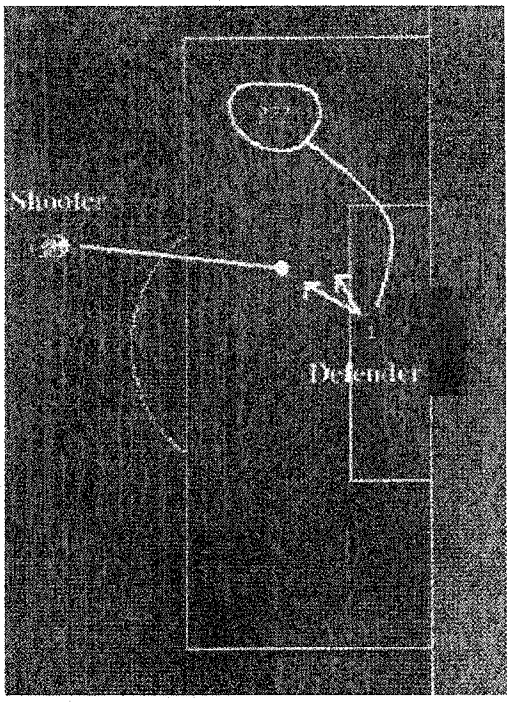

Figure 3.1: CMUnited interception.
(Left arrow – the defender moves directly towards the ball, causing it to entirely miss the ball. Right arrow – the defender moves in the correct direction, yet will lose sight of the ball in the process.)

Players must be able to intercept the opponent's shots and passes, but must also intercept passes from their teammates. The approach was to provide the clients with several training examples, and use Neural Networks [32] to train the interception skill. Since Neural Networks is typically an *unsupervised* learning technique, the initial training data is chosen at random. This is accomplished as follows: [31]

- While Ball Distance > 14, Turn(Ball Angle)
- When Ball Distance <= 14, set Turn Angle = Random [-45, 45]
- Record Ball Distance, Ball Angle, and Turn Angle
- Turn(Ball Angle + Turn Angle)
- Dash()
- Record result (from the coach: *GOAL, MISS, SAVE*).

Essentially, until the ball is within a certain range, the agent just turns and follows the ball. However, when the ball is within this range, the player turns a random angle away from the ball and dashes. Obviously the player will miss most of the time, but continuous training of 750 positive examples increases the performance dramatically from 24% saves to 86% [31].

To automate this process, a coach was used to record the result as mentioned above. A *SAVE* occurs when the agent successfully intercepts the ball. The shot is labeled a *GOAL* if the ball gets by the defending player, and is still between the goal posts (i.e. on course for a goal). Finally, the shot is deemed a *MISS* if the ball gets by the defender, but the trajectory would not allow for a goal. Of these, only a *SAVE* is considered a positive result, and hence only these are used for training.

The goal of the learning is for the player to determine an appropriate turn angle, based upon the ball distance, ball angle and previous ball distance. The Neural Network was trained for 3000 epochs, and consisted of a fully-connected net with 4 sigmoid hidden units, as well as a learning rate of 10^{-6} [31]. The weights connecting the input nodes to the hidden nodes used a linearly decreasing weight decay of .1%, and a linear output of no decay.

3.1.2 Learning a Higher-Level Decision

Once the players have learned how to control the ball, they can then learn how to cooperate as a team and make decisions accordingly. Therefore, the clients can use the previously learnt ball-interception skill in order to learn the more complex behavior of passing. In order to pass the ball, this requires action between two players. The *passer* must kick the ball to the *receiver*, who then controls the ball. Hence, the receiver must be

able to *intercept* the pass, and therefore uses the same trained Neural Network (in the case of CMUnited).

The passer however, must determine the likelihood that the pass will succeed. By using this assessment, the player learns whether or not to pass the ball. Adding an extra layer, more decisions can be made. Instead of just determining whether a pass may or may not succeed, the player can choose to either pass, dribble or shoot. However, when determining whether or not to pass the ball, there are several features, or attributes, that must be taken into account. In order to learn the appropriate action, for the given features, Decision Trees (DT) are used.

To gather the training data, like the ball interception, a coach is used to facilitate the trials. These trials involve the following: [31]

- Coach randomly places players on the field.
- Passer announces to the blackboard its pass intention.
- Each potential receiver reply with their view of the field when they are ready to receive.
- A receiver is chosen randomly during training, or by a DT during testing, by the passer.
- Various attributes are recorded by the passer describing the trial (see below).
- The passer announced the intended recipient of the pass.
- The receiver, and four defenders attempt to intercept the ball using the previously learnt skill.

- The coach classifies the trial as a *SUCCESS* if the receiver intercepts the ball; *FAILURE* if one of the opponent defenders intercept the ball; or *MISS* if the receiver and the defenders are unable to intercept the ball.

The attributes available to the Decision Tree are: [31]

- Distance and Angle to the receiver.
- Distance and Angle to other teammates, sorted by angle from the receiver.
- Distance and Angle to opponents, sorted by angle from the receiver.
- Counts of teammates, opponents, and players within given distances and angles of the receiver.
- Distance and Angle from receiver to teammates, sorted by distance.
- Distance and Angle from receiver to opponents, sorted by distance.
- Counts of teammates, opponents, and players within given distances and angles of the passer from the receiver's perspective.

Therefore, the goal of learning is to determine, using these attributes, whether a pass to a particular receiver will lead to a *SUCCESS*, *FAILURE*, or a *MISS*. In addition, the Decision Tree returns a confidence estimate, allowing the passer to choose the best candidate for a pass. As a result, even if there are only three possible receivers, each of which indicate a *FAILURE*, the confidence values can be used to select the pass with the greatest chance of success (i.e. the lowest confidence of a *FAILURE*).

There are still more examples of further layered learning. For instance, right now all the receivers do is to attempt to intercept the ball. However, the possible receiver could learn to anticipate when a pass may be coming from a teammate, and therefore obtain a

more strategic position to optimize the level of success. Another layer could be used to develop abilities to thwart opponents by moving to prevent or intercept a possible pass.

3.1.2 Layered Agent Architectures

Like CMUnited [32], many of the teams make use of some form of a layered architecture, especially those that build off of the CMUnited architecture such as 11Monkeys [13], FCPortugal [23] and TsinghuAeolus [11]. Even though each use the core layered skills of CMUnited [32], their approaches differ quite extensively. 11Monkeys [13] focuses on two types of planning: deliberative and reactive planning, consisting of three planning layers: Strategy, Group and Individual. These layers are further discussed in the section on *Strategic Planning*. FCPortugal [23] concentrates on dynamic positioning of players (see section on *Positioning*), whereas TsinghuAeolus [11] examines the problem of *Adversary Management*. This management consists of improvements to the basic dribbling and kicking mechanisms.

The real-time decision making strategy of TsinghuAeolus is structured into several modules, with attention being focused on visual control, handle-ball, offense positioning, and defense positioning [11]. *Figure 3.2* describes the overall strategy architecture outlined for TsinghuAeolus in [11].

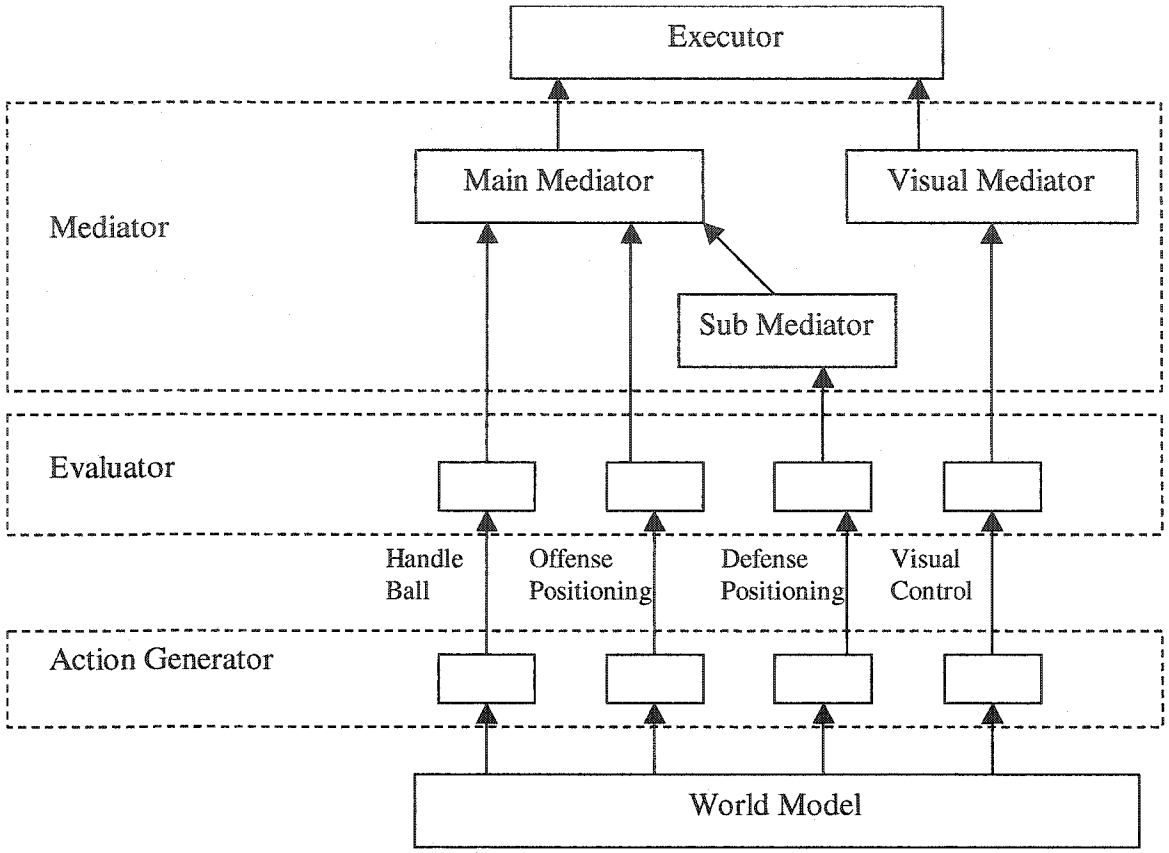

Figure 3.2: TsinghuAeolus Strategy Architecture

First the action generator is used to generate possible actions. An example of a typical action would be the pass action. Because of the enormous amount of pass routes possible, each consisting of a pass angle and kick power, an analytical approach is used to remove useless passes [11]. Next, using neural networks, the evaluation module is used to evaluate and assign a priority to each action, thereby determining how valuable the action is. Finally the mediator receives all the priority assigned actions from the evaluator. It is the mediator's job to find the optimal combination between conflicting and compatible actions. When the actions actually conflict, the mediator selects the best one based on the priority. The specifics of the TsinghuAeolus methodology are further discussed in the section on *Adversary Management*.

In addition, Essex Wizards [9], Mainz Rolling Brains [6] and FC Foo [8] also make use of a layered architecture. Each are similar in the types of items they have layered, but their main difference is the machine learning algorithms in which they use. Essex Wizards [9] make use of Reinforcement Learning, Mainz Rolling Brains [6] uses Self-Organizing Maps, and FC Foo [8] learns with Decision Trees

With Essex Wizards [9][10], the architecture consists of a multi-threaded approach that is used to achieve real-time performance, and a modular approach for overall agent implementation. The various threads include what are referred to as Sensors and Actuators. The modules are: Play Mode – current server play mode; Parameters – holds game constants and player options; Memory – current state; and Behaviours – selecting the best course of action depending on the Sensors, Play Mode, Parameters and Memory modules.

Mainz Rolling Brains [6] is not much different than those of the other teams with an architecture of three layers constituting a technical layer, transformation layer, and a decision layer. The decision layer distinguishes between five different tasks, each a module on its own:

- Standard situation module – i.e. kick off
- Goal shot module
- Pass module (selection and realization of passes)
- Ball handling module (intercepting, dribble, opponent tackling)
- Positioning module

Each module evaluates the possibility of success, and returns a grading, and the module with the highest grading will be acted upon.

However, instead of using techniques such as Reinforcement Learning to classify the behaviours as Essex Wizards [9][10], Self-Organizing Maps are used. Self-organizing maps (SOMs) is a data visualization technique used to reduce the dimensions of data through the use of self-organizing neural networks. The problem that data visualization attempts to solve is that humans simply cannot visualize high dimensional data. The way SOMs go about reducing dimensions is by producing a map of usually 1 or 2 dimensions, plotting the data similarities thereby grouping similar data together. So SOMs accomplish two things, they reduce dimensions and display similarities.

FC Foo [8] on the other hand uses another multi-layer approach of a reactive-deliberative architecture, consisting of four layers. The layers themselves represent various levels of abstraction and deliberation.

As with most other approaches, the lowest level of FC Foo [8] is essentially reactive, whereas all the other levels are more deliberative in nature. The teamwork is based on the use of finite automata and roles, where the roles represent a series of attributes used to describe some of the behaviour of a player. The decision making aspects use decision trees to select the appropriate action in the Decision layer.

Initially, each player is given a specific role which defines where the player is positioned on the field, when to take free kicks, etc. However, these roles are not static, in that they can be changed during the course of the game, and are only really used when a decision has to be made.

The finite automata is used within the Strategy layer to decide what decision tree to use. It describes when to change states, decision trees, as well as when to communicate

with various other players. The finite automata themselves are triggered by referee calls, ball position, and messages from other players.

The other two layers: Skills layer and Primitive Actions layer are used to calculate the command that is sent to the server. The following is a summary of FC Foo's architecture.

1. The *server interface* communicates with the server.
2. The *information processor* processes the sensor data provided by the server, and updates any relevant memory with this new information.
3. The *memory* keeps track of any information that is relevant to the agent.
4. The ***strategy layer*** pertains to team level planning.
5. The ***decision layer*** is responsible to perform the behavior action decided by the strategy layer.
6. The ***skills layer*** contains the different executable skills an agent contains, and is the "backbone" of the agent.

The ***primitive actions layer*** prepares a command that is ready to be sent to the server.

As can be seen, there are many teams that are designed, or built off of teams, using a layered architecture. Furthermore, three of the teams described above have been previous champions, so it stands to reason that a layered approach is not only practical, but also successful. In the next sections we describe how various teams examine different problems and their potential solutions.

3.2 Behaviour-Based

A popular architecture in RoboCup is called Behavior-based architecture, which is used by CMUnited-99 (the RoboCup-99 champion), and is capable of perception,

cognition and action. They build a model of the current state, and select appropriate actions based on this state. It maintains three states: world state, locker-room agreement and the internal state [24][30][33][37]. The world state is the representation of the agent's view of the world around it, evolving from sensors as well as the predicted outcomes of their actions. The locker-room agreement defines the teamwork structure, and is only accessed during private synchronization accomplished before game play or during half-time. Finally the internal state is where the agent stores any internal variables, and may reflect previous and current world states [32] [37]. In addition, the architecture also contains two types of behaviors: internal and external. Internal behaviors update the agent's internal state based on its current internal state, the world state and the team's locker-room agreement. The external behaviors reference the world and internal states, sending commands to the actuators [30][33]. The actions hence alter the agent's future percepts.

Essex Wizards [9][10] can also be categorized as being *Behaviour-Based* as the Behaviour module in each agent is responsible for carrying out both low-level and high-level behaviours. The most basic low-level primitives are *Kick*, *Turn*, and *Dash*. However, two extended primitives have been added to help provide higher level behaviour: *Akick* (Advanced Kick) – used for moving the ball through a series of moves to a location where the desired kick can be made; *Move* – uses a series of *Turns* and *Dashes* as well as collision avoidance to move a player to a specified location.

The high-level behaviours are built on top of the low-level ones, and are implemented as a hybrid of Q-learning, a recent form of Reinforcement Learning that does not require a model of the environment, and rule based decisions. These behaviours

consist of: *Dribble* – moving the ball while maintaining control with the same agent; *Look/Scan* – executes a series of *Turns* or *Turn-neck* commands to locate the ball; *Intercept* – used to predict the trajectory of the ball and an interception point to move to; *Clear Ball* – clears the ball from the defensive zone if no other option is available; *Send Ball* – used in the offensive side of the field to send the ball in behind the last row of defenders to avoid off-side; *Pass Ball* – scans the field to determine if a good pass is possible, and to which teammate to pass to; *Position Selection* – examines the current view of the field suggesting a good place to move the player to.

Even though both Essex Wizards [9] and CMUnited [30] can be classified as *Behaviour-Based*, they do also have the ability to position players on the field. However, there are teams that concentrate solely on this problem.

3.3 Strategic Planning

In order to focus on the problem of planning, 11Monkeys [13] is based off of the basic skills of CMUnited. 11Monkeys focuses on two types of planning: deliberative and reactive planning, consisting of three planning layers: Strategy, Group and Individual.

The Strategy layer is associated with planning the global team strategy to use, depending on the opponent model in use. This layer is used to cover all teammates and consists of stamina management, action algorithms to determine the appropriate action, and team formation. In addition, static role assignment is also assigned, specifically Goalie, Defensive, Half, etc. The Group layer is concerned with creating cooperation between close teammates, by assigning dynamic roles to agents such as a support player or ball handler. Therefore it typically only involves the three or four teammates nearest the ball. Finally, the Individual layer causes agents to act reactively to the soccer server

stimuli. Unlike the other two layers, this layer deals with the one-on-one state. Therefore, agents select the most suitable pre-planned action depending on the environment. As an example, the ball is cleared from the defensive area if there is no clear passing path available. [13]

Headless Chickens III [25] emphasized the development of the Strategy Editor used as a high-level team specification environment. It was intended for use by end users, and not programmers. The editor allows players to be positioned on the field, specifying the direction a player should kick and/or dribble when they obtain control of the ball. One can also supply different player formations and passing or dribbling patterns for a variety of situations. In addition, each player can be given a style of play (i.e. defensive or offensive).

The architecture is split into two levels, one for skills and the other for strategies. The skills layer uses sensory information from the world environment. The strategy layer however only uses the world information as abstracted fuzzy predicates. Each fuzzy predicate is associated with a Java class that uses the sensory information to assign a value to the predicate between one and a hundred. The higher this value, the more the "predicate seems to be true".

The strategy is separated into two parts, the individual strategy editor and the team strategy editor. As the names imply, the individual editor is used to define strategies for a single player, whereas the team editor is used to instantiate the templates created in the individual editor, and how they relate to one another.

Unlike some of the other teams, Headless Chickens agents do not communicate with one another because of the reactive nature. For coordination, it is the responsibility of the

designer at design time to ensure that the players are positioned appropriately at various stages and situations during the game. In other words, it's up to the designer to predict the course of the game.

Once a team has elements consisting of behaviours, positioning, and planning, the team can begin to compete more effectively. It is at this point that a team can focus on more advanced techniques such as *Adversary Management.*

3.4 Adversary Management

TsinghuAelous [11] implemented key technology that allowed them to successfully win the RoboCup 2001 and 2002 championships, as well as to finish second in the 2003 tournament. This technology included the implementation of basic adversarial skills that were developed using Dynamic Programming in combination with a heuristic search algorithm, as well as a reactive strategy architecture.

As with many of the other architectures previously examined TsinghuAeolus was created by referencing the source code of CMUnited [11]. The basic skills that the TsinghuAeolus lists as important elements in order to win matches are: interception, dribble and kick. However, their main focus was on adversary management [11]. This management consists of improvements to the basic dribbling and kicking mechanisms.

First, adversarial dribbling is completely hand-coded. In other words there is no learning associated with adversarial dribbling. There are only two simple rules for adversarial dribbling. Firstly, the ball must remain within the kickable area of the player every time cycle. Secondly, the ball must remain outside the kickable area of all opponents. Meeting both of these rules allows the player a greater level of control over the ball.

Second, adversarial kicking makes use of Reinforcement Learning in order to evaluate all possible kicks between two positions, and can be completed in several seconds [11]. This evaluation is taken as heuristic knowledge, and is used in conjunction with a searching algorithm similar to A*. The algorithm "is used to find a possible routine for the ball to both avoid the nearby opponents or sidelines and move effectively as intended" [11].

Further improvements have been made to TsinghuAeolus by combining Q-learning with their adversarial planning as a new approach to the kick problem [12]. The kick problem corresponds to accelerating the ball to the desired speed. Because of the ball's initial speed and direction, it is often the case that a series of kicks are needed in order to achieve the desired result [12]. Not only is a series of kicks necessary, but one also wants to create an optimal plan for a list of kicks. This plan can be evaluated by three factors:

- "Efficiency, which refers to whether to complete a kick task in cycles as few as possible.
- Robustness, which refers to whether to keep efficient with noise
- Adversarial, which refers to the ability to complete tasks under the other opponents' interference." [12]

The new solution proposed involves two steps. The first step, offline learning, involves discretizing the state and action spaces. The results of applying Q-learning to the discrete actions are then stored as heuristic knowledge. A heuristic search algorithm is then used to find the optimal kick strategy [12]. Although this sounds extremely similar to the process previously described, the method of Q-learning is applied only to this kick problem.

Throughout this chapter we have focused on various architectures, most of which either use some form of a neural network, fuzzy predicates, or decision trees, and do not rely on human input. However, there are approaches in which the human element play a more pertinent role.

3.5 The Human Element

Many of the past approaches have revolved around created self-sufficient teams that learn and adapt to their environments during a game of soccer. Most of these approaches are based off of CMUnited's layered learning code, using some form of neural networks.

There are however some teams out there, like ourselves, who are interested in some measurement of the *human element*. ULBS [2] is concerned with learning from a human in a real game, and Atsumi Laboratories [1] was interested in providing an interface for a human to play as part of the simulation league.

ULBS [2] also uses a hybrid approach, although the process of learning is different, which is to have each team member learn from a human player in a real game, however how this data is actually achieved was not described. The strategic and tactical behaviours are learned through a feed-forward neural network denoted as ANN [2], using a modified back-propagation algorithm.

ULBS [2] is a hybrid approach that also makes use of layering. A multi-layer agent architecture is used because of the efficient modeling of behaviour abstraction levels. It is structured into three layers: a) reactive – reacting to the environment through pre-planned actions; b) deliberative – consisting of two mixed components, a neural network for strategic planning, and a knowledge based system (KBS) for ball handling; c) cooperative planning [2] – referred to as the Group layer in [13].

The reactive layer is responsible for the behaviour of the agent when it controls the ball. The inputs consist of the *see* and *hear* information from the server, thereby updating the *world perception* (the agent's view of the environment). As mentioned, the deliberative layer consists of two components: ANN and a KBS. Both determine the tactical behaviour of the player, however the ANN is used when the player does not have possession of the ball, whereas the KBS is used when the player has control. Finally, the last layer is the most abstract of the three, and is influenced by the coach on a team-wide basis.

As a completely different approach, Atsumi Laboratories [1] developed a basic interface to allow a human to play as one of the soccer players, see *Figure 3.3*.

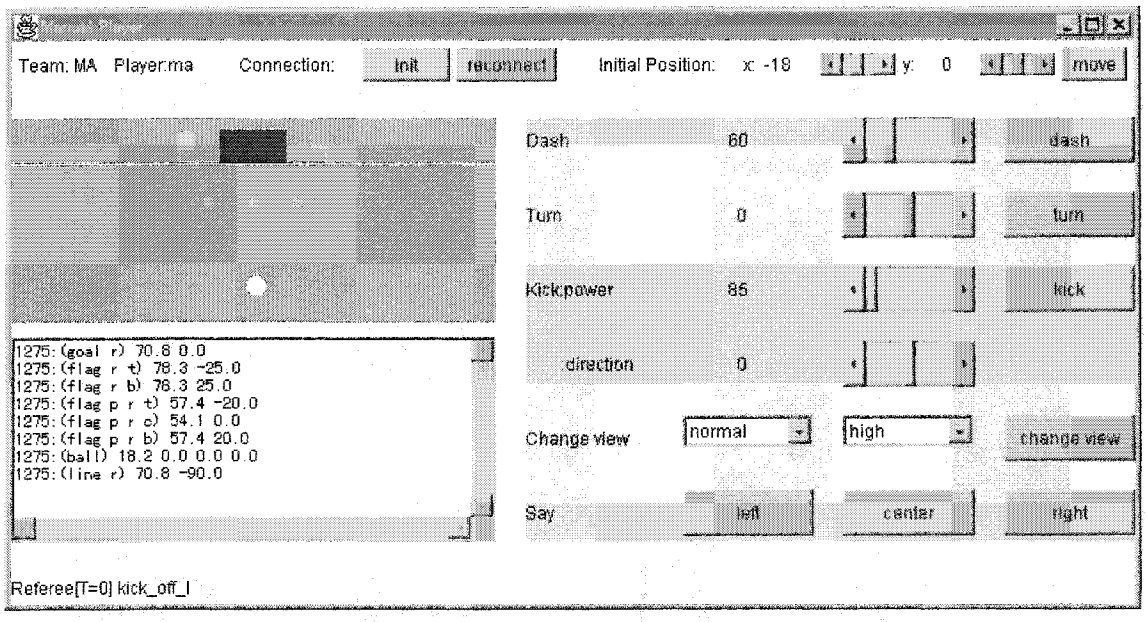

Figure 3.3: Manual Soccer Player developed at Atsumi Laboratories

However as indicated by [1], "It is difficult for a human to win against programmed soccer player by using this interface." Given the speed at which a computer team can make decisions, the time involved for a human player to manually choose specific values

using the scroll bars is too great. After these results were obtained, no further work along this line was completed.

Our approach is to extend upon this with a series of enhanced versions. The first version would not offer any more functionality than that of the Atsumi version, except for one major difference. The series of buttons and scrollbars to be manipulated is what makes the Atsumi Manual Soccer Player extremely slow. Our version uses keystrokes and mouse button clicks for the actions. The power of kicks and dashes, and the direction for turns are all determined by the position of the mouse pointer with respect to the icon representing the human player. Unfortunately their code is not available for download, and most of their documentation is in Japanese, but it is believed that these simple changes alone would result in an improvement in response time.

3.6 Learning From Conceptual Aliasing

To successfully teach an agent the teacher must have more knowledge on the learner's state [34]. The more information a teacher has about the state of the learner, the better it can adapt to ensure the points are understood. However, this knowledge is typically limited, because the teacher and learner are completely independent agents, which causes two problems: perceptual aliasing and action aliasing [34]. Perceptual aliasing is caused by different internal representations of the world between the teacher and learner. Action aliasing essentially means that instructions from the teacher may not directly correspond to action commands known to the learner. The two of these problems together is known as Conceptual aliasing.

To combat the problem, the teacher sends two messages to the learner. The first is teaching instructions that direct the learner to the goal state. The second is an evaluation

(good or not good), indicating whether the learner's reaction to the instructions provided result in a positive or negative state.

The algorithm can be summarized as follows: [34]

1. The teacher selects and sends an instruction to the learner
2. The learner then selects an action based on the instruction. If a success rate cannot be properly calculated (i.e. the learner hasn't received enough instructions), then an action is randomly chosen. Otherwise, an action is chosen based on the success of past teaching.
3. Learner executes, and the teacher observes the action.
4. Teacher provides the learner with feedback, good or not good.
5. The learner then adds the instruction, selected action, perception and the action's evaluation to it's list, and calculates the correlation among them.

The algorithm was tested using a coaching situation in RoboCup (directing an agent to the position of the ball), by sending three types of instructions: left, forward and right. Although the idea is interesting, the experiment used in the paper was too simple to be of use, and the agents really don't learn much.

Now that we have examined the relevant background information and state of the art, the next chapter focuses on our approach and how it differs from those previously presented.

3.7 Discussion

Throughout this chapter we have examined the approaches other teams have done, whether it's the introduction of a layered approach as in CMUnited [32], or the development of solutions to specific problems as described in TsinghuAeolus' approach

[11]. The main benefit of using a layered learning approach is the simplicity and the modularity. Since the approaches are split into layers (or modules), the development of each layer is simpler than an overall implementation would be. Furthermore, the modularity aspect allows the substitution of one or more modules with ease. However, none of the approaches thus far has attempted to make use of machine learning algorithms to learn from traces of plays by another agent or a human.

Our approach consists of something similar to that of Atsumi Labratories [1], where we created a basic interface to enable a human to play as a member of a team in the simulation league. Upon poor performance, which was expected, a more advanced interface was created to include what we term as *smart actions*, which make use of machine learning techniques to learn the more basic skills such as pass. As one plays using the interface, teammates will be flagged if a pass to the teammate should be successful. To pass, either click on the visual representation of the player, or use the keyboard to select the player number. In order to learn these skills, utilities have been developed to extract and learn from other teams such as TsinghuAeolus [11].

Our *LogServer* is used to create a log of commands sent between the soccer server and a set of players. The *Classifier* can be used to view visual information via logs, allowing a human user to classify various basic situations such as object distances and directions from real-valued pairs to discrete-valued pairs. In addition, data pertaining to more complicated scenarios such as passing and shooting can be extracted via the *LogExtractor*, which loops through all log files, extracting the visual information resulting in a kick action that was actually determined as a passing attempt. Each of these tools is explained in greater detail in *Chapter 4*.

Once the building blocks are in place, it should be relatively straightforward to add more smart buttons such as *Shoot*, and even the ability to maintain positioning. Furthermore, these learned skills could then be applied at another layer to develop strategies and tactics.

Chapter 4

Approach

Our long-term goal is to create a set of tools and interfaces that will enable us to learn individual game play, as well as strategies and positioning, directly from a human player. As a step to achieve this end result, and the main contribution of this thesis, is the creation of an interface agent based off of learned skills, allowing a human player to interact with the *Soccer Server* as an individual player on a team. To do this, we make use of supervised learning techniques in order to learn from the actions of other teams like the previously discussed TsinghuAeolus [11] in a variety of games and scenarios. It is our current belief that given the proper interface into the simulation world, a human player should be able to effectively compete and even possibly outperform teams based solely on computer decision making. However, an interface to simply supply the user with the ability to manually build the client commands to send to the server, as with the Atsumi Laboratories Interface [1], is insufficient. Their interface makes use of scroll bars to specify exact values to send to the soccer server for commands like *dash*, *kick*, and *turn*. This is a time consuming process when compared to the decision making speed of a computer team. Our initial interface, ITAS (In The Agent's Shoes [7] – *Section 4.2*), was also ineffective even though its usage is simpler than the aforementioned Atsumi Laboratories Interface [1].

Because of this insufficiency, it is necessary to improve upon the interface, which was accomplished by using supervised learning techniques to learn basic skills such as shooting and passing, and then incorporating these results within a more advanced interface. In order to create the advanced interface (*SmartITAS*), we have developed the following set of tools for the logging, extraction and classification of raw *Soccer Server* data, which we used in conjunction with the RoboCup 2001/2002 champion TsinghuAeolus [11]:

- **LogServer** – logs all messages from a player to the server and vice versa.
- **LogExtractor** – takes a set of *LogServer* files from a game, and extracts information pertaining to ball information, kicks, passes and goals. It has been created using an object-oriented approach to allow further extractions to be modularly added. Furthermore, this tool automatically creates classification files based on the extracted data.
- **Classifier** – takes a single player log file generated by the *LogServer* as input, allowing a user to classify instances including: ball kickable, whether objects are between the player and another object, and the conversion of real-valued distance and direction information into discrete values.
- **SmartITAS** – the advanced version of ITAS with the integration of learned classifications from the *LogExtractor* and/or the *Classifier*.

The incorporation of the learned skills into the interface is an iterative process. This is accomplished by introducing *smart buttons* into the interface that represent actions that can be executed, which are based entirely, or in part on, learning techniques. The reason we have chosen to include decisions learnt instead of hard-coding, is that the learning

process is more flexible, since the development of the *Soccer Server* is not static. Any changes to the *Soccer Server* could require changes within the code, whereas using a learned approach simply requires relearning. Take the decisions of whether the ball is kickable or not. In the 2D environment, this can simply be a configuration parameter. However with the move to 3D, determining if the ball is kickable can become more complex. Our current advanced interface contains two actionable items: shooting and passing. However, each of these actions required skills and/or observational learning. See *Section 4.6* for more detailed information on the resulting interface and its combined decision-making processes.

It is the development of our tool set as well as this iterative process of learning skills and play recognitions, and then incorporating them into the interface that is the main contribution of this thesis. Through each additional feature added, the interface continues to make playing the game as a human player easier. The simpler, yet more powerful, the interface is, the quicker the human response time, and hence results in a more competitive game. The resulting learning model can then be imported into a RoboCup team for the decision making process, or specifically for us, into our advance interface. *Figure 4.1* depicts all of our tools and how they interact with each other.

It is our future goal that the interface can become powerful enough using this process, such that it could provide an avenue to learn from human behaviour instead of computer players. This would allow developing tactics and scenarios based off of human playing, although this is beyond the scope of this thesis. In the next section, we give a brief explanation of what an interface agent is and how they have been used in other applications.

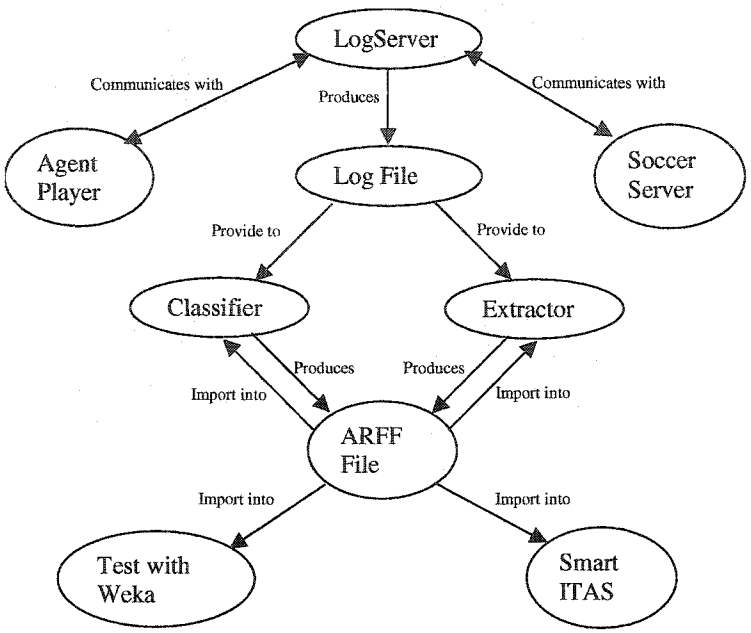

Figure 4.1: How it all fits together.

4.1 Interface Agents

Learning interface agents, as described in [17], are "computer programs that employ machine learning techniques in order to provide assistance to a user dealing with a particular computer application." Such a system works by "looking over the shoulder" of a user to detect patterns.

Previous interface agents have made use of knowledge engineering or end-user programming in order to acquire the necessary level of knowledge. As an example, [16] incorporate the use of agents, which are based off of a collection of rules programmed by the user for task-specific information processing. The problem is that the onus is on the user to not only create the rules that provide the agent with knowledge, but also to maintain these rules over time.

In order to solve the problems of the rule-based approach, machine learning techniques were incorporated into the systems as in [14]. In the Calendar Agent [14], the approach was to use memory-based reasoning in conjunction with rules. In addition, the results in [15] indicate that the use of machine learning techniques, "achieves a level of personalization impossible with knowledge engineering, and without the user intervention required by rule-based systems".

In [17], an email application was used as the domain. The interface agent would observe the user's behaviour on aspects relating to reading, printing, replying, forwarding, as well as message movement to folders, and assignment of message priority. These user patterns were captured using Memory Based Reasoning [27].

Schlimmer and Hermens [26] use a different application. Their approach is to create an interactive note-taking system with two distinctive features: prediction of user input, and construction of a custom button-box user interface upon request. To accomplish these features, the system learns two structures. Finite-state machines are used in order to characterize the syntax of the user's notes, which are constructed using grammatical inference, with decision trees situated at the states of the finite-state machine used to make predictions.

The goal in [5] was to introduce agent-oriented techniques within the domain of network management. The objective was to learn which action to trigger, depending on the event, or sequence of events received. The interface agent used is comprised of two main components: a learning system and a chronicle recognition system – a data structure that is used to store a sequence of events while associating each with temporal knowledge [5]. The chronicle recognition system takes event notifications, and attempts

to match these with chronicles stored in a confirmed database. The learning system creates chronicles and stores them in the unconfirmed database, based upon the actions of the network supervisor. These chronicles are then moved to the confirmed database when they have *matured*. The steps for the learning process are therefore: creation of chronicle, evaluation, and confirmation.

From these examples of interface agents, a short list of required elements can be extrapolated to deem what is necessary with respect to an *interface agent*.

- A user: typically a human user, but it could be another computer program
- A problem domain (i.e. Email, Automated note-taking, Network Management)
- An interface which houses the learning agent
- A data structure to capture the knowledge
- A recognition and execution engine appropriate for the data structure
- A machine learning algorithm appropriate for the data structure

In the next section we will describe our initial interface along with its positive and negative points.

4.2 Initial Interface: In The Agent's Shoes (ITAS)

The original idea behind ITAS [7] (*In The Agent's Shoes*) was to provide an interface for a human player to compete as one of the agents on a team.

The only other interface that has been created to allow a human player to compete as one of the soccer players, was developed at Atsumi Laboratories [1]. As previously described in *Section 2.1.2.1*, the client command protocol constitutes of an action and possibly parameter values associated with the action. An example of this is the *kick*

command, which expects two parameter values – *power* and *direction* of the kick. The Atsumi Laboratories interface uses a series of scrollbars to allow a human to select these desired values. This process of specifying values is considerably slow compared to the speed at which the computer players make their decisions, and although it allows such customization, the slowness makes it completely useless for providing a game of winning data.

ITAS takes the idea of an interface, and expands upon it by simplifying the amount of human interaction required. All actions are executed via mouse clicks on the screen and/or keyboard keys, and the values for the parameters are automatically calculated based upon where the mouse is clicked on the playing area, see *Figure 4.2* for a screen shot of ITAS.

ITAS has the three basic actions that correspond to the three main client actions: dash, turn and kick. Dashing works by using the left mouse button, with a power relative to the distance one clicks from your onscreen player (the red dot at the bottom of the screen surrounded by a series of green circles that indicate distance). Turning is accomplished by using the right mouse button, with an angle equal to the relative angle of the click from the direction the player is facing. Kicking is a little more complicated. Like both dashing and turning, the power and angle are taken from the mouse pointer's *position* on the screen at the time of the kick, which is executed using the space bar.

In addition to these basic actions, there is one additional action that has been added which is not available through the Asumi Laboratories Interface, and that is the *Chase ball* action. In order to speed up play, pressing the 'c' character signifies to the

application to continuously turn and dash towards the ball, until the action is cancelled by pressing 'c' a second time, thereby causing the player to run after the ball.

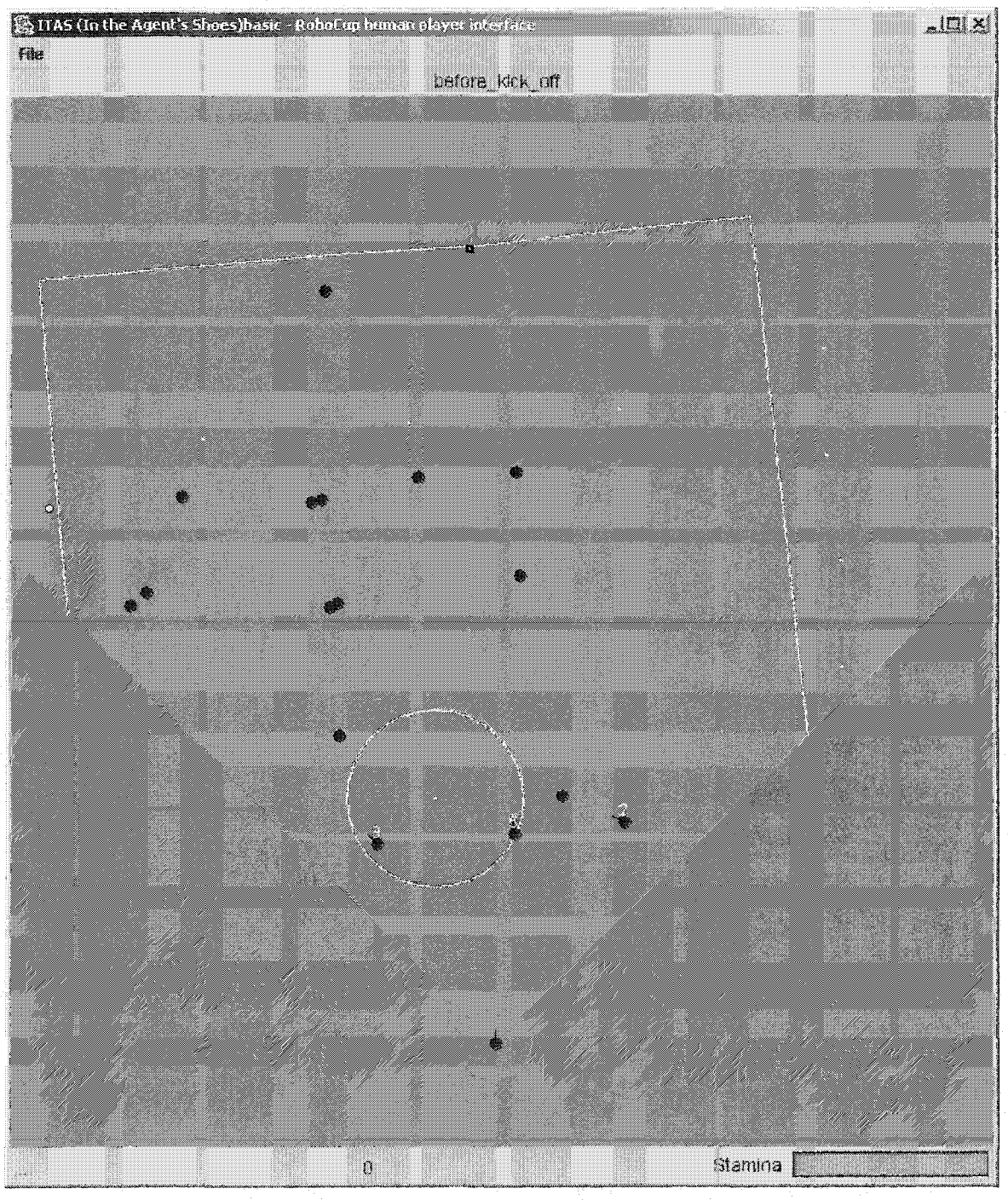

Figure 4.2: Screen shot of ITAS

Without having to specify each command, with their corresponding values manually, ITAS provides an interface that is simpler and quicker for response times. However, it was soon discovered that a human player is not able to make quick enough adjustments to effectively compete against the computer players. Simply providing an interface that allowed a person to kick, turn, and dash, is not sufficient. The reason being that agents corresponding to computer players can make decisions and execute actions on a much quicker scale than a human agent can when making use of the interface.

Through experiments, it was found that a human player(s), competing against computer agents, very rarely even had control of the ball, an average of only 3 times per game. Taking such statistics into account, 3 pass opportunities per game with an average game length of 10 minutes, in order to obtain any significant amount of data to learn (500 training examples), it would take a user in the magnitude of 278 hours, which is quite inefficient.

Since a human player is too slow at making decisions, with respect to those of the computer, any data that would be collected would be highly prone to errors. Therefore, the tasks themselves must be sub-divided and learnt separately, such as passing the ball. Although it was thought this would be a good candidate for an interface agent on it's own, the fact of the matter is, that again the speed at which a human can make decisions and act, comes into effect.

The goal is to not simply learn from any agent, but a successful one. In fact, it is extremely difficult for a person using this interface to compete against such teams as CMUnited [30], FCPortugal [23], and TsinghuAeolus [11], all former RoboCup champions, as can be seen from *Table 4.1* that depicts a series of ten games of five

humans competing using ITAS against five TsinghuAeolus players (no goalies). Therefore, the basic functionality of the interface is not enough to learn directly from a human.

Game #	Humans (5)	TsinghuAeolus (5)
1	0	29
2	0	30
3	0	26
4	0	31
5	1	29
6	0	27
7	1	29
8	0	29
9	1	26
10	0	28

Table 4.1: Game results of ITAS against TsinghuAeolus (no goalies)

It is believed though, that with a more advanced interface, a person could compete more effectively, and in turn allow an opportunity to learn based on human actions. This advanced interface will include learned functionality to visually indicate to the player when a *pass* to a teammate or a *shot* towards the goal should be successful. Furthermore, the *kick direction* and *power for passing* and *shooting* have also been learned and integrated into the interface. With the creation of such an interface, it could be extended into an Interface Agent, which learns a user's actions by "looking over the shoulder". Such an approach could be extremely useful for developing tactics and scenarios for game play. Therefore, to make a more advanced interface with smarter actions, supervised machine learning techniques are used on data obtained using our *LogServer* utility, as described in the next section.

4.3 Step 1: Using the LogServer

To obtain the data necessary to learn, the *LogServer* utility was created to allow an individual (artificial or human) player's actions with server responses to be logged to a file. It works just like the RoboCup soccer server does, by enabling one or more players to connect to the *LogServer.*

Instead of a player connecting directly to the soccer server, the player instead connects to the *LogServer* on port 7000. The *LogServer* is then responsible for establishing the socket connections necessary to forward commands from the player to the server, and back, all the while logging these commands to file. The benefit of this approach is that such a logging method could work with a computer agent, or even a human player using the interface. *Figure 4.3* shows a screen shot of the *LogServer*, *Figure 4.4* depicts the overall architecture of the *LogServer.*

The *LogServer* itself is an application that logs all communication between the soccer server and any clients attached via the *LogServer.* The log files are named according to the *init* command, which is exchanged between a client and the server – namely: *<teamname>_<player number>.lsf.*

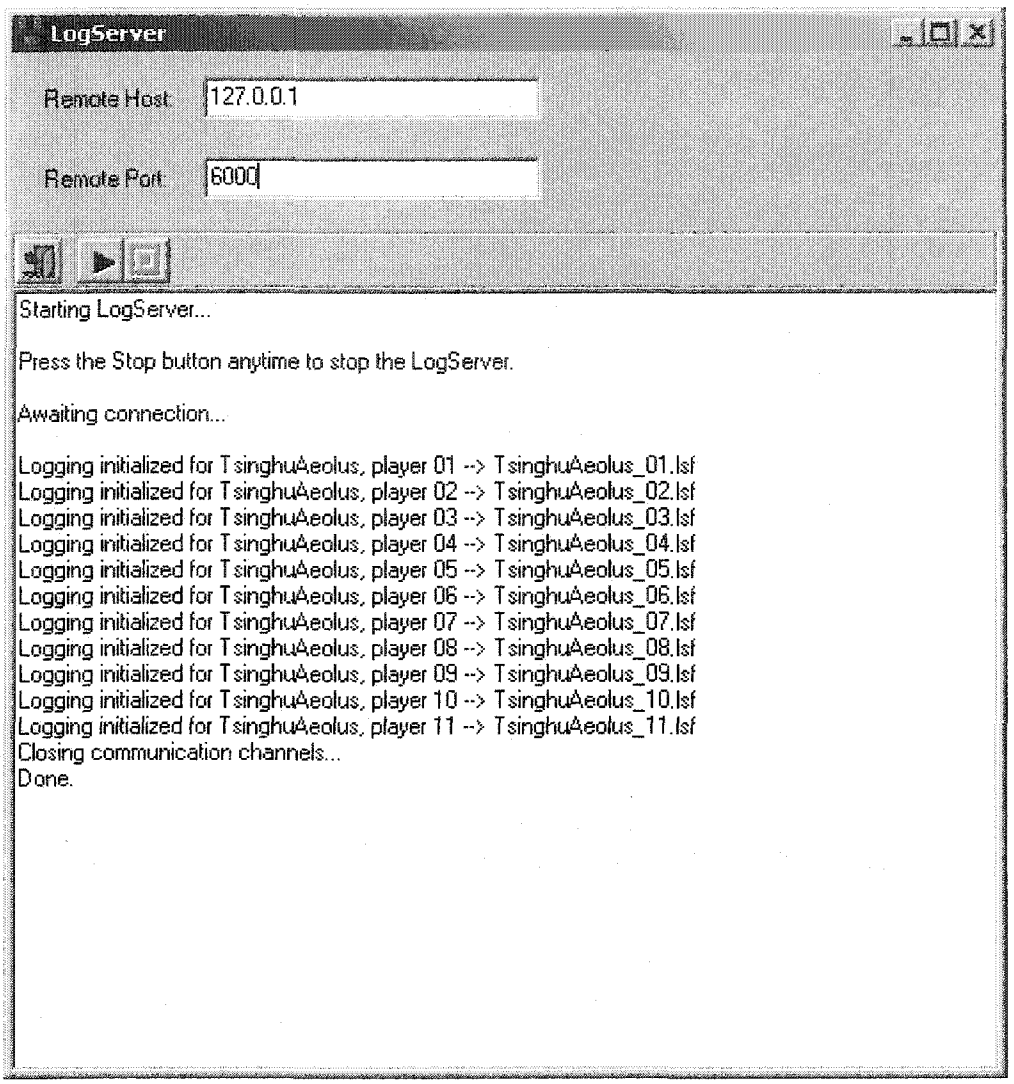

Figure 4.3: LogServer Utility

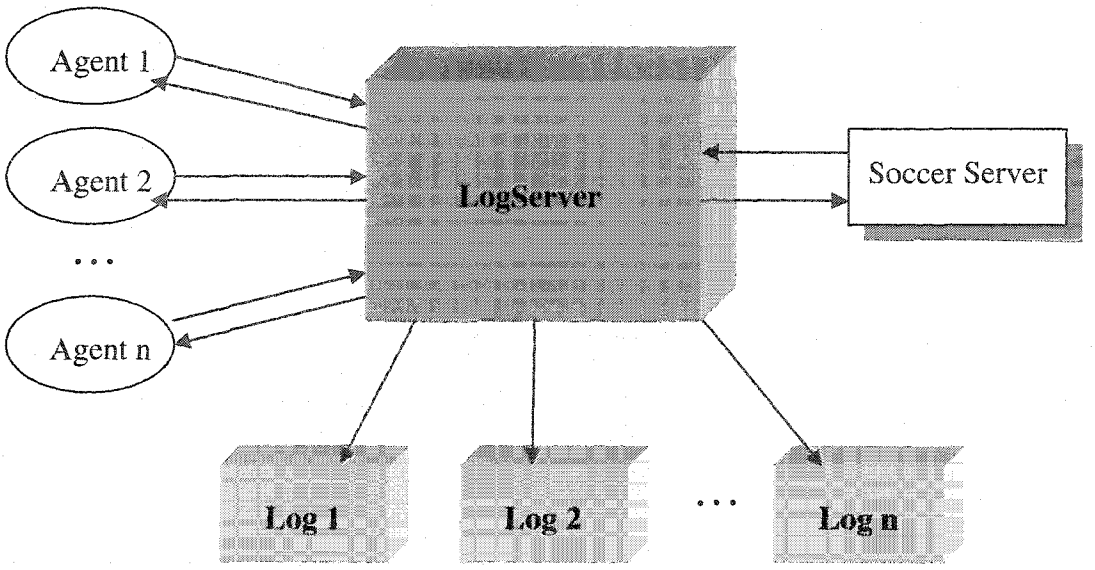

Figure 4.4: LogServer Architecture

The functionality of the *LogServer* is simple. It relays commands from the agent to the server, and vice-versa, while at the same time logging these commands to files. Each agent has their own log file, see *Appendix B*, as the *LogServer* is capable of serving multiple clients. Typically the agents would connect to the soccer server directly, however allowing them to connect to the *LogServer* in the same manner creates an easy way to obtain raw data to learn from. One downfall to using the *LogServer* is that many disk I/O operations can slow down the entire process. Hence, if this is a concern, the *LogServer* itself can be moved to an entirely separate computer dedicated to disk I/O. In addition, multiple *LogServers* can be used on multiple computers to even further decrease the effect of writing to a hard disk. Because of the file I/O, there was an initial concern regarding the effects the *LogServer* would have on the outcome of a game. *Table 4.2* and *4.3* show a set of ten games using two teams of TsinghuAeolus [11] players, with and

without the *LogServer*. In the following examples, all 22 players, *Soccer Server*, *Soccer Monitor* and *LogServer* were running on the same machine.

Game #	TsinghuAeolus 1	TsinghuAeolus 2
1	3	1
2	1	1
3	3	0
4	1	2
5	1	0
6	1	2
7	1	3
8	1	1
9	1	1
10	1	2

Table 4.2: Results of 10 games using the LogServer for both teams

Game #	TsinghuAeolus 1	TsinghuAeolus 2 (via LogServer)
1	1	0
2	2	1
3	2	0
4	0	0
5	1	3
6	0	2
7	1	1
8	1	1
9	2	1
10	1	0

Table 4.3: Results of 10 games with one team using the LogServer

Game #	TsinghuAeolus 1	TsinghuAeolus 2
1	1	1
2	0	0
3	2	1
4	1	3
5	2	2
6	2	1
7	3	2
8	1	0
9	1	1
10	0	2

Table 4.4: Results of 10 games without the LogServer

As can be seen, the *LogServer* did not have a direct effect on the end result of a game. The TsinghuAeolus [11] teams combined were scoring an average of 2.7 goals total with the *LogServer*, and 2.6 goals without as described in *Table 4.2* and *Table 4.4*. In *Table 4.3*, we see a set of games where one team of TsinghuAeolus is using the *LogServer* and another team of TsinghuAeolus is not, resulting in an average score per game of 0.9 goals with the *LogServer* and 1.1 goals without.

Upon creation of the log file(s), the data must be extracted and formatted into a file that a machine learning repository like WEKA [36] can use.

4.4 Step 2: Generating Classification Data

Two tools have been developed to obtain the formatted data files for use in WEKA. The first is the *Classifier*, which is an interface for a human user to manually classify instances, while the *LogExtractor* can be used to automatically generate these files by applying some preprocessing steps on the data, as well as to make use of previous data classifications.

4.4.1 Manually Classifying using the Classifier

A *Classifier* was created that allows a human player to step through a soccer game, one cycle at a time, as perceived by an individual player. This is accomplished by loading a log file that has been generated by the *LogServer* utility. The information displayed is exactly what the player would see, migrated to graphical form. *Figure 4.5* depicts a typical example of the server *see* information, and *Figure 4.6* shows the layout of the *Classifier*.

```
(see 1891 ((f l t) 70.8 -8) ((f p l t) 50.4 -15) ((f t l 20)
46.1 14) ((f t l 30) 53.5 6) ((f t l 40) 61.6 1) ((f t l 50)
70.1 -4) ((f l t 20) 70.8 -20) ((f l t 30) 73.7 -13) ((b) 16.4 -
8 0.328 2.6) ((p "TsinghuAeolus") 30 -4) ((p "TsinghuAeolus" 7)
16.4 -9 0 0 59 146) ((P) 2.2 148) ((p "Aeolus" 6) 14.9 -6 -0 -
0.2 132 -147) ((p "Aeolus" 9) 30 2 0 -0.1 -115 -133) ((l t) 53.5
-32))
```

Figure 4.5: Typical see information

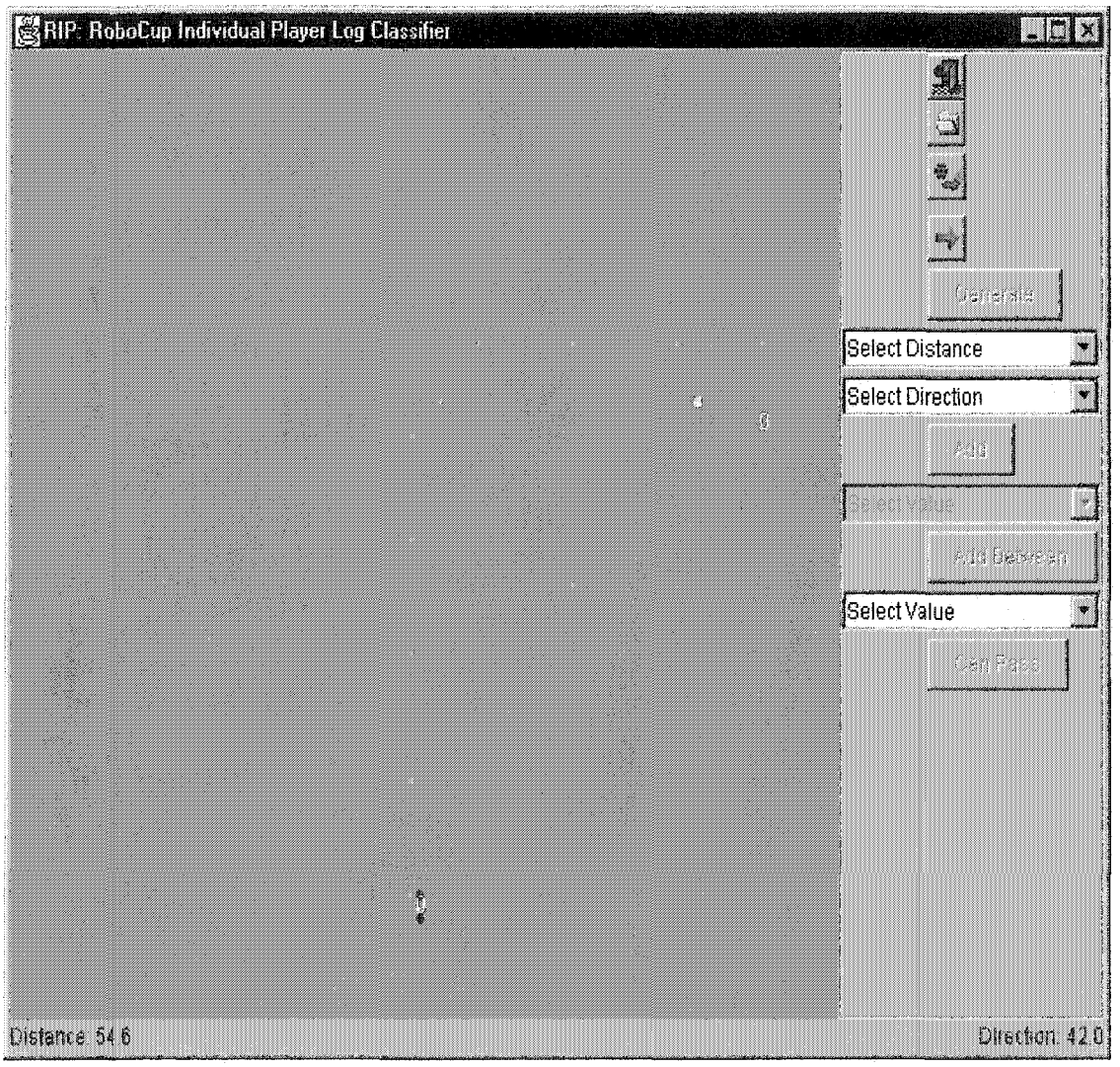

Figure 4.6: Screen-shot of the Classifier Application

The *Classifier* allows the human to select various objects, and to classify each, depending on what object(s) is selected. Currently the *Classifier* allows the following features to be classified manually: ball kickable, conversion of real-valued distance and directions into discrete values, whether a selected object is between the player and another object(s), and finally the ability to indicate if a pass to another player should be successful. Although this works, it can be quite tedious to obtain the vast quantities of classifications necessary in order to learn from. Unless what you are trying to learn is very straightforward and simple, typically one needs large number of training examples to obtain accurate results.

What the *Classifier* is useful for is classifying those instances that depend on a human interpretation. As an example, we look at object distances. The *Classifier* allows object distances to be represented as discrete values (near, close, far and distant) rather than by a real number. *Appendix C* contains a sample portion of a converted log file to see how these discrete values could be used, and *Appendix D* displays the *Positioning* ARFF file that is generated by the *Classifier* – although this is not currently used in our approach. The reason being that a more accurate result can be learned from real-valued than discrete values when it comes to distances and directions. However, the discretization of such values helps describe fewer situations, and is valuable when using a real-time situation recognition algorithm, as is being developed by Kevin Lam.

In our own case, we made use of only the *object between* classification. This classification is used for determining whether a pass or a shot can be executed with success. Therefore, the *Classifier* must be used to generate the *object between* classification even before using the *LogExtractor*, as this is more of a human

interpretation than a straight-forward analysis (*Figure 4.7*). For details regarding the knowledge represented by the object between classification, refer to the *Section 5.6.*

Figure 4.7: Is A Between B and C?

4.4.2 Programmatically Classifying Using The LogExtractor

As mentioned, many of the classifications can be automated with a little preprocessing. Unlike the *Classifier*, the *LogExtractor* can use multiple log files at once to generate large amounts of data. Therefore, where the *Classifier* is only effective in obtaining data that can be extrapolated from a single log file for manual classification, the *LogExtractor* can run comparisons programmatically on many log files to obtain more complex information. An example of such a use is to determine whether a *pass* was made between teammates. Although passing in soccer is a key element, it is difficult to determine when to pass, because of the dynamic nature of the game of soccer. In addition, when examining the logs of a game, *kick* actions do not necessarily correspond to a pass, but may represent a clearing attempt, dribbling or even shooting.

It is important to note that care needs to be taken to ensure that all the log files submitted to the *LogExtractor* are from the same game (they can be from both teams, but need to be from the same game). Since there is no way to determine with certainty that

an individual log file corresponds to an individual game, the *LogExtractor* assumes that all such files represent a single game.

There are several preprocessing steps, which are applied to the initial log files from a game. The first is the removal of inconsequential data. Depending on the team that the log files were created for, some of the information stored within the files is irrelevant for the current extraction procedures. This information consists of all *say* and *hear* commands from teammates and opponents, because most of the teams that use the *say* command have developed their own protocol, and without the knowledge of the specific protocols used, the commands themselves are meaningless. If the protocols were known however, such information may have some merit for learning.

The first iteration of the *LogExtractor* incorporated the *BallKickable* extraction methods for learning whether the ball can be kicked. Even though this could be achieved simply with a single *if*-statement, its main purpose served as a way to prove that the automated extraction idea could work. After all, if we could not learn when to be able to kick the ball from the data, then there would be little hope in learning more complex actions from the log files.

The second iteration corresponded to the extraction of each *kick* command executed by every player on both teams. It was identified early that our *CanPass* and *CanShoot* classifications will need passes and shots in order to learn. Since there are only *kick* commands (no *pass* or *shoot*), it was imperative that all kicks were extracted. Another reason for this was to enable ease of future extractions that would also make use of kicks (i.e. interception and dribbling). The *LogExtractor* was created using object-oriented techniques, thereby allowing any current (and future) extraction modules that make use

of kicks to inherit from a *KickExtraction* class, which not only extracts all the kicks both teams make, but also orders each by time cycles, indicates which teammate kicked the ball, and the visual information received from the *Soccer Server* before the kick was executed. Currently there are two extraction routines that inherit the functionality of the *KickExtraction*: *PassExtraction* and *ShootExtraction.*

The *PassExtraction* simply examines all kicks, and keep those corresponding to passes. A kick is determined as a pass if the next player to kick the ball was not the same player (called *dribbling*) but another teammate – therefore the ball was not intercepted; and if the time between successful kicks is below a given threshold – which will exclude any players that have to chase after the ball. Through observations of game play, this threshold was set to 15 time cycles.

The *ShootExtraction* differs in that only those kicks that represent a successful shot are extracted. Shooting is similar to passing, except the check isn't to see who was the first one able to receive the intended pass, but to ensure that the player who kicked, was the last player to kick the ball before a goal was scored. Therefore, the preprocessing involves searching for when the team scores, then comparing the cycle number of the goal to the most recent kicks of each player, selecting the player with a kick cycle number closest to that of the goal.

However, the usefulness of the *LogExtractor* does not stop there. In addition to performing extractions, it also uses these extractions to automate the classification process in order to determine the following end decisions: can a pass to a teammate be successful, would a shot be successful, what direction to kick the ball, and finally how hard to kick the ball.

The *CanPass* decision routine is where layered learning starts to come into play, as passing involves determining a variety of other factors: is the ball kickable, is the recipient a teammate, is the recipient reachable with a pass, is the recipient a goalie, and is there an opponent between (i.e. in the close vicinity of the pass route). All these questions have to be answered, and each lends itself well to a decision tree. Each of these can then be combined, or layered, to learn the *CanPass* skill. Note that this is only deciding whether a pass could be made to a specific player, it does not learn how to pass.

Similarly, the *CanShoot* decision routine also makes use of layered learning. Although not as complicated as passing, the decision to shoot is based off of many of the same questions as passing: is the ball kickable, is the goal reachable with a shot, and is there an opponent between the player and the goal (again, in the vicinity of the shot route). For more detailed information on *CanPass*, *CanShoot*, *KickDirection* and *KickPower* refer to *Chapter 5*. A typical screen shot of the output from the *LogExtractor* can be seen in *Figure 4.8*.

The end result of the *LogExtractor* and the *Classifier* utilities, are formatted data files that can be used in a machine learning repository such as WEKA [36]. The purpose here is to not reinvent the wheel, and therefore we make use of algorithms through WEKA [36] that have been built and tested by others.

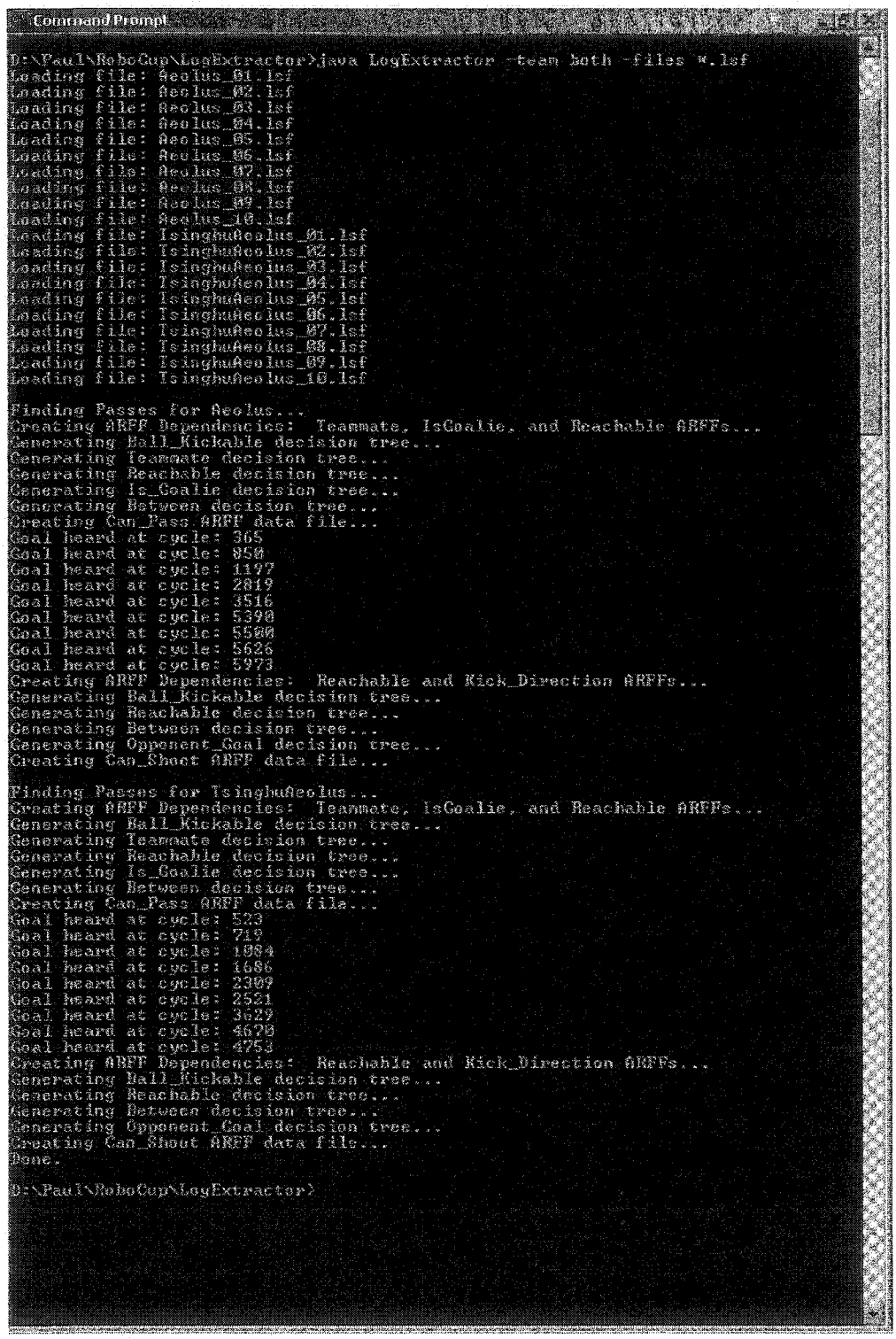

Figure 4.8: Typical Screen Shot of the LogExtractor

4.5 Step 3: Use of Relevant Machine Learning Algorithms

The University of Waikato in New Zealand has developed a machine learning toolkit implemented in Java called WEKA [36], which stands for the Waikato Environment for Knowledge Analysis. The purpose is to provide state-of-the-art learning algorithms, which can be applied to various datasets created by the user from the command line. In addition, there are also tools available for transforming the datasets. This set of tools allows the user to preprocess a dataset, use it as input into a learning scheme, and analyze the resulting classifier. Not only can the learning schemes be executed from the command line, but because WEKA was written in Java using packages, they can also be embedded and used within other Java code.

The learning methods within WEKA are called *classifiers*, that contain a series of command line options - both generic ones, as well as specific ones for individual learning schemes. In addition to the learning schemes, there are items called *filters*, which are used to preprocess the data. Like classifiers, the filters also have a standard command line interface, such that there are both generic and specific options that can be supplied at the command line.

When storing data, it is not uncommon to have it stored in a database or a spreadsheet. However, WEKA expects the data to be in its own format, known as ARFF. The reason is that WEKA needs type information about each attribute, which is difficult to automatically deduce from the attribute values themselves. The good news is that the conversion to ARFF format is quite straightforward. Most programs that deal with spreadsheets and databases contain the ability to export the data into a comma-delimited

format; in other words a list of records whose fields are separated by commas. All that remains is to edit the generated file in a word processor, and add the dataset's name using the @relation tag, the attribute information via the @attribute tag, and a @data tag to indicate the first line of the data. *Figures 4.9* and *4.10* are small examples of typical data format files expected by WEKA.

```
@relation Object_Between

@attribute Distance_Diff real
@attribute Direction_Diff real
@attribute Between {Yes, No}

@data
22.300001,8.0,Yes
26.999998,1.0,Yes
16.499998,3.0,Yes
34.5,17.0,No
24.199999,23.0,No
17.599998,12.0,No
9.0,4.0,No
24.900002,7.0,Yes
18.400002,1.0,Yes
```

Figure 4.9: Sample Between.arff – See Appendix E

```
@relation Can_Pass

@attribute BallKickable {Yes, No}
@attribute Teammate {Yes, No}
@attribute Reachable {Yes, No}
@attribute IsGoalie {Yes, No}
@attribute OpponentBetween {Yes, No}
@attribute CanPass {Yes, No}

@data
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,Yes,No
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,Yes,No
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,No,Yes
Yes,Yes,Yes,No,Yes,No
```

Figure 4.10: Sample Can_Pass.arff – See Appendix F

There are a number of learning algorithms within WEKA, but for the purposes of this dissertation, only k-Nearest-Neighbour (IBk) and Decision Trees were used. The reason we only make use of decision trees and k-nearest-neighbour is that we are currently not focusing on selecting the best learning algorithms. Our focus is to develop a toolset and process for enhancing our interface. The actual learning techniques can be interchanged relatively easily. For a complete list of algorithms available to WEKA, see [36]. The Instance-Based Learner (IBk) is an implementation of the *k*-nearest-neighbors classifiers that employs a distance metric. By default, a value of k=1 is used, but this can be changed, or determined automatically. It can also handle weighted instances, output a class distribution for categorical classes, and be updated incrementally.

WEKA has its own implementation of the C4.5 decision tree learner, called the J4.8 algorithm, which is a later, and slightly improved version of C4.5 Revision 8. This was the last public release before the commercial implementation of C4.0. Like IBk, it can handle weighted instances and output a class distribution for categorical classes, however it cannot be updated incrementally.

The data files can be fed into WEKA, in order to generate a Decision Tree, or k-Nearest-Neighbour in our case. Once the classifications have been verified and tested using WEKA, they can then be integrated into our interface.

4.6 Step 4: Integrate Tested Classifications

Once the classifiers have been learned, they can then be used within the interface. This involves creating either buttons to act as actions such as shooting and passing, or it

could be the incorporation of what we call a *play recognition.* An example of this is the *Can Pass* decision tree. This is used to indicate to a human player that a pass could be completed to a specific player. Each cycle, each teammate visible is examined using the can pass module. If this module determines that the teammate could be passed to, the teammate's colour on the field with change to indicate this.

The new interface is tested at this point to determine if the added functionality has made an improvement since the previous version, and if necessary the entire process is completed, steps 1 through 4, until an effective interface is developed. *Figure 4.11* shows the SmartITAS advanced interface.

Notice the differences between the original ITAS (*Figure 4.2*) and the advanced SmartITAS (*Figure 4.11*). The SmartITAS contains a single button to shoot, which becomes enabled when the *CanShoot* decision tree indicates a shot should be successful. In addition, based on the outcome of the *CanPass* decision tree, teammates change colour giving the user a visual cue that a pass to those teammates highlighted should succeed. In order to accomplish the pass, the user need only select the desired teammate with the mouse. All kick direction and power values are determined using information previously obtained with the k-nearest-neighbour algorithm – see *Chapter 5.*

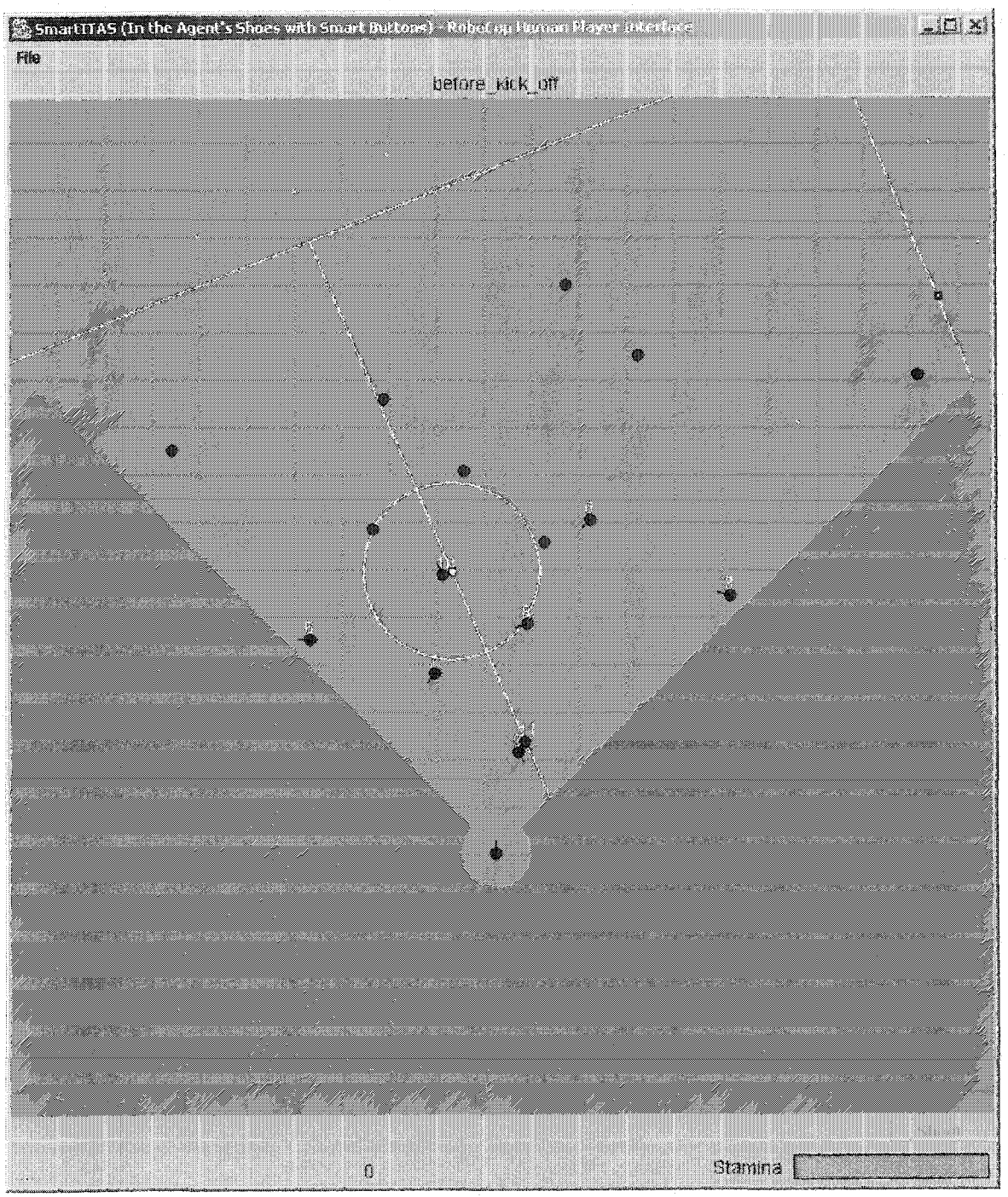

Figure 4.11: SmartITAS Interface

4.7 Step 5: Regenerate Log Files using Learned Actions

Not only can the learned classifiers be used within agents or our advanced interface, but they can also be applied to the log files themselves. This step involves replacing various parts of the log file (eventually the whole thing) with a more symbolic representation – see *Appendix C*. As can be deduced, these symbols are those learned using the Decision Trees or k-Nearest-Neighbour. Currently, each *see* command is parsed, and the distances and direction of the objects, along with their token symbol and/or teamname, are used as input values to the decision tree, which will then allow the replacement of real-valued attributes with a more symbolic one.

As an example, instead of having (b 1.0 30.0) – ball with distance 1.0 and direction 30.0, it may be replaced by (ball Left_Close), and (Ball_Kickable). Furthermore, instead of having the action *kick*, it could be replaced with a more descriptive meaning such as *shoot* or *pass_to*. The incorporation of these layers, would lend itself to easier learning for higher layers such as tactics and scenarios. Although we have a mechanism available to substitute real-valued attributes with discrete ones, we currently do not make use of this in our learning, as this will be future work.

In the next chapter we will detail our experimental results, as well as give a detailed explanation as to the machine learning algorithms we used, and how the results of each are related and layered to create more complex decision making modules.

Chapter 5

Experimental Results

Using the *LogServer* and data mining techniques presented in the previous chapter, the game action logs of TsinghuAeolus [11] were analyzed and relevant visual and sensory information were extracted and combined with machine learning algorithms through WEKA. This enabled the development of basic skills that could be used as stepping stones to more advanced learning techniques. Thus, the first *smart feature* to add to our advanced interface is *CanPass* – allowing a human player a choice of successful pass candidates. Similarly the *CanShoot* feature was added to aid the player in determining a good time to shoot. The first two sections will show a candid approach that illustrates the need for layered learning. The remaining sections detail how a layered learning approached helped solve the problem.

5.1 Initial Candid Approach

In order to obtain the logs necessary, TsinghuAeolus [11] was set up to play 10 complete games against Krislet – a dumb, yet effective team, in which all players constantly chase the ball. The *LogServer* was used to allow logging of the 10 TsinghuAeolus field players (excluding the goalie). As explained earlier, the *LogServer* acts as a router by forwarding commands between players and the server, while at the same time logging these commands to disk. The end result was 10 log files consisting of commands / responses pertaining to each individual player, see *Figure 5.1.*

```
(turn_neck 1.65)
(turn 34.34)
(turn_neck -40.84)
(attentionto TsinghuAeolus 6)
(see 1303 ((f c) 27.7 18) ((f c t) 61.6 19) ((f l t) 82.3 -
 20) ((f p l t) 60.9 -17) ((f t 0) 66.7 20) ((f t l 10) 67.4
 11) ((f t l 20) 70.1 3) ((f t l 30) 73.7 -4) ((f t l 40)
 78.3 -11) ((f t l 50) 84.8 -17) ((b) 3 0 1.98 -0.2) ((p
 "Krislet") 40.4 -10) ((p "Krislet") 49.4 -5) ((p "Krislet"
 6) 18.2 -16 0 -0.1 -76 -160) ((p "Krislet") 36.6 18) ((p
 "TsinghuAeolus") 36.6 13) ((p "TsinghuAeolus" 6) 14.9 5 -0 1
 113 -167) ((p "TsinghuAeolus") 40.4 -5) ((l t) 66 -69))
(sense_body 1304 (view_mode high narrow) (stamina 3860 1)
 (speed 0.02 -79) (head_angle 44) (kick 18) (dash 791) (turn
 824) (say 1296) (turn_neck 1668) (catch 0) (move 46)
 (change_view 425) (arm (movable 0) (expires 0) (target 0 0)
 (count 0)) (focus (target r 6) (count 1172)) (tackle
 (expires 0) (count 0)))
(dash 80.00)
(turn_neck 44.20)
```

Figure 5.1: Sample portion of a logged game.

Using the *LogExtractor*, the "relevant" *see* information pertaining to a pass is extracted. Since there is no actual *pass* command, a pass is deduced from a kick command, and whether the ball is handled by another teammate. Essentially, the *LogExtractor* examines each log file looking for instances of the kick command. It then examines the kick list, extracting kicks that result in a teammate gaining control of the ball within a defined time threshold. Through experimentation, it was determined that most passes were completed within 15 time cycles, and hence this was used as our threshold value. The end result is a list of kicks that represent successful passes. The pass extraction algorithm is described in *Figure 5.2*, outlined in pseudo code.

```
For each player do
Begin
    Open log file of the player;
    Repeat until at end of file;
    Begin
        If see information, then store as current see information;
        If kick command, then store kick and see information into the
          kick list;
    End;
End;
For each kick in the kick list do
Begin
    If current kick and next kick are different players but same team and
       the time cycle between then is <= 15 then
        store current see and kick info combination in the pass list;
End;
```

Figure 5.2: Pseudo-code algorithm of the Pass log extraction

Validation on this data is achieved by re-running the games, paying particular attention at the cycle numbers indicated as passes from which players. The RoboCup server provides additional functionality that allows a played game to be logged to a binary file, which can be opened and replayed. This process thereby allows verification of the passes that were extracted from the aforementioned mentioned algorithm. Validating one of our games resulted in 262 passes being extracted using the *LogExtractor*, with 280 passes being viewed during the game, resulting in approximately 93.4% of the passes being extracted. However, of the 262 passes, none were deemed as being incorrectly classified.

The next section examines the *CanPass* results of classification directly against the raw data and shows a need for a more concise layered approach.

5.2 Can Pass Classification from Raw Data

5.2.1 Sample Data

The resulting data set consisted of 47 input attributes: 22 players with both their respective distance and direction measurements – real values, the ball's distance and direction – real values, and the current player's number – integer from 1 to 11. The data set also contained one discrete-valued output classifier termed *Pass*, which could take one of the following values: pass_to_1, pass_to_2, pass_to_3, pass_to_4, pass_to_5, pass_to_6, pass_to_7, pass_to_8, pass_to_9, pass_to_10, pass_to_11, no_pass. The output therefore indicates if the player passed, and to which player.

P	O1_Dis	O1_Dir	...	O11_Dis	O11_Dir	...	T11_Dis	T11_Dir	B_Dis	B_Dir	Pass
1	30.0	-6.0		22.2	12.6		?	?	10.2	3.1	no_pass
2	14.3	-13.2		?	?		32.4	12.1	0.6	0.2	pass_to_1
3	?	?		42.1	-3.5		16.8	-3.4	0.8	-1.2	pass_to_11
...											...
11	11.9	10.3		?	?		?	?	2.2	0.9	no_pass

Table 5.1: Example table of raw attributes and values.

All this raw data, similar to the example shown in *Table 5.1*, comes directly from the *see* information without any preprocessing. Of the 47 input attributes, the first indicates the player number that the see information belongs to, and in the case of a pass, is the state of the field before the player decided to pass the ball. The next 22 attributes pertain to each opponent's distance and direction from the player indicated in the first attribute. Hence, O1_Dis is the distance of opponent player 1, and O1_Dir is the direction to opponent player one. Similarly, attributes 24 to 45 indicate each teammate's distance and direction, and attributes 46 and 47 pertain to the ball's distance and direction. Finally, the last column is the output classification, which results in either a *no_pass*, or an indication

of which player the pass is/should be intended for. Any information that is not available from the see information is indicated in the chart by a question mark for "unknown".

5.2.2 Decision Tree

The J48 algorithm in WEKA was used to generate a decision tree over the use of k-nearest neighbour, as it is quick, and can handle many attributes. The resulting decision tree is a complex, and unreadable tree. Even from this diagram, see *Figure 5.3* for the visual decision tree and *Figure 5.4* for the descriptive tree, it can be determined that either the raw chosen non-categorical attributes were inadequate or 952 instances are simply not enough entries to accurately learn upon the raw data without any preprocessing. Unfortunately the selection of non-categorical attributes are not available directly from the raw data.

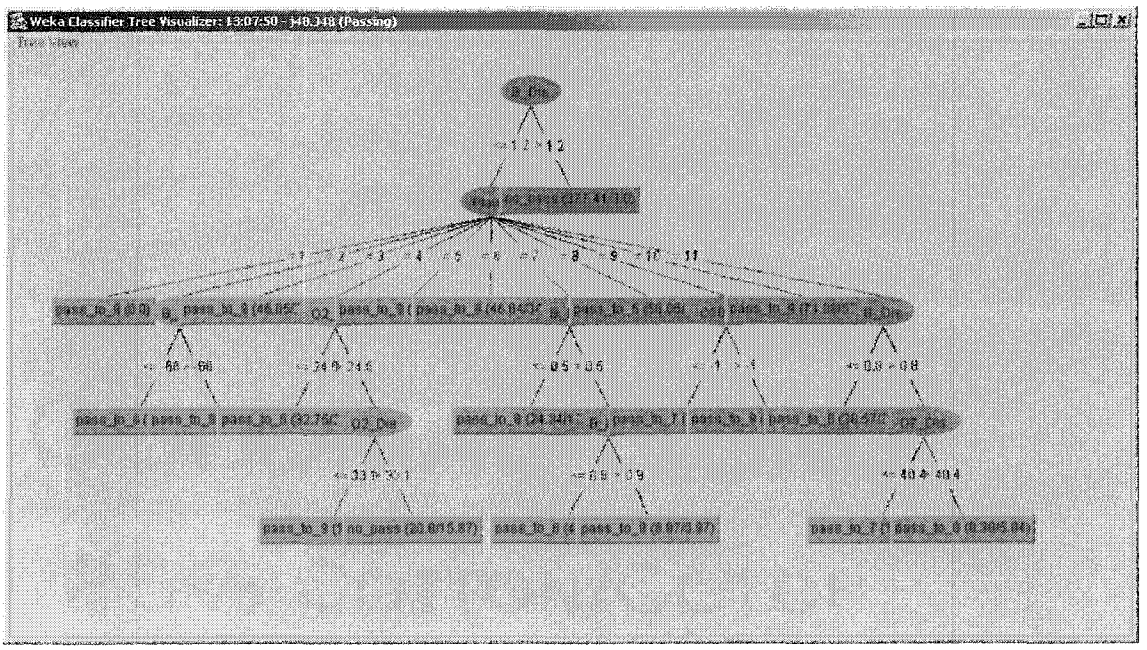

Figure 5.3: Decision tree on the 952 instances of raw data.

```
B_Dis <= 1.2
|  Player = 1: pass_to_9 (0.0)
|  Player = 2
|  |  B_Dir <= -66: pass_to_8 (7.14/3.14)
|  |  B_Dir > -66
|  |  |  O5_Dis <= 22.2
|  |  |  |  B_Dis <= 0.9
|  |  |  |  |  B_Dis <= 0.6: pass_to_9 (5.64/3.82)
|  |  |  |  |  B_Dis > 0.6: pass_to_7 (5.3/3.94)
|  |  |  |  B_Dis > 0.9: pass_to_8 (4.74/2.28)
|  |  |  O5_Dis > 22.2: pass_to_9 (18.82/10.0)
|  Player = 3: pass_to_9 (46.05/25.05)
|  Player = 4
|  |  O2_Dis <= 24.5: pass_to_5 (32.75/26.16)
|  |  O2_Dis > 24.5
|  |  |  O2_Dis <= 33.1: pass_to_9 (10.92/7.22)
|  |  |  O2_Dis > 33.1: no_pass (20.8/15.87)
|  Player = 5: pass_to_9 (59.45/42.45)
|  Player = 6: pass_to_8 (46.64/34.64)
|  Player = 7
|  |  B_Dis <= 0.5
|  |  |  O11_Dis <= 18.2: pass_to_8 (11.64/6.72)
|  |  |  O11_Dis > 18.2: no_pass (12.7/9.26)
|  |  B_Dis > 0.5
|  |  |  B_Dis <= 0.9: pass_to_6 (40.99/29.99)
|  |  |  B_Dis > 0.9: pass_to_9 (8.97/5.97)
|  Player = 8: pass_to_5 (56.05/42.05)
|  Player = 9
|  |  O10_Dir <= -1: pass_to_7 (27.99/19.14)
|  |  O10_Dir > -1: pass_to_8 (22.85/15.91)
|  Player = 10
|  |  O5_Dir <= -7
|  |  |  O5_Dir <= -9: no_pass (26.67/21.17)
|  |  |  O5_Dir > -9
|  |  |  |  O9_Dis <= 22.2: pass_to_9 (6.36/4.84)
|  |  |  |  O9_Dis > 22.2: pass_to_8 (6.32/4.05)
|  |  O5_Dir > -7
|  |  |  O5_Dir <= 15: pass_to_9 (19.02/12.46)
|  |  |  O5_Dir > 15
|  |  |  |  O4_Dir <= 9: no_pass (8.25/6.45)
|  |  |  |  O4_Dir > 9: pass_to_7 (8.26/5.54)
|  Player = 11
|  |  B_Dis <= 0.8
|  |  |  O8_Dis <= 22.2: pass_to_6 (17.45/13.19)
|  |  |  O8_Dis > 22.2
|  |  |  |  B_Dis <= 0.6: pass_to_5 (12.8/9.7)
|  |  |  |  B_Dis > 0.6: pass_to_8 (8.31/5.67)
|  |  B_Dis > 0.8
|  |  |  O7_Dis <= 40.4
|  |  |  |  B_Dis <= 0.9: pass_to_8 (5.4/4.17)
|  |  |  |  B_Dis > 0.9
|  |  |  |  |  B_Dir <= 34: pass_to_7 (4.83/2.21)
|  |  |  |  |  B_Dir > 34: pass_to_4 (3.08/1.86)
|  |  |  O7_Dis > 40.4: pass_to_8 (8.38/5.84)
B_Dis > 1.2: no_pass (377.41/3.0)
```

Figure 5.4: Descriptive textual output of the Can Pass decision tree.

5.2.3 Discussion

The decision tree was generated using a confidence factor of 0.95 in order to obtain accurate information to base results upon. The process of using the raw data may very well work, but it would require a considerable amount of training examples, considerably more than the 952 instances used in this case, or a more guided approach in the selection of the non-categorical attributes. However, it is believed that decomposing the problem into several smaller sub-problems would reduce the overall effort and improve the quality of the results. Hence, the problems with this candid approach can be greatly improved upon by introducing layered learning.

Therefore, in order to reduce the apparent complexity, our approach involved breaking the passing problem into a series of sub-problems that could be learned separately, and then used in conjunction with each other to learn passing as well as shooting. As a side effect, it was suspected that such a breakdown would allow successful learning on a smaller instance data set. The identified sub-problems are listed as follows:

- Ball Kickable (decision tree): Uses the ball's distance and direction – returns True/False if the player can kick the ball.
- Reachable (decision tree): Uses a recipient's distance to determine if the ball could actually reach the intended player.
- Opponent Goal (decision tree): Uses the visual information to determine if the opponent's goal can be seen.

- Opponent Between (decision tree): Uses all visible opponents Distance and Direction, compared to the teammate's Distance and direction – returns True/False if an opponent is in-between the player and the teammate recipient.
- Can Pass (decision tree): uses decision trees Ball Kickable, Reachable, Opponent Goal, and Opponent Between. It also makes use of direct *see* information pertaining to whether the intended recipient is a teammate and not a goalie. The resulting decision tree returns True/False if a pass to a particular player should result in success.
- Can Shoot (decision tree): uses decision trees Ball Kickable, Opponent Goal, Reachable, Opponent Between – returns True/False if a shot may be successful.
- Kick Power (k-nearest neighbour): Uses previously stored distance and power combinations to choose an appropriate power for a kick. This will be used not only for passing, but could be used for shooting as well.
- Kick Direction (k-nearest neighbour): Uses previously stored desired direction and actual kick direction combinations to select an appropriate angle for a kick. Like *Kick Power*, this could be used for shooting as well.

By reducing the problem into several sub-problems, consisting of a limited number of attributes and discrete-valued classifications, we are able to use well-known machine learning techniques like decision trees and k-nearest neighbour.

5.3 "Ball-Kickable" Classification

5.3.1 Sample Data and Decision Tree

The data used in order to classify whether the ball is kickable or not, is quite simple. The distance from the ball is the only factor in determining whether the ball can be

kicked. Typically many teams hard-code this information, either within their algorithms, or within a configuration file that is read in before each game.

Distance	Ball_Kickable
7.4	No
1.6	No
2.0	No
1.0	No
3.0	No
5.0	No
0.5	Yes
0.7	Yes
0.8	Yes
0.2	Yes

Table 5.2: Example data table for Ball Kickable classification

Decision trees were used in this case, again with a 0.95 confidence factor. For such a simple case for learning, just about any machine learning technique would suffice, including k-nearest neighbour, so there really is no reason for using decision trees other than the fact that its simple, and leads quite easily to visual representation. *Table 5.2* is a subset of the training data used to generate the decision tree depicted in *Figure 5.5*.

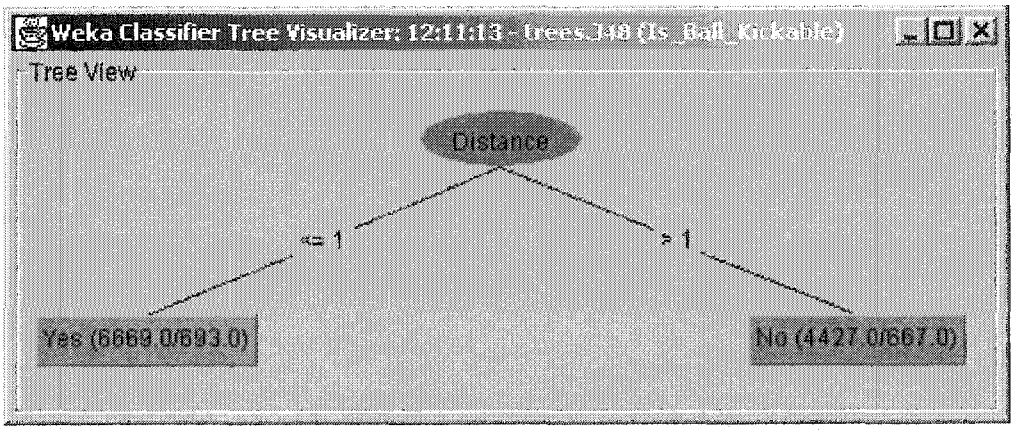

Figure 5.5: Decision tree for Ball_Kickable.

5.3.2 Discussion

The distance used to determine whether a ball can be kicked typically does not change and machine learning is not necessary in this case. However we wanted our

approach to completely make use of machine learning techniques wherever possible. In addition, with the new introduction of the 3D environment, the originally simple ball kickable implementation becomes more complex, and hence motivates the use of decision trees even more.

5.4 "Reachable" Classification

5.4.1 Sample Data and Decision Tree

The *Reachable* classification is used to determine if a teammate can be reached by a pass depending on the distance the intended recipient is from the player with control of the ball. *Table 5.3* displays a subset of the training data that was used to generate the decision tree shown in *Figure 5.6*. The subset chosen correctly represents the classifications that would result using the associated decision tree.

Distance	Reachable
27.1	Yes
18.2	Yes
10	Yes
12.2	Yes
10	Yes
5.5	Yes
5.5	Yes
24.5	Yes
6.7	Yes
82.3	No
57.4	No
55.4	No
73.2	No
40.9	No
49.7	No
82.8	No
83.4	No
53.2	No

Table 5.3: Example data table for Reachable classification

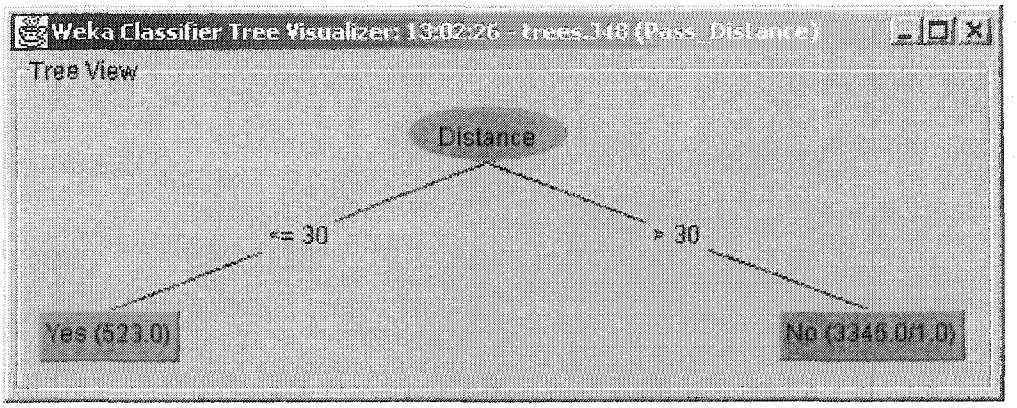

Figure 5.6: Decision tree for Reachable

5.4.2 Discussion

The learning process here determines whether the ball can reach some desired object, based on the object's distance from the player in control of the ball. Typically, this would be used for passing, but it would also port well into the *can shoot* learning. The *reachable* tree is completely derived from the passing data extracted from the log files.

5.5 "Opponent Goal" Classification

5.5.1 Sample Data and Decision Tree

The *Opponent Goal* classification is a straightforward decision, and is based entirely off of *see* information received from the *Soccer Server*. A comparison is made between the field side of the current player with the field side of the goal in question. If the side (right or left) of the player differs to that of the goal, then the goal is the opponent. Similarly if they are the same, then the goal corresponds to that of the players' own team.

TeamSide	GoalSide	OpponentGoal
r	r	No
r	r	No
r	r	No
l	l	No
l	r	Yes
l	r	Yes
l	r	Yes
l	l	No
l	l	No
r	l	Yes
r	l	Yes

Table 5.4: Example data table for Opponent Goal classification

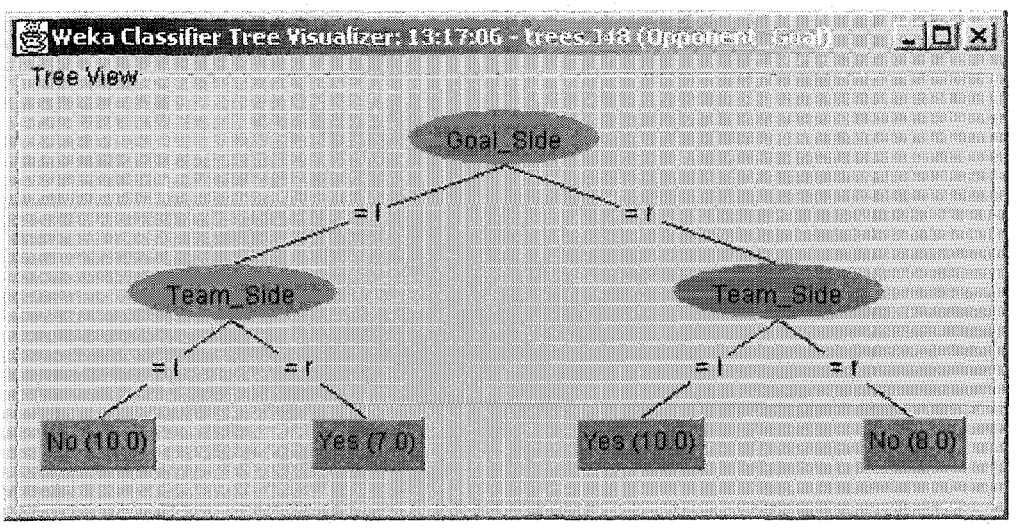

Figure 5.7: Decision tree for Opponent Goal

Table 5.4 contains a subset of the data that was used to train the decision tree in *Figure 5.7*. It can be seen that the data represented in *Table 5.4* conform to the results of the *Opponent Goal* decision tree.

5.5.2 Discussion

It can be seen from the tree in *Figure 5.7* that this will successfully determine whether a goal area belongs to the opponent or not. This is critical when deciding whether to shoot the ball. As with the other decision trees, this one was learned using a 0.95 confidence factor.

5.6 "Object Between" Classification

5.6.1 Sample Data and Decision Tree

The data for this classification requires some preprocessing. As inputs, it takes two attributes, the distance difference and direction difference of the object in question, compared to that of another object – both relative to the player. The *Classifier* was also used to obtain this data, allowing a person to manually indicate that an object is or is not between the player and another object, see *Figure 5.8* for an example scenario.

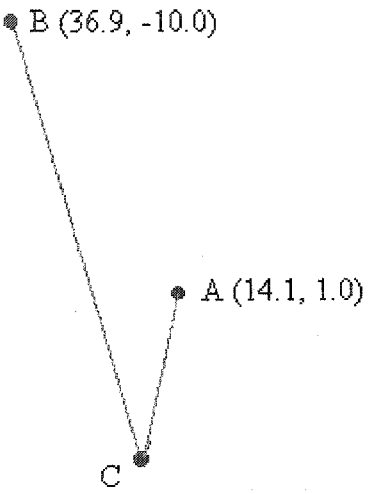

Figure 5.8: Example scenario for Object Between = No

The scenario in *Figure 5.8*, depicts an example where Object Between = No for the first instance in the data table in *Table 5.5*. It shows that object *A* is 14.1 units from player *C*, at an angle of 1.0 degrees. Object *B* is 36.9 units from player *C* at an angle of -10.0 degrees. Therefore, the data preprocessed to indicate if object *B* is between player *C* and object *A*, the distance difference is 14.1 – 36.9 = -22.8, and the direction difference is 1.0 – (-10.0) = 11.0.

Therefore, the resulting attributes consist of two real-valued input attributes for distance and direction differences, and one discrete-valued output classifier Between, with allowable values of *Yes* and *No*. *Figure5.9* describes the resulting decision tree using a set of training data similar to the information in *Table 5.5*.

Distance_Diff	Direction_Diff	Between
-22.8	11.0	No
-12.0	29.0	No
-6.8	21.0	No
17.1	23.0	No
29.8	8.0	Yes
19.8	3.0	Yes
9.7	1.0	Yes

Table 5.5: Example data table for Object Between classification

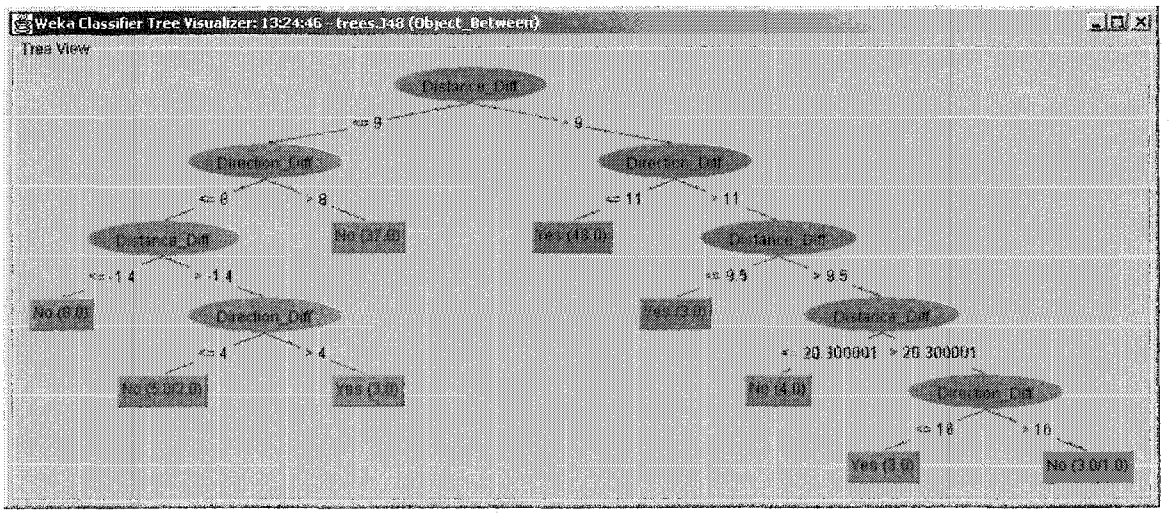

Figure 5.9: Decision tree for Object Between

5.6.2 Discussion

The decision tree depicted above is not entirely accurate. In theory, there should only be three levels of nodes. The reasoning behind this is that with the preprocessed calculations, the values should only fall into one of three buckets. Either an extreme negative number, extreme positive, or a neutral position. Those values which are either less than or equal to the extreme negative, or greater than or equal to the extreme positive, should be classified as "No" (object is not between). If the value falls within the thresholds, then the resulting classification should be "Yes" (object is between). The reasoning for the wide variety in the current decision tree results from a training set that was too small to deal with inconsistent data. The current training set consisted of only 114 samples, and it is believed a set of 500 or more would result in a much better approximation.

5.7 "Can Pass" Classification

5.7.1 Sample Data and Decision Tree

The data for this classification was the same information that was extracted using the *LogExtractor*. The only difference is that the see information was massaged using the techniques described above. In addition, this data pertains only to indicate whether a pass can be made to a *particular* player, and does not indicate which player a pass should be attempted to. That decision will be left up to the human player of the interface. A subset of the training data used is shown in *Table 5.6*, with the resulting decision tree outlined in *Figure 5.10*.

BallKickable	Teammate	Reachable	IsGoalie	OpponentBetween	CanPass
No	Yes	Yes	No	No	No
Yes	Yes	Yes	No	Yes	No
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	Yes	No
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	Yes	No
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	No	Yes
Yes	Yes	Yes	No	Yes	No
Yes	Yes	Yes	No	Yes	No
Yes	Yes	Yes	No	Yes	No
Yes	Yes	Yes	No	Yes	No

Table 5.6: Partial Data table for Can Pass classification

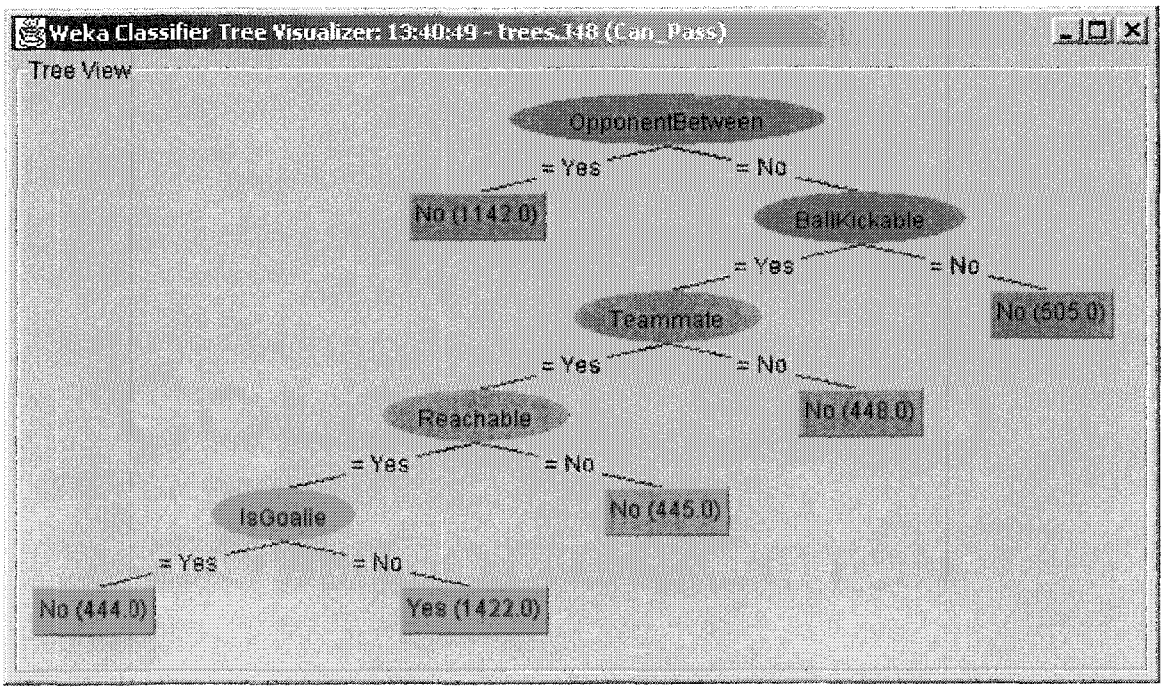

Figure 5.10: Can Pass Decision Tree

5.7.2 Discussion

It may seem as though all these decision trees are more work and initially that is true. However, such preprocessing allows the *Can Pass* decision tree to be much more simplified in its structure, and hence easier to understand to the human observer -- although this is not the objective, but a nice side effect. Comparing this decision tree to the one produced directly from the raw data in *Section 5.2*, we can see just how effective the above learning techniques work together to solve the problem of when a player can pass. Now that the basics have been learned, it should be an easy matter to learn when a player can shoot.

5.8 "Can Shoot" Classification

BallKickable	OpponentGoal	Reachable	OpponentBetween	CanShoot
Yes	No	No	Yes	No
Yes	No	Yes	No	No
Yes	Yes	No	No	No
Yes	Yes	Yes	No	Yes
Yes	Yes	Yes	Yes	No
Yes	No	Yes	No	No
Yes	Yes	No	No	No
Yes	Yes	Yes	Yes	No
Yes	Yes	Yes	No	Yes
Yes	No	Yes	No	No
Yes	Yes	Yes	No	Yes
Yes	No	Yes	No	No
Yes	Yes	No	Yes	No
Yes	Yes	Yes	Yes	No
Yes	Yes	Yes	No	Yes
Yes	No	Yes	No	No

Table 5.7: Example data table for Can Shoot classification

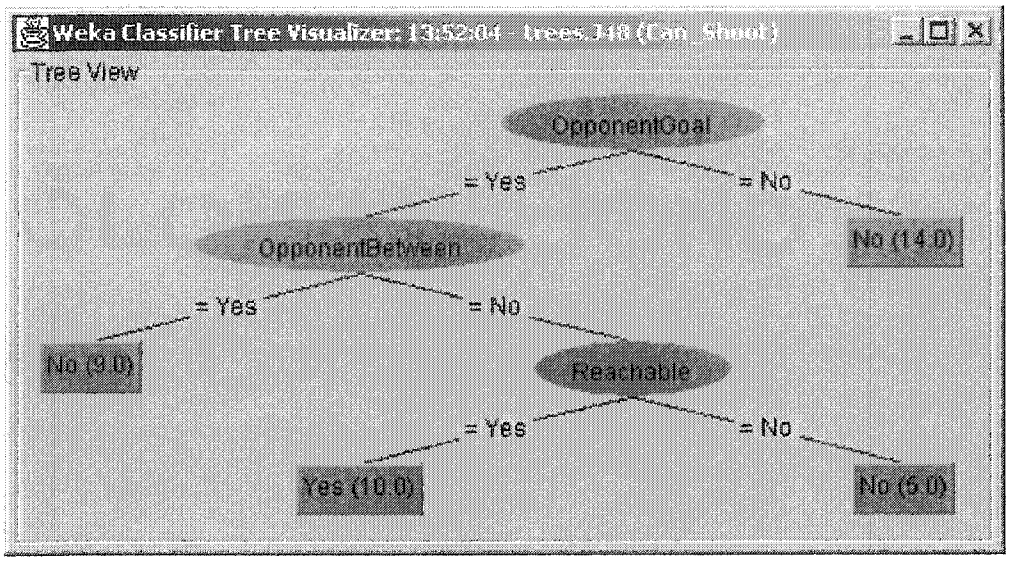

Figure 5.11: Decision tree for Can Shoot

The decision tree was generated using training data represented by *Table 5.7*. Unfortunately as can be seen from *Figure 5.11*, the initial *Can Shoot* decision tree is missing one key element – *Ball Kickable*. While everything else about the decision tree is correct, we neglected to include negative training examples for when the ball was not kickable. Making the necessary modifications to the *LogExtractor* corrected this (see *Table 5.8* and *Figure 5.12*). The mistake was not noticed at first in the interface, because of the *Reachable* portion of the decision tree. However, once the player became close enough to the opponent's goal, the interface provided the option to shoot, even when the player did not have the ball.

BallKickable	OpponentGoal	Reachable	OpponentBetween	CanShoot
Yes	Yes	Yes	No	Yes
Yes	No	Yes	No	No
Yes	Yes	Yes	No	Yes
Yes	No	Yes	No	No
Yes	Yes	Yes	Yes	No
Yes	Yes	Yes	No	Yes
Yes	No	Yes	No	No
Yes	Yes	No	No	No
No	No	No	Yes	No
No	No	Yes	No	No
No	Yes	No	No	No
No	Yes	Yes	No	No
No	Yes	Yes	Yes	No
No	No	Yes	No	No
No	Yes	No	No	No
No	Yes	Yes	Yes	No

Table 5.8: Can Shoot data with negative Ball Kickable instances – See Appendix G

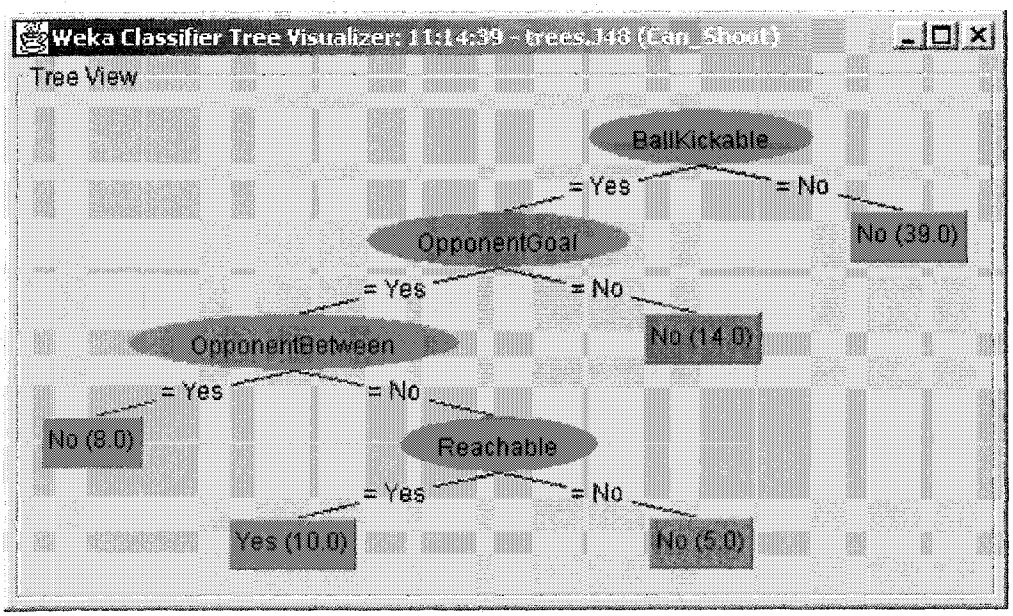

Figure 5.12: Corrected Can Shoot decision tree

In order to effectively shoot and pass, we also had to learn which direction to kick the ball, and at what power. The next two k-nearest neighbour classifications provided these values.

5.9 Kick Classifications

5.9.1 Kick Power

The *Kick Power* classification is used to learn how hard a kick is to be taken depending on the distance to either the goal or a player. The learning mechanism used is a k-nearest neighbour algorithm based off of training data, of which a subset is shown in *Table 5.9*.

Distance	Kick Power
22.2	40.04
24.5	100
22.2	92.74
22.2	100
9	80.75
6.7	82.28
10	100
9	74.35
18.2	65.23
16.4	97.79
12.2	95.01
18.2	61.79
14.9	58.42
13.5	85.84

Table 5.9: Example data table of Kick Power classification

Figure 5.13 shows a graphical representation of the *Kick Power* data. There does not appear to be any uniform structure, because of the *stamina* parameter associated with players during a soccer game. The power used for kicking and dashing decrease the stamina available for a player. Hence, the more effective teams like TsinghuAeolus [11] make decisions regarding kick power usage with stamina in mind.

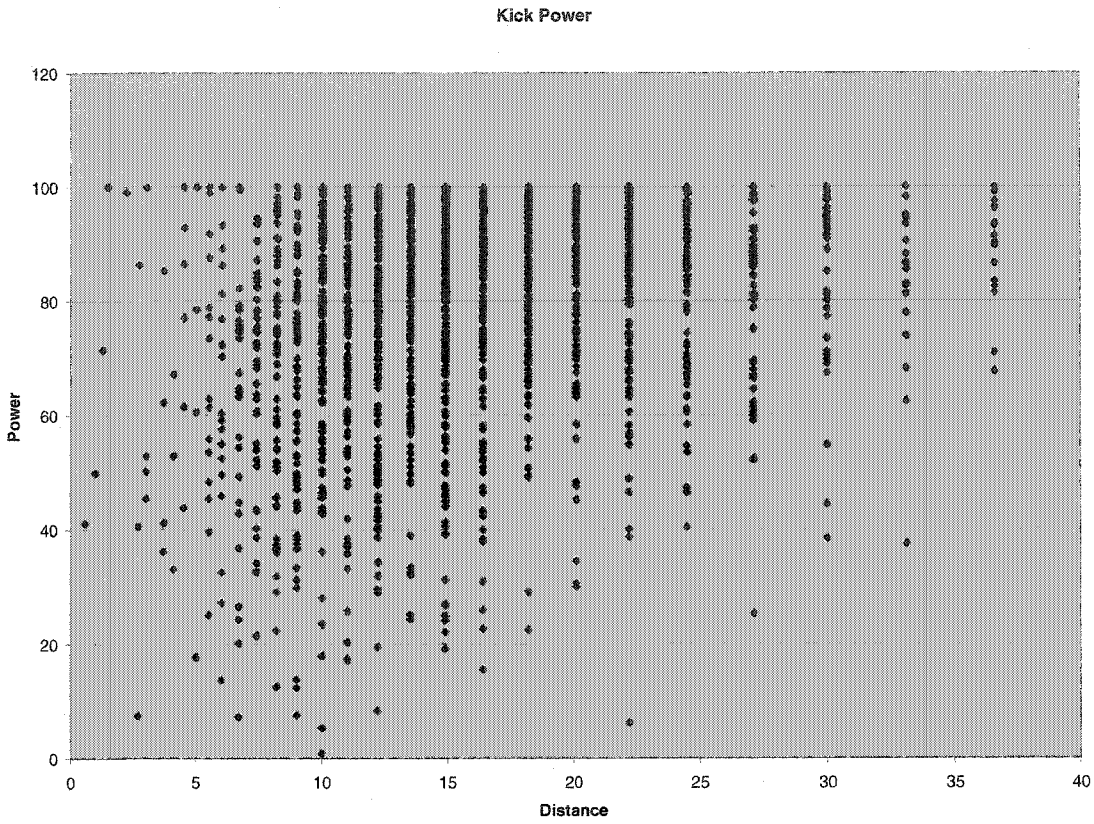

Figure 5.13: Graphical representation of k-nn for Kick Power

5.9.2 Kick Direction

Similarly with *Kick Power*, *Kick Direction* classification also uses a k-nearest neighbour algorithm. The purpose is to learn the direction in which a player should kick a ball, based off of the direction in which the intended recipient is from the player. *Table 5.10* shows a small subset of the sample data used to train with.

Direction	Kick Direction
-53.87	-44.39
-48.31	-38.41
157	141.71
-52.56	-35.6
113.34	108.65
-64.19	-58.85
-43.86	-48.81
55.32	58.93
-91.9	-81.92
49.86	55.7
96.36	111.28
59.21	40.2
78.24	72.75
95.44332	113.76
42.85781	24.45

Table 5.10: Example data table for Kick Direction classification

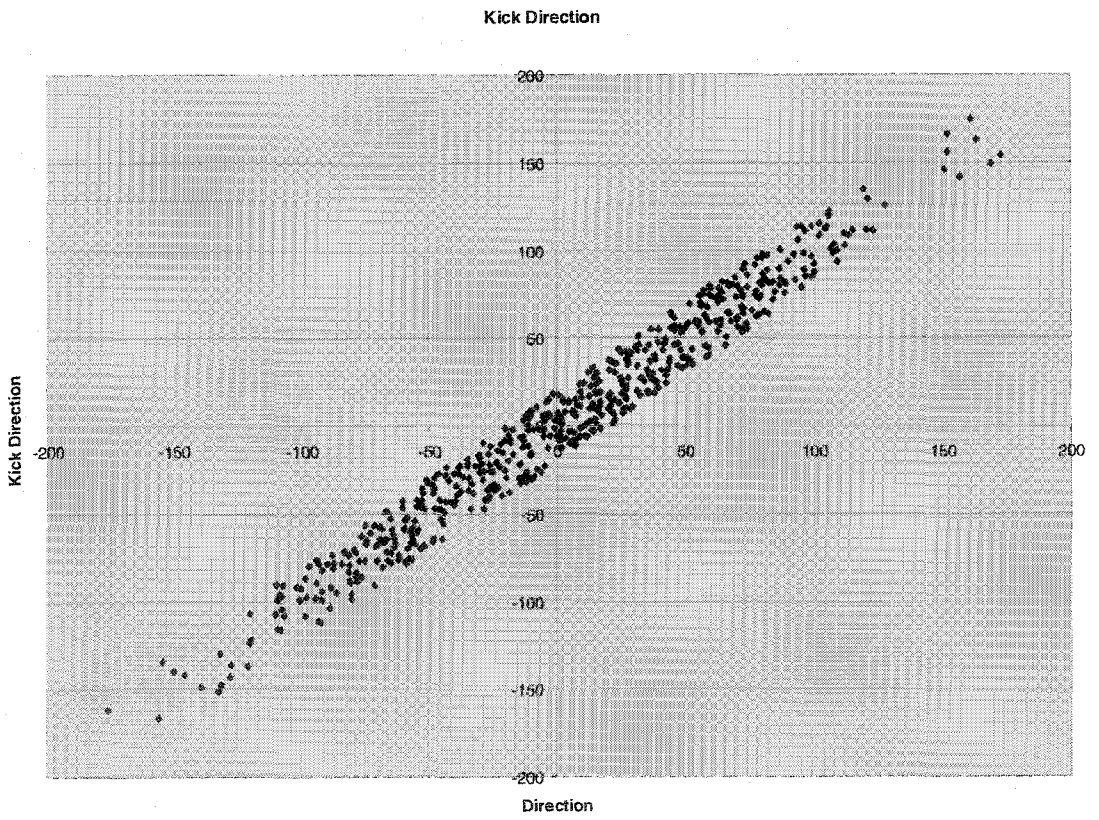

Figure 5.14: Graphical representation of k-nn for Kick Direction

Figure 5.14 shows the graphical representation of the data used for training. Unlike the *Kick Power* classification, the *Kick Direction* training data has a more linear

appearance, which was expected. Theoretically, if the intended recipient is at an angle of θ degrees from the player, then the direction in which a kick is taken should be close to θ. The reasoning why the data is not perfectly linear, is because teams like TsinghuAeolus [11], take the velocity of the recipient into account as part of the passing algorithm, thereby being close to θ, but not exactly equal.

5.10 Integration of Knowledge into SmartITAS

Upon the generation of the aforementioned classifications, the *CanPass*, *CanShoot*, *KickDirection* and *KickPower* were integrated into the *ITAS* interface to create our *SmartITAS*. This new version of the interface provides the user with visual stimuli corresponding possible pass and shoot options. The idea is to give the user a set of options that would lead to more successful plays. Those teammates who could successfully receive a pass change colour during game-play to aid in passing, and the *Shoot* button enables/disables automatically depending on the *CanShoot* decision tree. *Table 5.11* depicts a series of games of five on five between users using the *SmartITAS* interface and TsinghuAeolus [11] players.

Game #	Humans (5)	TsinghuAeolus (5)
1	0	25
2	1	23
3	0	21
4	1	26
5	2	20
6	1	22
7	1	22
8	0	23
9	3	19
10	1	21

Table 5.11: Game results of SmartITAS against TsinghuAeolus (no goalies)

Comparing the results of the table above, to those of the original *ITAS* in *Table 5.1*, we can see an improvement. In a series of 10 games of *ITAS* against TsinghuAeolus, the total goals were: 3 for *ITAS* and 284 for TsinghuAeolus, thereby resulting in an average of 0.3 goals for *ITAS* and 28.4 goals for TsinghuAeolus per game. Conversely, the *SmartITAS* performed marginally better by scoring a total of 10 goals, with 222 goals scored against for an average of 1 goal for, and 22.2 goals against per game. The goalies are not present because our interfaces are not sufficient enough to allow a human player to act as a goalie (i.e. there are no goalie specific commands implemented like *catch_ball*). Hence for testing purposes, we wanted the games to be as evenly matched as possible.

The improvement in game play is mostly due to the options the interface provides the user, instead of a learned kicking mechanism. The ability to provide the user with a set of viable options gives the human player a way to quickly discount choices that would not prove successful. This is only the start of the process, as the groundwork set here shows how such an approach can evolve and improved upon.

5.11 Summary

When future actions are learned like interception, dribbling, and keep-away, and incorporated into the interface, it may then be possible to provide a more effective interface agent. Even so, currently our *SmartITAS* does provide an improvement over the original *ITAS* as it provides the user with the visual stimuli necessary to react quicker with a greater success rate. The layered process to learning works, and may lead to future competitive versions, which could be used to learn tactics and scenarios. The current

problem with our interfaces is the same as those of inferior teams in that it is difficult to either obtain the ball from the opposing team, or more importantly to keep the ball once possession has been obtained. However, even with these issues the *SmartITAS* was able to build and improve upon the original *ITAS* version.

Chapter 6

Conclusions

6.1 Discussion

The original idea was to create an interface agent, allowing a human to play the part of one of the soccer players. The interface agent would allow learning to be accomplished from the actions of the human player. However, a basic interface allowing just *turn*, *dash*, and *kick* does not provide the speed at which is needed for a human player to effectively compete against computer players, as shown in our results as well as in [1].

This led to the idea of developing a more advanced, "*smart*", interface that would provide the user with a set of decisions for which the corresponding actions could be successful. Such implementations consist of the *Can Pass* and *Can Shoot* decisions that provide visual stimuli to the player, indicating successful pass and shoot possibilities.

The *Can Pass* decision tree is executed for all teammates, and possible passes are indicated on the interface by a colour change of their teammates. This is accomplished by not only taking into account whether the human player has control of the ball and if the intended recipient is indeed a teammate, but more importantly, it takes into account the position of opponents. If an opponent is close enough to the *pass line* between teammates (i.e. is in a position to intercept the ball), the pass probability will be deemed as a failure, and therefore the teammate will not be available as a viable pass option. This is because many teams have the ability to intercept passes, and as such one does not want

to make any unnecessary passes to those players where a probability of an interception is high.

Similarly, the *Can Shoot* decision tree provides the user with visual stimuli by enabling / disabling a "Shoot" button on the interface. As with the *Can Pass* methodology, the interception possibilities of the opponent players (including the goalie) are taken into account. Only when there is a clear shot will the *Shoot* button become enabled. Even so, we do not restrict a shot only when this button is enabled. At any time the user can manually shoot the ball with a kick command towards the goal.

These additions improve the overall players performance by providing the user with the opportunity to focus more on positioning and actual game play instead on whether a pass or a shot might succeed.

In order to create the layers for these end decisions, three main utilities that were developed to aid in obtaining the data and learning. The first is the *LogServer*, which acts like a router in that a soccer agent connects to it, and any commands from the agent are routed to the soccer server with the responses going back through the *LogServer* and being routed to the original agent. The important aspect is the ability for the *LogServer* to log all commands between the agent and the server to a file, allowing machine learning and data mining to be completed on the log file. The downfall to the *LogServer* is that in general, I/O operations are CPU intensive and hence the more logging one is doing, the slower the responses. To get around this, the *LogServer* was created in such a way that one can run multiple instances of the *LogServer* on multiple computers – especially if one intends to log an entire game of 22 players. Essentially, the *LogServer* provides the means to "eavesdrop" on other teams like TsinghuAeolus [11].

The *Classifier* is a tool for manual classification of objects by a human user. It provides the ability to classify if the ball is kickable, the conversion of real-valued distance and direction pairs into discrete-valued pairs, as well as the ability to classify if an object is in between another two objects. Each one of these classifications result in a decision tree, and can be used as building blocks to the more advanced decision trees such as *Can Pass* and *Can Shoot*. See *Appendix A* for further information on how to use the *Classifier*.

The *LogExtractor* is a tool used for preprocessing of data by scanning through all the generated log files using a set of extraction routines created using object-oriented techniques. Currently there are such routines for the extraction of ball information, kicks, passes, and shots resulting in goals. These passes and shots were verified through the use of the *Soccer Server*'s ability to replay games that have been logged (via the *Soccer Server*). Part of the information extracted using these routines contains the visual see information, and hence the cycle number. When replaying a game, the extracted cycle numbers can be manually compared to those during the game for kicks, passes and shots.

Each of the extraction routines, except the kick extraction, also contains mechanisms to generate classification files to be used in a machine learning repository tool like WEKA [36]. After it is all said and done, the *LogExtractor* creates a set of classification files that are integrated into our interface, along with references to WEKA, in order to provide the user with a more intelligent version of ITAS (*SmartITAS*). This version improves performance slightly over the original ITAS as can be seen by our results. However, the improvement is not enough for a team of humans to effectively compete

against former champions, but the process described herein can be applied to future work on the interface.

6.2 Future Work

The current work involved aspects relating to passing and shooting. However, there is much more to a soccer game than being able to decide when and how to pass and shoot. Further important additions to the interface would constitute the ability to *intercept*, *adversary management* (dribbling, advanced kicking, and ball-keep away), and goalie specific options so that a human can play as the goalie. The process for implementing decision relating to passing and shooting can be applied to these future additions for the creation of even smarter actions.

With the addition of each smart actions a human player should get progressively better at competing against computer teams. Once the interface becomes more effective, it could be further improved by incorporating positioning aspects and memory management (the ability to keep track of objects/players outside the view cone – using a combination of say commands to broadcast player positions as well as confidence level values).

All these additions lead directly into the architectural approach of layered learning, where these individual skills can be combined into further machine learning techniques to incorporate strategies and scenarios. Teams like CMUnited [24][29][30][32][33], FCPortugal [23] and Cyberoos2000 [22] already include some tactics based upon how the team is doing as the game progresses. These teams tend to change to a defensive approach if they are winning late in a game, and a more offensive approach if they are losing. The same sort of aspect could be achieved by combining positioning, passing,

shooting, as well as the score and length of time remaining. With the necessary skills in place, the possibilities of tactics are nearly endless.

Finally, with each iteration of the human interface, it is expected that a human team would continually improve against the computer teams to a point where a human team would surpass the abilities of their computer opponents, at which time the interface itself could be used as an interface agent to learn actions and scenarios from human players instead of those computer agents created by other researchers. Such a process lends well to learning with chronicles [4][5], because they tend to be based off of event-driven inputs that can correspond to predicate logic.

References

1. M. Atsumi, "RoboCup of Atsumi Laboratory", *http://www.intlab.soka.ac.jp/~matsumi/document/research/robocup/index-e.html*, September 1997.
2. C. Candea, M. Oancea, D. Volovici, "Emulating Real Soccer", *Proceedings of the International Conference Beyond 2000*, Sibiu Romania, Pages 35-38, (1999) – [ULBS 99] *http://airg.verena.ro/members/ciprianc/robocup99.zip*
3. M. Chen et. al, "RoboCup Soccer Server User Manual", *The RoboCup Federation*, June 2001.
4. Cordier, Marie-Odile, Dousson, Christophe, "Alarm Driven Monitoring Based On Chronicles", *Proceedings of SAFEPROCESS 2000*, Budapest, Hungary, 2000 (Pages 286-291)
5. B. Esfandiari, G. Deflandre, J. Quinqueton, C. Dony, "Agent Oriented Techniques for Network Supervision", *Annals of Telecommunication, 51 n. 9-10*, 1996, Pages 521 – 529
6. F. Flentge, C. Meyer, B. Schappel, T. Uthmann, "Enhancing the Adaptive Abilities Mainz Rolling Brains 2001", *http://www.rollingbrains.de/mrb2001teamdesc.ps*, Germany (2001) – [Mainz Rolling Brains]
7. T. Hassan, B. Esfandiari, "ITAS and the Reverse RoboCup Challenge", *RoboCup-2004: Robot Soccer World Cup VIII*, Springer Verlag, Berlin – To Appear

8. F. Heintz, "FCFoo – A Short Description", *http://www.ida.liu.se/~frehe/publications/FCFoo.ps.gz*, (1999) – [FC Foo]

9. H. Hu, K. Kostiadis, M. Hunter, N. Kalyviotis, "Essex Wizards 2000 Team Description", *RoboCup-00: Robot Soccer World Cup IV*, Springer Verlag, Berlin (2001) – [Essex Wizards 2000] *http://privatewww.essex.ac.uk/~kkosti/Files/EW2000.ps.gz*

10. H. Hu, K. Kostiadis, M. Hunter, M. Seabrook, "Essex Wizards '99 Team Description", *RoboCup-99: Robot Soccer World Cup III,* Springer Verlag, Berlin (2000) – [Essex Wizards '99] *http://privatewww.essex.ac.uk/~kkosti/Files/EW1999.ps.gz*

11. Y. Jinyi, C. Jiang, C. Yunpeng, L. Shi, "Architecture of TsinghuAeolus", *RoboCup-2001: Robot Soccer World Cup V*, Springer Verlag, Berlin, Pages 491-494 (2002) – [TsinghuAeolus]

12. Y. Jinyi, C. Jiang, S. Zengqi, "An application in RoboCup combining Q-learning with Adversarial Planning", *IEEE Robotics and Automation Society*, 2002

13. S. Kinoshita, Y. Yamamoto, "11Monkeys Description", *http://homepage2.nifty.com/kinoshita/archive/11monkeys_strategy.ps.gz*, Japan, (2001) – [11Monkeys]

14. R. Kozierok, P. Maes, "A Learning Interface Agent for Scheduling Meetings", *Proceedings of the ACM SIGCHI International Workshop on Intelligent User Interfaces*, Orlando Florida: ACM Press, 1993, Pages 81 – 88

15. R. Kozierok, P. Maes, "Learning Interface Agents", *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington DC: AAAI Press, Pages 459 – 465

16. K. Lai, T. Malone, K. Yu, "Object Lens: A Spreadsheet for Cooperative Work", *ACM Transactions on Office-Information Systems*, 5(4), 1988, Pages 297 – 326

17. Y. Lashkari, M. Metral, P. Maes, "Collaborative Interface Agents", *Proceedings of the Twelfth National Conference on Artificial Intelligence, Vol. 1*, AAAI Press, Seattle, WA, August 1994.

18. T. Matsumura, "Team Erika", *http://www.futamura.info.waseda.ac.jp/~matsu/erika/erika99-description.ps.gz*, Japan (1999) – [Erika]

19. A. Merke, "FrameView 2D", http://i11www.ira.uka.de/~amerke/frameview/, December 2001.

20. T. Mitchell, "Machine Learning", *The McGraw-Hill Companies, Inc.*, United States, 1997

21. I. Noda, H. Matsubara, "Soccer Server and Researches on Multi-Agent Systems", *Proceedings of the IROS-96 Workshop on RoboCup*, November 1996.

22. M. Prokopenko, M. Butler, T. Howard, "Cyberoos2000: Experiments with Emergent Tactical Behaviour", *http://www.cmis.csiro.au/aieb/Cyberoos/rc_publications/cyberoos2000.ps*, Australia (2000) – [Cyberoos2000]

23. L. Reis, J. Lau, L. Lopes, "FC Portugal Team Description: RoboCup 2000 Simulation League Champion", *RoboCup-2000: Robot Soccer World Cup IV*,

Springer Verlag, Berlin (2001) – [FC Portugal]

http://www.ieeta.pt/robocup/documents/FCPortugalChampion.ps.zip

24. P. Riley, P. Stone, D. McAllester, M. Veloso, "ATT-CMUnited-2000: Third Place Finisher in the RoboCup-2000 Simulator League", *RoboCup-2000: Robot Soccer World Cup IV*, P. Stone, T. Balch, G. Kreatzschmarr (eds.), Springer Verlag, Berlin (2000) – [ATT-CMUnited-2000]

http://www.research.att.com/~pstone/Papers/2000robocup/ATTCMUnited.ps.gz

25. P. Scerri, J. Ydren, T. Wiren, M. Lönneberg and P. Nilsson "Headless Chickens III", *RoboCup-99: Robot Soccer World Cup III*, Springer Verlag (2000) – [Headless Chickens] http://www.ida.liu.se/~pausc/RC99/teamdesc.ps

26. J. C. Schlimmer, L. A. Hermens, "Software Agents: Completing Patterns and Constructing User Interfaces", *Journal of Artificial Intelligence Research, Vol 1*, 1993, Pages 61 – 89

27. C. Stanfill, D. Waltz, "Toward Memory-based Reasoning", *Communications of the ACM, 29(12)*, 1986, Pages 1213 – 1228

28. P. Stone, "Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer", *The MIT Press*, Cambridge, Massachusetts, 2000

29. P. Stone, P. Riley, M. Veloso, "Defining and Using Ideal Teammate and Opponent Agent Models: A Case Study in Robotic Soccer", *Proceedings of the IEEE Fourth International Conference on MultiAgent Systems*, Pages 441-442 (2000)

30. P. Stone, P. Riley, M. Veloso, "The CMUnited-99 Champion Simulator Team", *RoboCup-99: Robot Soccer World Cup III*, M. Veloso, E. Pagello and H. Kitano (eds.), Springer Verlag, Berlin (2000) – [CMUnited-99] http://www.research.att.com/~pstone/Papers/99springer/champ99-extended.ps.gz

31. P. Stone, M. Veloso, "A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server", *Applied Artificial Intelligence (AAI) Volume 12*, 1998

32. P. Stone, M. Veloso, "The CMUnited-97 Simulator Team", *RoboCup-97: Robot Soccer World Cup I*, H. Kitano (ed.), Springer Verlag, Berlin (1998) – [CMUnited-97] http://www.research.att.com/~pstone/Papers/97springer/simulator/simulator.ps.gz

33. P. Stone, M. Veloso, P. Riley, "The CMUnited-98 Champion Simulator Team", *RoboCup-98: Robot Soccer World Cup II*, M. Asada and H. Kitano (eds.), Springer Verlag, Berlin (1999) – [CMUnited-98] http://www.research.att.com/~pstone/Papers/98springer/simulator/champ98.ps.gz

34. S. Suzuki, T. Tamura, M. Asada, "Learning From Conceptual Aliasing Caused by Direct Teaching", *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Volume 6, Pages 698-703 (1999)

35. K. Sycara, "Multiagent Systems", *AI Magazine*, 19(2), (1998)

36. University of New Zealand, "WEKA", http://www.cs.waikato.ac.nz/~ml/weka/ .

37. M. Veloso, P. Stone, "Individual and Collaborative Behaviors in a Team of Homogeneous Robotic Soccer Agents", *Proceedings of the 1998 IEEE International Conference on Multi Agent Systems*, Pages 309-316 (1998)

38. B. Zhang, X. Chen, G. Liu, Q. Cai, "Agent Architecture: A Survey on RoboCup-99 Simulator Teams", *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, Volume 1, Pages 189-193 (2000)

Appendix A: How to use the Classifier

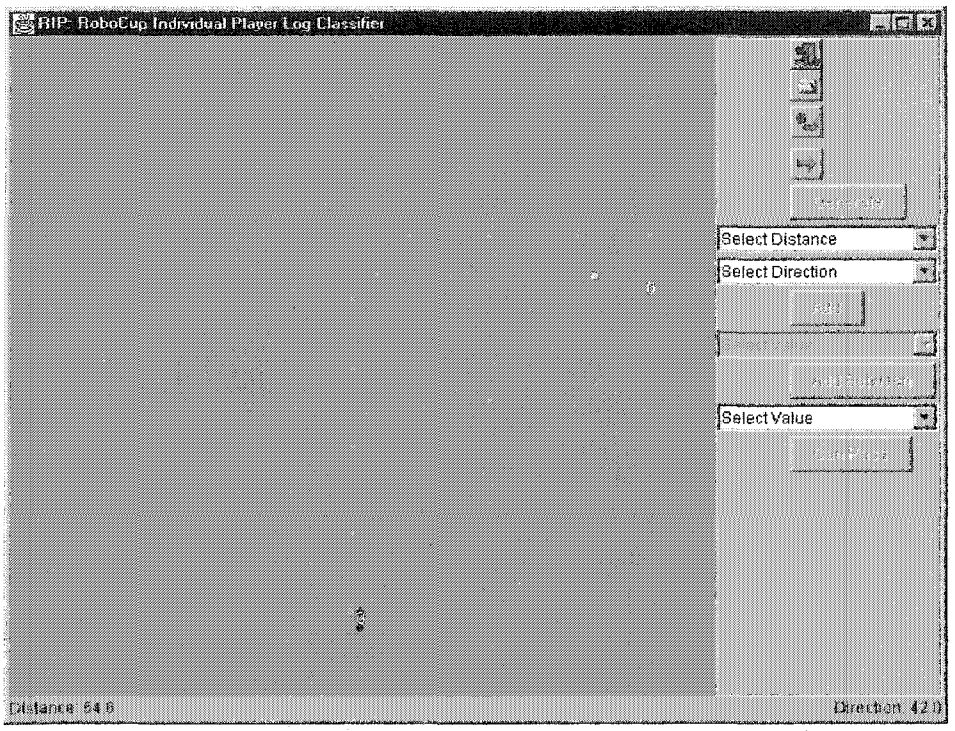

Figure A.1: Screen shot of the Classifier tool.

Figure A.1 depicts a screen shot of the *Classifier*. It can be used to accomplish manual classification of objects using log files of visual see information. Any file containing *see* information compatible with version 5.24 up to version 9.0.3 of the *soccer server* have been tested. Once a log file has been opened, the green arrow button can be used to cycle between each uniquely logged see information and displayed accordingly.

To classify objects, simply select an object on the screen. Depending on the object selected, the appropriate options will become enabled on the right hand side of the screen. For instance, if the ball is selected, the *ball kickable* icon button will become enabled. Each classification adds an entry in memory, and does not get written to file until the user

selects *Generate*. This button will generate the necessary .ARFF files to use in WEKA if desired.

The classifications available are: ball kickable, object position, and object between. As mentioned, ball kickable relies on the ball being selected, and indicates that the ball is kickable within the distance the ball is from the player. Object position is available on all objects, and is used to convert real-valued coordinates (distance and direction), to one of 20 discrete-valued classifications, i.e. *Left_Near, ExtremeRight_Far, Front_Distant*, etc. Because this could take some time to get an accurate sampling, the ability exists to select multiple objects and classify them with the same classification.

Finally, an object can be classified as being between the human player (lower center portion of the screen), and another object. This requires the user to select the object to classify first, call it *A*, followed by the reference object second, call it *B*. Essentially, the selection means: Object *A* is between / not between myself and object *B*.

Appendix B: Sample Log File

```
(see 12 ((g l) 79 -8) ((f c) 27.7 3 -0 0) ((f c t) 49.9 43) ((f l t) 90
14) ((f l b) 83.1 -32) ((f p l t) 68.7 10) ((f p l c) 62.8 -6) ((f p l
b) 63.4 -24) ((f g l t) 80.6 -3) ((f g l b) 79 -13) ((f t l 50) 90 17)
((f t l 40) 81.5 21) ((f t l 30) 73.7 25) ((f t l 20) 66 31) ((f t l
10) 59.7 38) ((f b l 50) 82.3 -36) ((f b l 40) 73 -38) ((f b l 30) 64.1
-42) ((f l t 30) 91.8 10) ((f l t 20) 88.2 4) ((f l t 10) 85.6 -1) ((f
l 0) 83.9 -8) ((f l b 10) 83.9 -15) ((f l b 20) 84.8 -22) ((f l b 30)
86.5 -28) ((b) 27.1 3 -0 0) ((p) 73.7 -7) ((p) 54.6 15) ((p) 49.4 2)
((p "Essex_Wizards") 49.4 -12) ((p "Essex_Wizards") 49.4 -26) ((l l)
81.5 75))
(sense_body 13 (view_mode high normal) (stamina 3940 1) (speed 0.08 2)
(head_angle 30) (kick 0) (dash 6) (turn 4) (say 0) (turn_neck 5))
(dash 70.00)
(turn_neck 1.65)
(sense_body 14 (view_mode high normal) (stamina 3915 1) (speed 0.18 -
21) (head_angle 31) (kick 0) (dash 7) (turn 4) (say 0) (turn_neck 6))
(dash 70.00)
(turn_neck 1.35)
(see 14 ((g l) 79 -11) ((f c) 27.1 0 -0 0.1) ((f c t) 49.4 41) ((f l t)
89.1 11) ((f l b) 82.3 -35) ((f p l t) 68 7) ((f p l c) 62.2 -9) ((f p
l b) 62.8 -27) ((f g l t) 79.8 -6) ((f g l b) 78.3 -16) ((f t l 50)
89.1 14) ((f t l 40) 80.6 18) ((f t l 30) 73 23) ((f t l 20) 66 28) ((f
t l 10) 59.1 35) ((f t 0) 54.1 44) ((f b l 50) 81.5 -39) ((f b l 40)
72.2 -42) ((f l t 30) 91.8 7) ((f l t 20) 88.2 1) ((f l t 10) 85.6 -4)
((f l 0) 83.9 -11) ((f l b 10) 83.1 -18) ((f l b 20) 83.9 -25) ((f l b
30) 86.5 -31) ((b) 27.1 0 -0 0.1) ((p) 73.7 -10) ((p "Essex_Wizards")
54.6 12) ((p "Essex_Wizards") 49.4 0) ((p) 49.4 -15) ((p) 49.4 -28) ((l
l) 82.3 72))
(sense_body 15 (view_mode high normal) (stamina 3890 1) (speed 0.25 -
33) (head_angle 33) (kick 0) (dash 8) (turn 4) (say 0) (turn_neck 7))
(turn 121.57)
(turn_neck -55.50)
(see 15 ((g l) 78.3 -12) ((f c) 26.6 0) ((f c t) 49.4 40) ((f l t) 88.2
10) ((f l b) 81.5 -36) ((f p l t) 67.4 6) ((f p l c) 61.6 -10) ((f p l
b) 62.2 -29) ((f g l t) 79 -7) ((f g l b) 77.5 -17) ((f t l 50) 89.1
13) ((f t l 40) 80.6 17) ((f t l 30) 73 22) ((f t l 20) 65.4 27) ((f t
l 10) 59.1 34) ((f t 0) 53.5 43) ((f b l 50) 81.5 -40) ((f b l 40) 72.2
-43) ((f l t 30) 90.9 6) ((f l t 20) 87.4 0) ((f l t 10) 84.8 -5) ((f l
0) 83.1 -12) ((f l b 10) 83.1 -19) ((f l b 20) 83.1 -26) ((f l b 30)
85.6 -33) ((b) 27.1 0 -0 0.3) ((p) 73.7 -11) ((p) 54.6 11) ((p
"Essex_Wizards") 44.7 -1) ((p) 44.7 -15) ((p) 44.7 -29) ((l l) 82.3
71))
(sense_body 16 (view_mode high normal) (stamina 3935 1) (speed 0.1 -36)
(head_angle -22) (kick 0) (dash 8) (turn 5) (say 0) (turn_neck 8))
(dash 70.00)
(sense_body 17 (view_mode high normal) (stamina 3910 1) (speed 0.17 11)
(head_angle -22) (kick 0) (dash 9) (turn 5) (say 0) (turn_neck 8))
```

Appendix C: Converted Log File

```
(see 12 (Goal Front_Distant) (Flag Front_Near) (Flag ExtremeRight_Far) (Flag
Right_Distant) (Flag Left_Distant) (Flag Right_Distant) (Flag Front_Distant)
(Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant) (Flag
Right_Distant) (Flag Right_Distant) (Flag Right_Distant) (Flag Right_Distant)
(Flag Right_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Right_Distant) (Flag Left_Distant) (Flag Left_Distant)
(Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant)
(Ball Front_Near) (Opponent Front_Distant) (Opponent Right_Distant) (Opponent
Front_Distant) (Opponent Front_Distant) (Opponent Left_Far) (Line
Right_Distant))
(sense_body 13 (view_mode high normal) (stamina 3940 1) (speed 0.08 2)
(head_angle 30) (kick 0) (dash 6) (turn 4) (say 0) (turn_neck 5))
(dash 70.00)
(turn_neck 1.65)
(sense_body 14 (view_mode high normal) (stamina 3915 1) (speed 0.18 -21)
(head_angle 31) (kick 0) (dash 7) (turn 4) (say 0) (turn_neck 6))
(dash 70.00)
(turn_neck 1.35)
(see 14 (Goal Front_Distant) (Flag Front_Near) (Flag ExtremeRight_Far) (Flag
Right_Distant) (Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant)
(Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant) (Flag
Right_Distant) (Flag Right_Distant) (Flag Right_Distant) (Flag Right_Distant)
(Flag Right_Distant) (Flag ExtremeRight_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Ball
Front_Near) (Opponent Front_Distant) (Opponent Front_Distant) (Opponent
Front_Distant) (Opponent Front_Distant) (Opponent Left_Far) (Line
Right_Distant))
(sense_body 15 (view_mode high normal) (stamina 3890 1) (speed 0.25 -33)
(head_angle 33) (kick 0) (dash 8) (turn 4) (say 0) (turn_neck 7))
(turn 121.57)
(turn_neck -55.50)
(see 15 (Goal Front_Distant) (Flag Front_Near) (Flag ExtremeRight_Far) (Flag
Right_Distant) (Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant)
(Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant) (Flag
Right_Distant) (Flag Right_Distant) (Flag Right_Distant) (Flag Right_Distant)
(Flag Right_Distant) (Flag ExtremeRight_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Ball
Front_Near) (Opponent Front_Distant) (Opponent Front_Distant) (Opponent
Front_Far) (Opponent Front_Far) (Opponent Left_Far) (Line Right_Distant))
(sense_body 16 (view_mode high normal) (stamina 3935 1) (speed 0.1 -36)
(head_angle -22) (kick 0) (dash 8) (turn 5) (say 0) (turn_neck 8))
(dash 70.00)
(sense_body 17 (view_mode high normal) (stamina 3910 1) (speed 0.17 11)
(head_angle -22) (kick 0) (dash 9) (turn 5) (say 0) (turn_neck 8))
(see 17 (Goal Front_Distant) (Flag Front_Near) (Flag ExtremeRight_Far) (Flag
Right_Distant) (Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant)
(Flag Left_Distant) (Flag Front_Distant) (Flag Left_Distant) (Flag
Right_Distant) (Flag Right_Distant) (Flag Right_Distant) (Flag Right_Distant)
(Flag Right_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Flag
Left_Distant) (Flag Left_Distant) (Flag Left_Distant) (Ball Front_Near)
(Opponent Front_Distant) (Opponent Front_Distant) (Opponent Front_Far)
(Opponent Front_Far) (Opponent Left_Far) (Line Right_Distant))
(turn -108.41)
```

Appendix D: "Position" ARFF File

```
@relation Object_Position

@attribute Direction real
@attribute Distance real
@attribute Position {ExtremeLeft_Close, ExtremeLeft_Near,
ExtremeLeft_Far, ExtremeLeft_Distant, Left_Close, Left_Near, Left_Far,
Left_Distant, Front_Close, Front_Near, Front_Far, Front_Distant,
Right_Close, Right_Near, Right_Far, Right_Distant, ExtremeRight_Close,
ExtremeRight_Near, ExtremeRight_Far, ExtremeRight_Distant}

@data
0.0,3.0,Front_Close
0.0,6.0,Front_Close
-6.0,17.1,Front_Near
-4.0,27.1,Front_Near
-3.0,37.0,Front_Far
27.0,37.0,Right_Far
7.0,55.1,Front_Distant
3.0,49.4,Front_Distant
17.0,56.8,Right_Distant
26.0,60.9,Right_Distant
40.0,72.2,Right_Distant
0.0,7.5,Front_Near
50.0,33.1,ExtremeRight_Far
56.0,24.0,ExtremeRight_Near
76.0,40.0,ExtremeRight_Far
63.0,43.4,ExtremeRight_Far
53.0,48.9,ExtremeRight_Far
44.0,55.1,ExtremeRight_Distant
-21.0,22.2,Left_Near
-50.0,33.1,ExtremeLeft_Far
-56.0,24.0,ExtremeLeft_Near
-76.0,40.0,ExtremeLeft_Far
-63.0,43.4,ExtremeLeft_Far
-53.0,48.9,ExtremeLeft_Distant
-44.0,55.1,ExtremeLeft_Distant
-34.0,59.7,Left_Distant
-38.0,62.8,Left_Distant
-33.0,70.8,Left_Distant
-29.0,79.8,Left_Distant
0.0,49.4,Front_Distant
-5.0,73.7,Front_Distant
5.0,73.7,Front_Distant
1.0,27.1,Front_Near
-1.0,30.0,Front_Far
41.0,40.4,Right_Far
-36.0,44.7,Left_Far
-1.0,40.4,Front_Far
-2.0,66.7,Front_Distant
-16.0,66.7,Left_Distant
20.0,66.7,Right_Distant
31.0,76.7,Right_Distant
-29.0,76.7,Left_Distant
```

Appendix E: "Between" ARFF File

```
@relation Object_Between

@attribute Distance_Diff real
@attribute Direction_Diff real
@attribute Between {Yes, No}

@data
22.8,11.0,Yes
-22.8,11.0,No
-12.0,29.0,No
-6.799999,21.0,No
-2.5,11.0,No
-5.699997,34.0,No
17.100002,23.0,No
29.800001,8.0,Yes
19.800001,3.0,Yes
9.700001,1.0,Yes
-19.800001,3.0,No
-22.699999,8.0,No
-27.800001,13.0,No
-10.1,2.0,No
10.599998,12.0,Yes
6.299999,5.0,Yes
-7.5,34.0,No
-2.2999992,19.0,No
-6.299999,5.0,No
-10.599998,12.0,No
17.6,1.0,Yes
46.6,11.0,Yes
33.800003,10.0,Yes
-17.6,28.0,No
-17.6,14.0,No
-17.6,1.0,No
-27.499998,11.0,No
-39.6,7.0,No
-33.800003,10.0,No
-34.5,29.0,No
17.799995,34.0,No
8.099998,28.0,No
1.2999954,21.0,No
16.899998,1.0,Yes
4.8999996,1.0,Yes
18.2,12.0,Yes
52.999996,22.0,Yes
25.799995,3.0,Yes
27.2,19.0,Yes
22.300001,18.0,Yes
48.1,21.0,Yes
-13.899998,6.0,No
-24.7,9.0,No
-16.2,13.0,No
-8.299999,19.0,No
```

Appendix F: "Can Pass" ARFF File

```
@relation Pass_Possible

@attribute Distance real
@attribute Player_Between {Yes, No}
@attribute Can_Pass {Pass, No_Pass}

@data
54.6,No,No_Pass
22.2,No,Pass
60.3,No,No_Pass
60.3,No,No_Pass
60.3,No,No_Pass
81.5,Yes,No_Pass
60.3,No,No_Pass
60.3,No,No_Pass
60.3,No,No_Pass
66.7,Yes,No_Pass
24.5,No,Pass
40.4,Yes,No_Pass
66.7,Yes,No_Pass
8.2,No,Pass
30.0,Yes,No_Pass
40.4,No,No_Pass
36.6,No,No_Pass
60.3,Yes,No_Pass
6.0,No,Pass
22.2,No,Pass
33.1,Yes,No_Pass
16.4,No,Pass
33.1,No,Pass
12.2,No,Pass
7.4,No,Pass
30.0,No,No_Pass
36.6,No,No_Pass
7.4,No,Pass
33.1,No,Pass
33.1,No,Pass
24.5,No,Pass
30.0,No,Pass
60.3,Yes,No_Pass
30.0,No,Pass
36.6,No,No_Pass
60.3,No,No_Pass
22.2,No,Pass
36.6,No,Pass
27.1,No,Pass
18.2,Yes,No_Pass
13.5,No,Pass
24.5,No,Pass
14.9,No,Pass
20.1,Yes,No_Pass
20.1,No,Pass
22.2,No,Pass
20.1,No,Pass
36.6,No,No_Pass
60.3,Yes,No_Pass
22.2,No,Pass
44.7,No,No_Pass
14.9,No,Pass
9.0,No,Pass
```

Appendix G: "Can Shoot" ARFF File

```
@relation Can_Shoot

@attribute BallKickable {Yes, No}
@attribute OpponentGoal {Yes, No}
@attribute Reachable {Yes, No}
@attribute OpponentBetween {Yes, No}
@attribute CanShoot {Yes, No}

@data
Yes,Yes,No,Yes,No
Yes,Yes,Yes,Yes,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,No,No,No
Yes,Yes,Yes,Yes,No
Yes,Yes,Yes,Yes,No
Yes,Yes,Yes,Yes,No
Yes,No,Yes,No,No
Yes,Yes,No,No,No
No,Yes,Yes,Yes,No
Yes,No,Yes,No,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,Yes,Yes,No
Yes,Yes,Yes,No,Yes
Yes,No,Yes,No,No
Yes,Yes,No,No,No
No,No,No,Yes,No
No,No,Yes,No,No
No,Yes,No,No,No
No,Yes,Yes,Yes,No
No,Yes,Yes,Yes,No
No,No,Yes,No,No
No,Yes,No,No,No
No,Yes,Yes,Yes,No
No,Yes,Yes,Yes,No
No,No,Yes,No,No
No,Yes,No,No,No
No,Yes,Yes,Yes,No
No,No,Yes,No,No
No,Yes,Yes,No,No
No,No,Yes,No,No
No,Yes,Yes,No,No
No,No,Yes,No,No
No,Yes,Yes,No,No
No,No,Yes,No,No
No,Yes,Yes,No,No
No,No,Yes,No,No
No,Yes,Yes,Yes,No
No,Yes,Yes,No,No
```