Global State Evaluation in StarCraft

Graham Erickson and Michael Buro
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{gkericks|mburo}@ualberta.ca

Abstract

State evaluation and opponent modelling are important areas to consider when designing game-playing Artificial Intelligence. This paper presents a model for predicting which player will win in the real-time strategy game StarCraft. Model weights are learned from replays using logistic regression. We also present some metrics for estimating player skill which can be used a features in the predictive model, including using a battle simulation as a baseline to compare player performance against.

1 Introduction

Purpose

Real-Time Strategy (RTS) games are a type of video game in which players compete against each other to gather resources, build armies and structures, and ultimately defeat each other in combat. RTS games provide an interesting domain for Artificial Intelligence (AI) research because they combine several difficult areas for computational intelligence and are implementations of dynamic, adversarial systems (Buro 2003). The research community is currently focusing on developing bots to play against each other, since RTS AI still preforms quite poorly against human players (Buro and Churchill 2012). The RTS game StarCraft (en.wikipedia.org/wiki/StarCraft) is currently the most common game used by the research community, and is chosen for this work because of the online availability of replay files and the open-source interface BWAPI (code.google.com/p/bwapi).

When human players are playing RTS games, they have a sense of when they are winning or losing the game. Certain aspects of the game which can be observed by the player are used to tell players if they are ahead or behind the other player. The goal of a match is to get the other player to give up or to destroy all that player's units and structures, and achieving that includes but isn't limited to having a steady income of resources, building a large and diverse army, controlling the map, and outperforming the other player in combat. Human players have a good sense of how such features contribute to their chances of winning the game, and will adjust their strategies accordingly. They also have a good sense of determining the skill of their opponent, based on decisions the other player made and their proficiency at combat. We want to enable a bot to do similar. The purpose of our work is to identify quantifiable aspects of a game which can be used to determine 1) if a particular game-state is advantageous to the player or not; and 2) the relative skill level of the opponent.

Motivation

Search algorithms have been used successfully to play the combat aspect of RTS games (Churchill and Buro 2013). Classical tree search algorithms require some sort of evaluation technique; that is, search algorithms require an efficient way of determining if a state is advantageous for the player or not. Currently, there is work being done to create a tree search framework that can be used for playing full RTS game (Ontañón 2012). Evaluation can be done via simulation (Churchill, Saffidine, and Buro 2012) for combat, but for the full game different techniques will be needed. Also, in the context of a complex search framework that uses simulations, state evaluation could be used to prune search branches which are considered strongly disadvantageous. As we will show in Section 4, the type of evaluation we are proposing can be computed much faster than performing a game simulation.

The most successful RTS bots still use hard coded rules as parts of their decision making processes (Ontañón et al. 2013). Which policies are used can be determined by making the bot aware of certain aspects of the opponent. For example, if you have determined that the opponent is implementing strategy A, and you have previously determined that strategy B is a good counter to A, then you can start running strategy B (Dereszynski et al. 2011). Likewise, UAlbertaBot (code.google.com/p/ualbertabot), which won last year's AIIDE StarCraft AI competition, currently uses simulation results to determine if it should engage the opponent in combat scenarios or not. This is based on the assumption that the opponent is proficient at the combat portion of StarCraft. If there is evidence that the opponent is not skilled at combat, one might be willing to engage the opponent even when their army composition is superior (or if they are strong, not engage the opponent unless the player has a large army composition advantage).

Objective

The primary objective of this work is to present a model for evaluating RTS game states. More specifically, we are providing a possible solution to the game result prediction problem: given a game state predict which player will go on to win the game. Our model uses logistic regression (en.wikipedia.org/wiki/Logistic_regression/) to give a response or probability of the player winning (which can be looked at as a value of the state for the player). Presenting our model will then come down to describing the features we compute from a given game state. The features come in two distinct types: features that represent attributes of the state itself (which can be correlated with win status), and features which represent the players skill (which is a much more abstract notion). Our model assumes perfect information; StarCraft is an imperfect information game, but for the purposes of preliminary investigation we assume that the complete game-state is known.

The next section presents background on machine learning in RTS games. In Section 3 we describe the data set we developed the model on, and how we parsed and processed the data set. Then in Section 4 we present the features we use in the model. Section 5 presents our experiments that we used to evaluate the approach, and shows the results along with a discussion of the findings. Limitations of the model, along with future plans are discussed in Section 6.

2 Background

Although there has been a recent trend of using StarCraft replays as data for machine learning tasks (Weber and Mateas 2009) (Synnaeve and Bessière 2011) (Dereszynski et al. 2011), little work has been regarding state evaluation in RTS games. (Yang, Harrison, and Roberts 2014) tries to predict game outcomes in Massively Online Battle Arena (MOBA) games, a different but similar genre of game to RTS. They represent battles as graphs and extract patterns that they use to make decisions about which team will win. Opponent modelling has also seen little work, with (Schadd, Bakkes, and Spronck 2007) being the notable exception (their work was not with StarCraft though). (Avontuur, Spronck, and van Zaanen 2013) extracted features from StarCraft II replays and showed that they can be used to predict the league a player is in.

As presented in (Davidson, Archibald, and Bowling 2013), a *control variate* is a way reducing the error of an estimate of a random variable. They apply control variates (in conjunction with a baseline scripted player) to Poker, as a way of estimating a player's skill. We apply the idea to the combat portion of StarCraft. We use a StarCraft combat simulator to replay battles with a baseline player, and the control variate technique to reduce the variance of the resulting skill feature estimate.

This project uses logistic regression. The data is represented as a matrix X with n examples (rows) and k features (columns). Each row has a corresponding target value, 1 for a player win 0 for a player loss, shown in the column vector Y. The logistic regression algorithm takes X and Y and gives a column vector K of k weights such that

$$X \cdot K = Y'$$

$$T(g(Y')) \approx Y$$

$$g(s) = \frac{1}{1 + e^{-s}}$$

Y' *predicts* Y and the weights K can be used to predict new targets given new examples by applying the weights to the features as a linear combination and then putting those sums through a sigmoid function (g). The result is a response value for each example, i.e., a real number between 0 and 1 that can be thresholded (e.g., T = Heaviside step function) to act as a prediction for the new examples.

3 Data

For our data set, we used a collection of replays collected by Gabriel Synnaeve (Synnaeve and Bessiere 2012). Since professional tournaments usually only release videos of matches and not the replay files themselves, the replays were taken from amateur ladders (GosuGamer (www.gosugamers.net), ICCUP (iccup.com/en/starcraft), and TeamLiquid (www.teamliquid.net)). Synnaeve et al. released both the replay files themselves and parsed text files (http://emotion.inrialpes.fr/people/synnaeve/), but we decided to write our own parser (github.com/gkericks/SCFeatureExtractor), because of the specific nature of our task and to reduce the possible sources of error. Synnaeve et al. collected ∼8000 replays of all different faction match-ups, but we decided to focus on just the ∼400 Protoss versus Protoss matches because the Protoss faction is the most popular faction among bots and our in-house StarCraft bot plays Protoss.

We wrote our replay parser in C++ and it uses BWAPI to extract information from replay file which use a compact representation that needs to be interpreted by the StarCraft engine itself. BWAPI then lets one inject a program like our parser into the StarCraft process. The parser outputs two text files: one with feature values for every player and one with information about the different battles that happened. The first file also includes information about the end of the game, including information about who won the game (which is not always available from the replays) and the final game score, which is computed by the StarCraft game engine. The exact way that the game score is computed is obfuscated by the engine, and the score could not be computed for the opponent in a real game, because of the imperfect information nature of StarCraft, so the game score itself is not a viable feature. In the first file, feature values are extracted for each player according to some period of frames, which is an adjustable parameter in the program.

Battles

Our technique for identifying battles, as shown by Algorithm 1, is very similar to one presented in (Synnaeve and Bessiere 2012). The main difference is what information is logged. Synnaeve et al. were concerned with analyzing army compositions, but we want to be able to actually recreate the battles (the details of which are explained in

Algorithm 1 Technique for extracting battles from replays. Note that UPDATE is called every frame and ON_ATTACK is called when a unit is found to be in an attacking state.

```
Global: List CurrentBattles = []
Global Output: List Battles = []
function ON_ATTACK(Unit u)
    if u in radius of a current battle then
        return
    end if
    U ← Units in MIN_RAD of u.position
    B ← Buildings in MIN_RAD of u.position
    if U does not contain Units from both players then
        return
    end if
    Battle b
    b.units ← U ∪ B
    UPDATE_BATTLE(b, U)
    CurrentBattles.add(b)
end function
function UPDATE
    for all b in CurrentBattles do
        U ← getUnitsInRadius(b.center, b.radius)
        if U = ∅ then
            end(b); continue
        end if
        b.units ← U                          ▷ also log unit info
        if ∃ u ∈ U such that ATTACK(u) then
            UPDATE_BATTLE(b,U)
        end if
        if b.timestamp−CurrentTime() ≥ Δ then
            end(b); continue;
        end if
    end for
    move ended battles from CurrentBattles to Battles
end function
```

Section 4). In Algorithm 1, ON_ATTACK is function that gets called when a unit is executing an attack (during a replay) and UPDATE is a function called every frame. All ON_ATTACK instances for a single frame are called before UPDATE is called. When a new unit is encountered in both the ON_ATTACK and UPDATE functions, the unit's absolute position, health, shield, and the time the unit entered the battle are recorded. When the battle is found to be over (which happens when one side is out of units or no attack actions have happened for some threshold time Δ), the health and shields are recorded for each unit, as well as the time that the battle ended at. Another significant difference is that we start battles based on attacks actions happening (whereas Synnaeve et al. start battles only when a unit is destroyed).

Preprocessing

From viewing the replays themselves, it became apparent that some of the replays would be problematic for our type of analysis. Some games contained endings where the players would appear to be away from their computers, or where one player was obviously winning but appeared to give up.

Algorithm 1 continued

```
function UPDATE_BATTLE(Battle b, UnitSet U)
    center ← average(u.position : u ∈ U)
    maxRad ← 0
    for u ∈ U do
        rad ← distance(u, center) + range(u)
        if rad ≥ maxRad then
            maxRad ← rad
        end if
    end for
    b.center ← center
    b.radius ← maxRad
    b.timestamp ← CurrentTime()
end function
```

Games	Number of Games
Original	447
Kept	391
No Status Close Score	30
Conflict Type A	24
Conflict Type B	1
Corrupt	1

Table 1: A breakdown of how many games were discarded

Such discrepancies could cause mis-labelling of our data (in terms of who won each game), so we chose to filter the data based on a set of rules. Table 1 shows how many replays were discarded in each step.

When extracting the features from the replays BWAPI has two flags (of interest) that can be true or false for each player: *isWinner* and *playerLeft*. If *isWinner* is present the game is kept and that player is marked as the winner. If *isWinner* is not present, then two things are considered: the *playerLeft* flags (which come with a time-stamp denoting when the player left) and the game score. The game score is a positive number computed by the StarCraft engine. If neither player has a *playerLeft* flag, then we look at the game score. If the game score is close (determined by an arbitrary threshold; for this work, we used a value of 5000), the game is discarded. We chose a relatively large threshold because we want to be confident that the player we picked to be the winner is actually the winner. Otherwise, the player with the larger score is selected as the winner. If there is one *playerLeft* flag, the opposite player is taken as the winner, unless that conflicts with the winner as suggested by the game score, in which case the game is discarded (Conflict Type A). If there are two *playerLeft* flags, the player that left second is taken as the winner unless that conflicts with the winner as suggested by the game score, in which case the game is discarded (Conflict Type B). If a replay file was corrupted (i.e., caused StarCraft to crash at some point) we discarded it as well.

4 Features

After the replays are parsed and preprocessed, we represent the data in the form of a machine learning problem. For our

matrix X the examples come from the sample states from the games in the Protoss versus Protoss data set. States were taken every 10 seconds for each replay, so each game gave several examples. Since the match-up is symmetric, we let the features be in terms of the difference between feature values for each player and put in two examples for every game state. For example, if player A has D_A Dragoons and player B has D_B Dragoons and player A wins the game, then one example will have $D_A - D_B$ as the number of Dragoons and a target value of 1 and the other example will have $D_B - D_A$ as the number of Dragoons and a target value of 0. This ensures that the number of 1s and 0s in Y is equal and the learned model is not biased towards either target value.

Economic

Economic features are features that relate to a player's resources. In StarCraft there are two types of resources, minerals and gas. For both resource types we include the current amount that the player has R_{cur} (unspent). A forum post on TeamLiquid (www.teamliquid.net/forum/starcraft-2/266019-do-you-macro-like-a-pro) showed that the ladder StarCraft 2 players were in was correlated with both average unspent resources U and income I. We included both quantities as features in our model. Average unspent resources is computed by taking the current resources a player has at each frame, and dividing that total by the number of frames. Income is a rate of the in-flow of resources and can be computed as simply the total resources R_{tot} divided by the time passed (T). We stress that it is important that these features are included together because an ideal player will keep their unspent resources low (showing that they spend quickly) and keep their income high (showing that they have a good economy).

$$U = (\sum_{t \leq T} R_{cur})/T$$

$$I = \frac{R_{tot}}{T}$$

Military

Every unit and building type has its own feature in the model. As discussed previously, these features are differences between the counts of that unit type for each player. Features for the Scarab and Interceptor unit types were not included, as those units are used as ammo by Reavers and Carriers respectively and the information gained by their existence would already be captured by the existence of the units that produce them. In an earlier version of the model features for research and upgrades were included as well, but we decided to remove them from the final version due to scarcity and the balanced nature of the match-up. The set of all unit count features is UC.

Map Coverage

We chose a simple way of computing map coverage, as a way of approximating the amount of the map that is visible to each player. Each map is divided into a grid, where tiles contain 4 build tiles (build tiles are the largest in-game tile). A tile is considered occupied by a player if the player has a unit on the tile. Tiles can be occupied by both players. Our map coverage score is then a ratio of the total number of occupied tiles to the total number of tiles. For the feature included in the final version of the model, we just count units (not buildings) and only consider tiles that have walkable space (so the score is map specific). This score can then be computed for each player at a given state, and the included feature, MC, is the difference of those two scores. If P is the set of walkable tiles, then for player p MC can also be formalized as:

$$MC(p) = \sum_{pos \in P} f(pos, p)$$

$$f(pos, p) = \begin{cases} 1 & \text{if } pos \text{ is occupied by } p \\ 0 & \text{otherwise} \end{cases}$$

Micro Skill

Skill is an abstract concept and different aspects of a player's skill can be seen by observing different aspects of a game. The combat portion of the game (also known as micro-managing) takes a very specific type of skill. We devised a feature to capture the skill of a player at the micro portion of the game. The feature makes use of the ideas in (Davidson, Archibald, and Bowling 2013), where a scripted player is used as a baseline to compare a player's performance against. We grab battles (as shown in Section 3), play them out using a StarCraft battle simulator (*SparCraft*, developed by David Churchill (code.google.com/p/sparcraft), and compare the results of the scripts to the real battle outcomes. We use a version of *SparCraft* edited to support buildings as obstacles, as well as units entering the battle at varying times.

We used a scripted player as the baseline. For this work, we use the *NOK-AV* (No-OverKill-Attack-Value) script and a version of the LTD2 evaluation function to get a value from a battle (Churchill, Saffidine, and Buro 2012). *NOK-AV* has units attack an enemy in range with the highest damage-per-frame / hit-points, and has units switch targets if the current target has been assigned a lethal amount of damage already. Note that although we include buildings in the battles, buildings are not acknowledged explicitly by the script policy, and thus are just used as obstacles.

We need a way of comparing the outcome of the real battle to that of the baseline, in a way that can be used as a feature in our model. For a single player, with a set of units U, the Life-Time-Damage-2 (LTD2) (Kovarsky and Buro 2005) score is:

$$\text{LTD2}_{start}(U) = \sum_{u \in U} \sqrt{\text{HP}(u)} \cdot \text{DMG}(u)$$

LTD2 is an evaluation function which favours having multiple units to single units given equal summed health and rewards keeping units alive that can deal greater damage quicker. LTD2 is sufficient for calculating the army value at the end of the battle, but since units can enter the battle at varying times, we need a way of weighting the value of each unit. Let T be the length of the battle and st(u) be the

time unit u entered the battle (which can take values from 0 to T). Then:

$$\text{LTD2}_{end}(U) = \sum_{u \in U} \frac{T - \text{st}(u)}{T} \cdot \sqrt{\text{HP}(u)} \cdot \text{DMG}(u)$$

Let one of the players be the player (P) and the other to be the opponent (O). Which player is which is arbitrary, as there are two examples for each state (where both possible assignments of player and opponent are represented). Let P_{out} and P_s be the player's units at the end and the start of the battle respectively, and O_{out} and O_s to be the opponent's units at the end and the start of the battle respectively. The value for the battle is then:

$$V^P = (\text{LTD2}_{end}(P_{out}) - \text{LTD2}_{end}(O_{out})) - (\text{LTD2}_{start}(P_s) - \text{LTD2}_{start}(O_s))$$

To get a baseline value for the battle, we take the initial army configuration for each player (including unit positions and health) and play out the battle in the simulator until time T has passed. At that point, let P_β and O_β be the remaining units for both player and opponent respectively. Then the value given by the baseline player is:

$$V^\beta = (\text{LTD2}_{end}(P_\beta) - \text{LTD2}_{end}(O_\beta)) - (\text{LTD2}_{start}(P_s) - \text{LTD2}_{start}(O_s))$$

For a particular state in a game where there have been n battles with values $V_1^P, V_2^P, \ldots, V_i^P, \ldots, V_n^P$ and $V_1^\beta, V_2^\beta, \ldots, V_i^\beta, \ldots, V_n^\beta$, we can get a total battle score (β) by:

$$\beta_{tot} = \sum_{i=1}^{n} (V_i^P - V_i^\beta)$$

$$\beta_{avg} = \frac{\beta_{tot}}{n}$$

Note that β_{tot} the $(\text{LTD2}_{start}(P_s) - \text{LTD2}_{start}(O_s))$ parts for each V_i will cancel out. They are left in the definitions above because we can also represent the feature as a baseline control variate (Davidson, Archibald, and Bowling 2013):

$$\beta_{var} = \frac{1}{n} \sum_{i=1}^{n} (V_i^P - \frac{\widehat{\text{Cov}}[V_i^P, V_i^\beta]}{\widehat{\text{Var}}[V_i^P]} \cdot V_i^\beta)$$

We then use β_{var} as a feature in our model. We also devise β_{avg} and β_{var} as estimates of a player's skill at the micro game. Although we do not explore the idea experimentally here, we maintain that β_{var} could be used in an RTS bot's decision making process: for high values (showing that opponent is skilled) the bot would be conservative about which battles it engaged in, and for low values (showing that the opponent is not skilled) the bot would take more risks in hopes to exploit the opponent's poor micro abilities.

Macro Skill

In contrast to the micro game, the macro game refers to high-level decision making, usually affecting economy and unit production. The macro game is much more complicated (and it encompasses the micro game) and we do not have a scripted player for the macro game, so the baseline approach to skill estimation does not apply. Instead, we have identified features that suggest how good a player is at managing the macro game. The first, inspired by (Belcher, Ekins, and Wang 2013) is the number of frames that supply is maxed out for SF. Supply is an in-game cap on how many units a player can build. They must construct additional structures to increase the supply cap (S_{max}). Thus, if a player has a large amount of frames where their supply S_{cur} was maxed out, it shows that the player is bad at planning ahead.

$$SF = \sum_{t \leq T} f(t)$$

$$f(t) = \begin{cases} 1 & \text{if } S_{cur} = S_{max} \text{ at time t} \\ 0 & \text{otherwise} \end{cases}$$

The other feature is the number of idle production facilities PF. A production facility is any structure that produces units and if a player has a large amount of production facilities which are not actively producing anything, it suggests that the player is poor at managing their unit production. A third feature is the number of units a player has queued Q. Production facilities have queues that units can be loaded into, and it is considered skillful to keep the queues small (since a player must pay for the unit the moment it is queued). However, both PF and Q could never be used as part of a state evaluation tool, since that information is not known even with a perfect information assumption.

5 Evaluation and Results

For the purposes of experimentation, we did 10-fold cross validation on games. Recall that our example matrix X is made up of examples from game states from different games at different times. If we permuted all the examples and did cross validation we would be training and testing on states that are from the same games (which introduces a bias). So instead we chose to split the games into 10 different folds. When we leave a particular fold out for testing, we are leaving out all the states that were taken from the games in that fold. So for each fold, logistic regression is ran on all the other folds giving a set of feature weights. Those feature weights can then be used as the weights for a linear combination of the features of each of the examples in the testing fold and then putting those sums through a squashing function (sigmoid function). The result is a response value for each example i.e., a real number between 0 and 1 that acts as a prediction for the test examples.

We report two different metrics for showing the quality of our model. The first is accuracy, which is just the proportion of correct predictions (on the test folds) to the total number of examples. Predictions are taken from the response values by thresholding them. We used a threshold value of 0.5, which is standard. The second is the average log-likelihood (Glickman 1999). For a single test example, whose actual target value is given by y and whose response value is r, the log-likelihood is defined as follows:

$$L(y, r) = y \cdot \log(r) + (1 - y) \cdot \log(1 - r)$$

Features	0-5	5-10	10-15	15-
R_{cur},I,U	54.42 (-0.686)	57.76 (-0.672)	62.98 (-0.647)	64.17 (-0.625)
UC	51.96 (-0.712)	57.84 (-0.682)	66.67 (-0.705)	66.46 (-0.644)
MC	51.27 (-0.693)	55.20 (-0.685)	61.45 (-0.657)	71.39 (-0.561)
β_{var}	50.23 (-0.693)	53.25 (-0.690)	55.09 (-0.690)	52.82 (-0.690)
SF, PF, Q	51.26 (-0.695)	49.96 (-0.695)	51.75 (-0.694)	54.97 (-0.709)
A	53.91 (-0.708)	58.81 (-0.680)	66.36 (-0.712)	69.22 (-0.613)
B	54.05 (-0.708)	58.66 (-0.681)	66.44 (-0.712)	69.87 (-0.608)
C	53.81 (-0.710)	58.72 (-0.681)	66.41 (-0.708)	72.59 (-0.587)

Table 2: Individual feature (group) and feature set prediction performance reported as accuracy(%) (avg L) in each game time period; A = economic/military features R_{cur}, I, U, UC; B = A + map control feature MC; C = B + skill features β_{var}, SF, PF, Q

Feature Set	0-5	5-10	10-15	15-20
R_{cur},I,U	53.75 (-0.6875)	58.85 (-0.6708)	62.82 (-0.6510)	60.23 (-0.6562)
UC	52.03 (-0.6936)	58.43 (-0.6735)	65.76 (-0.6329)	63.96 (-0.6516)
MC	51.27 (-0.6943)	55.20 (-0.6872)	61.45 (-0.6588)	64.02 (-0.6385)
β_{var}	50.23 (-0.6931)	53.25 (-0.6896)	55.24 (-0.6899)	56.14 (-0.6868)
SF, PF, Q	52.02 (-0.6925)	50.74 (-0.6939)	52.82 (-0.6916)	55.21 (-0.6857)
A	53.19 (-0.6917)	58.74 (-0.6726)	65.28 (-0.6367)	63.58 (-0.6612)
B	52.60 (-0.6916)	58.56 (-0.6727)	64.89 (-0.6377)	63.99 (-0.6617)
C	52.73 (-0.6914)	58.70 (-0.6669)	65.77 (-0.6267)	65.23 (-0.6510)

Table 3: Feature set prediction performance [accuracy(%) (avg L)]; If time interval is [k,l] training is done on examples in [k,∞) and tested on examples in [k,l]

We report the average log-likelihood across examples. Values closer to zero indicate a better fit.

Through experimentation we want to answer the following questions: 1) Can our model be used to accurately predict game outcomes? 2) Does adding the skill features to our model improve the accuracy? 3) What times during a game is our model most applicable to?

To explore these questions we tested a few different feature subsets on examples from different times in the games. Especially in the early game, result prediction is a very noisy problem because there is little evidence in the early game as to who is going to win because the important events of the game have not happened yet. We decided to experiment by running separate cross validation tests just using examples that fall within certain time intervals. For time intervals, we use 5 minute stretches and just include any example with a time-stamp greater than 15 minutes together. Note that not all games are the same length, so for the later time intervals not all games are included. 5 minute interval lengths were chosen because we wanted to have a reasonable amount of examples in each interval while still doing a fine-grained analysis. Table 4 shows how examples were divided based on time-stamps.

Table 2 shows the performance of using the feature sets individually. The map coverage feature MC preforms very well in the late game because a good MC value near the end of the game is the result of having a good economy earlier in the game and having a strong army. In general, prediction rates improve in the later game stages. β_{var} has a drop in accuracy in the late game because many of the battles at that stage of the game include unit types which our simulator does not support and so late game battles (which are important to the game outcome) are ignored. Table 2 also shows feature sets being added, culminating in the full model in line C. Notice that the skill features make a difference in the late game, due to there being differences in skill being more noticeable as the game progresses (SF takes to grow, as players need to make mistakes for it to be useful). When choosing intervals we had problems with overfitting. UC especially is prone to overfitting if the training set is too small. Table 3 shows how we tested with larger training sets to avoid overfitting. Results are overall slightly lower because the early intervals are trained with all or most of the timestamps, and examples from 20- are never tested on.

Time (min)	Games	Examples
0-5	391	23418
5-10	386	22616
10-15	364	19836
15-20	289	14996
20-	211	31060

Table 4: A breakdown of how examples were split by timestamp

6 Conclusions and Future Work

Our results show that prediction of the game winner in an RTS game domain is possible, although the problem is noisy. Prediction is most promising in later game stages. This gives hope for quick evaluation in future search algorithms. Also promising is the use of a baseline to estimate player skill, although work is still needed to improve its performance as

a state evaluator. Future work includes extending our simulator to support more unit types, improving our baseline player to get a closer player skill estimate, reproducing our experiments on a larger data set, and altering our model to work with the imperfect information game.

References

Avontuur, T.; Spronck, P.; and van Zaanen, M. 2013. Player skill modeling in starcraft ii. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Belcher, J.; Ekins, D.; and Wang, A. 2013. Starcraft 2 oracle. *University of Utah CS6350 Project*. http://www.eng.utah.edu/ cs5350/ucml2013/proceedings.html.

Buro, M., and Churchill, D. 2012. Real-time strategy game competitions. *AI Magazine* 33(3):106–108.

Buro, M. 2003. Real-time strategy games: A new AI research challenge. In *IJCAI 2003*, 1534–1535. International Joint Conferences on Artificial Intelligence.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 112–117.

Davidson, J.; Archibald, C.; and Bowling, M. 2013. Baseline: practical control variates for agent evaluation in zero-sum domains. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 1005–1012. International Foundation for Autonomous Agents and Multiagent Systems.

Dereszynski, E.; Hostetler, J.; Fern, A.; Hoang, T. D. T.-T.; and Udarbe, M. 2011. Learning probabilistic behavior models in real-time strategy games. In AAAI., ed., *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Glickman, M. E. 1999. Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 48(3):377–394.

Kovarsky, A., and Buro, M. 2005. Heuristic search applied to abstract combat games. *Advances in Artificial Intelligence* 66–78.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *TCIAIG*.

Ontañón, S. 2012. Experiments with game tree search in real-time strategy games. *arXiv preprint arXiv:1208.1940*.

Schadd, F.; Bakkes, S.; and Spronck, P. 2007. Opponent modeling in real-time strategy games. In *GAMEON*, 61–70.

Synnaeve, G., and Bessière, P. 2011. A Bayesian model for plan recognition in RTS games applied to StarCraft. In AAAI., ed., *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2011)*, Proceedings of AIIDE, 79–84.

Synnaeve, G., and Bessiere, P. 2012. A dataset for StarCraft AI & an example of armies clustering. In *AIIDE Workshop on AI in Adversarial Real-time games 2012*.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG)*.

Yang, P.; Harrison, B.; and Roberts, D. L. 2014. Identifying patterns in combat that are predictive of success in moba games. *Proceedings of Foundations of Digital Games 2014* to appear.