Extracting Policies from Replays to Improve MCTS in Real Time Strategy Games

Zuozhi Yang and **Santiago Ontañón***
Drexel University, Philadelphia, Pennsylvania 19104
{zy337, so367}@drexel.edu

Abstract

In this paper we study the topic of integrating supervised learning models into Monte Carlo Tree Search (MCTS) in the context of RTS games. Specifically, we focus on learning a tree policy for MCTS using existing supervised learning algorithms. We evaluate and compare two families of models: Bayesian classifiers and decision trees classifiers. Our results show that in experiments under same iteration budget for MCTS, the models with higher classification performance also have better gameplay strength when used within MCTS. However, when we constrain computation budget by time, faster models tend to outperform slower, more accurate, models. Surprisingly, the classic C4.5 algorithm stands out in our experiments as the best model since it has good classification performance and fast classification speed.

Introduction

Board and computer games have always been popular in the AI research community since they offer a challenging and rich testbed for both machine learning and search techniques. A prominent example is AlphaGo (Silver et al. 2016), which demonstrated that it is possible to deal with large search spaces by integrating machine learning and search. Real-time strategy (RTS) games have become a more active research area as it offers a greater challenge because of their enormous state space and branching factor, real-time nature, and partial observability. In this paper we study the topic of integrating supervised learning models into Monte Carlo Tree Search (MCTS) in the context of RTS games. Specifically, we focus on learning tree policy for MCTS using existing supervised learning algorithms. Due to the large state space and branching factor, it is hard to obtain good policy real-time from MCTS. Thus, learning a reliable policy offline and bias the tree search towards a more promising search space can significantly improve the performance of MCTS.

In this paper we build upon previous work on Informed MCTS, where Bayesian models were trained from data to inform the tree policy (Ontañón 2016). Moreover, given that RTS games have much more limited decision budget between each two game decisions compared to other games, like board games, the model we build must make fast prediction at testing time, apart from the requirement of accuracy. In consideration of the time constraint, we evaluate and compare two family of models: Bayesian classifiers and decision trees classifiers. Our results show that in experiments under same iteration budget for MCTS, the models with higher classification performance also have better gameplay strength when used within MCTS. However, when we constrain computation budget by time, faster models tend to outperform slower, more accurate, models. Specifically, the C4.5 model stands out in our experiments as the best model since it has good classification performance and fast classification speed.

The rest of this paper is structure as follows: (1) In the background section we introduce RTS games research, the research platform we employ, and work on applying MCTS to RTS games. (2) Then we describe and compare the two algorithm families that we study. (3) After that, we specify the details of our experiments on comparing the models in classification performance and gameplay strength as tree policy for MCTS. (4) Finally, we discuss the empirical results and layout the possible lines of future work.

Background

Real-time strategy (RTS) games have been receiving an increased amount of attention as they are even more challenging than games like Go or Chess in at least three different ways: (1) the combinatorial growth of the branching factor (Ontañón 2017), (2) limited computation budget between actions due to the real-time nature, and (3) lack of forward model in most of research environments like Starcraft. Many research environments and tools, such as TorchCraft (Synnaeve et al. 2016), SCIILE (Vinyals et al. 2017), μRTS (Ontañón 2013), ELF (Tian et al. 2017), and Deep RTS (Andersen, Goodwin, and Granmo 2018) have been developed to promote research in the area. Specifically, in this paper, to stay focused on the problem of interest of this paper, we chose μRTS as our experimental domain, as it offers a forward model for game tree search approaches such as minimax or Monte Carlo Tree Search.

Real-Time Strategy Games

RTS is a sub-genre of strategy games where players aiming to defeat their opponents (destroying their army and

*Currently at Google

base) by strategically building an economy (gathering resources and building a base), military power (training units and researching technologies), and controlling those units. The main differences between RTS games and traditional board games are: they are simultaneous move games (more than one player can issue actions at the same time), they have durative actions (actions are not instantaneous), they are real-time (each player has a very small amount of time to decide the next move), they are partially observable (players can only see the part of the map that has been explored, although in this paper we assume full observability) and they might be non-deterministic.

Furthermore, comparing to traditional board games, RTS games have a very large state space and action space at each decision cycle. For example, the branching factor in StarCraft can reach numbers between 30^{50} and 30^{200} (Ontañón et al. 2013).

μRTS

μRTS[1] is a simple RTS game designed for testing AI techniques. μRTS provides the essential features that make RTS games challenging from an AI point of view: simultaneous and durative actions, combinatorial branching factors and real-time decision making. The game can be configured to be partially observably and non-deterministic, but those settings are turned off for all the experiments presented in this paper. We chose μRTS, since in addition to featuring the above properties, it does so in a very minimalistic way, by defining only four unit types and two building types, all of them occupying one tile, and there is only one resource type. Additionally, as required by our experiments, μRTS allows maps of arbitrary sizes and initial configurations.

There is one type of environment unit (minerals) and six types of units controlled by players, which are:

- Base: can train Workers and accumulate resources
- Barracks: can train attack units
- Worker: collects resources and construct buildings
- Light: low power but fast melee unit
- Heavy: high power but slow melee unit
- Ranged: long range attack unit

Additionally, the environment can have walls to block the movement of units. A example screenshot of game is shown in Figure 1. The squared units in green are Minerals with numbers on them indicating the remaining resources. The units with blue outline belong to player 1 (which we will call *max*) and those with red outline belong to player 2 (which we will call *min*). The light grey squared units are Bases with numbers indicating the amount of resources owned by the player, while the darker grey squared units are the Barracks. Movable units have round shapes with grey units being Workers, orange units being Lights, yellow being Heavy units (now shown in the figure) and blue units being Ranged.

[1] https://github.com/santiontanon/microrts

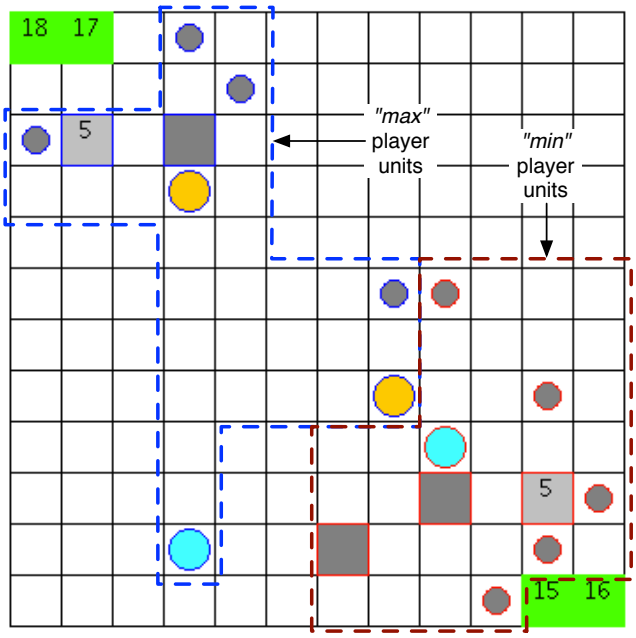

Figure 1: A Screenshot of μRTS.

Monte Carlo Tree Search in RTS Games

Monte Carlo Tree Search (Browne et al. 2012) (Coulom 2006) is a method for sequential decision making for domains that can be represented by search trees. It has been a successful approach to tackle complex games like Go as it takes random samples in the search space to estimate state value. However, most of the successful variants of MCTS, e.g. UCT (Kocsis and Szepesvári 2006), do not scale up well to RTS games due to the combinatorial growth of branching factor with respect to the number of units. Sampling techniques for combinatorial branching factors such as Naïve Sampling (Ontañón 2013) or LSI (Shleyfman, Komenda, and Domshlak 2014) were proposed to improve the exploration of MCTS exploiting combinatorial multi-armed bandits. Inspired by AlphaGo, another approach to address this problem is the use of Naïve Bayes models to learn a action probability distribution as a tree policy prior to guide the search (Ontañón 2016). Other work to deal with this problem involves limiting the search space by introducing action abstractions. For example, instead of searching directly in the raw unit action space, Portfolio greedy search (Churchill and Buro 2013) and Stratified Alpha-Beta Search (Moraes and Lelis 2017) search in the abstracted action spaces generated by hard-coded scripts.

In this paper, we extend the work of policy distribution learning (Ontañón 2016) by comparing existing machine learning models and applying the trained model to tree policy of MCTS.

Naïve Monte Carlo Tree Search. Naïve Monte Carlo Tree Search (NaïveMCTS) (Ontañón 2013) is a variant of MCTS specifically designed to handle RTS games. For that purpose, NaïveMCTS can also handle durative and simultaneous actions. Most importantly, the key feature of

NaïveMCTS is that rather than using standard ϵ-greedy or UCB1, it uses a sampling policy for Combinatorial Multiarmed Bandits (CMABs) called *Naïve Sampling* in order to scale up to the combinatorial branching factors of RTS games.

Specifically, in our experiments, we use the implementation of Informed NaïveMCTS built in into μRTS, which can be used to incorporate machine learning models into the tree policy, as described below.

Informed Tree Policy. MCTS algorithms can be broken down into the following four stages (Browne et al. 2012):

1. Selection: Starting at root node, recursively select child nodes according to a pre-defined *tree policy* until a leaf node L is reached.
2. Expansion: If the selected node L is a not a terminal node then add a child node C of L to the tree (also using the tree policy).
3. Simulation: Run a simulation from C according to a *playout policy* until a terminal state is reached.
4. Backpropagation: Update the statistics of the current move sequence with the simulation result.

The *tree policy* serves as the criteria of child node selection within the search tree and balances exploration and exploitation. It can also be informed by a given prior distribution and then bias the search towards the desired action space. For example, the PUCB algorithm (Predictor+UCB) (Rosin 2011) extends the standard UCB1 strategy commonly used as the tree policy with an existing distribution. A variation of PUCB was used in AlphaGo, where Silver et al. (Silver, Sutton, and Müller 2012) used temporal difference method to learn a value function and to inform the tree policy. This prior distribution can be trained offline, e.g. from existing game replays. AlphaGo, for example, trained a neural network using the human expert plays as supervision, and then refined it using reinforcement learning. Unlike the game of Go, however, the RTS games are real-time, which means the computational budget per action is much smaller than in Go. Therefore, large neural network models become impractical since they are currently too slow for the available computational budget. Thus, in this paper, we select a collection of statistical learning algorithms which are potentially fast at classification time as our subjects of study.

Methods for Tree Policy Learning

We use supervised learning to model and predict the move probability of script bots and bias the tree search to mimic and hopefully improve upon script bots with the help of tree search. A supervised learning model $p_u(a_i|s)$ takes a feature vector s as the representation of the game state from the point of view of a unit u and output the probability distribution over all n possible actions $(a_1, a_2, a_3, \cdots, a_n)$ the unit can perform. The model is trained beforehand and prediction is made each time the tree policy needs to be used. Thus our learning algorithm for modeling the problem must be fast at classification time. Also, in order to increase the sampling quality, the classifier should also provide good class probability estimations. Therefore, we selected a relatively small set of categorical features (eight features) to represent the game states. Moreover, two types of classification algorithms what are previously known to be suitable to categorical data and fast at classification time are studied in this paper: (semi) Naïve Bayes and (ensembles of) decision trees.

Naïve Bayes and Semi Naïve Bayes

The Naïve Bayes classifier is a simple probabilistic classifier that makes a strong assumption that all the features are independent. This allows us to write the joint density as the product of all the component densities:

$$p(\boldsymbol{x}|y = c, \boldsymbol{\theta}) = \prod_{i=1}^{D} p(x_i|y = c, \boldsymbol{\theta}_{ic})$$

Naïve Bayes classifiers are highly scalable and can be trained and tested both in linear time. However, the assumption of independent features can be to violated for most of the situations. Thus, the research community has developed many semi-Naïve Bayes algorithms that have a weakened independence assumption.

The semi-Naïve Bayes algorithms tested in our paper is called averaged one-dependence estimators (A1DE) (Webb, Boughton, and Wang 2005). Sahami introduces n-dependence estimators, where the probability of each feature is conditioned by the class and n other features. (Sahami 1996) The averaged one-dependence estimators work like this: for each feature $\boldsymbol{x}$, a classifier that assumes the other features are independent given the label and feature $\boldsymbol{x}$. The model makes predictions by averaging the outcome of all classifiers. Therefore, A1DE can be also viewed as a ensemble methods for Naïve Bayes classifiers. We also test a more relaxed variation of A1DE called A2DE, which build estimators conditioned by each pair of features.

Decision Trees and Ensembles

The C4.5 classifier (Quinlan 2014) iteratively builds a decision tree by splitting the instances by the feature with the most information gain (reduction of entropy) at each internal node. Then it assigns class predictions at leaf nodes. Then the algorithm prunes the tree by a threshold confidence interval of 25%. We can estimate the class probability naturally from the label frequency of the leaves. Moreover, this estimation can be skewed towards 0 or 1 since the leaves are mostly dominated by one class. In our experiments, Laplace smoothing is applied in order to produce more accurate class probability estimation.

Bootstrapped aggregating (bagging) (Breiman 1996) is an ensemble method that helps to improve accuracy and stability by combining results from multiple training sets resampled with replacement. In our experiments, we apply 10 iterations of bagging to C4.5.

Random forest (Liaw, Wiener, and others 2002) is a meta classifier that fits a collection of random tree classifiers on resamples of the dataset and thus improve the predictive accuracy and control overfitting by averaging. The internal random trees select a subset of features to build the model. In our case, each random tree selects a subset of $\log(n) + 1$

Table 1: Model Comparison on Classification Accuracy in the 12 Datasets

	Naïve Bayes		A1DE		A2DE		J48		Bagging		Random Forest	
	Accuracy	Exp. l.l.	Accuracy	Exp. l.l.	Accuracy	Exp.l.l	Accuracy	Exp. l.l.	Accuracy	Exp. l.l.	Accuracy	Exp. l.l.
WR	0.6119	-1.0749	0.6609	-0.8921	0.6698	**-0.8527**	0.6821	-0.8574	0.6828	-0.9074	**0.6882**	-0.9810
LR	0.7955	-0.6048	0.8210	-0.5059	**0.8276**	**-0.4876**	0.8188	-0.4972	0.8248	-0.5598	0.8263	-0.6504
HR	0.8490	-0.4765	0.8671	-0.3950	**0.8732**	**-0.3825**	0.8666	-0.3995	0.8703	-0.4379	0.8716	-0.4855
RR	0.8411	-0.5223	0.8585	-0.4110	0.8602	**-0.3954**	0.8593	-0.4070	0.8606	-0.4389	**0.8648**	-0.4450
LSI500	0.3637	-1.4711	0.3713	-1.4048	**0.3715**	-1.3902	0.3637	**-1.3488**	0.3618	-1.5125	0.3595	-1.9905
LSI1000	0.3743	-1.4626	**0.3795**	-1.3904	0.3776	-1.3721	0.3628	**-1.3392**	0.3619	-1.5155	0.3606	-1.9683
LSI2000	0.3832	-1.4621	**0.3920**	-1.3877	0.3915	-1.3708	0.3749	**-1.3314**	0.3764	-1.5202	0.3741	-1.9860
LSI5000	0.3977	-1.4382	**0.4062**	-1.3578	0.4051	-1.3400	0.3866	**-1.3119**	0.3903	-1.4692	0.3878	-1.9382
NaïveMCTS500	0.3747	-1.4711	**0.3832**	-1.4017	0.3811	-1.3861	0.3677	**-1.3385**	0.3672	-1.5314	0.3635	-1.9821
NaïveMCTS1000	0.3839	-1.4544	**0.3926**	-1.3809	0.3893	-1.3641	0.3740	**-1.3312**	0.3747	-1.4991	0.3709	-1.9431
NaïveMCTS2000	0.3916	-1.4527	**0.3995**	-1.3831	0.3985	-1.3665	0.3798	**-1.3296**	0.3827	-1.5034	0.3802	-1.9374
NaïveMCTS5000	0.4005	-1.4422	0.4123	-1.3652	**0.4126**	-1.3451	0.3969	**-1.3100**	0.3974	-1.4677	0.3963	-1.9043

Table 2: Comparison of Model Classification Speed

Model	Instances/ms.	Complexity
Naïve Bayes	303.58	$O(kn)$
A1DE	174.46	$O(kn^2)$
A2DE	43.07	$O(kn^3)$
J48	298.05	$O(log(n))$
Bagging+J48	64.37	$O(m \cdot log(n))$
Random Forest	30.97	$O(m \cdot log(n))$

where n is the number of features. In our experiments 100 random trees are built by the random forest algorithm. The class probability estimation is generated from the proportion of votes of the trees in the ensemble.

In practice, we observed that learning a different model for each different unit type in the game (workers, bases, barracks, etc. in μRTS), resulted in better estimation of the probabilities. So, for the experiments reported in the remainder of this paper, we generated the probability models in the following way: (1) For each unit type we train a model using the subset of the training data corresponding to the unit type. (2) If this subset is empty, then just train with the whole training set (i.e., if we have no training data to model the way a specific unit is controlled, we just train a model with the whole training set for such unit, hoping it will reflect what the script we are collecting data from would have done).

Experiments and Results

Bots for Data Collection

We generated training data in the following way. We collected data from a round-robin tournament consists of six bots in 8 different maps, four of which are 8×8 maps and the other four are 12×12. Four of the bots are built-in hard-coded bots: WorkerRush, LightRush, HeavyRush, and RangedRush. The other two were Monte-Carlo search bots: LSI and NaïveMCTS, whose computational budgets are configurable. We run the experiments for LSI and NaïveMCTS bots for four times with computation budgets of 500, 1000, 2000, and 5000 playouts per game frame respectively. The description of each bot is as below:

- *Rush bots:* hardcoded deterministic bots that implement a rush strategy that constantly produces one type of unit and sends them to attack the opponent's base. Specifically, we used the *WorkerRush*, *LightRush*, *RangedRush* and *HeavyRush* bots of μRTS, which implement rushes with worker, light, ranged, and heavy units respectively.
- *LSI:* Monte Carlo search with Linear Side Information (Shleyfman, Komenda, and Domshlak 2014)
- *NaïveMCTS:* MCTS algorithm with a tree policy specifically designed for games with combinatorial branching factors. The tree policy exploits Combinatorial Multi-Armed Bandits (Ontañón 2017) to handle the combinatorial explosion of possible moves.

The feature vector $\boldsymbol{x}(u, s)$ used to represent each game state contains only eight features: the number of resources available to the player, the cardinal direction (north, east,south, west) toward where most friendly units are, the cardinal direction toward where most enemy units are, whether we have a barracks or not, and four features indicating the type of the unit in the cell two positions north, east, south or west (or whether these cells are empty or are a wall). Adding more features could certainly improve performance, which will be part of our future work. We employed these 8 features, as used in previous work (Ontañón 2016), for comparison purposes.

The 12 datasets collected are described below:

- I_{WR}: consisting of all the unit-actions of WorkerRush (32679 instances)
- I_{LR}: consisting of all the unit-actions of LightRush (30139 instances)
- I_{HR}: consisting of all the unit-actions of HeavyRush (24385 instances)
- I_{RR}: consisting of all the unit-actions of RangedRush (31279 instances)
- I_{LSI}^{500},I_{LSI}^{1000},I_{LSI}^{2000},I_{LSI}^{5000}: consisting of all the unit-actions of LSI under computing budget of 500, 1000, 2000, and 5000 respectively (85918, 88816, 85611, and 68236 instances respectively)
- I_{NMCTS}^{500},I_{NMCTS}^{1000},I_{NMCTS}^{2000},I_{NMCTS}^{5000}: consisting of all the unit-actions of NaïveMCTS under computing budget of 500, 1000, 2000, and 5000 respectively (73249, 83925, 78896, and 68158 instances respectively)

Table 3: Win rates against the baselines of the models trained by I_{WR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.987	1.000	1.000	0.993	0.993	0.993
WorkerR	0.668	0.650	0.662	0.637	0.675	0.668
LightR	0.843	0.875	0.806	0.900	0.868	0.850
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.656	0.662	0.637	0.612	0.662	0.706
NMCTS	0.475	0.531	0.506	0.637	0.587	0.687
Average	0.829	0.840	0.827	0.848	0.848	**0.863**

Table 4: Win rates against the baselines of the models trained by I_{LR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.993	1.000	1.000	1.000	0.987	0.993
WorkerR	0.662	0.581	0.625	0.6	0.687	0.681
LightR	0.862	0.862	0.787	0.781	0.862	0.831
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	0.993	1.000	1.000	1.000	1.000
LSI	0.618	0.656	0.618	0.650	0.650	0.687
NMCTS	0.456	0.525	0.500	0.643	0.593	0.606
Average	0.824	0.827	0.816	0.834	0.848	**0.850**

Table 5: Win rates against the baselines of the models trained by I_{HR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	1.000	0.994	1.000	1.000	1.000	1.000
WorkerR	0.631	0.625	0.637	0.713	0.688	0.744
LightR	0.844	0.869	0.856	0.881	0.869	0.919
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	0.993	1.000	1.000	1.000	1.000
LSI	0.544	0.569	0.650	0.738	0.725	0.637
NMCTS	0.512	0.487	0.550	0.575	0.606	0.675
Average	0.816	0.818	0.837	0.863	0.861	**0.872**

Table 6: Win rates against the baselines of the models trained by I_{RR}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.994	1.000	0.994	0.994	1.000	0.994
WorkerR	0.675	0.600	0.637	0.681	0.694	0.713
LightR	0.800	0.825	0.875	0.812	0.906	0.838
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.588	0.637	0.725	0.700	0.706	0.738
NMCTS	0.619	0.494	0.644	0.588	0.619	0.600
Average	0.834	0.820	0.859	0.847	**0.866**	0.860

Table 7: Win rates against the baselines of the models trained by I_{LSI5000}

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Rndom	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	1.000	1.000	1.000	1.000	0.994	1.000
WorkerR	0.650	0.675	0.700	0.706	0.675	0.725
LightR	0.856	0.944	0.894	0.931	0.894	0.844
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.850	0.825	0.688	0.688	0.688	0.744
NMCTS	0.675	0.688	0.662	0.562	0.725	0.625
Average	0.879	**0.891**	0.868	0.861	0.872	0.867

Table 8: Win rates against the baselines of the models trained by $I_{\text{NMCTS5000}}$

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	1.000	1.000	1.000	0.994	0.994	0.994
WorkerR	0.731	0.700	0.738	0.600	0.688	0.688
LightR	0.875	0.831	0.881	0.887	0.887	0.875
HeavyR	1.000	1.000	1.000	1.000	1.000	1.000
RangedR	1.000	1.000	1.000	1.000	1.000	1.000
LSI	0.800	0.762	0.769	0.700	0.775	0.794
NMCTS	0.650	0.725	0.756	0.688	0.688	0.669
Average	0.882	0.877	**0.893**	0.859	0.879	0.877

Empirical Comparison of Models

For all six models, we used the Weka implementation[2]. In this section we compare the six machine learning models described before from the following aspects:

- action prediction (accuracy and probability estimation)
- classification speed
- gameplay strength as tree policy under iteration budget
- gameplay strength as tree policy under time budget

Unit Action Classification First we evaluate the models on the classification and class probability estimation performance. Table 1 shows the predictive accuracy and expected log-likelihood of the six models in each of the 12 datasets using a 10-fold cross validation. There is a total of 69 different actions a unit can perform in μRTS, but each individual can perform only between 5 and 33 unit-actions. We can see that the models predict the behavior of the four hard-coded bots better than the Monte Carlo search bots. But as the computing budget increases, the prediction performance also improves for the Monte Carlo search bots. This indicates that the Monte Carlo search bots converge to more stable strategies as the computing budget increases.

We also report the expected log-likelihood of the actions in the dataset given the model. This is a better metric to consider than classification accuracy given that we want to estimate the probability distribution of the actions, and not just predicting the most likely action. The best possible log-likelihood would be 0. We can observe that for Bayesian models, the more we relax the independence assumption, the better the accuracy and the log-likelihood. However, for decision trees, bagging and random forest outperformed J48 in accuracy but not in the log-likelihood.

Table 2 report the speed in classification time, where k is the number of classes, n is the number of features, and m is the number of iterations and trees built for bagging and random forest respectively. The fastest models are Naïve Bayes and J48. The classification speed is very important in time-constrained MCTS experiments as we will show later.

[2]The open sourced implementation of C4.5 in Weka is J48.

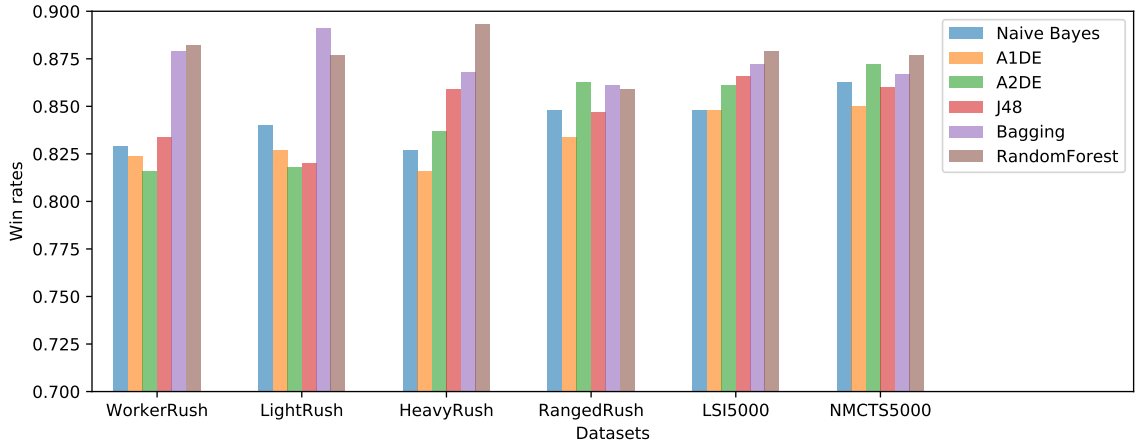

Figure 2: Win Rates by Dataset and Model under iteration budget.

Win rates under Iteration Budget We now evaluate our models as the prior distribution for the tree policy. We use the trained models in the tree policy as the sampling prior for the Naïve Sampling process and we used RndBiased bot as the default policy. In this experiment, the NaïveMCTS bots coupled with models trained from different datasets are tested against the eight baseline bots that were used for training trace generation. The individual results for each dataset are reported in Tables 3-8. Figure 2 shows an aggregated bar chart view of the average win rates grouped by training set and model. The overall average performance of every model is reported in Table 9.

The best model overall is random forest with 0.865 win rate against the baselines, and being the best performing model in three out of six experiments trained separately for each dataset. A general observation is that decision tree models especially the tree ensembles have better gameplay performance than Bayesian models when the training dataset is one of the rush bots. Meanwhile the Bayesian models outperformed the decision tree models in the more complex datasets, I^{5000}_{LSI} and I^{5000}_{NMCTS}. This seems to be because decision trees are better at predicting the deterministic behavior of the rush bots, but struggle to estimate the probability distribution of moves of the more stochastic search bots.

Win rates under Time Budget Due to the nature of RTS games, the computational budget is normally constrained by time. In that sense, complex models that are working well under iteration budget might lose their edges because they run less iterations than simpler but faster models. Thus, we run the gameplay strength experiments against seven[3] of the baseline bots again with computational budget being 50 milliseconds per cycle. The results are reported in Table 10.

We can observe that the model performance is significantly affected by the time budget. Although the more sophisticated models like A1DE, bagging, and random forest have a better performance in experiments with iteration budget, they do not have the same performance in experiments with time budget. The reason is that those models are significantly slower than Naïve Bayes and C4.5, according to Table 2. The C4.5 model stands out as the best performing model (0.853 overall win rate) under experiments with time constraints due to its speed and better class probability estimation performance comparing to Naïve Bayes model.

[3]The current version of LSI does not support time budgets.

Table 9: Overall Win rates per Model under Iteration Budget

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Overall	0.844	0.846	0.850	0.852	0.862	**0.865**

Table 10: Overall Win rates per Model under Time Budget

Tree Policy	NB	A1DE	A2DE	J48	Bagging	RF
Random	1.000	1.000	1.000	1.000	1.000	1.000
RndBiased	0.992	0.987	0.982	0.991	0.991	0.980
WorkerR	0.522	0.514	0.512	0.551	0.514	0.495
LightR	0.634	0.648	0.608	0.703	0.665	0.624
HeavyR	0.991	0.993	0.995	0.997	0.996	0.987
RangedR	0.993	0.997	0.992	0.996	0.992	0.988
NMCTS	0.708	0.688	0.634	0.736	0.594	0.579
Average	0.834	0.833	0.818	**0.853**	0.822	0.808

Conclusions and Future Work

The long term goal of this paper is to study the dynamics of the integration of machine learning models and MCTS methods and factors that affect the performance of their interaction in the context of RTS games. Specifically, this paper presents a comparison study on two types of machine learning models, Bayesian models and decision tree models, integrated in MCTS as part of the tree policy. We compare the models in terms of classification performance, classification speed, and gameplay strength as part of the tree policy under iteration and time budget.

Our empirical results showed that both classification accuracy and speed play a significant role when integrating

machine learning models into MCTS. Specifically, a random forest model has the best classification performance and best gameplay performance as tree policy when we models are compared under iteration constraints. However, in experiments under time constraint, random forest model has much worse performance compared to faster models. Instead, the C4.5 model, which has logarithmic complexity in classification time, has the best performance.

There are a number of future work directions we intend to pursue. For example, the models in this paper ignore the interdependence of actions (when selecting an action for a unit, knowing which action was selected for another unit on the same game state might be relevant). Training a model that takes action interdependence into account should potentially result in better performing agents. Another possible improvement can come from models learned online using techniques like contextual bandit algorithms (Chu et al. 2011; Mandai and Kaneko 2016).

References

[Andersen, Goodwin, and Granmo 2018] Andersen, P.-A.; Goodwin, M.; and Granmo, O.-C. 2018. Deep rts: A game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

[Breiman 1996] Breiman, L. 1996. Bagging predictors. *Machine learning* 24(2):123–140.

[Browne et al. 2012] Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

[Chu et al. 2011] Chu, W.; Li, L.; Reyzin, L.; and Schapire, R. 2011. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 208–214.

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

[Coulom 2006] Coulom, R. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, 72–83. Springer.

[Kocsis and Szepesvári 2006] Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

[Liaw, Wiener, and others 2002] Liaw, A.; Wiener, M.; et al. 2002. Classification and regression by randomforest. *R news* 2(3):18–22.

[Mandai and Kaneko 2016] Mandai, Y., and Kaneko, T. 2016. Linucb applied to monte carlo tree search. *Theoretical Computer Science* 644:114–126.

[Moraes and Lelis 2017] Moraes, R. O., and Lelis, L. H. 2017. Asymmetric action abstractions for multi-unit control in adversarial real-time games. *arXiv preprint arXiv:1711.08101*.

[Ontañón et al. 2013] 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. *IEEE Transactions on Computational Intelligence and AI in games* 5(4):293–311.

[Ontañón 2013] Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[Ontañón 2016] Ontañón, S. 2016. Informed monte carlo tree search for real-time strategy games. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, 1–8. IEEE.

[Ontañón 2017] Ontañón, S. 2017. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research* 58:665–702.

[Quinlan 2014] Quinlan, J. R. 2014. *C4. 5: programs for machine learning*. Elsevier.

[Rosin 2011] Rosin, C. D. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61(3):203–230.

[Sahami 1996] Sahami, M. 1996. Learning limited dependence bayesian classifiers. In *KDD*, volume 96, 335–338.

[Shleyfman, Komenda, and Domshlak 2014] Shleyfman, A.; Komenda, A.; and Domshlak, C. 2014. On combinatorial actions and cmabs with linear side information. In *ECAI*, 825–830.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484–489.

[Silver, Sutton, and Müller 2012] Silver, D.; Sutton, R. S.; and Müller, M. 2012. Temporal-difference search in computer go. *Machine learning* 87(2):183–219.

[Synnaeve et al. 2016] Synnaeve, G.; Nardelli, N.; Auvolat, A.; Chintala, S.; Lacroix, T.; Lin, Z.; Richoux, F.; and Usunier, N. 2016. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

[Tian et al. 2017] Tian, Y.; Gong, Q.; Shang, W.; Wu, Y.; and Zitnick, C. L. 2017. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)*.

[Vinyals et al. 2017] Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

[Webb, Boughton, and Wang 2005] Webb, G. I.; Boughton, J. R.; and Wang, Z. 2005. Not so naive bayes: aggregating one-dependence estimators. *Machine learning* 58(1):5–24.