CS 229 Project: Exploring the Efficacy of Machine Learning in General Game Playing

Aaron Brackett, Antonio Tan-Torres, and Martin Kalev

Introduction

General Game Playing is a relatively new field in which people try to make game players capable of playing a game well without knowing the rules until runtime. With game-specific players such as AlphaGo and other AI's, it's pretty easy to see the benefit Machine Learning can have—these programs can have data from millions of matches of their specific game, if not more, and be able to choose optimally based on this and heuristics specific to their specific game. With General Game Playing, things are not that simple. Because you do not know the rules of the game until runtime, you do not have a sufficiently large dataset for some of the more advanced machine learning techniques. In addition, you lack the insight on the game that you would have if you were playing a specific game, and thus also lack the specific heuristics and specific features to learn. But that's not to say that it's impossible, just not as clear cut as it is in other situations. The goal of this project is to explore different methods of machine learning in General Game Playing, the best ways to implement them, and how effective they are.

Choosing a Move: A Heuristic Approach

The best method for General Game Playing is by exploring the tree using Monte Carlo Tree Search, a method in which you send out probes to find terminal states of the game, and decide which move is best based on the average score returned to you by these probes. Even in light of our research this is still the best method. This method is so good because it generates its own heuristic based on terminal states, which are specific to each game, rather than the general heuristics used in fixed-depth searches. But, there could be cases in which even MCTS is not fast enough to explore the tree for long enough that it is a reliable way in choosing a good move. Or there could be cases in which you do not know what moves will be available in the next state (or 10 or 20 states down the line) and that you cannot just play a game and must rely on a heuristic.

Typical heuristics include maximizing or minimizing your mobility, maximizing or minimizing opponent's mobility, and simply using the goal value of the state as a heuristic. All of these have their flaws. If making the right decision early leads you on a path where the only choices possible terminate in a winning state, the maximizing your mobility will miss it every time. You can think of similar scenarios for the other states. The benefit machine learning will bring is that it will come up with a useful heuristic specific to its game. Unlike goal proximity, which could return a score of zero even if it is one proposition away from winning, our machine learning heuristic will be able to properly evaluate how close a state is winning, based on the propositions present and the data it's learned.

Input Features

In order to perform any machine learning, we first needed to come up with meaningful features and a good way to represent them. Our first thought for features was to have a vector the size of all of the base propositions that made up the game, and to have the values of the indices of all of the propositions that were true when the game terminated to be a one if

they were true, and a zero (or negative one) otherwise. Classifying each vector proved to be slightly less straightforward than we thought. While in a two player game, it’s pretty easy to determine whether or not you won. If your score was higher (or equal) to the other player, then that counts as a win. This philosophy can be expanded to three or more players as well. But in a single player game, it's not always true when you've "won." An easy, but not necessarily complete fix is to say that you've won if you've scored the maximum number of points possible. But this does not account for a game like "Hunter," where you are awarded points for capturing pieces but can never capture all of the pieces, and thus can never achieve the maximum number of points. So we decided to settle on counting a "win" in single player as having 70% or more of the maximum number of points. This means that it is possible for the player to trend towards maybe getting the second most amount of points possible, but we thought this was better than the alternative, as there is never anyway to know if the maximum amount of points possible is the same as the maximum amount of points allowed.

Implementation

To obtain our data, we run a bunch of simulated games during the start clock. In each terminal state, we check to see which propositions are true, and match up the propositions to the correct index by using a sorted list of all of the propositions present for the game. If a proposition is true, we set the value at that index to 1. If it is false, we set it to zero. We then have a separate array of classification for each data point.

For computing a heuristic for a given state using Naive Bayes, we found all of the propositions that would be true if we chose a certain move. We then used the Naive Bayes formula to find the probability that each move was a winning state, and returned that as our heuristic.

For SVM, most of the work was done in the start clock, where we kernelized the data and ran the training algorithm, using Gaussian kernels and the same techniques used in Problem Set #2. When it came to choose a move, we ran the propositions that would result from a move through our SVM. Whereas normally it doesn’t matter how positive or negative the output of the SVM is, here we treated more positive values (and less negative values, in dire situations) as better moves.

Naive Bayes

The first algorithm we implemented was Naive Bayes. With this method, no extra work needs to be done in the start clock, which is a huge plus. However, the more data you have, the longer it takes you to process it. The same is true for propositions. Games with a low number of propositions (like “Buttons and Lights”) can typically be run very fast, causing the dataset to be rather large, but still kind of quick to go through, because there are so few propositions. But there is not always an inverse relationship between the number of propositions and number of games we are able to simulate. The game “Rainbow”, for example, will end in the same number of moves as buttons and lights, but has around 20x more propositions. In games like this, it is super easy for the player to timeout on the start clock typically given to games of this size. This is the biggest flaw with Naive Bayes: the more data you get, the harder it is to process in a general game playing setting.

Support Vector Machines

SVM's had the opposite problem of Naive Bayes. SVM took almost no time in the play clock, which is probably the more important clock when it comes to General Game Playing. Still, without knowing anything about the game, including the start clock, it is hard to know when to stop collecting data and start kernelizing it. Given more time, you could probably work out some sort of algorithm using the number of propositions that typically worked, but you also have to deal with the other processes going on during the start clock, such as building the state machine, which is essential in getting all of the propositions, so the problem cannot be solved with threading.

In-Depth Comparison

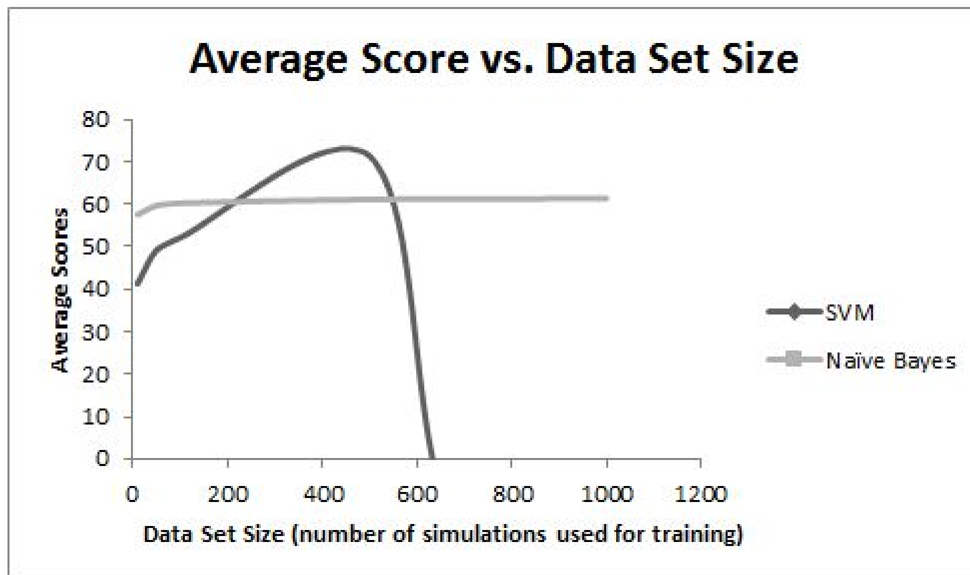
In order to truly compare the two, we chose two single player games, "Hunter" and "Hamilton," that we thought best tested the strengths of our players. These games contained manageable numbers of propositions, and could be simulated enough time to get some good data. They also had scores that were functions of how well you did, rather than the 100 or 0 scoring present in a lot of the single player puzzle games.

Hunter is a single player game played on a 5x3 board. The game begins with a single knight in the upper left corner of the board and pawns on all other squares. On each step, the knight moves like a knight in Chess, capturing pawns on any squares to which it moves. The goal of the game is to capture as many pawns as possible in 14 moves. Note that it is impossible to capture all pawns in 14 moves, and thus, it is impossible to reach 100 points.

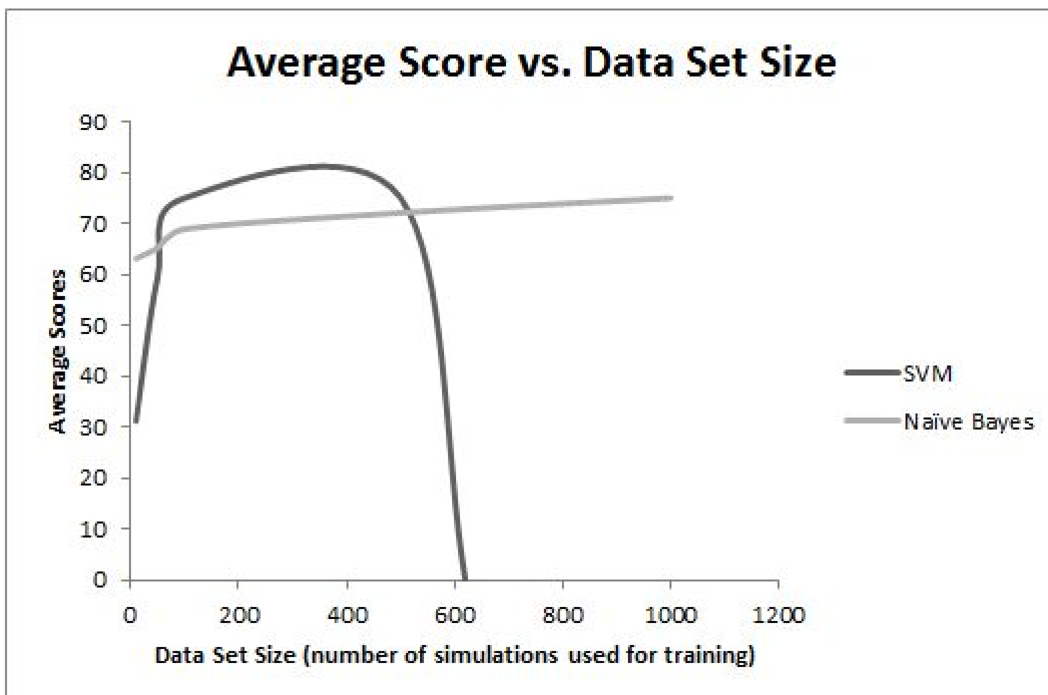
Hamilton is a single player game played on a graph with two interconnected nodes. The player starts out on one of the twenty nodes, and on each step moves to an adjacent node. The objective of the game is to visit as many nodes as possible in twenty steps. The reward at the end of the game is an increasing function of the number of nodes visited.

For each trial, we capped the number of matches it could simulate in the metagame phase. We then averaged all 10 trials to get that player's score for that number of data points. It is important to note that the start clocks and play clocks were consistent across all players, but were guaranteed to be sufficient for the players to perform at their best ability (i.e. no start clock timeout for SVM, no play clock timeout for NB).

Hunter



Hamilton



Both games show similar trends. Naive Bayes is pretty good right out of the bat, showing no improvement in "Hunter" and minimal improvement in "Hamilton." SVM starts off slow, but ramps up to surpass NB. With SVM, there is a pretty direct correlation between number of games and how well it does, so it has a lot more potential than NB (assuming the

start clock is sufficient). The main problem we ran into, as well as what causes the obvious oddity on the graphs, is that Java would run out of memory when trying to kernelize the larger data sets. An unfortunate flaw, but a fatal one.

Conclusion

Overall, I think that given certain settings, our machine learning algorithms can be successful. However, at some point it stops feeling like General Game Playing and more of trying to fit a round peg into a small hole. The results produced were good, but a lot of adjustments were made in order to preserve the spirit of testing over the spirit of GGP. For example, we adjusted the start and play clocks (but kept them consistent) so that we could get some interesting data out of it, without our players just timing out. Interestingly enough, procuring the data took almost no time, it was the act of kernelizing and performing Naive Bayes which took the most time. Given the typical clocks of GGP (20/15, we were using 120/45), the machine learning parts seem unfeasible, but more optimized code could make it more feasible, especially in competitions built to accommodate machine learning algorithms. But right now, I think machine learning is best saved for specific problems and games.

However, there is an upside of this that I think could be really helpful. Our algorithms still proved to be useful on *unknown* games. Given a new game and sufficient time, our players, especially an SVM uninhibited by Java, could prove to be quite fearsome. The makers of GGP are thinking about having a special tournament where you get a 10 minute start clock, and then play a bunch of that game in a row, and I think machine learning would thrive in this environment.

There can still be massive improvements made. In our proposal we discussed two suggestions provided by Michael Genesereth and Bertrand Decoster. Prof. Genesereth suggested that instead of strictly using propositions, we might try some of the heuristics we looked at earlier in class (mobility, opponent mobility, etc.) and use machine learning to come up with the best weights for each heuristic. We then also talked and considered implementing a boosting algorithm using these heuristics as weak learners (although we need to flesh out more details here). Bertrand suggested using combinations of propositions as features, so that instead of just having singular propositions, we can logically combine and negate propositions to see if these complex features are more useful than our simple ones. Unfortunately, coming up with a way to get meaningful data took way longer than any of us had expected, as we had to try out a variety of combinations of games, start clocks, and play clocks, and single games could take upwards of 10 minutes to play out. But I think implementing their suggestions, while probably not making the players any faster, would make them better at accurately playing unknown games, given a sufficient amount of games.