Applying Machine Learning to the Board Game Pylos

Stanislav Fort[1], Lucia Gan[2], and Allen Zhao[3]

Abstract— **We are exploring the application of machine learning and AI methods to the board game Pylos. Pylos is a deterministic, zero-sum, strategy game for two players. To increase the chances of a win, the next move chosen by a player can be optimized by a machine learning algorithm. We represent the available moves by a decision tree where each move is represented by a score extracted from the game board. We pick the optimal next move by first generating data via an optimally ordered alpha-beta pruned minimax decision rule, then implementing a support vector regression to optimize the score of potential moves. The experimental results show that the ordered alpha-beta pruned minimax strategy with scores determined by an SVM is the most successful game play technique. After running the algorithm for 100 trials, the SVM agent won the majority of games played against the agent employing a hard-coded evaluation function, which was the previous best AI that we created.**

I. Introduction

Pylos is a two-player strategy game played on a square board with 16 indentations laid out on a 4x4 base, seen in Fig. 1. Each player starts with fifteen balls, one player with light balls and the other player with dark. Players take turns placing balls onto the game board. If four balls of any color are placed on the game board in a 2x2 square formation, a ball can be placed on top of them forming a second level of play with 3x3 available positions to place a ball. As the game continues, players may position balls on this second level and once a square is formed, a ball can be played on the third level with 2x2 positions. Once the third level has been filled, a final ball can be placed on the fourth level. The player that places the final ball wins the game.

If a move results in a 2x2 square of any color, the player can remove one of their balls from anywhere on the board that is not supporting another level and place it on top of the square, saving the balls in the player's reserve for future moves. This locks all the underlying balls in place. Additionally, if a player forms a straight or diagonal line out of their own color balls, they may return either one or two of their balls that are not supporting another level to their reserve. These mechanics introduce much of the complexity in the game. Otherwise, the second player to place a ball would win every game. This also introduces the possibility of loops that can result in the game going on indefinitely, making a general game AI difficult to derive.

[1]sfort1@stanford.edu
[2]luciagan@stanford.edu
[3]arzhao@stanford.edu

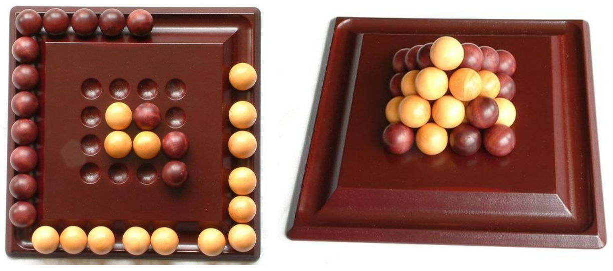
Fig. 1. Pylos is played on a 4x4 indented game board. Players may position balls on multiple levels until a final ball is placed on the top.

II. Related Work

We chose Pylos for this machine learning application because it is deterministic and previously unsolved. Autonomous play of Pylos has been explored by only one previous publication. However, they solved the game by generating a 30 GB database with all possible positions from which a win can be forced [1]. This obviously defeats any human player. We are planning to implement our solution with more understanding of the game as opposed to the brute force method of a full game tree analysis.

III. Methodology

Pylos is a competitive zero-sum game, meaning that one player's win results in their opponent's loss. How each player acts in the game depends on how they think the other player will respond, which makes the task of programming the game play difficult. Choosing the player's next move is a decision that can be optimized to produce a winning solution. In this problem, a player picks their next move by generating a list of all legal next moves and computing a score for each potential move. The player picks the move based on a variant of the minimax decision rule. The input to our algorithm is a board configuration. We then use an SVM regression to output the best next move.

A. Board Representation

The game board is represented by a vector that reflects the current board configuration. A 0 indicates a vacant spot, a 1 indicates the position where player 1 places a ball, and a 2 indicates the position where player 2 places a ball.

[0,
0, 0, 0, 0,
1, 0, 0, 1, 0, 0, 2, 0, 0,
2, 1, 2, 2, 2, 1, 2, 2, 2, 2, 1, 0, 1, 1, 1, 0]

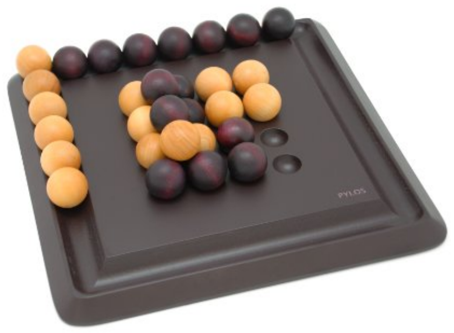

Fig. 2. Vector (top) that represents a corresponding board configuration (bottom).

This 1x30 integer vector is transformed into 1x60 Boolean vector where the first 30 values represent the positions occupied by player 1's balls and the last 30 represent player 2's balls. This vector was translated into a 64-bit integer hash function. Each game state is represented by an integer, which improved performance ten-fold compared to the vector of integers.

B. Decision Tree

From any given state, there are a finite number of moves for a player to consider. We chose to represent these moves in a decision tree and use an algorithm to pick the best next move in a game. The tree contains all possible moves from each position; the nodes represent positions in a game and the edges represent moves, seen in Fig. 3.

Pylos is too complex a game to enumerate an entire tree efficiently. Since players can remove their own pieces from the board, the game may not terminate after a finite number of steps and begin looping, resulting in a draw. Therefore, the decision tree for this game can be essentially infinite given these loops. At a single node, the branching factor can be over 100, which means that the player has over 100 legal moves to choose from at each turn. To put this in perspective, the average branching factor is about 35 in a game of chess and 250 for the game Go [2]. In order to determine the best move out of all the potential moves available to a player, we need to consider an algorithm that assigns node scores based on the current state of the board.

For the AI to practically play the game, we must limit the depth of the search down the tree to only a few moves ahead in the game, since it is too computationally expensive to expand all the way to the end of each game to determine whether a move results in a win. We settled on a layer-based scoring approach, whereby the AI makes a heuristic estimate for the score of each possible approach for the next step and follows each one until the game ends.

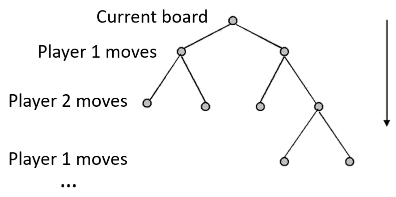

Fig. 3. Decision tree contains all possible moves from any board configuration. Each layer alternates between players.

C. Score Evaluation

We created four different, hard-coded player agents to simulate game play using various strategies, described below. The hard-coded player simulations all use a version of the decision tree search and a static evaluation function, which takes in a game state and computes a score correlating to the likelihood of a win. However, the agents differ in how they treat and combine these scores into more meaningful metrics. The score is extracted from the current state of the board. We experimented with evaluators that gave more points for balls placed at higher levels and balls placed in more strategic positions on the board. However, as is quite typical in other similar games such as chess or checkers, it was more useful to use the simple maximizing heuristic:

# opponent's balls - # player's balls op het bord

and get better performance by a deeper search. The evaluator wants to maximize the number of opponent's balls on the board while minimizing the number of the player's own balls on the board.

D. Player Simulations

1) Random Agent: The easiest strategy to implement is to simply pick a legal move at random. We benchmarked the performance of the rest of the agents against this random algorithm.

2) Maximizing Agent: A slightly more sophisticated strategy looks at every available move and evaluates them. Once we get a list of legal moves, we choose the move that results in the best board configuration based on the static evaluation function that computes the difference between the number of balls each player has on the board. The Maximizing Agent returns the move that results in the best immediate score. This agent plays similarly to a beginner who is still getting familiar with the rules.

3) Minimax Agent: The strategy of the Maximizing Agent is short-sighted. Ideally, the AI would be able to look ahead an infinite number of turns when determining a move, but we are limited by computing power. The minimax algorithm improves on the maximizing strategy by considering the implications of a move several turns in advance. The objective of minimax is to minimize the threat posed by the other player on their next move.

Minimaxation is a very common AI approach. The Minimax Agent searches to a specified depth (looks at a specified number of moves ahead) to find the best immediate move for a player. This algorithm assumes the opponent will always play their best possible move. Nodes are assigned values via the static evaluation function that estimate the degree of belief that a move will result in a win. Between every move, a list of legal moves are compiled and scored using the static evaluator. If it is player 1's turn (max node), the AI returns the maximum score from the list to maximize the probability of player 1's win and if it is player 2's turn (min node), the AI returns the minimum score from the list to maximize the probability of player 1's loss.

4) Ordered Pruned Minimax Agent: The minimax algorithm is effective but impractical in practice. In an actual game, the algorithm cannot search deeper than two turns/layers ahead using minimax without incurring significant delays in game play. This is because it evaluates many subtrees that can be ignored. To optimize the look ahead search, we used alpha-beta pruning, which causes the algorithm to quit evaluating branches that result in better opponent scores since those branches have no effect on the final outcome. This brings down the run time significantly.

Alpha-beta pruning is an algorithm that keeps track of two values, alpha and beta, which represent the maximum score achievable by any moves we have encountered and the score that the opponent can keep us under by playing other moves, respectively. During the game, if we find a move that causes alpha to be greater than beta, then we can quit searching this subtree since the opponent can prevent us from playing it. The performance depends on how well we can estimate the scores for the prospective branches and the order in which we encounter them.

The effectiveness of alpha-beta pruning depends on the order in which the children nodes are visited. For a max node, no pruning will occur if we encounter the worst child first, and for a min node, no pruning will occur if we encounter the best child first. We want to visit children nodes in the best order. To order the children, we rank them based on previous minimax evaluations of nodes from previous explorations. If the child is selected optimally, alpha-beta pruning only explores that one child and prunes away the rest of the children. This means that for the same computation time, the tree can be explored more deeply than before, which improves the AI's performance.

The Ordered Pruned Minimax Agent is the most advanced hard-coded agent and exploits knowledge from previous searches to decide the sequence of children nodes to explore, gradually searching deeper in the game tree. Furthermore, it is time-limited so it parses through layers and updates scores until stopped; there is no set depth of search. The minimax scores are stored in a dictionary and used in future moves.

E. Support Vector Machine

The AI's performance is based upon its ability to develop node scores that accurately represent the correlation of game states with a successful result. Once we acquired results from playing the four hard-coded agents against one another, we processed the data using a Support Vector Machine. The Gaussian kernel function we used transforms the data into a higher dimensional feature space, making it possible to perform the linear separation.

We let two Ordered Pruned Minmax Agents play each other for 7 hours, limiting each move to 0.5 s, and kept a record of all relevant elements (board configuration, search depth, node score) encountered during game play. This generated over 3,000,000 data points. However, we were only interested in evaluations that were generated using information greater than a certain depth, as those scores contain information about possible future evolutions correlated to the current board. We selected 3,000 examples out of the original 3,000,000 as our training data; their score was calculated using information about children looking more than 3 turns ahead.

We used the Python scikit-learn library to train an SVM regression with a Gaussian kernel on the 3,000 data points. We experimented with different kernels, but Gaussian worked the best due to its nonlinearity. The advantage of SVM is that it does not try to fit as closely as possible to the hyperplane and instead relaxes at a certain distance (which we set to 0.2) from the desired point. Since the score function is integer-valued, it does not need to have a more precise fit.

IV. Results

A. Agent Play Results

We quantified the performance of our AI by simulating games between various agents. We allowed each agent to play every other agent 100 times, generating the data below. We limited the time for Agent 4 to parse through layers to 0.05 s. This data showed that Agent 4 was the most effective, so we used it to generate the SVM model. For the table below, we show data corresponding to the wins, draws, and losses of an agent from the header row column against a player in the left column. For example, the second set of data (24, 68, 8) represents the fact that Agent 2 beat Agent 1 in 24 games, drew in 68 games, and lost 8 games - all out of 100.

TABLE I
AGENT VS AGENT WINS, DRAWS, LOSSES

	Agent 1	Agent 2	Agent 3	Agent 4
Agent 1	44, 7, 49	24, 68, 8	98, 0, 2	51, 49, 0
Agent 2	-	0, 100, 0	53, 44, 3	37, 63, 0
Agent 3	-	-	47, 0, 53	58, 4, 38
Agent 4	-	-	-	41, 43, 16

As expected, all other agents won the majority of games played against Agent 1 (Random Agent). Agent 2 won 24% of games against Agent 1 while Agent 3 won 98% of games against Agent 1. Agent 4 won only 51% of games against Agent 1 but lost 0 games.

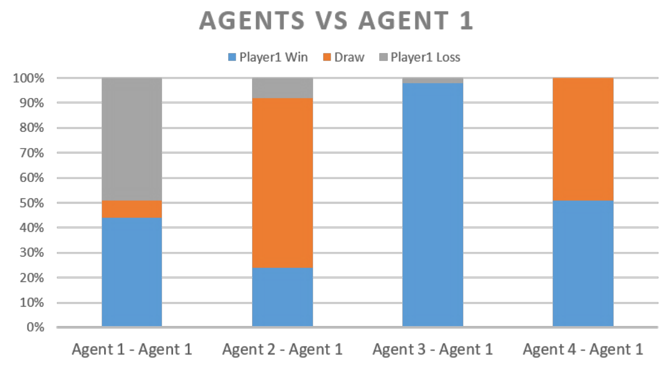

Fig. 4. Results of 100 games played by each agent against Agent 1.

Agent 3 won 53% of games against Agent 2; meanwhile, Agent 4 won 37% of games against Agent 2 but lost 0 games.

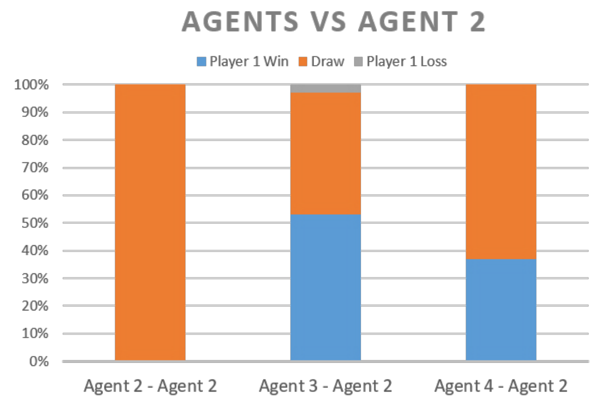

Fig. 5. Results of 100 games played by each agent against Agent 2.

Agent 4 won 58% of games against Agent 3, which had the best previous track record. Thus, we deemed Agent 4 to be the most advanced player simulation.

B. Support Vector Machine

After playing two of the Agent 4 simulations against each other, we processed the data using a Support Vector Machine regression. We culled 3,000 data points generated from 3,000,000 games, aggregating the agents scores and end results (win/loss). Specifically, we set the score x for each child node to be:

$$x = y * 10.0 + z$$

where y represents the outcome of a game and takes on values [1, 0, -1] (win, draw, or loss, respectively). 'z' is the estimated value for that node, found during the

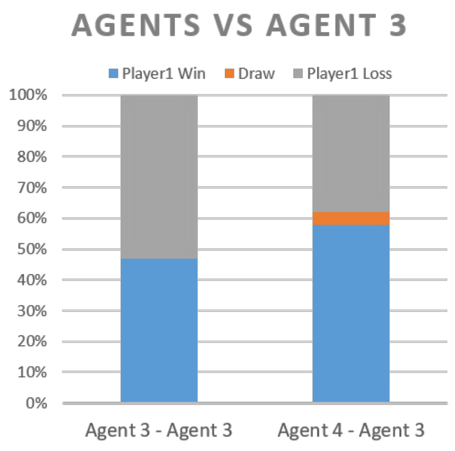

Fig. 6. Results of 100 games played by each agent against Agent 3.

Agent vs Agent phase. We use a Gaussian Radial Basis Function kernel (due to its versaitility) to train an SVM regression with a validation set of 100 examples. Our resulting cross validation thus only has one fold, as that provided sufficient results. The specific parameters that we used for the SVR function in scikit-learn were: C = 1.0, epsilon = 0.2, and kernel = rbf. 'C' is the penalty term for the error (kept at default value for simplicity), 'epsilon' is the relaxation distance from a desired point (set to 0.2 for limited relaxation), and the kernel chosen is 'rbf' for the Gaussian Radial Basis Function. The scores are refined through self-play and through this process, we converged on an optimal set of scores.

We saw from the SVM scores that the game preferred moving to certain board configurations, visualized in Fig. 7 & 8 (where the 1x1, 2x2, 3x3, and 4x4 squares represent playable levels on the game board). The algorithm had a preference for placing balls in the middle of the board and forcing its opponent to place balls along the outside edge. The agent wanted to place balls in special configurations like squares and lines that allow the player to return balls to their reserve. We observed higher order correlations between the ball positions and future moves. This shows that the SVM learned something valuable about the connection between the state of the board and how good it is to move to this state in a game.

Our player simulation using the SVM scoring model played against Agent 4, with varied time limits for layer exploration. The two agents played 100 games with different time limits for evaluation, generating the data below. For every time limit, Agent 5 won the game against Agent 4 (our best hard-coded static evaluator agent) more often than it lost. When Agent 5 was tested against human players (e.g. the team), the algorithm won 100% of the time. Thus, we consider our project to create an effective Pylos player (with machine learning) a success.

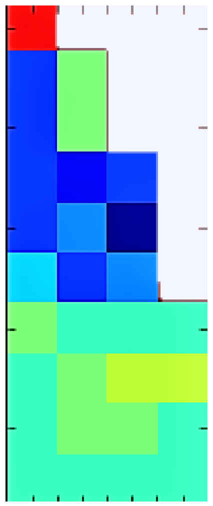

Fig. 7. Visualization of SVM scores with one of the player's balls on the board. Cooler colors represent better scores. The AI wants the player to place balls in better configurations and on upper levels.

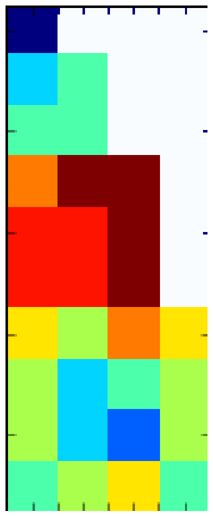

Fig. 8. Visualization of SVM scores with one of the opponent's balls on the board. The AI does not want the opponent to gain an advantage by putting balls on an upper level, so those are given poorer scores.

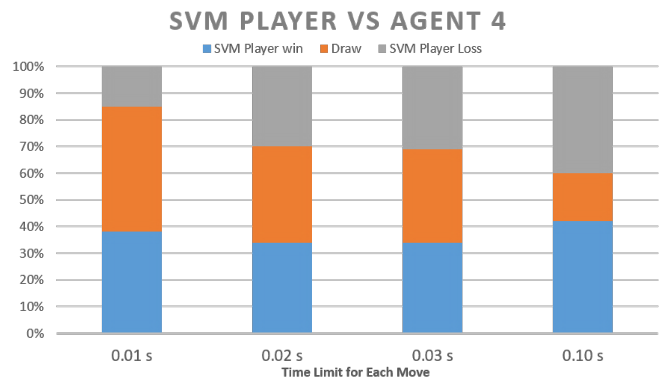

Fig. 9. Results of 100 games played between SVM player and Agent 4.

V. Future Work

The problem with the minimax and alpha-beta pruning algorithms is that they take an impractically long amount of time to search more than a few turns ahead. Additionally, we selected features that we thought could evaluate whether a move is likely to result in a win, based on prior knowledge we had from playing the game. Given that Pylos has a high number of available moves per turn, using these algorithms can result in somewhat slow gameplay. In the future, we could look to a technique that is used for playing complex games (such as Go) known as Monte Carlo Tree Search (MCTS), which could improve the performance of the AI. MCTS is a method in which the algorithm randomly traverses a tree to find the end result of the game and decides which move is best based on which edges were chosen most often. We would like to compare the performance (in terms of speed and win rate) between our SVM agent and a MCTS player. Finally, we would like to implement a GUI for the game so that a human can click on buttons to make a move instead of inputting code manually through terminal.

VI. Conclusions

In the beginnning, we set out to develop an agent that would learn to beat humans at Pylos. After hard-coding four different agents that worked well - but used static evaluation functions - we were able to take our existing data and generate an SVM model. The player that we built out of this model was able to play significantly better than any hard-coded evaluator that we created, making our project goal a success. However, it is not perfect and there is a good amount of work to be done before it can be as powerful in Pylos as Alpha Go is in Go.

References

[1] O. Aichholzer, et al, "Playing Pylos with an autonomous robot," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, 2010, pp. 2507-8.

[2] D. Silver, et al, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484-489, Jan. 2016.