GoLD: Go Life and Death Solver

Joshua Gang, John Blackmore, Zachary Daniels

***Abstract*—We introduce a novel approach to solving life and death problems in Go. We combine a tree search-based agent with a learned heuristic. We train and test on a large, crowd-sourced dataset of life and death problems from goproblems.com. Experiments are run to evaluate the quality of the classification model, features used, and general agent, and we offer an interpretation of these results.**

I. Introduction

A. Artificial Intelligence and Game Playing

Games in general, specifically board games, have been the focus of many efforts in the field of artificial intelligence for many years. Games can be thought of as a means of simplifying many real world problems, making them significantly easier to solve as more and more constraints are imposed on them. Games provide a set of rules and a limited world for them to operate on, which allows us to abstract real world problems even further. They allow for the rapid application of artificial intelligence techniques, which allows many experiments to be run and results in a fast pace for developing new techniques with relative ease. For example, games such as Chess and Go have allowed for many advancements in search.

While Chess captured the public eye for many years, with machines such as IBM's Deep Blue, it is a much simpler problem than Go, and as such, not all of the techniques applied to Chess can be applied to Go. Chess, for instance, is played on a 64 square board, and has a branching factor of about 35, with approximately $1.58 * 10^{55}$ total board states. Go, on the other hand, is played on a 361 square board, and has a branching factor of about 250, for a total of approximately $2.08 * 10^{170}$ board states. As a result, Go requires the use of more complex strategies.

B. The General Game of Go

Go is an ancient board game, originating in China over 2,500 years ago. The game is played on a 19x19 board by two players, one who controls the black stones and one who controls the white stones. Each player, on their turn, places one stone on the board. Once a stone is placed on the board, it stays there until the end of the game unless it is captured by the opponent. When the game is over, the winner is determined by whoever has the most stones on the board and the largest amount of area surrounded. An example board of go is seen in Figure 1.

C. Life and Death Problems in Go

The game of Go has many different strategic parts of it and can be thought of as many different subproblems, all of which need to be solved in order to play the larger game. Life and death is one of such subproblem, whose purpose is to determine whether or not a particular group of stones is "alive", meaning that it won't be removed from the board, or "dead", meaning that it will be removed from the board and count as points for the opponent. These problems include both figuring out how to make groups live by the formation of two or more "eyes", regions of surrounded empty space that due to suicide rules, stones are forbidden to be placed in, or by figuring out how to kill the groups by preventing the formation of two or more eyes.

We decided to focus on life and death because it is simpler compared to the rest of Go. Since we are only focusing on a small subset of the board, our potential search space vastly decreases. We also don't need to worry about how our moves affect the overall status of the board, so we don't have to worry about how the moves that we make here would affect the status of other groups, which allows for simpler evaluation functions since there is much less information we have to work with. All that we have to concern ourselves with is whether or not this group lives or dies.

An example can be seen in Figures 2 and 3. In this board state, black is one move away from causing his groups to live unconditionally. By moving at (0, 0), he captures the stone at (0, 1) and his groups live. No matter what white does, since he cannot move into either (0, 1) or (2, 1) due to the anti-suicide rule (as white's stone would be immediately captured), black's stones will never be removed from the board. Thus, this problem is solved by black moving at (0, 1). If black wasn't to move there, then white would go to (2, 1) and black's right group of stones would be captured, so black would be dead.

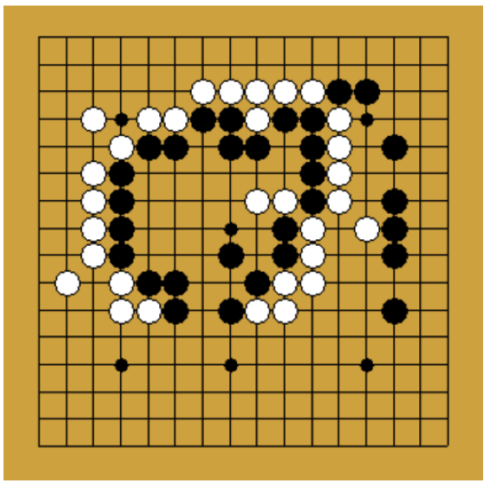

Fig. 1: An example board in Go

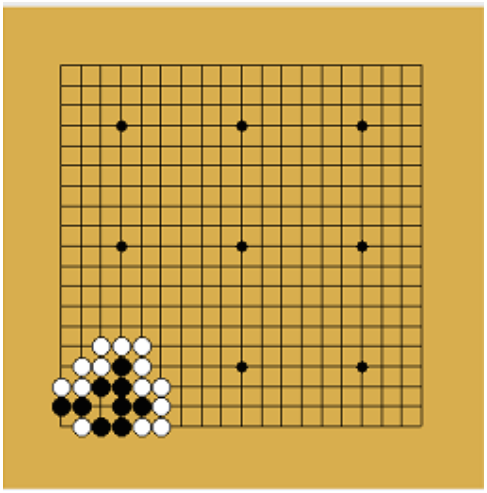

Fig. 2: An example life and death problem

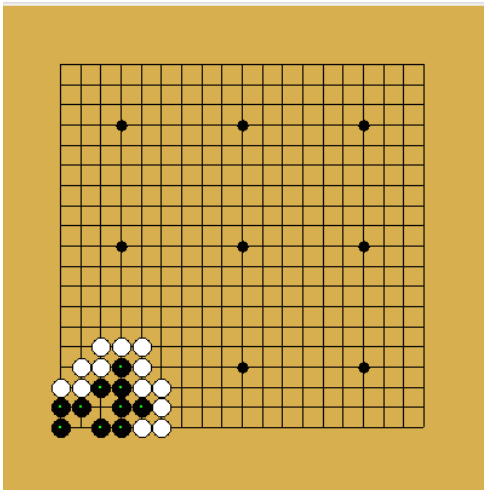

Fig. 3: A solved life and death problem: Green dots denote a live group.

II. Literature Review

Go is a well-studied and central problem in artificial intelligence. As a result, there has been a large amount of work done on constructing intelligent agents for playing the game of Go. These agents implement a wide variety of game-playing strategies. Tree search is perhaps the most classic approach to building game-playing agents, and it has been applied to playing Go [1]. However, the search space of Go is so large that traditional tree search without modification or without supplemental strategies is a poor method for building Go-playing agents.

Another classic approach to building game-playing agents is by encoding expert knowledge and rules into the agent. Fotland encoded expert strategies in his software, "The Many Faces of Go" [2]. However, expert rule-based agents require extensive knowledge about the domain, and as such, are only as good as their knowledge base. Likewise, because knowledge is essentially hard-coded in such systems, these systems are not easily generalizable to other domains.

Others have taken more experimental approaches to creating Go-playing agents. Enzenberger used neural networks with board states as inputs to predict which moves to make [3]. Graepel et al. represented Go boards as graphs and then applied a traditional machine learning approach on features extracted from the graphs [4]. Stern et. al explored probabilistic and Baynesian methods for playing Go and also explored reasoning over graphical models [5] [6].

The most popular modern approaches to building intelligent agents for Go use reinforcement learning combined with Monte Carlo methods. The most successful of these methods involve combining temporal difference learning [7] with Monte Carlo search trees [8] [9]. Using this method, Gelly et al.were able to construct an agent capable of beating professional players on 9x9 boards [8].

There has also been extensive work done on life and death problems in Go. This work is split into two major areas:

1) General algorithms for solving life and death problems
2) Methods for determining the life and death status of groups of stones

Because life and death problems restrict the game of Go, tree search methods have seen more success in life and death problems than the general Go problem [10].

Determining the life and death of groups on a Go board is a difficult problem. In order to determine the life status of a set of stones, one needs to search for specific patterns known as "eyes". For a group to be alive, it must have two eyes. The problem arises from the fact that there is no standard definition of what constitutes an eye [11]. There have been a number of approaches to solving this problem. Benson designed an algorithm for determining the unconditional life of a group of stones [12]. However, life and death games often end before a group reaches unconditional life (or death) because human players are capable of seeing several moves into the future. There have been multiple attempts at constructing eye databases using Monte Carlo methods [13] [14] [15]. Heuristics for determining the life and death of groups have been explored [16] [2], and Chen and Chen have explored using static analysis to determine eyes and select future moves [16].

There have been successes in solving life and death problems in Go. Kishimoto evaluated his own methods and those of GoTools [15], a longstanding, top life and death solver for Go, on a specially crafted dataset consisting of one-eye problems (a simplification of the general, more difficult two-eye problems that we address in our research) on a 9x9 board [10] [1]:

1) GoTools: 80% (some problems removed)
2) Alpha/Beta: 85%
3) TsumeGo Explorer: 96% (some problems removed)
4) DF-PN: 93%
5) DF-PN(R) Limited: 85%
6) DF-PN(R): 97%

III. Project Plan and Design

A. Project Plan

We planned to follow the following sequence of steps:

1) Obtain the dataset
2) Parse and standardize the dataset: convert white-to-live problems to black-to-live and black-to-kill problems to white-to-kill
3) Split the dataset into training, development, and test sets
4) Determine the heuristic evaluation function: we elected to use a learned heuristic
5) Determine the type of learning agent
6) Determine the features to use: should we focus on knowledge-based or general features?
7) Construct a terminal test
8) Determine methods and definitions for evaluating success and failure
9) Build the tree search architecture
10) Build the learning agent
11) Integrate the learning component with the tree search component
12) Evaluate the system

B. Dataset

We obtained our data from goproblems.com [17]. The dataset consists of 5309 life and death problems. The problems vary in difficulty from 30k (beginner) to 6d (upper professional). The games are crowd-sourced, i.e. a person submits a game and a set of solutions. As others play the game, they verify the solutions, add their own, and add interesting (but not completely enumerated) incorrect paths. These games are meant to be teaching tools. The data for each game is represented as a game tree consisting of solution paths and incorrect paths. Because the games are crowd-sourced, there are errors in the data. Likewise, problems are labeled in natural language text, which means error arises in parsing the data. Additionally, not all games are traditional black-to-live and white-to-kill problems; for example, some games have more specific instructions such as "black to live in the corner". Many games lack the problem type label, have an unparsable label, or are ill-formatted. These errors account for the loss of about 3000 problems. After the data is obtained, all problems must be converted to one of two forms: *black-to-live* or *white-to-kill*.

We also split the dataset into three subsets: a training set of 1771 games, a development set of 1769 games, and a test set of 1769 games. We evaluate our models by training on the training set and testing on the development set. After the system is fully built, we train on the combined training and development set and test on the test set.

The size of our data is as follows where (*number*, *percent*) represents (number of correct moves, percent correct moves):

- Total: 166,586 moves (31,533/18.9%)
- Training Set:
  - Black-to-Live: 25,128 moves (4,918/19.6%)
  - White-to-Kill: 32,620 moves (5,298/16.2%)
- Development Set:
  - Black-to-Live: 22,059 moves (4,543/20.6%)
  - White-to-Kill: 31,189 moves (5,557/17.8%)
- Test Set:
  - Black-to-Live: 20,632 moves (4,949/24.0%)
  - White-to-Kill: 34,958 moves (6,268/17.9%)

C. Project Architecture Outline

The GoLD project is organized into the following components, or packages (we denote some example functions and classes):

- **models** - All data classes and functions that operate on them. Board, Problem, determineLife(), SearchTree.
- **features** - Feature class and implementing subclasses, calculate features given a board state and subsequent move. HuMomentsFeature, DistanceFromCenterFeature, FeatureExtractor.
- **learn** - All machine learning model and related classes, model building, analysis, tuning, etc. Model, ModelBuilder
- **ui** - User Interface - Launcher, viewer
- **utils** - Any utility programs. ProblemTester, csv_analyzer.

To execute a problem, the ProblemTester (utils) will load the problem file into a data structure via MoveTreeParser (models) including the starting board state. Then the SearchTree (models) will try to determine the best next move, in turn, until the terminal test is satisfied or too many moves have been taken (one more than twice the moves of the longest path in the problem). When completed, the user has the option of launching the UI to show the result, from start to finish. In addition, the resulting path is written to file, which can be read and displayed by the UI.

There are several benefits to this architecture. Since the UI was not an essential component for much of our work, the logic for maintaining the board had to be kept separate. It was straightforward, however, to process an instance of Board and display it on the screen.

Having all machine learning feature implementations inherit from the Feature class offered a clear layer of abstraction, allowing easy addition and removal of features from a standardized vector.

The user interface is implemented using TKinter, a UI package for Python programming language.

Machine learning components leverage scikit-learn and numpy.

Feature implementations use scikit-learn, scipy, and numpy.

All other code was developed in Python for this project.

The main component interfaces are defined as follows:

- Board.place_stone(x,y,isblack) - Places a stone on the board
- SearchTree.decideNextMove() - Decide the move and return (x,y)
- Model.scale(data), .getScoreCorrect(instance) - Scale features and compute posterior probability
- MoveTreeParser(problemFile) - Parses file into data structure, including starting board state
- Launcher.showPath(boards[]) - Display a series of board states with a slider to move back and forth

- life.determineLife(board,color) - Returns groups that are *unconditionally alive*. For black-to-live problems, our terminal test is if black has any groups that are unconditionally alive.

IV. System Overview

We present a broad, conceptual overview of our system in Figure 4. We describe specific parts of the system in detail in this section.

A. Terminal Test

With regards to groups, there are five different states: Unconditional Life, Conditional Life, Death, Ko, and Seki.

Unconditional Life means that no matter what the other player does, this group of stones cannot be removed from the board. This is can be done simply in $O(n^2)$ time, through Benson's Algorithm [12], although faster, more complicated implementations are probably possible. Benson's Algorithm will tell us whether or not a group is unconditionally alive. Issues with this become finding which regions of the board are properly enclosed by that group as there can be many corner cases as to whether or not the region is properly enclosed that can be difficult to encapsulate. Testing for unconditional life can be useful as a terminal test as it's not reliant on any reading ahead or pattern recognition: either a group is unconditionally alive, or it isn't.

Conditional life means that for all intents and purposes, the group is alive. What this means is that for every potential move that the enemy has, to try and kill the current group, there exists a move that will ensure that the group remains alive. Eventually, all conditional life problems will turn into unconditional life problems. There are two issues with conditional life, however. The first is that, in its simplest implementation, it requires that you read ahead to see if it will become unconditional life, which can take a very long time as you (ideally) have to search through every possible move. If you repeat this as your terminal test over and over again while you're searching, then it can greatly slow down your search. The second issue is that while there are shapes that are known to reduce to unconditional life, encoding what those shapes are in such a way that a computer can understand them is difficult. Many humans use this form of pattern recognition when they determine whether or not a group is alive.

The fact that most people actually end with conditional life was a source of noise in our data as there were many problems that ended in life but not unconditional life.

Death is simply that a group cannot ever form two or more eyes. This is also very difficult to determine algorithmically without doing a full on search such that for all potential moves that the current player could do, there exists a move that the opponent can do such that the group will remain dead. Much like conditional life, there are some recognized patterns that are dead, but you run into the same problems.

Ko is one of the rules of Go that essentially encapsulates a refusal to allow states to repeat themselves. There are some problems that end in Ko, and the fate of these groups ultimate rely on the rest of the board. When presented with these, the typical problem just terminates. We decide to ignore Kos, and as such the problem just continues and doesn't move into the Ko. Whoever first forms the Ko, with our models and search, will effectively "win" the Ko.

Seki is a state of mutual life and death. Whoever makes the first move in a Seki situation "loses", and that player's groups die. There are problems that end in Seki, but Seki is usually seen as a suboptimal solution for the problem. As such, we decide to ignore this as well, as it made things simpler: we just look for life or absence of life.

Our terminal test in the end just ended up being a test for unconditional life, as it was the most time efficient and easiest to encode test for when to end searching. Nothing beneficial occurs after unconditional life, so we can stop at that point without losing anything. It allows for us to not accidentally think that something is alive when in fact there exists a sequence of moves that will lead to death as the path from conditional life to unconditional life is both not only a whole other complete search by itself but can be thought of as another life and death problem. Namely, we have this group, and instead of trying to make it just alive, can we make it unconditionally alive?

To account for most of the problems not ending in unconditional life, we perform a deeper search in the tree to see if it would terminate within an alloted depth. If it doesn't, then we say that we can't solve the problem.

B. Tree Search

To determine the next move for an AI in a game such as Go, it is common practice to construct a search tree, where the root node represents the current game state, and all the child nodes represent the game state after a "move" has been made. Unfortunately, in the game of Go, there are too many possible moves to search very deep, limiting the effectiveness of conventional search strategies.

Since the Go board has 361 spaces, there are as many as 361 possible moves. Even if we assume an average of 100 possible moves, the search space grows to one million at a depth of only 3. The time complexity of search for this problem is defined as follows

$\mathcal{O}(B^d)$

where B is the branching factor, or number of possible moves, and d is the depth we allow the tree to grow for each move to be decided.

The terminal state we're searching for is unconditional life, as described above. If encountered while searching moves for white, those moves leading to unconditional life would be pruned from the tree. If encountered while searching moves for black, that move is returned and the search cut off, since there is no way to improve upon the terminal state for black.

To mitigate the overwhelming cost of search, we implemented the following strategy:

1) Limit number of moves through the use of heuristics.
2) Limit breadth of tree through the use of pattern classification.
3) Limit the depth of the tree as needed.

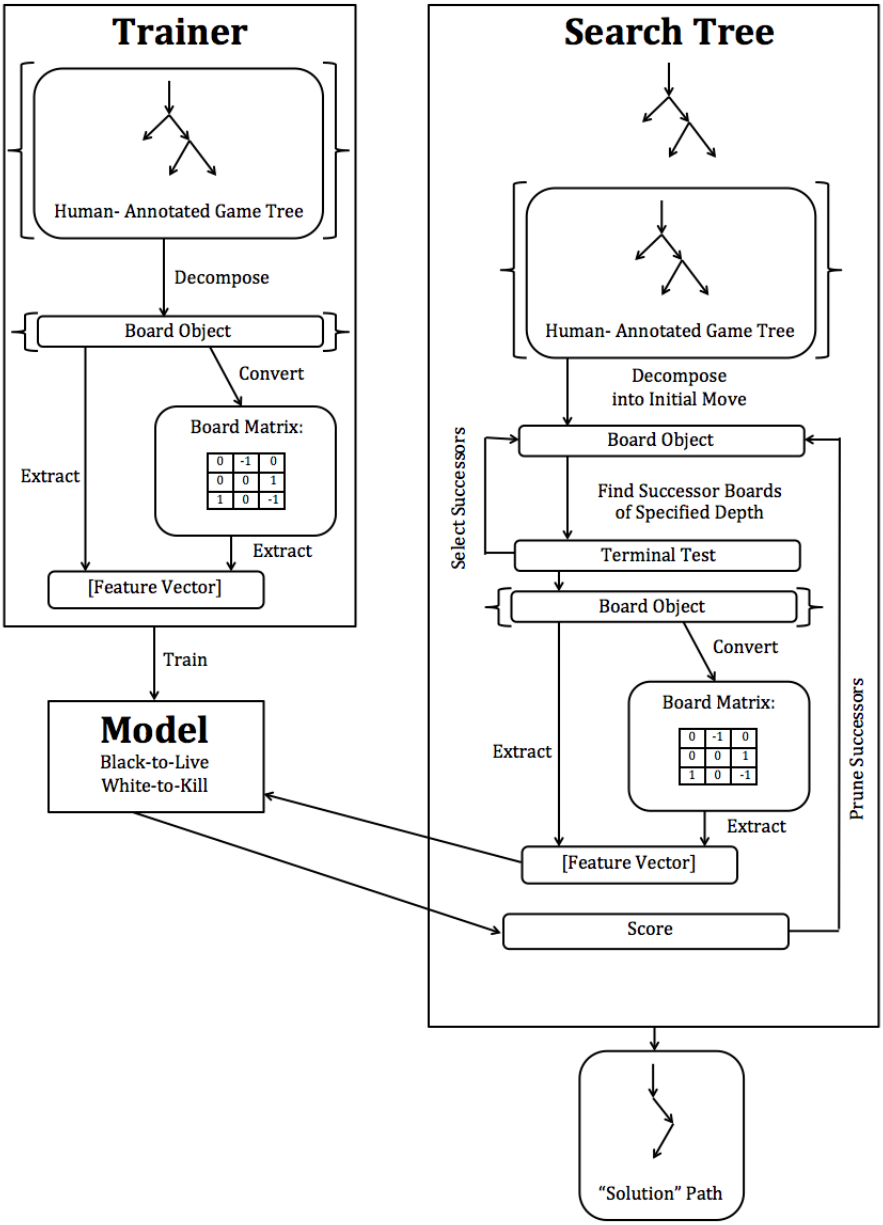

Fig. 4: Conceptual overview of the system architecture

Heuristics: (Goal: Reduce B). Any information we can use to quickly discard, or prune, bad moves or moves with no utility value, from the search tree. For these problems, we limit our search to spaces near existing groups of stones, with the understanding that moves far away from the existing stones have little or no bearing on the life-and-death problem at hand.

Further, we simply count stones for each side before and after the move under consideration. This will almost always result in increasing black stones by one for black's turn, increasing white stones by one for white's turn. Exceptions will only occur when stones can be taken. We assume that taking stones is invariably good and losing stones is invariably bad, and therefore we can restrict our AI to search those moves that maximize the number of stones for itself.

Classification: (Goal: Further reduce B). After heuristics have been applied, in many cases, there are still too many possible moves to search. We can impose a limit to the number of moves that are searched, using an evaluation function to decide which are more likely to provide us with good moves, or moves leading to a terminal state for black, or leading away

from terminal states for white. For the evaluation function, we employ a classifier trained from documented game states from our problem data. The evaluation function, a binary classifier, returns a continuous value between 0 and 1, where values closer to 0 are generally less likely to be good moves than values closer to 1. This classification is costly, however, and consequently we limit this evaluation to the first move considered. For example, if we set a limit of 20 moves to search, and after applying heuristics, there are 30, we evaluate all 30 moves, and search the 20 moves with the highest evaluation. By classifying only the first level, however, we are only able to reduce the cost linearly. If we evaluate 100 moves and choose 50, for example, we may be cutting away roughly half the search tree.

Depth Limit: (Goal: Reduce d) This just imposes a hard limit on how deep to search before stopping. In order to be able to work through problems and have results for this report, a maximum depth of 3 was used.

C. Machine Learning

We define our learning problem to be the following:

> **Given:** a current board-state, *A*, and a direct successor board-state, *B*
> **Predict:** whether or not the move which transitions the state of the board from *A* into *B* is a "good" move, i.e. will it eventually lead to a solution state?
>
> **Mathematically:** we seek the function:
> predict: $(A,B)\rightarrow\{0,1\}$
> where 1 denotes a move along the solution path and 0 denotes otherwise

Note that in our final system, as mentioned in previous sections, we do not directly use the prediction output by our classifier model but instead seek the probability of a move being "good". When obtaining such a probability value is not feasible, an alternative score that expresses similar information is instead used, i.e. distance from the separating hyperplane.

D. Features

In order to build a classifier model, we must first find a way to represent the state of the board and how that state changes when a specific move is made. We express this representation as a set of quantitative, descriptive features. In total, we experimented with 244 individual features; however, these features fall into seven general categories which can further be grouped into three general classes.

Expert Features: Expert features are domain-specific features based on how an experienced Go player would attempt to solve life and death problems.

Stone Count: The stone count feature set contains perhaps the simplest metrics for evaluating the "goodness" of a move. These features all relate to the number of stones a player currently has on the board, how the number of stones of one player compares to the other, and how the number of stones changes after a move is made.

Liberty Count: A liberty is an empty point immediately touching a stone in the non-diagonal directions or an empty point connected to a continuous string of same-colored stones. In order for a group to be alive at any given moment, it must have at least one liberty. The liberty count feature set contain features that measure the number of liberties each player has and how the number of liberties change after a move is made.

Distance from Center: The distance from center feature attempts to find a way to quantize the knowledge that the correct move for a life and death problem is going to take place inside an existing group, and not by trying to capture the stones on the outside of a group.

Number of Live Groups: It's important to be able to describe the general status of groups on the board. One way to characterize such a sentiment is by counting the number of groups that are extremely unlikely or impossible to be captured no matter how many future moves are made. The number of live groups feature set embodies this number and other features related to this number.

General Features: General features are domain-agnostic features. Their inspiration comes from other areas such as image processing.

Hu Moments: Image moments are properties of images that describe the structure of the pixels in the image based on specific, weighted averages of the pixel intensities. Raw moments describe general image properties such as the image's area and centroid. Central moments are built off of raw moments and are notable because they are standardized around the centroid and thus, are translation invariant. Furthermore, scale-invariant moments also exist which are further built on top of central moments. Lastly, Hu moments are a collection of moments built off of scale-invariant moments and are notable because in addition to being translation and scale invariant are also rotation invariant and in one case, skew invariant as well [18]. We treat boards as having three pixel intensities {black, white, empty} and directly compute the eight Hu moments.

Sparse Dictionary Encoding (Sparse Coding): Sparse coding is a technique whereby a collection of vectors X is used to learn a dictionary of vectors D. Each vector in X can then be approximately expressed as a linear combination of the vectors in D, and sparsity constraints are used to keep most of the coefficients in the linear combination as zero. Formally, sparse coding is used to find an over-complete set of basis vectors while enforcing an additional sparsity criterion to prevent degeneracy [19] [20]. Our feature involves splitting boards into smaller sub-patches, learning a dictionary from the training data, and encoding these patches using sparse-coding. The coefficients for all patches in a board are pulled and used as features. This set of features attempts to capture the structure of the board and inherent patterns in the board.

Mixed Features: Mixed features are those features that are grounded in Go players' strategies for addressing the problem of life and death, but take a generalized, learning-based approach to extracting features instead of using hard-coded, specific rules for feature extraction.

Local Shape Presence: Certain shapes in the board are indicative of good and bad states. The local shape presence feature

examines the board for specific stone patterns. These shapes are selected without human input. Every 2x2 and 2x3 shape is generated. However, this set of shapes is too large to be used to extract features from the full dataset due to machine restrictions. Instead, these shapes are convolved in every orientation over boards taken from a subset of the training data, and the presence of every shape is recorded along with the 'goodness' of the board state. A tree-based classifier is trained on this data, and the features with the top-n Gini importance scores are selected as the most discriminative shapes [21]. The presence of these selected shapes and change in presence of these shapes after a move is made are then used as features. Local shape features are common place in reinforcement learning-based Monte-Carlo methods for solving go problems; however, these methods do not discard any shapes, use even larger shape sizes, and add the concept of weights to shape presence [8].

E. Models

We took an experimentalist-view to model selection. We evaluated our data on eight different types of classifiers: *linear discriminant analysis*, *quadratic discriminant analysis*, *logistic regression*, *k-nearest neighbors*, *naïve Bayes*, *linear support vector machines*, *random forests*, and *AdaBoost*.

Training: Our training data consists of individual moves from the training set (see section above describing the dataset) marked as belonging to a solution path or belonging to an incorrect path. We train two models: one for *black-to-live* problems and one for *white-to-kill* problems. We have training data specifically corresponding to each of these problem types and trained on all moves with known labels in the training set.

Issues: As we experimented we came upon a number of issues.

Unbalanced Data: The number of training samples for bad moves outnumber the good moves by a factor of between three and five depending on the subset of data being examined. The result of this is that the classifier is much more biased to say a move is bad than it is to say a move is good, and in some instances, refuses to classify all but the very best of the good moves as being good. We examine three measures to correct this:

1) Some classifiers such as the support vector machine allow us to modify class weights. In this case, we modify the class weights so they are inversely proportional to number of samples for their class, i.e. the more training samples that exist for a specific class, the less weight the class is given.
2) In other cases, we play with the class priors as a means of weighting one class over the other. While this works, it is not necessarily a scientifically-sound method since the data no longer reflects the true state of nature.
3) Lastly, we down-sample the class with the larger sample size so it matches that of the small sample size.

Each of the above methods experimentally seems to help curb the issue of unbalanced data.

Incomparable Distances: A common problem in machine learning is that feature values are incomparable, and thus, they can't be combined in a single model particularly well. We attempt to alleviate this problem by scaling each feature to have zero-mean and unit-standard deviation.

Hyperparameter Tuning: Some models require parameters to be tuned before they produce useful results. For example, one must determine the number of neighbors for k-nearest neighbors or the number of estimators (trees) for random forest. In order to tune such parameters, we apply grid search with cross-validation.

Poor Feature Discriminability: Sometimes features can interfere with one another causing less than optimal models. In other cases, expensive to compute features can be removed while minorly affecting the performance of the classifier. We look at several ways to reduce the number of features to address these two concerns:

1) We manually investigate the discriminability of subsets of features. The results of these investigations are presented and discussed in the following section of this paper.
2) We run the recursive feature elimination algorithm on our features [22].
3) We explore dimensionality reduction using PCA and PCA with whitening.

We find mixed results when eliminating features or performing dimensionality reduction.

V. Results and Discussion

A. Metrics for Feature Analysis and Model Evaluation

For feature analysis and model evaluation, we use five metrics: *precision*, *recall*, *f-measure*, *accuracy*, and *the area under the receiver-operator curve*. We define these metrics mathematically where TP represents the number of true-positives, FP represents number of false-positives, TN represents the number of true-negatives, and FN represents the number of false-negatives:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$fmeasure = 2 * \frac{precision * recall}{precision + recall}$$

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

We briefly describe the significance of each measure:

- Precision measures how often a move we've identified as being good actually is good.
- Recall measures how often we're able to identify a good move as being good.
- F-Measure attempts be a measure that combines precision and recall in such a way that the flaws of these two metrics are addressed.
- Accuracy measures how often we're able to correctly identify good moves as being good and bad moves as being bad.
- Area under the Receiver-Operator Curve (AUROC) measures how often we rank a randomly selected positive

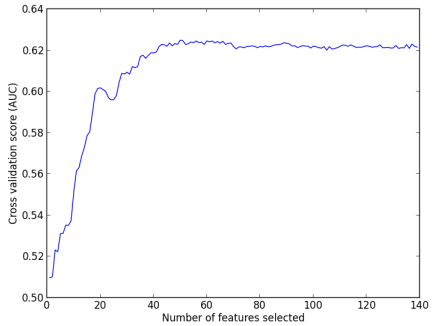

Fig. 5: Typical area under the ROC curve values as recursive feature elimination is applied

example higher than a randomly selected negative example.

Prioritizing accuracy results in a classifier that is overly biased to classifying moves as bad unless the move is very good because of the unbalance between classes in the dataset. When prioritizing precision, we face similar issues, only the really good moves are classified as good, resulting in a classifier that's biased to classifying moves as bad. Prioritizing recall results in the opposite: models that are overly biased to classifying moves as being good. AUROC corresponds to our problem: we're using the probability or score given by the classifier to try and rank the good moves over the bad moves. A high AUROC is an indication that the model is doing what it should do. F-Measure is another good metric because it attempts to overcome the flaws of precision and recall. As such, we focus on achieving high AUROC and high f-measure without sacrificing recall too much.

B. Feature Analysis

One of the most common problems in machine learning is that features interfere with one another instead of complementing one another leading to lower discriminability. A related concern is that sometimes adding features neither improves nor worsens the classification model. We run several experiments to determine how good our selection of features is and if it can be improved.

Our first experiment involves running the recursive feature elimination algorithm (RFE) [22] on our models and seeing how the AUROC is affected. Figure 5 shows the typical behavior of the AUROC as RFE is run with cross-validation on the training set using random forest as the classification model. We find that as features are added, AUROC improves, but there comes a point when adding more features doesn't improve the classifier, i.e. the classifier's performance plateaus using only a minority of the features.

We then aim to identify which of the features contribute most to the performance of the classification models. We explore individual feature groups, expert vs general features, board size-invariant vs board size-dependent features, and the effects of applying RFE. The data for these experiments can be found in Tables I and II. Data is collected and averaged over three trials to reduce the effects of randomness. Random forest with down-sampling and scaling is used as the classification model. The model is trained on the training set and tested on the development set.

Individual Features: We find that for both *black-to-live* and *white-to-kill* models most features at least slightly contribute to the performance of the classifier. Exceptions exist. For *black-to-live* models, the number of live groups features and local shapes features have AUROC very near 0.5. Likewise, they have noticeably low f-measure values. The sparse dictionary feature also has a low f-measure value and border-line AUROC value. When the number of live groups feature and local shapes features are removed from the model, but the other features are used to build the model, we find classifier performance decreases slighty, meaning these two features help with discriminability a tiny bit. Further evidence that local shapes features are not highly discriminating by themselves can be seen by examining how RFE eliminates all but one feature from the original 106 local shape features in both the *black-to-live* and *white-to-kill* models.

We also see that the Hu moments features end up being the best features on both models. We also see that in the *white-to-kill* model, all features are somewhat discriminating.

Expert vs General Features: When comparing models trained on the expert and general features separately, we see that performance is almost identical on the *black-to-live* data and the expert features have the edge on the *white-to-kill* data. In both cases, combining the expert and general features leads to improved classifier performance.

Board Size-Invariant vs Board Size-Dependent Features: When engineering features we incorporated both features invariant to the size of the board (e.g. ratio of black stones to white stones) and those dependent on board size (e.g. total number of black stones). We see that using only the board size-invariant features results in a models that generalizes slightly better and improves the performance of the classifier when the performance is judged on the AUROC and f-measure values.

Recursive Feature Elimination: We find mixed results when applying RFE. Sometimes it helps the classifier performance a bit as in the *white-to-kill* model, while in other cases , it slightly decreases classifier performance as in the *black-to-live* model. It does significantly cut the number of features by about 50 to 70 percent while minimally affecting classifier performance. As such, if speed of feature extraction is a more important consideration than classifier performance, using the feature-selected models might be beneficial.

C. Model Evalutation:

We evaluated the model varying the following parameters:

- Model type: {LDA, QDA, Logistic Regression, k-NN, Naïve Bayes, Linear SVM, Random Forest, AdaBoost}
- Scaling: {True, False}
- Down-Sampling: {True, False}
- PCA: {None, Regular, With Whitening}

We test each combination three-times to reduce the effects of randomness. We train the models on the training set and test on

Feature Set	Precision	Recall	F-Measure	Accuracy	AUROC	Number of Features
Stone Count	0.217	0.535	0.309	0.512	0.527	9.00
Stone Count-feat	0.217	0.546	0.310	0.505	0.525	8.67
Stone Count-inv	0.215	0.563	0.311	0.493	0.525	6.00
Liberty Count	0.217	0.512	0.305	0.523	0.522	10.00
Liberty Count-feat	0.215	0.524	0.305	0.512	0.523	9.67
Liberty Count-inv	0.214	0.496	0.299	0.525	0.523	6.00
Distance From Center	0.218	0.530	0.309	0.516	0.522	1.00
Distance From Center-feat	0.214	0.531	0.305	0.507	0.522	1.00
Number of Live Groups	0.250	0.045	0.076	0.778	0.499	4.00
Number of Live Groups-feat	0.264	0.051	0.085	0.778	0.504	2.00
Number of Live Groups-inv	0.252	0.044	0.075	0.779	0.502	2.00
Local Shapes Presence	0.211	0.411	0.279	0.568	0.509	106.00
Local Shapes Presence-feat	0.213	0.140	0.169	0.719	0.504	1.00
Hu Moments	0.239	0.445	0.310	0.598	0.559	32.00
Hu Moments-feat	0.240	0.440	0.311	0.603	0.560	25.67
Sparse Dictionary	0.211	0.408	0.278	0.569	0.517	82.00
Sparse Dictionary-feat	0.212	0.310	0.248	0.625	0.507	24.00
All	0.257	0.420	0.319	0.634	0.581	244.00
All-feat	0.251	0.407	0.310	0.632	0.576	79.33
All-inv	0.264	0.421	0.325	0.643	0.591	235.00
All-noLoc	0.247	0.452	0.320	0.608	0.575	138.00
All-noLive	0.250	0.419	0.313	0.626	0.575	240.00
All-noLocLive	0.247	0.445	0.317	0.610	0.570	134.00
Expert	0.234	0.515	0.322	0.558	0.563	24.00
Expert-feat	0.236	0.523	0.325	0.558	0.561	15.33
Expert-inv	0.232	0.504	0.318	0.560	0.559	15.00
General	0.242	0.443	0.313	0.604	0.562	114.00
General-feat	0.237	0.428	0.305	0.603	0.558	46.00

TABLE I: Results of experiments on various subsets of the features for the *black-to-live* model. '-feat' means recursive feature elimination was run, '-inv' means only board-invariant features were used, '-noLoc' means no local shape features were use, '-noLive' means the live group feature wasn't used, '-noLocLive' means neither local shape nor live group features were used.

Feature Set	Precision	Recall	F-Measure	Accuracy	AUROC	Number of Features
Stone Count	0.1881	0.5524	0.2807	0.4945	0.5236	9.00
Stone Count-feat	0.1870	0.5465	0.2786	0.4949	0.5238	8.67
Stone Count-inv	0.1903	0.5844	0.2871	0.4821	0.5295	6.00
Liberty Count	0.1904	0.5208	0.2789	0.5194	0.5258	10.00
Liberty Count-feat	0.1856	0.5106	0.2722	0.5127	0.5169	5.33
Liberty Count-inv	0.1918	0.5057	0.2781	0.5314	0.5285	6.00
Distance From Center	0.1862	0.5248	0.2748	0.5060	0.5200	1.00
Distance From Center-feat	0.1860	0.5216	0.2741	0.5070	0.5202	1.00
Number of Live Groups	0.1846	0.9887	0.3111	0.2183	0.5202	4.00
Number of Live Groups-feat	0.1837	0.9858	0.3097	0.2156	0.5140	1.00
Number of Live Groups-inv	0.1844	0.9870	0.3108	0.2186	0.5183	2.00
Local Shapes Presence	0.1859	0.5001	0.2710	0.5197	0.5215	106.00
Local Shapes Presence-feat	0.2023	0.4813	0.2849	0.5687	0.5345	1.00
Hu Moments	0.1946	0.5529	0.2878	0.5116	0.5443	32.00
Hu Moments-feat	0.1986	0.4927	0.2830	0.5546	0.5490	18.00
Sparse Dictionary	0.1890	0.5639	0.2831	0.4903	0.5286	82.00
Sparse Dictionary-feat	0.1872	0.5676	0.2816	0.4831	0.5232	21.00
All	0.2046	0.6089	0.3063	0.5076	0.5704	244.00
All-feat	0.2061	0.5999	0.3068	0.5162	0.5740	194.00
All-inv	0.2051	0.6168	0.3078	0.5050	0.5702	235.00
All-noLoc	0.1976	0.5850	0.2954	0.5019	0.5515	138.00
All-noLive	0.2038	0.5998	0.3042	0.5103	0.5661	240.00
All-noLocLive	0.1968	0.5819	0.2941	0.5015	0.5456	134.00
Expert	0.2037	0.5974	0.3037	0.5109	0.5706	24.00
Expert-feat-direct	0.1950	0.7008	0.3000	0.4216	0.5485	13.67
Expert-inv	0.2023	0.5663	0.2981	0.5239	0.5662	15.00
General	0.1927	0.5769	0.2889	0.4931	0.5360	114.00
General-feat	0.1964	0.5806	0.2935	0.5011	0.5466	46.67

TABLE II: Results of experiments on various subsets of the features for the *white-to-kill* model. '-feat' means recursive feature elimination was run, '-inv' means only board-invariant features were used, '-noLoc' means no local shape features were use, '-noLive' means the live group feature wasn't used, '-noLocLive' means neither local shape nor live group features were used.

Difficulty	Number of Problems	Number Correct	Percent Correct
30k	2	2	100%
25k	N/A	N/A	N/A
20k	3	2	67%
18k	3	3	100%
15k	5	3	60%
12k	7	5	71%
10k	7	4	57%
8k	N/A	N/A	N/A
7k	N/A	N/A	N/A
5k	6	3	50%
total	33	22	**67%**

TABLE V: Results of system on various difficulties when trained on the training set and tested on the development set

Difficulty	Number of Problems	Number Correct	Percent Correct
30k	N/A	N/A	N/A
25k	3	1	33%
20k	2	1	50%
18k	3	0	0%
15k	4	2	50%
12k	6	1	17%
10k	13	8	62%
8k	6	4	67%
7k	9	5	56%
5k	N/A	N/A	N/A
total	46	22	**48%**

TABLE VI: Results of system on various difficulties when trained on the combined training and development set and tested on the test set

the development set. We record the precision, recall, f-measure, accuracy, and AUROC. From this, we select random forest with scaling and down-sampling as our model. We then repeat the experiment, except this time we train on both the training and development set and test on the test set. We get the results in Tables III and IV (only showing the best classifier for each model type), and verify that the initial results are indicative of the true performance of the classifiers, and random forest with scaling and down-sampling is a "good" model with AUROC of around 0.59 for both *black-to-live* and *white-to-kill* models with decent f-measure and recall values. We find in general, the simpler models such as LDA and naïve Bayes are the worse performing models, and the more complex models perform better on this problem domain.

In general, no classifier performs especially well. This is likely due to several reasons. First, the data is noisy because of the way it is constructed and parsed. Second, our features are not particularly discriminating as can be seen from the results of the feature analysis experiments and the low results on the model evaluation.

D. Analysis of Life and Death Solving Agent

We tested our agent in two ways. First, after training our agent on data from the training set, we ran it on problems from the development set that ranged in difficulty from 30k (absolute beginner) to 5k (high-level amateur). The results are shown in table V We found that the agent was able to solve most of the easier problems. This is likely due to the relatively small number of moves needed to solve the problem or because the "good" moves are obvious. Having a small number of moves required for a solution allows the agent to find terminal states at low depths, and thus, solving the problem becomes easier than having to rely on the classification model. The agent was even able to solve a good portion of the more difficult problems, but this might be because the search depth was too deep and the board size was too large, and thus, the agents were able to make slightly better than random moves until one agent happened by chance to hit a correct move and started moving along the solution path, or the depth of the tree was reduced enough after several moves that terminal states started to appear in the search.

We see slightly different results in our second experiment. The second experiment involved training our agent on both the training set data and development set data and running it on the test set. The results are show in in table VI. We find that the agent doesn't perform as well on lower difficulty problem, but continues to do decently on higher difficulty problems. The reason for such a disparity might be the result of having much more data for higher difficulty problems than lower difficulty problems or because the lower difficulty problems might be just deep enough to give the *white-to-kill* model an advantage of spotting terminal states first and being able to block black's progression. Likewise, the good results in the higher-level difficulty might be caused by the slighty better than random movements described above.

E. Expert Analysis of Agent Results

As a Go Player, looking at the paths that our model and search produced, I was rather impressed that it was able to "solve" as many problems as it did. If I were to rate our model in terms of strength, then I would say that it's about the level of a 23k person: a bit stronger than someone who has just learned the rules of the game. How is it managing to solve problems that are well beyond it's strength range then? I believe that it's because both players that we have are at that same strength. The moves that they make on the higher difficulty boards are pseudo-random until one player stumbles onto something that leads to unconditional life, and that player isn't ever punished for making the bad moves because the opponent is just as bad as he is. I feel as though if instead of a two-model system, we had an oracle (whether this be a much much stronger model or a human player) who was controlling the "enemy", then our system would end up performing much worse than it currently does. Additionally, while there are times that the total solution path is present, they often get the order of the moves wrong. While the end result, in these cases, end up being the same, the order of the moves in Go matters greatly, and had it been playing against anyone stronger, it would have been strongly punished. Designing such an oracle would have been very hard to do, however.

VI. Marketing

Since our agent has performed quite well on the easier problems, it could prove a valuable tool for providing guidance to novice players, such as by suggesting moves, especially when the player is taking a long time to decide. The visual

Model	Precision	Recall	F-Measure	Accuracy	AUROC
LDA-scale-down-pca	0.2373	0.8114	0.3670	0.3286	0.5339
QDA-scale-pca	0.2818	0.5165	0.3646	0.5683	0.5719
LogReg-scale-pca	0.2785	0.5541	0.3707	0.5488	0.5736
kNN-scale-down	0.2602	0.5431	0.3519	0.5201	0.5280
NaïveBayes-scale-down	0.2840	0.6674	0.3248	0.4421	0.5490
LinearSVM-down	0.2839	0.5840	0.3812	0.5455	0.5861
RandForest-scale-down	0.2892	0.4851	0.3624	0.5905	0.5858
AdaBoost-scale-down-pcaW	0.2740	0.5314	0.3616	0.5500	0.5624

TABLE III: Best results for each classifier type on *black-to-live* using the training and development sets for training and testing on the test set

Model	Precision	Recall	F-Measure	Accuracy	AUROC
LDA-scale-down-pcaW	0.1939	0.0203	0.0368	0.8092	0.5012
QDA-scale-pca	0.2142	0.5613	0.3101	0.5522	0.5804
LogReg-scale-down	0.2087	0.5985	0.3095	0.5212	0.5771
kNN-scale-down-pca	0.1910	0.5449	0.2829	0.5046	0.5204
NaïveBayes-scale-down	0.2062	0.0140	0.0260	0.8136	0.5210
LinearSVM-scale-pca	0.2093	0.5842	0.3082	0.5297	0.5783
RandForest-scale-down	0.2171	0.5888	0.3172	0.5456	0.5930
AdaBoost-scale-down-pca	0.2110	0.6168	0.3145	0.5178	0.5842

TABLE IV: Best results for each classifier type on *white-to-kill* using the training and development sets for training and testing on the test set

indicator (green dot) indicating unconditional life could also be used as a learning tool to understand the basic concepts of the game.

We have successfully developed a framework for working on Life and Death Go problems. In its current form, it could be used as a Life and Death tutor, which players can play against to gain practice in handling those situations. With additional data, our agent can easily be adapted to a general game-playing agent which can operate at different skill levels.

Many games of general Go end up having situations where youll have to play through a life and death situations. The GoLD system can be a component for addressing these sub-problems.

VII. Future Work

There was a lot more that we could have done with our data set. A method of reducing noise that we didn't think of at the time was to further standardize the initial board states by rotating all of the boards to the same orientations and further splitting up the problems to corner, edge, and center cases. This might have allowed for more repeated instances of similar problems to show up, which would further improve both our learning and our caching speeds and would allow for better accuracy and faster search. We also threw out a lot of data due to there being problems with them either with formatting or just bad labeling. Is there a way for us to make use of this data?

All of our features needed a bit more work. Many of the features that we were contemplating turned out to be too costly, or not accurate enough. More work could have been done to attempt to generate features more representative of the problem and to compute them faster, either through updating a cache or through better implementation.

VIII. Conclusion

We found that the problem of solving life and death problems in Go was more difficult than expected. Simply determining life and death of groups of stones (our terminal test) ended up being our largest problem, and it turns out that this is a classic problem in the field. Finding discriminating features that generalize well is another difficult problem, one that remains unsolved by us. While we had several additional feature ideas based on experienced go player's expertise, we found that these features were too difficult to code or when translated to code, would be computationally inefficient. We also saw the effect that noisy, unbalanced, unlabeled, ill-formatted, and incomplete data can have on one's ability to develop good classification models and solve complex problems.

Go is a hard problem in general. The complexity of Go and size of search tree are especially bad [23] [16]: for a game of unbound length, Go is EXPTIME-Complete and for bound length, it is PSPACE-Complete. Life and death problems are no less difficult being PSPACE-Hard and EXPTIME-Complete.

Overall, however, we learned a lot about Go-playing agents and artificial intelligence techniques from our project, and our system is a good first step to building better systems.

References

[1] A. Kishimoto and M. Müller, "Search versus knowledge for solving life and death problems in go," in *AAAI*, pp. 1374–1379, 2005.

[2] D. Fotland, "Knowledge representation in the many faces of go," 1993.

[3] M. Enzenberger, "The integration of a priori knowledge into a go playing neural network," *URL: http://www.markus-enzenberger.de/neurogo.html*, 1996.

[4] T. Graepel, M. Goutrie, M. Krüger, and R. Herbrich, "Learning on graphs in the game of go," in *Artificial Neural NetworksICANN 2001*, pp. 347–352, Springer, 2001.

[5] D. H. Stern, T. Graepel, and D. MacKay, "Modelling uncertainty in the game of go," in *Advances in neural information processing systems*, pp. 1353–1360, 2004.

[6] D. Stern, R. Herbrich, and T. Graepel, "Bayesian pattern ranking for move prediction in the game of go," in *Proceedings of the 23rd international conference on Machine learning*, pp. 873–880, ACM, 2006.

[7] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski, "Temporal difference learning of position evaluation in the game of go," *Advances in Neural Information Processing Systems*, pp. 817–817, 1994.

[8] S. Gelly and D. Silver, "Achieving master level play in 9 x 9 computer go.," in *AAAI*, vol. 8, pp. 1537–1540, 2008.

[9] D. Silver, R. S. Sutton, and M. Müller, "Reinforcement learning of local shape in the game of go.," in *IJCAI*, vol. 7, pp. 1053–1058, 2007.

[10] A. Kishimoto, *Correct and efficient search algorithms in the presence of repetitions*. PhD thesis, University of Alberta, 2005.

[11] A. Hollosi, "Sensei's library." http://senseis.xmp.net/, May 2012.

[12] D. B. Benson, "Life in the game of go," *Information Sciences*, vol. 10, no. 2, pp. 17–29, 1976.

[13] T. Cazenave, "Generation of patterns with external conditions for the game of go," *Advance in Computer Games*, vol. 9, pp. 275–293, 2000.

[14] D. Dyer, "An eye shape library for computer Go," 1995.

[15] T. Wolf, "The program gotools and its computer-generated tsume go database," in *Game Programming Workshop in Japan*, vol. 94, pp. 84–96, 1994.

[16] K. Chen and Z. Chen, "Static analysis of life and death in the game of go," *Information Sciences*, vol. 121, no. 1, pp. 113–134, 1999.

[17] A. Miller, "Go problems." http://www.goproblems.com.

[18] M.-K. Hu, "Visual pattern recognition by moment invariants," *Information Theory, IRE Transactions on*, vol. 8, no. 2, pp. 179–187, 1962.

[19] B. A. Olshausen and D. J. Field, "Sparse coding with an overcomplete basis set: A strategy employed by v1?," *Vision research*, vol. 37, no. 23, pp. 3311–3325, 1997.

[20] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, and C. Suen, "Ufldl tutorial," 2012.

[21] B. H. Menze, B. M. Kelm, R. Masuch, U. Himmelreich, P. Bachert, W. Petrich, and F. A. Hamprecht, "A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data," *BMC bioinformatics*, vol. 10, no. 1, p. 213, 2009.

[22] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Machine learning*, vol. 46, no. 1-3, pp. 389–422, 2002.

[23] R. A. Hearn and E. D. Demaine, *Games, puzzles, and computation*. AK Peters Wellesley, 2009.