CMU-CS-86-173

A Pattern Classification Approach to Evaluation Function Learning

Kai-Fu Lee

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

October 1986

Abstract

We present a new approach to evaluation function learning using classical pattern-classification methods. Unlike other approaches to game-playing where ad-hoc methods are used to generate the evaluation function, our approach is a disciplined one based on Bayesian Learning. This technique can be applied to any domain where a goal can be defined and an evaluation function can be applied. Such an approach has several advantages: (1) automatic and optimal combination of the *features*, or terms, of the evaluation function; (2) understanding of inter-feature correlations; (3) capability for recovering from erroneous features; (4) direct estimation of the *probability of winning* by the evaluation function. We implemented this algorithm using the game of Othello and it resulted in dramatic improvements over a linear evaluation function that has performed at world-championship level.

erroneous features? this is new for me!!!

need to try the linear evaluation approach first

This research was partly sponsored by a National Science Foundation Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

In Bayesian learning, probability distributions are used to represent uncertainty about parameters or assumptions in a model. When new data becomes available, we update our probability distribution based on the new information, a process referred to as "Bayesian updating." This allows us to not only have an estimate of the parameters but also to quantify the uncertainty associated with that estimate.

Table of Contents

need to understand how they train the parameters

List of Figures

List of Tables

what is this algorithm?

1. Introduction

Most successful game-playing programs employ full-width search that applies a heuristic evaluation function at terminal nodes [14] [1] [11] [7]. A typical evaluation function has the form:

$$Eval = C_1 \times F_1 + C_2 \times F_2 + \cdots + C_n \times F_n, \quad (1.1)$$

this is what I've tried recently

where *Eval* is the static evaluation of a board configuration, and linearly combines a number of *features* $(F_1, F_2, \cdots F_n)$ weighed by coefficients $(C_1, C_2, \cdots C_n)$. Each feature is a well-defined measure of the "goodness" of the board position. In chess, reasonable features might be piece-count advantage, center control, and pawn structure. In Othello, reasonable features might be mobility, edge position, and disc centrality.

The above formulation suggests three ways to improve a game-playing program:

1. Finding a superior search strategy.
2. Selecting better features.
3. Combining the features appropriately.

I've not done much work on this

this is what leads us to this paper

We believe that the first two ways are already well-understood. Researchers have proposed several new full-width search strategies, such as Scout [10] and the zero-window search [1]. Unfortunately, these techniques provide only constant improvements in an exponential search space [7]. Selecting good features is extremely important. However, good features are usually not very difficult to derive from expert knowledge[1]. At present, the strongest game-playing programs are relying on fast hardware instead of new search algorithms [3] [1], and efficient feature analysis instead of discovering new features [1] [7]. Therefore, we suggest that research in full-width searching and feature selection has reached the saturation point, and that the success of future game-playing programs will depend crucially on how well the features are combined. In this paper, we will introduce an algorithm based on *Bayesian Learning* that automatically combines features.

need to learn more no these search

Unlike feature selection, feature combination is a very unintuitive process. On the one hand, one must establish a precarious balance among diversified strategies (such as choosing weights for positional advantage or piece advantage in chess). On the other hand, one must attend to interaction between related features (such as pawn structure and king safety in chess). Furthermore, one always faces the dilemma of having either too few features, or many features that include correlated or redundant ones.

maybe I should start learning from this

Samuel [12] was the first to propose algorithms that automatically combine features. He introduced a linear evaluation function learning algorithm, and subsequently devised a non-linear learning algorithm based

[1] Finding good features *automatically* is a very difficult problem, but it is beyond the scope of this paper.

on *signature tables* [13]. However, his algorithms suffer from a number of problems. The linear evaluation function has an inadequate understanding of relationships among the features. The *signature tables* have smoothness problems from extreme quantization. The greatest problem with both algorithms is that while they obviated the burden of tuning the coefficients, they imposed new burdens such as choosing the *signature tables* structure, determining the range and level of quantization, and selecting the amount and frequency of functions' adjustments during learning. The effort required to tune the programs far exceeded the conventional trial-and-error tuning of the coefficients with expert help. Finally, because of the effort required, Samuel's algorithms are highly domain dependent.

I thought about this too; a linear combination is just not enough

dont understand about quantization but I ignore it.

In this study we report a new learning algorithm. Like Samuel's, our algorithm learns to combine features into an evaluation. Unlike Samuel's, however, it has no problem with smoothness and is completely automatic, requiring no tuning whatsoever. Our algorithm is based on Bayesian Learning [4].

1. We accumulate a large number of games as training data. Each game is played by two expert players.
2. We label (either manually or automatically) each position as *winning* or *losing* according to some criteria. The criterion used in this study is the actual outcome of the game.
3. The training program computes a Bayesian discriminant function the labeled training data. The function tries to recognize feature patterns that represent *winning* or *losing* positions. Given a set of feature values of a position, it assigns a probability that the position is a *winning* one.
4. We build different classifiers for different stages of the game.

Our algorithm has five important advantages:

1. The learning process is completely automatic. There are no coefficients or parameters to tune, and no normalization is needed.
2. Assuming multivariate normal distribution [4], the quadratic combination is proven to be optimal on the training data.
3. The algorithm considers not only the features themselves, but also how they covary with each other. For example, if two correlated features were used, they will not both contribute fully to the evaluation.
4. The algorithm can recover from erroneous information in the evaluation. For example, adding a random feature would not degrade its performance.
5. It returns *the probability of winning* as the evaluation. This evaluation is one that all evaluation functions try to emulate. Furthermore, having such an evaluation validates comparison between values on different levels of the search tree.

well I need to prove this statement

We tested this algorithm in the domain of Othello. An Othello program, BILL 2.0 [7], was modified to learn an evaluation function from the features that it already used. Since the evaluation in BILL 2.0 was carefully tuned and since BILL 2.0 is virtually invincible, only moderate improvements were expected.

However, results showed that BILL 3.0 (using Bayesian Learning) defeated BILL 2.0 over twice times as often as it was defeated from nearly even initial positions. The average final score from this experiment was 37 to 27. We show that this gain is equivalent to two extra plies of search. In another experiment involving Othello problem solving, BILL 3.0 solved 11% more problems than BILL 2.0 using eight plies of search.

need to take note of all the cons to write it later on the report

In Chapter 2, we will first discuss conventional ways of constructing evaluation functions and their shortcomings, with emphasis on Samuel's work. In Chapter 3, Bayesian Learning and its application to evaluation function learning is described in detail. In Chapter 4, the results from Othello are presented. Chapter 5 contains analyses and discussion of the Bayesian Learning of evaluation function. Finally, Chapter 6 contains some concluding remarks.

2. The Construction of an Evaluation Function

2.1 The Role of the Evaluation Function in Search

The fundamental paradigm in game-playing programs has changed very little since Newell, Simon, and Shaw's discovery of the alpha-beta relationship [9]. Almost all programs still rely on full-width alpha-beta search, and all programs still use static evaluation at terminal nodes. from static -> dynamic evaluation

Since most programs employ similar search strategies, the evaluation function plays the most crucial part in game-playing programs. The evaluation function embodies the knowledge of the program, and is responsible for differentiating good moves and positions from poor ones. Furthermore, since most programs rely on the evaluation function for move ordering, a good evaluation function leads to a more efficient search.

a good evaluation function helps: - embodies the knowledge of the game
- search more efficiently
- find the optimal move in a certain position -

The static evaluation includes two stages: (1) evaluating certain *features* of a board position, and (2) combining these feature scores into an evaluation. Selecting the features is a domain-dependent task, and cannot be systematically studied. In this study, we will focus on the combination of the feature scores into an evaluation. In particular, we will later present an algorithm that accomplishes this task automatically.

maybe in the report I should list all the method/formula for each feature first; tell my point of view then talk about how I come up with my feature evaluation

Traditionally, a static evaluation is a linear combination of the features as shown in Equation (2.1):

$$Eval = C_1 \times F_1 + C_2 \times F_2 + \cdots + C_n \times F_n. \tag{2.1}$$

where *Eval* is the static evaluation of a board configuration, and is a linear combination of *features* $(F_1, F_2, \cdots F_n)$ weighed by coefficients $(C_1, C_2, \cdots C_n)$.

biggest drawback!!!

There are two problems with this representation. First, it assumes that the features are independent, and that they can be combined linearly. This is clearly a false assumption. In fact, we will later show that *every pair of features are correlated to some degree.* Second, the coefficients are usually derived by ad-hoc methods. In many cases, the implementor guesses these coefficients from his domain knowledge. Even when the implementor is knowledgeable, it is difficult to derive these coefficients because humans do not think in terms of alpha-beta search and static evaluation. When the implementor is not knowledgeable, he will be clueless. This was the initial motivation for Arthur Samuel, a novice checkers player, to write a checkers learning program.

2.2 Samuel's Evaluation Function Learning Experiments

One of the earliest and the most intensive studies on machine learning was conducted by Arthur Samuel in the domain of checkers from 1947 to 1967 [12] [13]. His objective was very similar to that of this study, namely, given a set of feature values of a board position, assign a score which measures the goodness of the

position. Although he performed many experiments, we will focus on the two most important ones: (1) polynomial evaluation learning through self-play, and (2) nonlinear *signature table* evaluation learning through book-move. In the two subsequent sections, these two procedures will be described and evaluated.

2.2.1 Polynomial Evaluation Learning through Self-play

In polynomial evaluation learning [12], Samuel arranged to have two copies of the checkers programs play against each other, and learn the weight for each feature in a linear evaluation function. One copy of the program, Beta, uses a fixed function throughout a game. The other copy, Alpha, continuously improves its evaluation function. Alpha learns by comparing its evaluation to that of a more accurate evaluation, which is derived by the use of a minimax search. If the search returns a sufficiently higher value than the static evaluation, it is assumed that the static evaluation is in error. Each negative feature in the static evaluation is penalized by lowering its weight.

wow I've not tried this

what's exactly is alpha, beta in this case? need to read that paper.

Alpha and Beta are originally identical, and Alpha continuously improves its weights. After each game, if Alpha defeats Beta, it is assumed to be better, and Beta adopts Alpha's evaluation function. Conversely, if Alpha loses three games to Beta, it is assumed to be on the wrong track, and the coefficient of its leading term is set to 0 in an attempt to put it back on the right track. Furthermore, manual intervention was used to restore previous states if "it becomes apparent that the learning process is not functioning properly."

how do I implement this?

The resulting evaluation function from this learning algorithm seemed to stabilize after a number of games when Alpha consistently defeated Beta. The final program was able to play a "better-than-average" game of checkers. This learning procedure was one of the first examples of machine learning. However, its validity is predicated upon several incorrect assumptions.

The first of these assumptions is that a good evaluation function can be defined as a linear combination of independent features. This assumption is false in general, and is particularly wrong with Samuel's learning procedure because he intentionally collected redundant features. If two identical features were considered by Samuel's procedure, both would be assigned the same weight, resulting in over-estimation of the feature's value. Furthermore, a linear evaluation function is unable to capture the relationship between features. By repeating Samuel's experiments, Griffith [5] showed that better performance was achieved by an extremely simple heuristic move ordering procedure.

The second assumption is that when search and static evaluation disagree, the static evaluation must be in error. While deep searches are more accurate than shallow searches in general, there are several ways this assumption can fail. It is possible that a problem with a position can only be discovered by looking several plies ahead. In this case the static evaluation should be allowed to remain in error, as it is usually impossible

to program such look-ahead knowledge into it. Furthermore, searches could suffer from the *horizon effect* [2], resulting in inaccurate evaluations.

It refers to a limitation or challenge in decision-making systems that occurs when these systems have a limited view or "horizon" of the future.

The third assumption is that if the evaluation function is found to be overly-optimistic, any positive component is assumed to be in error. This is clearly incorrect because in most positions, each player is ahead in some features and behind in others. The erroneous evaluation may be due to components that are negative, but not negative enough. Just checking the sign is simply not adequate.

Finally, this procedure assumes that if Player A defeats Player B, it must be because of its superior evaluation function. This assumption may be reasonable for expert players, but when novice programs play each other, a win may be the result of: (1) a superior evaluation function, (2) luck (when neither player understands the position), or (3) opponent's errors. Since Samuel's program played like a novice, it would often win due to luck or opponent's errors, resulting in erroneous credit assignment.

2.2.2 Signature Table Learning through Book Moves

Samuel recognized these problems, and subsequently devised another learning procedure that remedied most of them [13]. In order to handle nonlinear interactions among the feature, he introduced *signature tables*; and in order to cope with the incorrect assumptions in self-play, he used *book moves.*

Signature tables are multi-dimensional tables that combine features together nonlinearly. Each dimension represents some feature. To evaluate a board position, the value for each feature is used to index into the table, and the cell corresponding to this index contains the evaluation. The obvious problem with this scheme is that the tables become extremely large. Samuel deals with this problem by using a a hierarchical organization, and the features are grouped into sets of 4. Only intra-set interactions are considered. Then each set produces a value from the cell indexed by these four feature values. Tables in the next higher level use these values as if they were features. There were a total of three levels. But this could still result in very large tables, so Samuel quantized his feature values. In each of the lowest-level tables, the values were restricted to (-1, 0, 1) for three of the features, and (-2, -1, 0, 1, 2) for the other feature. This resulted in the final configuration as shown in Figure 2-1. This configuration contains 883 cells, which is reasonable for both storage and training.

what?????

what does each cell represent for?

These cells are trained from *book moves.* A large number of board positions from master play were presented to the program. The goal was to have the *signature table* learn to emulate master play. This is done by keeping counts for each cell corresponded with the feature combination of the move chosen by the master (A), and for each cell that corresponded with the feature combination of each legal move not chosen by the master (D). The actual cell value is updated periodically with $\frac{(A-D)}{(A+D)}$, which is a measure of how well the cell corresponded to book moves.

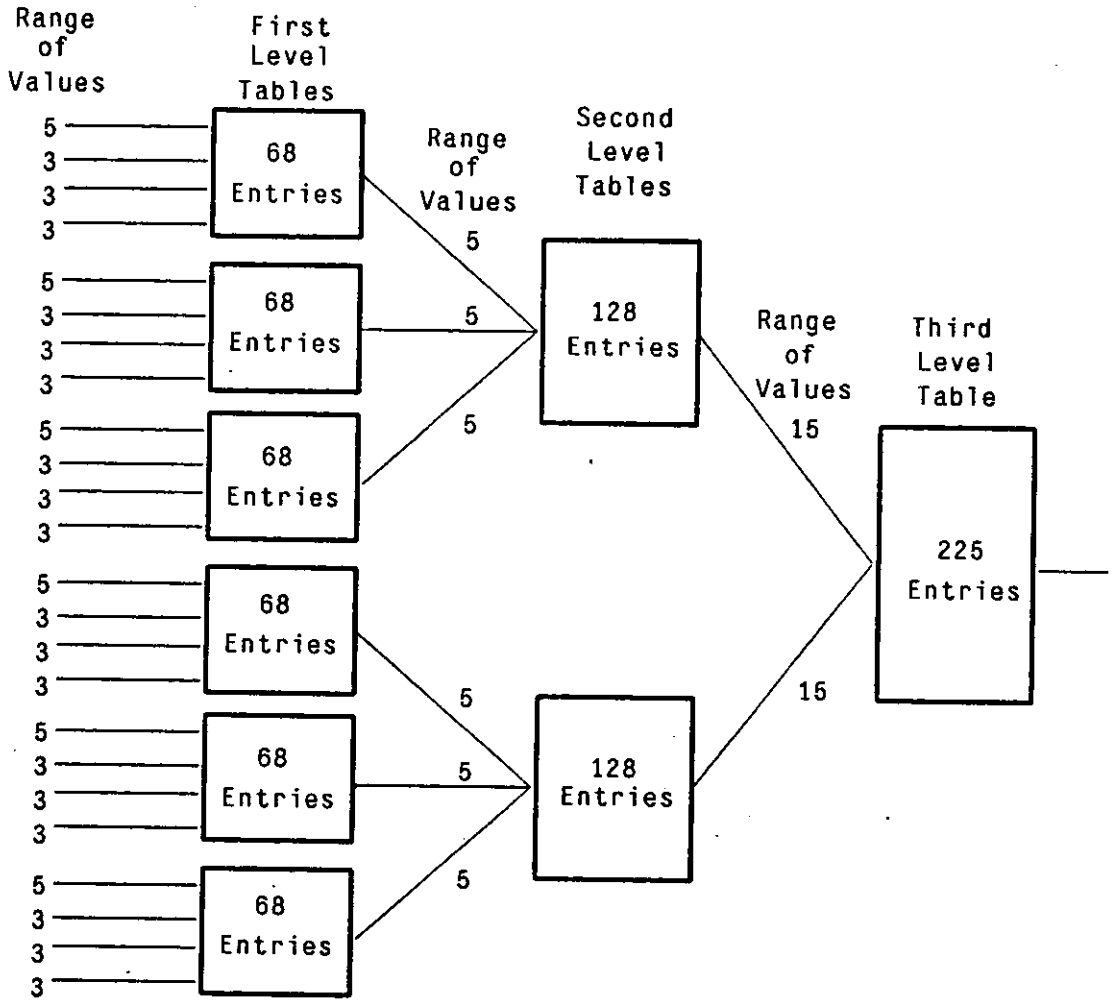


Figure 2-1: Samuel's final signature table scheme.

The *signature table* approach successfully added nonlinear learning; the *book learning* approach successfully eliminated the incorrect assumptions of self-play. The resulting program significantly outperformed a linear function trained by book learning.

However, there are a number of new problems with the signature table approach. First, this approach makes two assumptions: (1) the book move is always the best move, and (2) there are no equally good alternative moves. Since these were expert games, and games from the losing side were not used, they are reasonable assumptions, but certainly not infallible ones. These problems could probably be minimized by using a sufficiently large training database.

A major problem is that considerable accuracy is lost during the quantization process. Extreme quantization is very risky. For example, if one were to quantize material difference in chess into 3 or 5 values,

how could one expect to make the right moves when the loss of a bishop may be equal to the loss of a queen? Moreover, extreme quantization violates Berliner's *smoothness* principle [2], thereby introducing the *blemish effect*. In Berliner's words,

> a very small change in the value of some feature could produce a substantial change in the value of the function. When the program has the ability to manipulate such a feature, it will frequently do so to its own detriment.

The arbitrary quantization into so few levels has precisely this problem.

In spite of its sacrifice of smoothness, signature table learning still does not provide a general solution to the linearity problem. Consider the case where two features were identical (or highly correlated). If these features, F1 and F2, were organized as in Figure 2-2a, their redundancy will be successfully eliminated in Table 1. However, if they were organized as in Figure 2-2b, F1 would contribute to Table 1, and F2 would contribute to the Table 2. Furthermore, Table 3 could not eliminate this redundancy because it could not know whether its two inputs are affected by F1 and F2 or the other features. This results in over-estimation of the utility of this feature. For the same reason, we believe the higher level signature tables to be of little utility as they cannot identify the contributors to their inputs, and consequently cannot handle inter-table correlations.

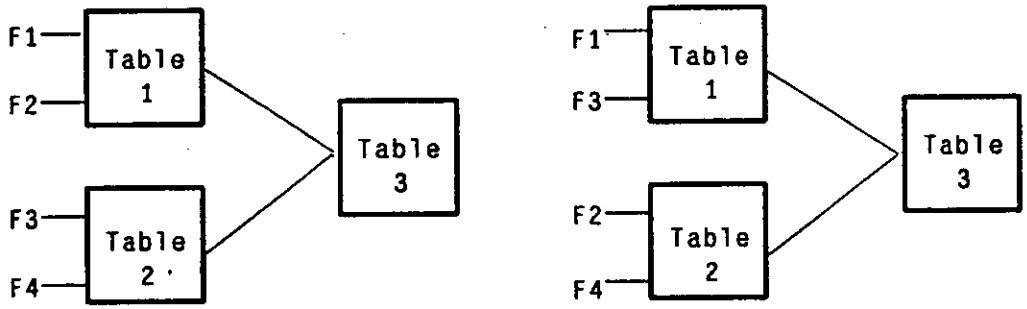


Figure 2-2: If F1 and F2 are the same feature, the signature table configurations in (a) would cancel their redundancy, but the one in (b) would not.

As a result, it is important to carefully arrange the structure of the signature tables so that covariances are captured. Furthermore, it is necessary to determine the range and levels for the quantization of each level, the initial values in the cells, the frequency that they must be updated, and many others. This is what we consider to be the greatest flaw of both procedures, namely, excessive human initialization and interventions are required. Each parameter adds the possibility of human errors affecting the learning process, and

increases the amount of time needed to derive and test these values. The result is either poor learning caused by human errors, or acceptable learning as a result of excessive human intervention.

All of this heuristic tuning made Samuel's procedure domain dependent. If one wanted to implement his procedures in another domain, considerable additional domain knowledge must be used in the learning algorithm, and considerable effort must be invested in the trial-and-error modification of the parameters. This is clearly undesirable.

Finally, the concepts learned by Samuel's algorithms are suboptimal. His self-play algorithm learns to distinguish *good features* from *bad features* based on search. His book-move algorithm learns to distinguish *moves chosen by experts* from *moves not chosen by experts*. However, in a game-playing domain, the optimal concept to learn is that which distinguishes *winning positions* from *losing positions*.

In summary, while Samuel's studies were a milestone in the early years of machine learning, we believe that the amount of supervision makes them impractical and domain dependent. These major flaws, coupled with Samuel's problematic assumptions, smoothness problems, and suboptimal learning severely limited the practicality and applicability of his procedures.

2.3 Other Work on Automatic Feature Combination

Griffith, who originally conceived of the idea of signature tables, reported a number of evaluation function learning results in checkers [5]. He compared linear evaluation function learning, two variants of signature table learning, and a heuristic move ordering algorithm. He showed that the heuristic move ordering algorithm, which has only extremely rudimentary checkers knowledge, outperformed the linear evaluation function, but was outperformed by the signature table algorithms.

my idea too, need to implement this

Another study was undertaken by Mitchell [8], who used regression analysis to create a linear evaluation function in Othello. The resulting program did not play as well as was hoped. We conjecture that this is due to the lack of nonlinearity in his program.

Both of these results indicate the inadequacy of linear evaluation functions. However, Griffith and Samuel's signature table approach is also not adequate because of problems with smoothness and the excessive tuning required.

3. Bayesian Learning of Evaluation Function

In this chapter, an algorithm that automatically combines the features is presented. First, Bayesian Learning is introduced for readers unfamiliar with the concept. Next, the game of Othello and the Othello program BILL are briefly described. Finally, we present the evaluation learning algorithm, which is based on Bayesian Learning, and applied to the domain of Othello.

3.1 Bayesian Learning

Bayesian Learning of discriminant functions is a standard technique used in pattern recognition. Typically, it is applied to recognition and classification of concrete objects, such as characters, images, speech and seismic waves. Assuming multivariate normal distribution, the discriminant function defines a decision boundary between classes. In a two-class problem, all points on this boundary are equally likely to belong to either class. This boundary is automatically computed from the features vectors of the training data, and takes into consideration variance and covariance of the features. It is composed of two stages, namely, training and recognition.

3.1.1 The Training Stage

The training stage is a straight-forward parameter estimation stage. A database of labeled training data is required. Each training sample consists of a feature vector and a label indicating which class the feature vector belongs to. The task of the training stage is to estimate the mean feature vector and the covariance matrix for each label (or class) from the training data.

The mean vector for a class c, μ_c, is simply the arithmetic average of each feature value for each training sample labeled as this class, and is computed as follows:

$$\mu_c = \frac{\sum_{i=1}^{N_c} \mathbf{x}_i}{N_c} \tag{3.1}$$

where N_c is the number of times class c was observed, and $\mathbf{x}_i$ is the i^{th} feature vector for class c.

The covariance matrix for class c, Σ_c, measures the degree that two features covary with each other, and is computed as follows:

$$\Sigma_c = \frac{1}{N_c} \sum_{i=1}^{N_c} (\mathbf{x}_i - \mu_c)(\mathbf{x}_i - \mu_c)^t \tag{3.2}$$

where t is the matrix transpose operation.

3.1.2 The Recognition Stage

From the mean vector and the inverse of the covariance matrix, the general multivariate normal density function p of a distribution can be computed as shown in Equation (3.3):

$$p(\mathbf{x} \mid c) = \frac{1}{2\pi^{-N/2}|\Sigma_c|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu_c)^t \Sigma_c^{-1}(\mathbf{x}-\mu_c)} \tag{3.3}$$

where $\mathbf{x}$ is the N-element feature vector, $|\Sigma_c|$ is the determinant of the covariance matrix, Σ_c^{-1} is the inverse of the covariance matrix, and μ_c is the mean vector for class c.

Bayes Rule shows that minimum-error-rate classification can be achieved using the discriminant function $g_c(\mathbf{x})$:

$$g_c(\mathbf{x}) = \log p(\mathbf{x} \mid c) + \log P(c) \tag{3.4}$$

where $P(c)$ is the *a priori* probability of class c. Substituting Equation (3.3) into Equation (3.4), we have:

$$g_c(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_c)^t \Sigma_c^{-1}(\mathbf{x}-\mu_c) - \frac{N}{2}\log 2\pi - \frac{1}{2}\log|\Sigma_c| + \log P(c) \tag{3.5}$$

To determine which class a new feature vector belongs to, g is computed for each class, and the new input is assigned to the class with the greatest g. Furthermore, if probabilities, P, are preferred to making hard decisions, they can be derived by simply normalizing $g(\mathbf{x})$:

$$P(c \mid \mathbf{x}) = \frac{e^{g_c(\mathbf{x})}}{\sum e^{g(\mathbf{x})}} \tag{3.6}$$

In Bayesian training, the "prior probability" refers to the initial belief or probability assigned to a hypothesis before considering any new evidence or data. It represents what we know or believe about the probability of a hypothesis based on prior information, knowledge, or experience.

3.2 The Game of Othello and Bill

Before explaining how Bayesian Learning can be applied to evaluation function learning, we first briefly describe the domain of our experiment, Othello.

Othello is a game played on an 8 by 8 board between two players, black and white. The board is initially set up as in Figure 3-1. Black starts the game by placing a black disc (a playing piece with the black side up) on any empty square on the board which allows white's disc(s) to be flipped. Each white disc captured between this black disc and any other black disc is flipped to black. The players alternately place discs on the board until neither player can make another move. The player with the most discs is declared the winner.

Othello has been a very popular game for computer implementation because of its relatively small branching factor, and the relative ease of writing programs that play reasonably. This ease is due to the

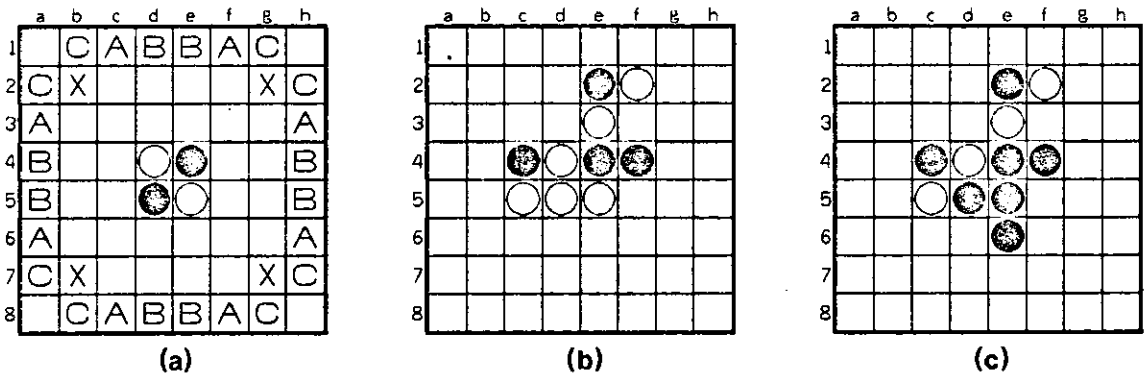


Figure 3-1: (a) shows the initial Othello board set-up and the standard names of the squares; (b) shows a sample board with legal moves (for Black) to C6, D6, D2, E6, and G2; (c) shows the board after Black plays to E6.

difficulty of Othello for human players. Othello is difficult because the board changes drastically after each move, and some strategies are counter-intuitive.

In 1981, Rosenbloom [11] demonstrated the feasibility of creating a world-championship level Othello program, IAGO. IAGO's evaluation consisted of a linear combination of edge stability, mobility, and potential mobility. In 1986, Lee and Mahajan [7] constructed another program, BILL, that consistently defeated IAGO. BILL placed second in the 1986 North American Championships, and is one of the best Othello players in the world. BILL's playing strength is mostly due to its use of an efficient yet accurate evaluation function that used tables with pre-compiled knowledge. BILL 2.0 uses four sets of these tables, hence four features. The features are:

need to read this

1. **Weighted Current Mobility** - Measures the quantity and quality of the player's moves. All moves are found using tables, and appropriate penalties are subtracted for poor moves.
2. **Weighted Potential Mobility** - Measures the goodness of the player's future moves. Tables that consider each neighbor of each disc are used.
3. **Weighted Squares** - Measures the goodness of each square occupied by each disc of the player. Both static measures (central squares are better than squares next to the corner) and dynamic measures (surrounded discs are better than peripheral discs) are used.

how to implement dynamic weight?

4. **Edge Position** - Measures the player's edge position using a table that contains every combination on each edge. This table is generated by a probabilistic minimax procedure [7].

this is the stability part that I've not done

The interested reader is directed to [11] or [7] for analyses of Othello strategies.

BILL 2.0 combines these features together linearly. The weights in the linear combination were determined by creating ten different versions and holding a tournament among these ten versions. The

version that won the tournament was chosen as the final set of coefficients. We will apply Bayesian Learning to the same four features, and measure the strength of the resulting program by comparing it against BILL 2.0.

3.3 Evaluation Function Learning

We will now present an evaluation function learning algorithm that uses Bayesian Learning as described in Section 3.1. The key difference between our application and typical applications is that instead of trying to recognize concrete objects, we are trying to recognize and classify board positions as *winning positions* or *losing positions.*

The basic approach of our algorithm is shown in Figure 3-2. Like Bayesian Learning, this algorithm consists of two stages, namely, training (learning) and recognition (evaluation), which will be described in the next two sections.

3.3.1 The Training Stage

In order to train a Bayesian discriminant function, a database of positions, where each position is labeled as a *winning position* or a *losing position*, is required. There are many ways to obtain such a database. In this study, the training data were taken from actual games between two experts, and each position of the winning player is marked as a *winning position*, and each position of the losing player as a *losing position.* While this is a simple and consistent method, there is a serious problem: A winning position could be lost by a poor subsequent move, and would be mislabeled as a losing position. We deal with this problem in two ways. First, *reliable* experts are needed. Since BILL 2.0 is a world-championship level player, we simply used it to generate the training data by self-play from initial positions. Second, 20 random initial moves were generated for each game and training commenced after 20 random moves (or after there are 24 discs on the board). It was hoped that one of the players would be ahead after the 20 moves, and that this player would go on to win the game. In cases with truly even positions, sufficient training data should result in equivalent number of *winning* and *losing* labels.

To generate the training data, BILL 2.0 was set up to play itself under the following conditions: the first 20 half-moves are made randomly, then BILL 2.0 is given 15 minutes for each side to play the remaining 40 half-moves[2]. When there are 15 half-moves left, an endgame search is performed to find the winner assuming perfect play on both sides. The game is terminated, and recorded as training data. 3000 games were played to estimate the parameters.

[2]There are usually exactly 60 half-moves to a game. The only exception is when neither side has a legal move.

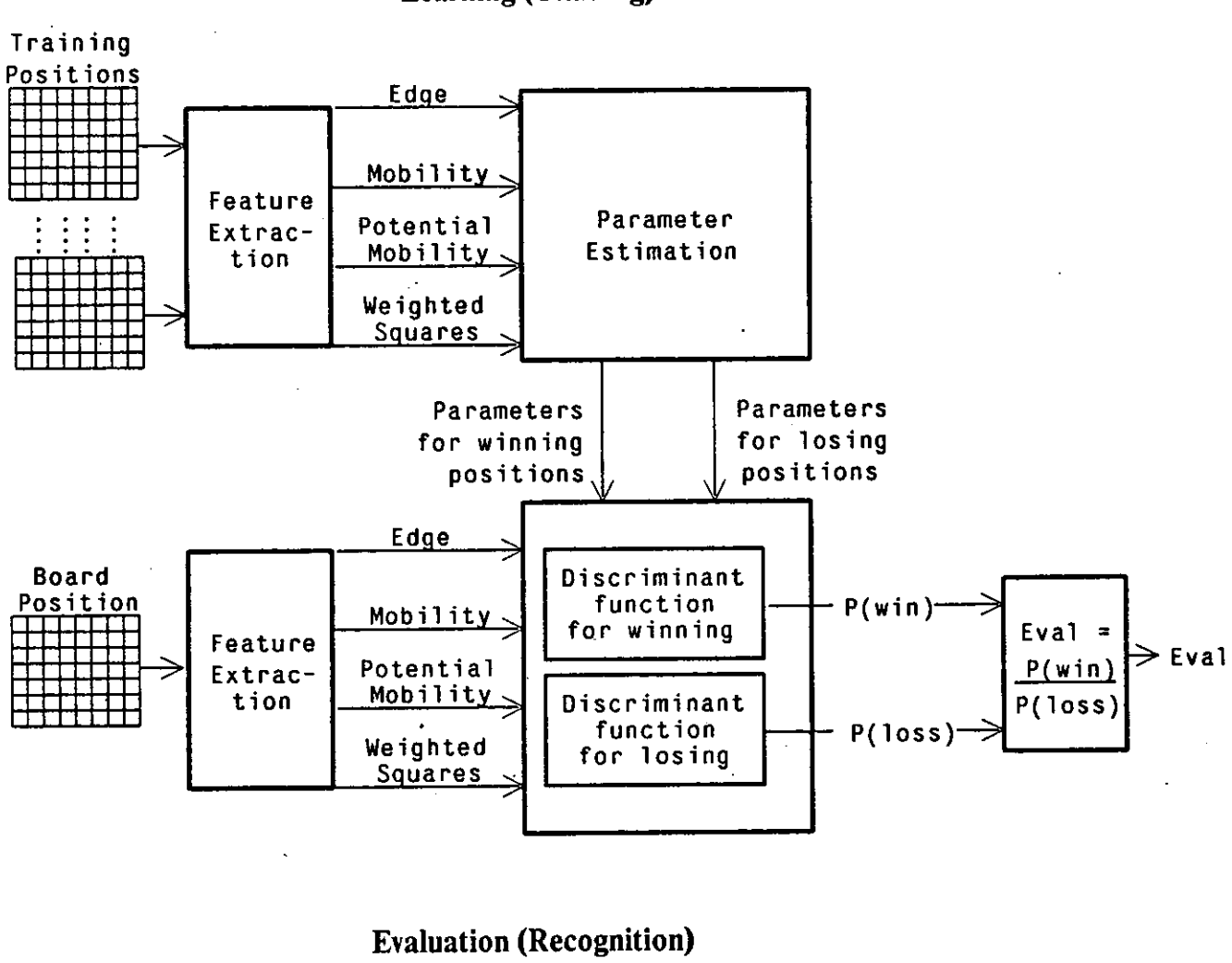


Figure 3-2: The learning and evaluation process of the proposed evaluation function learning algorithm based on Bayesian Learning.

It is well known that different strategies are needed for different stages of the game [2]. Therefore, we generated a discriminant function for each stage, where a stage is defined by the number of discs on the board. The discrimination function for a stage with N discs is generated from training positions with $N-2$, $N-1$, N, $N+1$, and $N+2$ discs[3]. Since there are almost always 60 moves per game in Othello, disc count provide a reliable estimate of the stage of the game. By coalescing adjacent data, the discriminant function is slow-varying, and similar to the *application coefficients* proposed by Berliner [2].

[3]Note that positions with *Discs* < 24 and those with *Discs* > 49 are not trained because of random initialization and endgame search. We solved this problem by copying the parameters for the *Discs* = 24 to *Discs* < 24, and those for *Discs* = 49 to *Discs* > 49.

Having generated the training data, the four features are extracted from each position in the database. Then, the mean feature vector and the covariance matrix between features are estimated for the two classes, *winning positions* and *losing positions*. Table 3-1 shows the mean vector, the covariance matrix, and the correlation matrix for the classes of *winning* and *losing* positions at Discs = 40. They clearly support our earlier claims that *every pair of features are correlated to some degree*, and that nonlinearity is crucial to the success of an evaluation function.

WIN				LOSS			
Mob.	Pot.	Wtd.	Edge	Mob.	Pot.	Wtd.	Edge
Mean				**Mean**			
533	266	242	599	-419	-200	-112	-334
Covariance Matrix				**Covariance Matrix**			
475871	185298	137911	121288	508045	196432	143986	118982
185298	133714	71658	26931	196432	138063	73963	28984
137911	71658	94977	41957	143986	73963	95262	42341
121288	26931	41957	444184	118982	28984	42341	436256
Correlation Matrix				**Correlation Matrix**			
1.00	0.73	0.65	0.26	1.00	0.74	0.65	0.25
0.73	1.00	0.64	0.11	0.74	1.00	0.64	0.12
0.65	0.64	1.00	0.20	0.65	0.64	1.00	0.21
0.26	0.11	0.20	1.00	0.25	0.12	0.21	1.00

Table 3-1: The mean vector, covariance matrix, and correlation matrix for the classes *win* and *loss* at move 40.

3.3.2 The Evaluation Stage

The evaluation of a board position involves the computation of the features, and then the combination of this feature vector, **x**, into a final evaluation. From Section 3.1.2, we know that the two discriminant functions for *win* and *loss* are:

$$g_{win}(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_{win})^t\Sigma_{win}^{-1}(\mathbf{x}-\mu_{win}) - \frac{N}{2}\log 2\pi - \frac{1}{2}\log|\Sigma_{win}| + \log P(win) \tag{3.7}$$

$$g_{loss}(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_{loss})^t\Sigma_{loss}^{-1}(\mathbf{x}-\mu_{loss}) - \frac{N}{2}\log 2\pi - \frac{1}{2}\log|\Sigma_{loss}| + \log P(loss) \tag{3.8}$$

The evaluation function should measure the likelihood that the board position belongs to the class *win*, or:

$$g(\mathbf{x}) = \frac{P_{win}}{P_{loss}} = g_{win} - g_{loss} \tag{3.9}$$

We now substitute (3.7) and (3.8) into (3.9). The constant $\frac{N}{2}\log 2\pi$ are canceled. Also, we will assume that the *a priori* probability of winning and losing are equal, and eliminate $\log P(win)$ and $\log P(loss)$. This results in our final evaluation function:

$$g(\mathbf{x}) = (\mathbf{x}-\mu_{win})^t\Sigma_{win}^{-1}(\mathbf{x}-\mu_{win}) - (\mathbf{x}-\mu_{loss})^t\Sigma_{loss}^{-1}(\mathbf{x}-\mu_{loss}) + \log|\Sigma_{win}| - \log|\Sigma_{loss}| \qquad (3.10)$$

The term $\log|\Sigma_{win}| - \log|\Sigma_{loss}|$ is a constant used to normalize the quantity, and is not strictly necessary if all evaluations use the same constant. But as stated above, a different set of parameters is estimated for each stage of the game; therefore, eliminating this term would result in different evaluation ranges on different levels of the search tree. That would be inconvenient because some thresholds in our search require a consistent range [7]. Thus, the term is retained. Furthermore, by retaining this term, when the program reports its evaluation, it is possible to compute the *probability of winning* directly from $g(\mathbf{x})$: how????

$$\frac{P(win\,|\,\mathbf{x})}{P(win\,|\,\mathbf{x}) + P(loss\,|\,\mathbf{x})} = \frac{e^{g_{win}(\mathbf{x})}}{e^{g_{win}(\mathbf{x})} + e^{g_{loss}(\mathbf{x})}} = \frac{e^{g_{win}(\mathbf{x}) - g_{loss}(\mathbf{x})}}{e^{g_{win}(\mathbf{x}) - g_{loss}(\mathbf{x})} + 1} = \frac{e^{g(\mathbf{x})}}{e^{g(\mathbf{x})}+1} \qquad (3.11)$$

divide both side with e^{g(x)}

Let us summarize our evaluation procedure. At every terminal node of the search, the four features are computed, and combined into $g(\mathbf{x})$ as shown in Equation (3.10). At the completion of each iteratively deepened search, $g(\mathbf{x})$ is converted to $P(win\,|\,\mathbf{x})$ as shown in Equation (3.11), which is a more meaningful measure for humans.

4. Results

In this chapter, we will describe two experiments with two versions of the Othello program, BILL. The first version is BILL 2.0, which combines the four features linearly, and is known to play at the world-championship level. The other version, BILL 3.0, uses the same four features, and combines them using Bayesian Learning described in the previous chapter.

There are many ways to evaluate and compare game-playing programs, including (1) playing the programs against each other, (2) giving problems with known solution to the programs. and (3) actual playing record or rating. The first two methods are used here, and the third was not possible due to the scarcity of opponents who play at BILL's level.

4.1 Actual Games

The most obvious measure of two programs is simply arranging them to play each other. We arranged BILL 2.0 to play BILL 3.0 under the following conditions: 100 nearly even positions with 20 discs on the board were selected from BILL 2.0's opening book. BILL 2.0 played BILL 3.0 twice from each position, once as black and once as white. Each side was given 25 minutes to make all of its moves. BILL 3.0 won 139 of the 200 games, tied 6, and lost 55. The average score was 36.95 to 27.05. This result showed that Bayesian Learning is significantly better than a fine-tuned linear evaluation function. The actual difference may be even greater because although all of the initial positions are close, one side could be destined to win. In that case, each program would score a win. this is how we know if our engine performs well or not

In order to find out exactly how much is gained from Bayesian Learning, different versions of BILL 2.0 that searched to different depths played each other from the same initial positions. The results are shown in Table 4-1. Since the above conditions allowed BILL to search 6 to 8 plies, it is valid to say that BILL 3.0 is about equivalent to BILL 2.0 with two extra plies of search. The effective branching factor in Othello is between 3.4 and 3.7, which implies that if BILL 2.0 were given 13 times as much time, it would be about as good a player as BILL 3.0.

4.2 Endgame Problems

Since Othello endgames can be solved many moves from the end of the game, it is possible to assess the strength of a program by the frequency with which it selects (without searching to the end) the move that leads to the optimal result. A problem with this scheme is that as endgame is approached, the less applicable are the known Othello strategies, and sometimes counter-intuitive moves have to be made. In order to minimize this problem, we acquired a database of 63 winning positions with 20 to 24 moves left, each with the

Players	Win	Tie	Loss	W/L	Avg. Score
7-ply vs. 6-ply	121	7	72	1.68	34.54 - 29.39
8-ply vs. 7-ply	115	6	79	1.46	35.04 - 28.89
7-ply vs. 5-ply	141	10	49	2.88	37.03 - 26.94
8-ply vs. 6-ply	130	13	57	2.28	36.38 - 27.59
Learn vs. Linear	139	6	55	2.53	36.95 - 27.03

Table 4-1: Results between two versions of BILL.

move that leads to the win with the largest margin. This database was generated by a hardware endgame searcher built by Clarence Hewlett [6].

The linear version and the Bayesian Learning version of BILL were given these problems, and each suggested a best move using 3 to 8 plies of search. The frequency that they agreed with the optimal move is shown in Table 4-2. Under tournament conditions, BILL could search about 8 plies at this stage of the game. Thus, BILL 2.0 and BILL 3.0 would solve 55% and 64% of these problems, respectively, should they occur in actual tournaments.

Search Ply	Bayesian	Linear
3	51%	41%
4	51%	43%
5	53%	46%
6	57%	53%
7	61%	53%
8	64%	55%

Table 4-2: Percentage of agreement between two versions of BILL and the move that guarantees the largest winning margin.

In order to evaluate these figures, one should be aware that some of the optimal moves may still be counter-intuitive. More importantly, there are often many moves that still preserve a winning position. Unfortunately, we were not provided with sufficient information to evaluate the frequency that wins were lost due to a poor move. But the above statistics show that Bayesian Learning is indisputably stronger.

Finally, it is possible to compare these figures against expert human play. 19 of these positions were

taken from six games between the top players in the world. It was found that the human experts made 9 correct moves, and 10 incorrect ones, for an accuracy of 47.36%. From this, it is clear that both versions of BILL played significantly better than human experts. Furthermore, BILL 3.0 is far better than BILL 2.0 and the human experts.

5. Discussion

5.1 Advantages of Bayesian Learning

In this section, we will discuss the advantages of the Bayesian Learning algorithm by comparing it against Samuel's algorithms. Table 5-1 shows a comparison of Samuel's two algorithms and the Bayesian Learning algorithm. It is clear that while all three algorithms were designed to solve the same problem, there are many major differences.

	Samuel's Polynomial	Samuel's Signature Table	Bayesian Learning
Nonlinearity	No	Yes	Yes
Can deal with redundancy?	No	Possibly	Yes
Smoothness	Yes	No	Yes
Completely automatic?	No	No	Yes
Optimality	No	No	Yes
General purpose	Probably Not	No	Yes
Concept learned	Good vs. bad features	Strong vs. weak positions	Winning vs. losing positions
Learning Method	Self-play	Book-moves	Games

Table 5-1: Comparison between Samuel's Algorithms and the Bayesian learning algorithm.

The first great difference is that of linearity. Samuel's polynomial learning algorithm learns a linear function. It assumes that all features are mutually independent, which we have shown to be false. If two identical features were presented to this algorithm, both would be assigned equal weights, resulting in gross over-estimation of its utility. The signature table learning is a reasonable nonlinear approximation. However, as pointed out in Section 2.2.2, the nonlinearity in signature table learning will function only if the table structure is carefully arranged. Even then, correlations between features that are in different tables will be lost. If two identical features were not positioned properly, their utility is still overestimated. The Bayesian Learning approach, on the other hand, understands nonlinear relationships between the features by considering covariances between every pair of features. It is always possible to detect redundant features, and account for all the overlap among the features.

In the previous chapter, we saw the dramatic improvement produced by using a nonlinear evaluation function. To illustrate why a linear evaluation is inadequate, the correlations between each pair of features

are plotted in Figure 5-1. Although the four features were extracted from different characteristics of the board, they were found to be highly correlated, particularly mobility, potential mobility, and weighted squares. It would be detrimental to combine these highly correlated features linearly.

Another difference is smoothness. Samuel's polynomial learning is smooth, as it uses natural features. But the extreme quantization in signature table learning deprived it of its smoothness. The lack of smoothness results in a search space where a minute change in one feature value could significantly alter the evaluation. In contrast, Bayesian Learning learns a smooth function, which measures the likelihood that a position belongs to the class of *winning* or *losing* positions.

Another serious problem with both of Samuel's procedures is that they require additional tuning and supervision. Bayesian Learning, on the other hand, is completely automatic. Since tuning of feature-combination is very unintuitive, automation is a very desirable property. Furthermore, Bayesian Learning provides the optimal quadratic combination assuming multivariate normal distribution[4].

One other problem with Samuel's procedures is that they do not adequately account for the stages of the game. We measured the utility of each feature in Figure 5-2, which shows the fraction of the training positions correctly classified as *winning* or *losing* positions for each feature if it were used in isolation. It is clear that the stage of the game affects the relative importance of the features. The correlation changes shown in Figure 5-1 also support the need for a fine and slow-varying measure of the stage of the game, which we provide by generating a discriminant function for each stage. Note that it would be possible to modify Samuel's procedure to generate many stages of learning; however, that would increase the already very large number (180,000) of training positions.

An interesting difference between Samuel's procedures and ours is the concept that is being learned. Samuel's polynomial learning tries to distinguish good features from bad ones by penalizing the ones that make poor decisions, where the goodness of the feature is determined by its agreement with a deeper search. The signature table learning tries to differentiate strong positions from weak ones, where the strong positions result from the move made by the expert, and weak positions result from legal moves not selected by the expert. Bayesian Learning tries to learn the concepts of *winning* and *losing* positions, where *winning* positions are those that lead to an eventual win, and *losing* positions are those that lead to a loss. Since the objective of any game is to win, it seems more plausible to model *winning vs. losing positions* than *good vs. bad features* or *moves chosen by experts vs. moves not chosen by experts.*

---

[4]We will show in Section 5.2.1 that this assumption is correct.

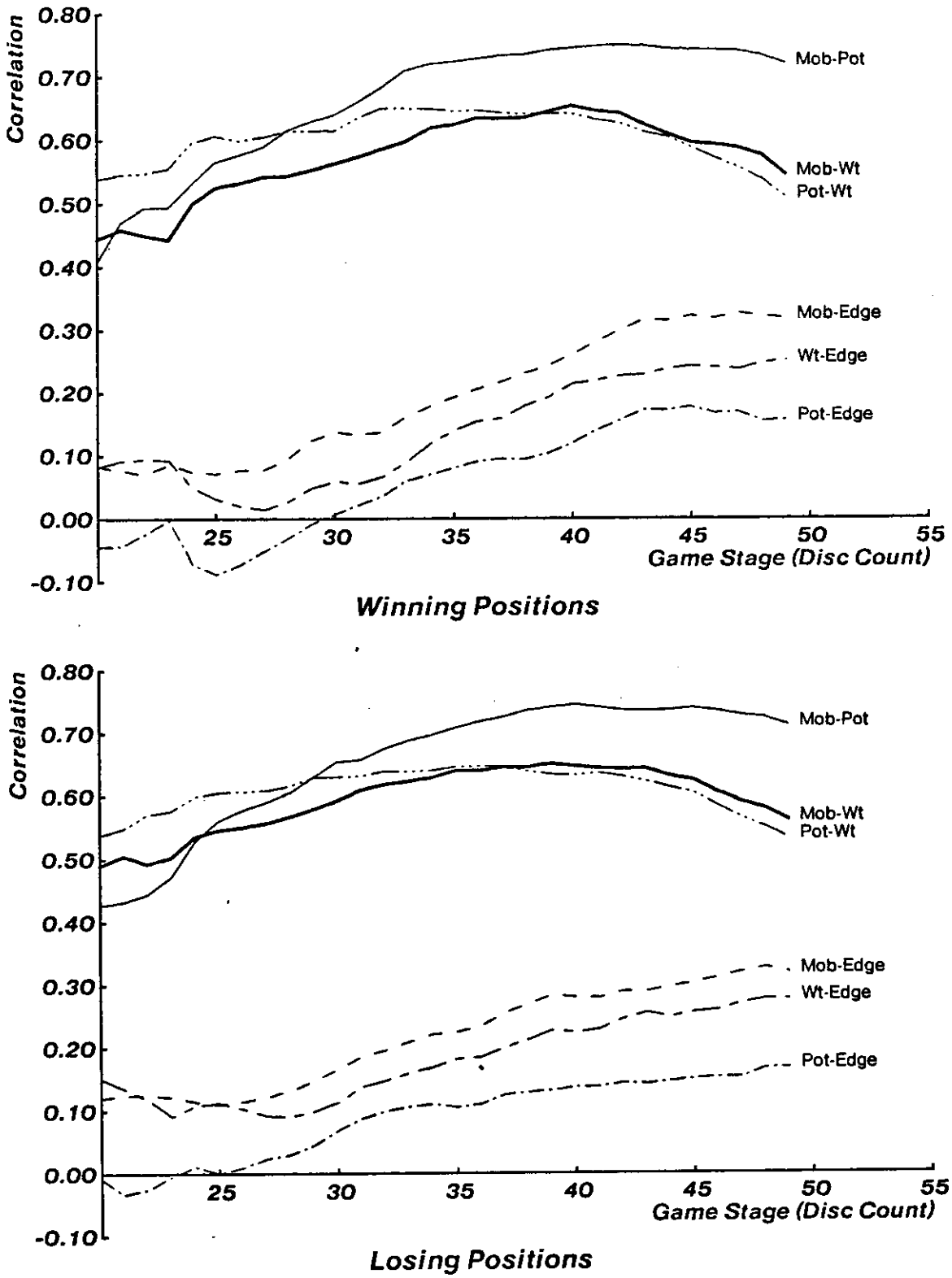


Figure 5-1: Correlation between every pair of features for winning and losing positions as a function of the stage in the game.

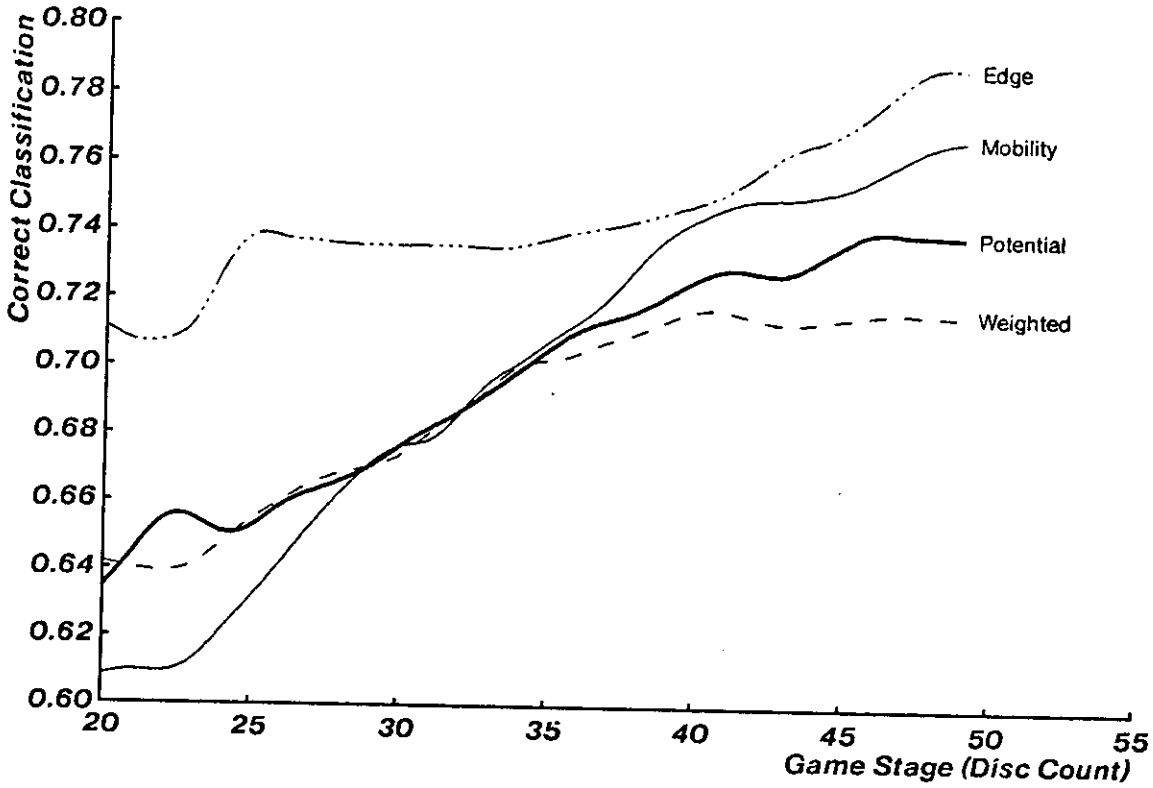


Figure 5-2: The fraction of training positions correctly classified by each feature used in isolation as a function of the stage of the game.

Finally, the method of training is different for all three algorithms. Samuel's polynomial learning algorithm used self-play to generate the training data. Since this is an incremental hill-climbing procedure, it is likely to converge to a local maximum. The signature table learning is more global; however, the training from book-move suffers from limitations. First, while expert moves usually provide good positive exemplars, using all moves not chosen by the expert as negative exemplars is misleading. Second, by learning to imitate expert moves, it is theoretically impossible for the evaluation function to play better (without searching) than the experts. In this study, the use of *winning* and *losing* positions provide good positive and negative exemplar learning. Furthermore, by modeling "moves that lead to a win" rather than "moves chosen by experts", it is theoretically possible for our evaluation to be superior to the experts who played the training games.

5.2 Problems with Bayesian Learning

5.2.1 Multivariate Normal Assumption

The simplicity and elegance of Bayesian Learning are largely due to its assumption of the underlying distribution of the data. In order for our learning algorithm to function properly, the distributions of the feature must be multivariate normal. To verify this assumption, the distribution of the four features from all 3000 training games were plotted in Figure 5-3. The thick curve is the distribution for winning positions, and the thin one is the distribution for losing positions. The positions were taken from positions with 24 empty squares on the board. It is clear from these figures that this assumption is quite reasonable.

5.2.2 Accuracy of Labeling

One point that can be raised is that the win/loss labeling procedure may not be very accurate, and any mistake in the labeling is likely to adversely affect the performance of Bayesian Learning.

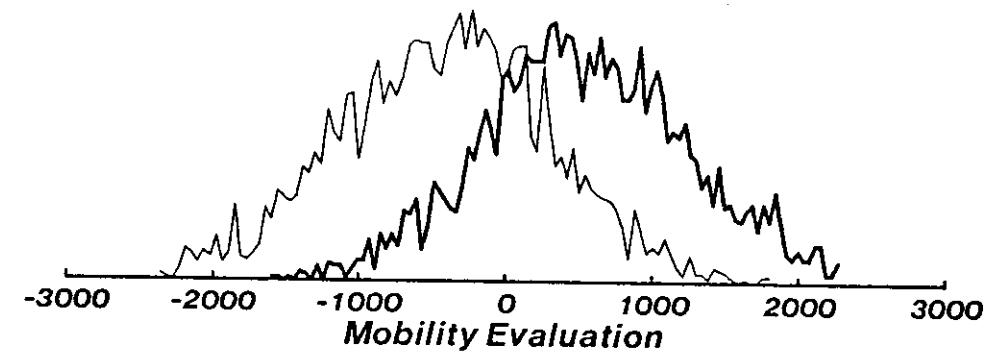
Although positions with 15 empty squares are always perfect because of BILL's endgame solving capability, the earlier positions could be in error. We feel, however, that our labeling method is reasonable because:

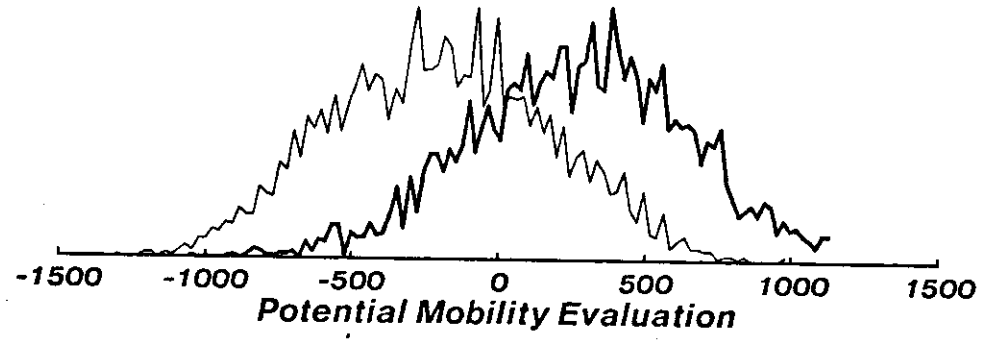
1. Many pattern classification procedures use hand-labeled training data, which are not always perfect.
2. Since BILL 2.0 probably played better than any expert, this is the best labeling mechanism available.
3. The first 20 random moves should create many positions that are not very close, and the side that is ahead will almost always win because BILL plays extremely well.
4. Nearly even positions are difficult to label; however, given sufficient training data, these positions will simply form a boundary where *win* and *loss* are difficult to differentiate.

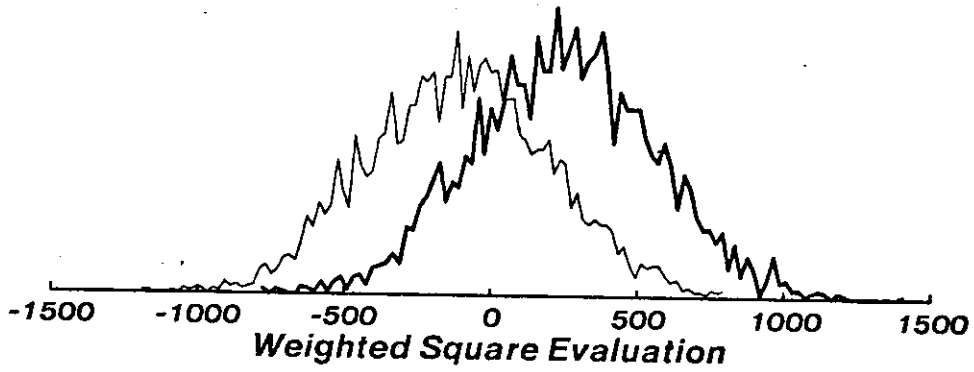
5.2.3 Efficiency of Bayesian Learning Evaluation

Perhaps the greatest problem with Bayesian Learning of evaluation function is that of efficiency. Each evaluation requires four multiplications of the inverse covariance matrix by the feature vector. Because the matrices are symmetric, the number of floating point multiplications needed to combine the features is $2N(N+1)$, where N is the number of features. Consequently, having a large number of features substantially decreases the speed of the program. Furthermore, accurate parameter estimation with many features requires much more training data.

We could deal with this problem by reducing the dimensions of the feature space using *principal components analysis* [4], which rotates the feature space into one that has independent features, and unimportant features (those with small variance) can be discarded. Another similar approach is to use *Fisher's linear discriminant* [4], which uses labeled training data to maximize the ratio between inter-class variance and intra-class variance. wtf is these methods?







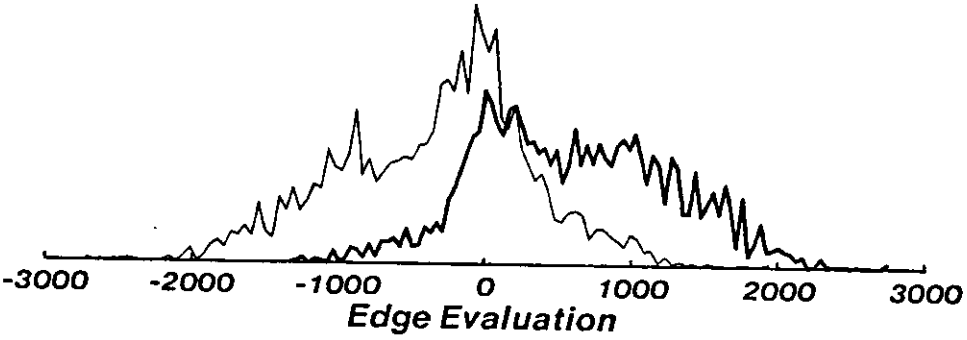


Figure 5-3: Win/Loss distribution of the four feature used.

5.3 Applicability to Other Domains

Bayesian Learning has already been applied to speech, vision, character recognition, and many other domains. This study is the first that uses Bayesian Learning to learn feature combination in an evaluation function. The key concept that enabled this application is our use of Bayesian Learning to maximally separate the classes of *winning* and *losing* positions. This algorithm is applicable to any other game or any other search-based application where static evaluation is used. In order to obtain superior results, the following conditions are necessary: (1) good features must be used, (2) it must be possible to define the goal in terms of classes, and (3) the multivariate normal distribution must provide a reasonable fit.

The first condition is needed for any program to be successful. The second condition is easy to satisfy in game-playing programs, because the classes of *winning* and *losing* positions are the ideal concepts for learning. It may be more difficult in other domains. The third condition is satisfiable in most domains.

Although we used a different version of BILL to generate the training data, it is not always desirable to do so. In games difficult for computers such as Go, self-generation will lead to many mislabeled positions. But in that case, using games between superior players should lead to even better performance; however, it was not possible here because it is questionable that such a player existed in Othello. But in Go, where humans are far superior to programs, training with expert games will improve the level of play even more drastically.

6. Conclusion

In this paper we presented a new algorithm for combining terms, or features, of an evaluation function. This algorithm is based on Bayesian Learning. First, a training database is obtained, and each position is labeled as *winning* or *losing*. These positions are used to train a discriminant function that evaluates positions by estimating the probability that a position is a *winning* one.

While machine learning of evaluation functions has been studied, algorithms such as Samuel's suffer from lack of smoothness, excessive human tuning, and lack of generality. The Bayesian Learning algorithm eliminates these problems, and has a number of desirable properties:

1. Completely automatic learning from training data.
2. Optimal quadratic combination.
3. Understanding of feature covariances.
4. Capability of recovering from erroneous features.
5. Evaluation directly estimating *the probability of winning.*

We demonstrated that Bayesian Learning significantly improved the playing ability of an Othello program that already played at the world-championship level. We believe that it can be applied to any domain where a static evaluation is needed, and will not only drastically reduce the tuning time, but also dramatically improve the performance of the program.

Acknowledgments

The author wishes to thank Sanjoy Mahajan for suggestions and for programming support; Hans Berliner for helpful discussions and reading drafts of this paper; and Roy Taylor and Beth Byers for reading drafts of this paper.

References

1. Berliner, H., Ebeling, C.. "The SUPREM Architecture: A New Intelligent Paradigm". *Artificial Intelligence 28.* , 1. (February 1986), 3-8.

2. Berliner, H. On the Construction of Evaluation Functions for Large Domains. Proceedings of IJCAI-79, 1979, pp. 53-55.

3. Condon, J.H., Thompson, K. Belle Chess Hardware. In *Advances in Computer Chess 3*, Clark, M.R.B., Ed., Pergamon Press, Oxford, 1982.

4. Duda, R., Hart, P.. *Pattern Classification and Scene Analysis.* New York: Wiley, 1973.

5. Griffith, A.K. "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to the Game of Checkers". *Artificial Intelligence 5,* , 1. (Spring 1974), 137-148.

6. Hewlett, C. "Report on a Hardware Computing System Dedicated to the Game of Othello". *Othello Quarterly 8,* , 2. (Summer 1986), 7-8.

7. Lee, K., Mahajan, S. BILL : A Table-Based Knowledge-Intensive Othello Program. Carnegie-Mellon University, April, 1986.

8. Mitchell, D. Using Features to Evaluate Positions in Experts' and Novices' Othello Games. Master Th., Northwestern University,July 1984.

9. Newell, A., Simon, H., and Shaw, C. Chess playing programs and the problem of complexity. In *Computers and Thought*, Feigenbaum, E.A., Feldman, J., Ed., McGraw-Hill, 1963.

10. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, 1984.

11. Rosenbloom, P. S. "A World-Championship-Level Othello Program". *Artificial Intelligence 19* , 3. (November 1982), 279-319.

12. Samuel, A. L. "Some Studies in Machine Learning Using the Game of Checkers". *IBM Journal* , 3. (1959), 210-229.

13. Samuel, A. L. "Some Studies in Machine Learning Using the Game of Checkers. II". *IBM Journal* , 11. (November 1967), 601-617.

14. Slate, D. J., Atkin, L. R. CHESS 4.6 -- The Northwestern University Chess Program. In *Chess Skills in Man and Machine,*
Springer-Verlag, 1977, pp. 101-107.

Figures and Tables

A Pattern Classification Approach
to Evaluation Function Learning

Kai-Fu Lee
Computer Science Department
Carnegie-Mellon University

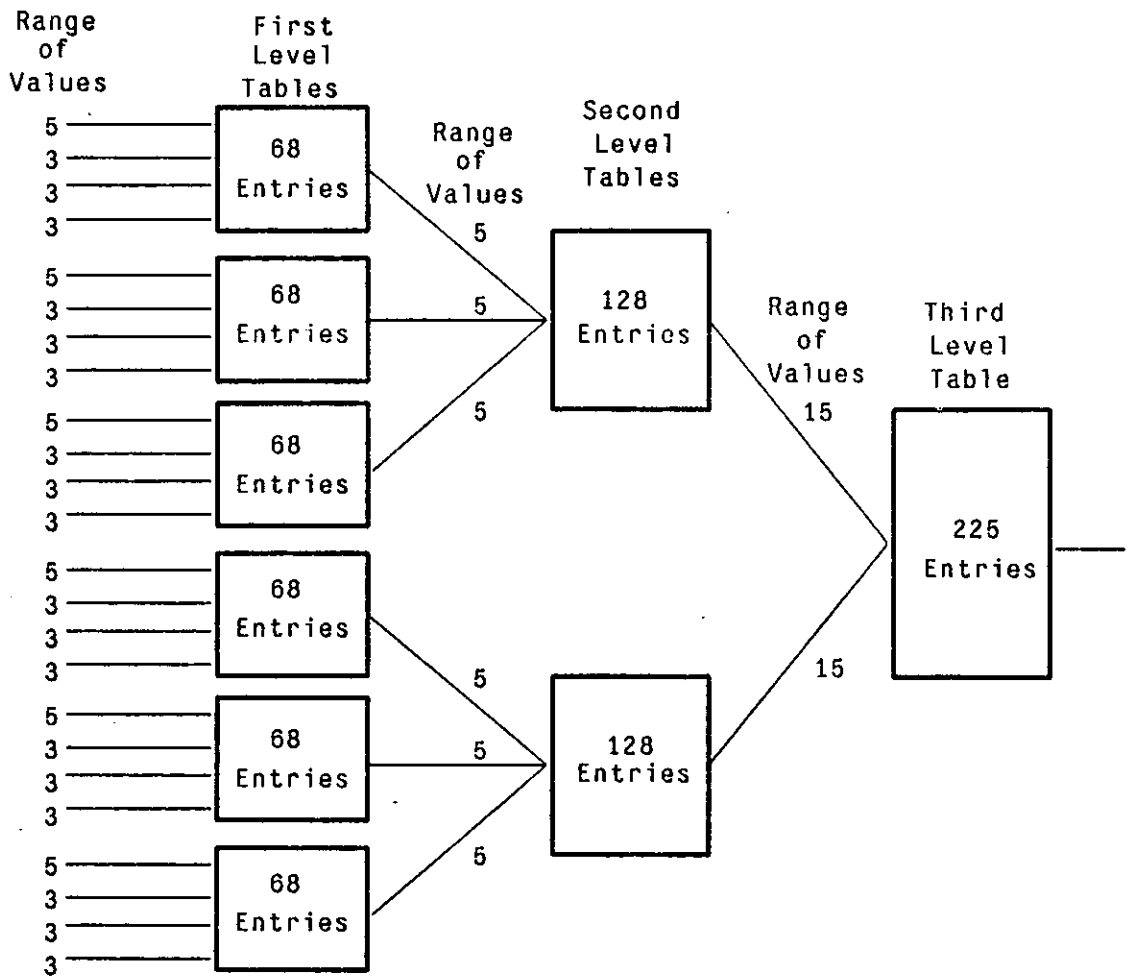


Figure 2-1: Samuel's final signature table scheme.

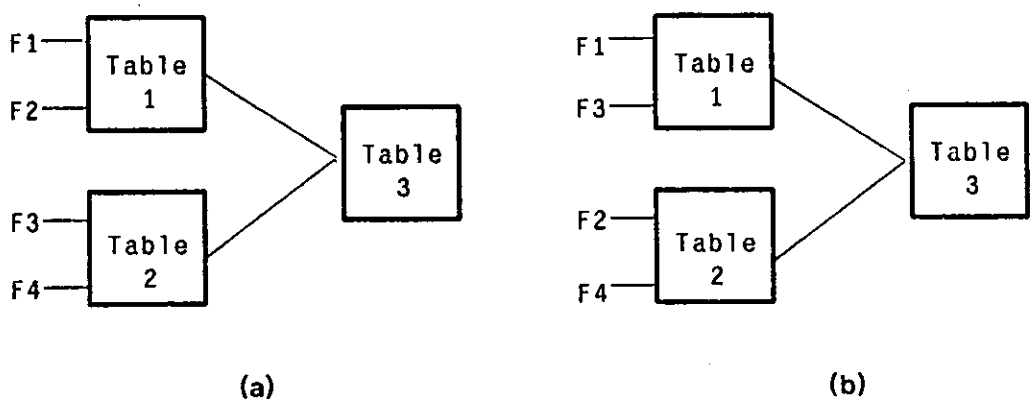


Figure 2-2: If F1 and F2 are the same feature, the signature table configurations in (a) would cancel their redundancy, but the one in (b) would not.

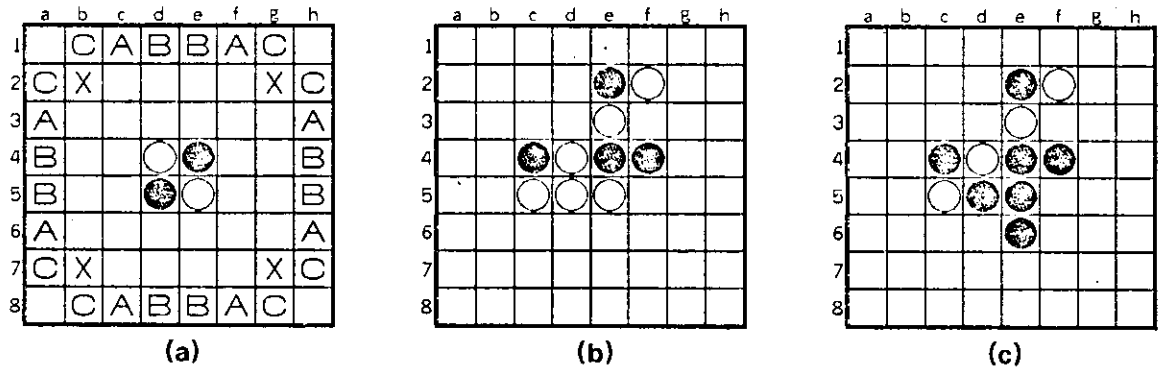


Figure 3-1: (a) shows the initial Othello board set-up and the standard names of the squares; (b) shows a sample board with legal moves (for Black) to C6, D6, D2, E6, and G2; (c) shows the board after Black plays to E6.

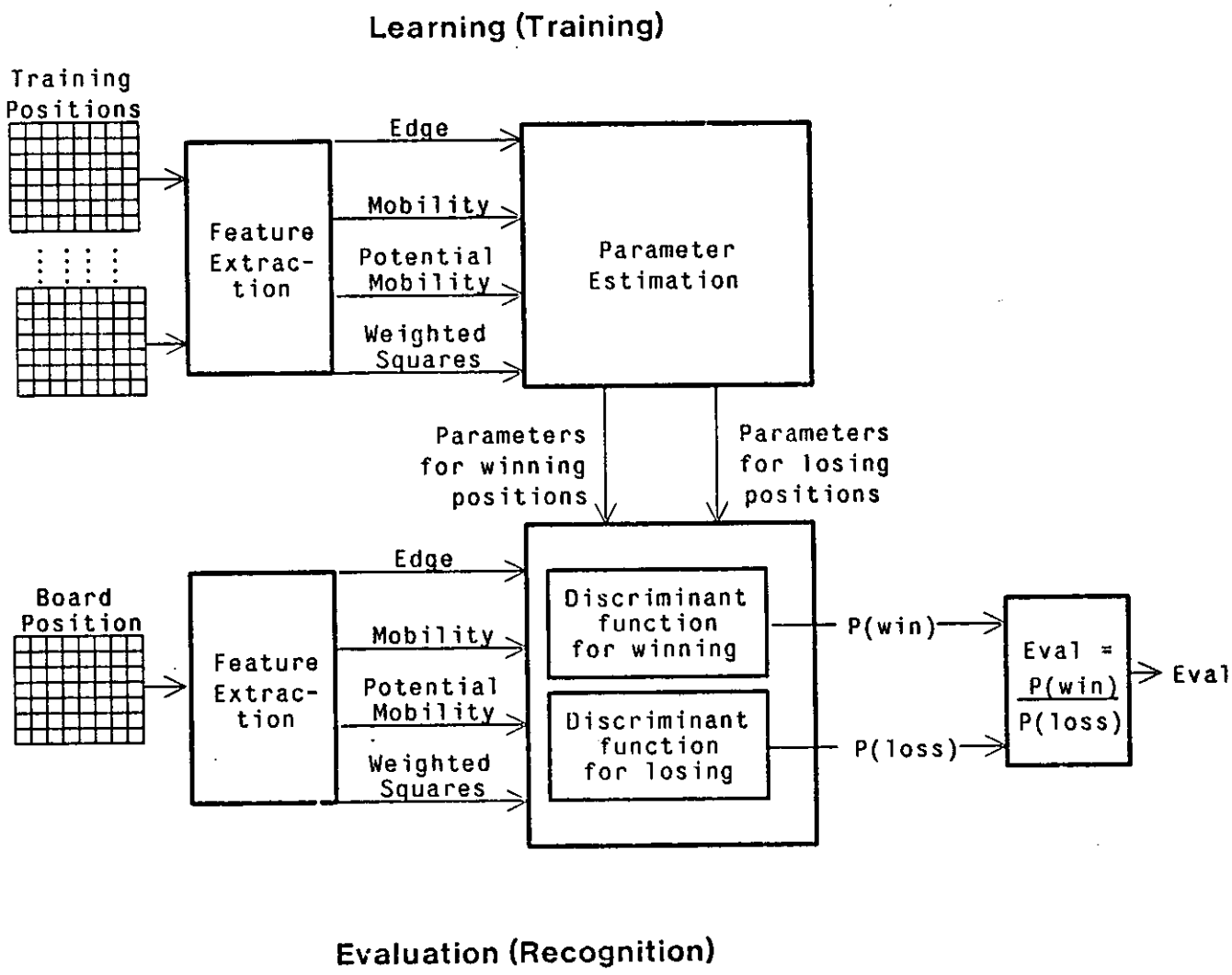


Figure 3-2: The learning and evaluation process of the proposed evaluation function learning algorithm based on Bayesian Learning.

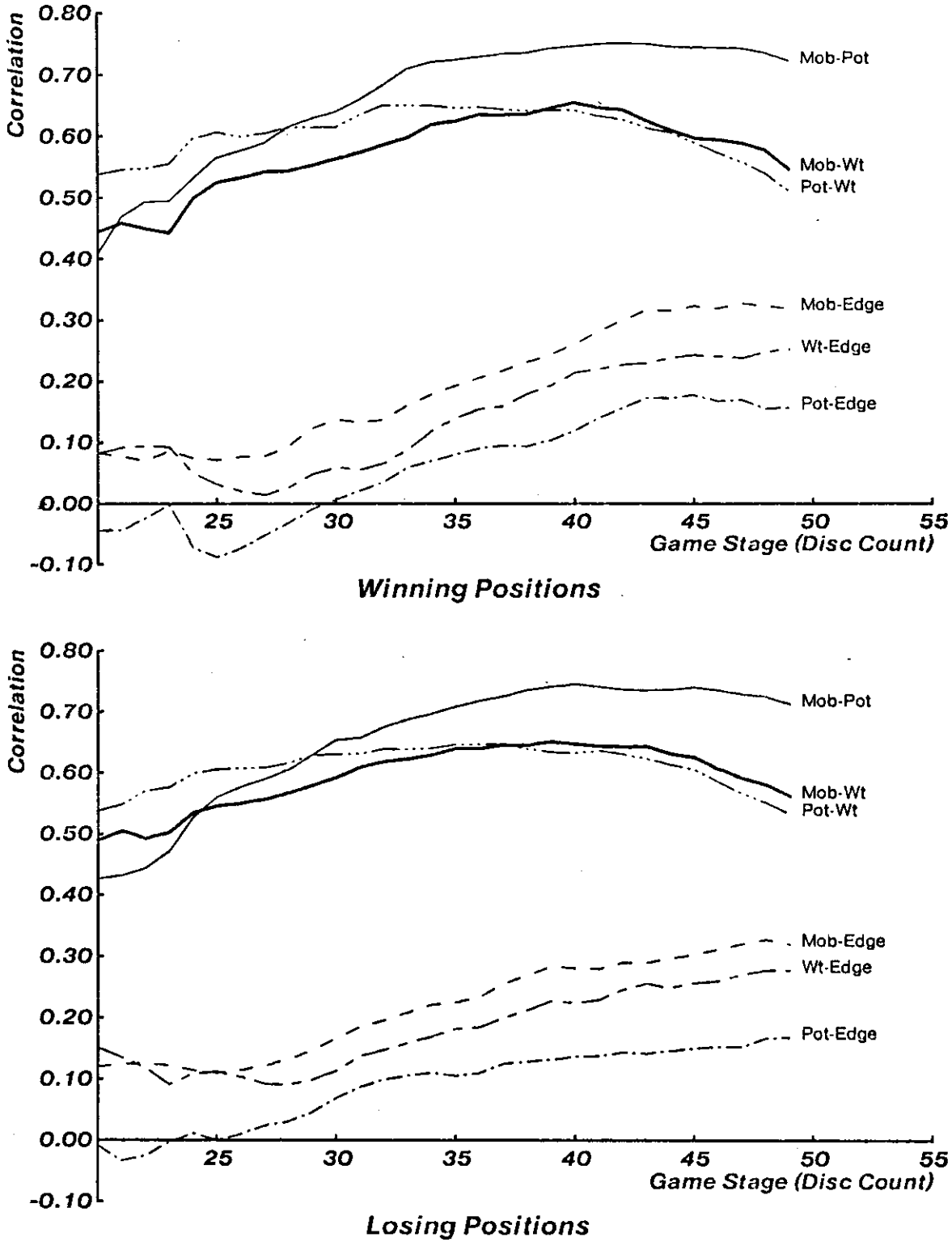


Figure 5-1: Correlation between every pair of the features for winning and losing positions as a function of the stage in the game.

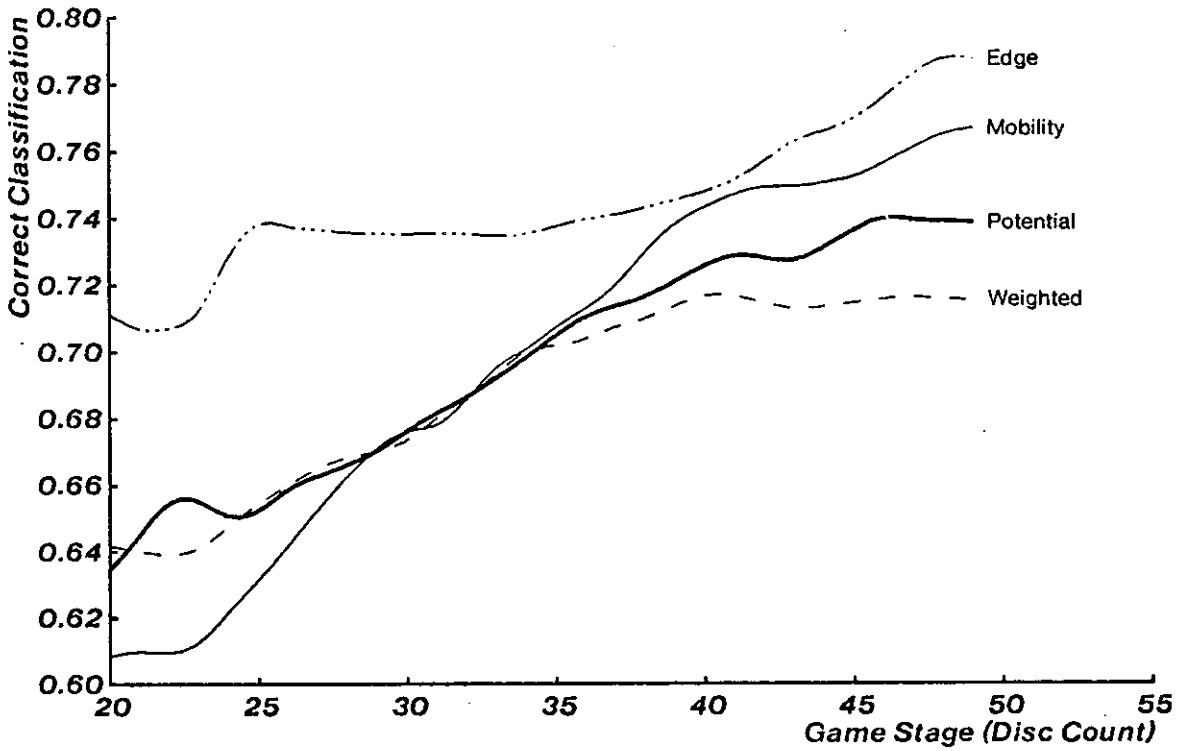


Figure 5-2: The fraction of training positions correctly classified by each feature used in isolation as a function of the stage of the game.

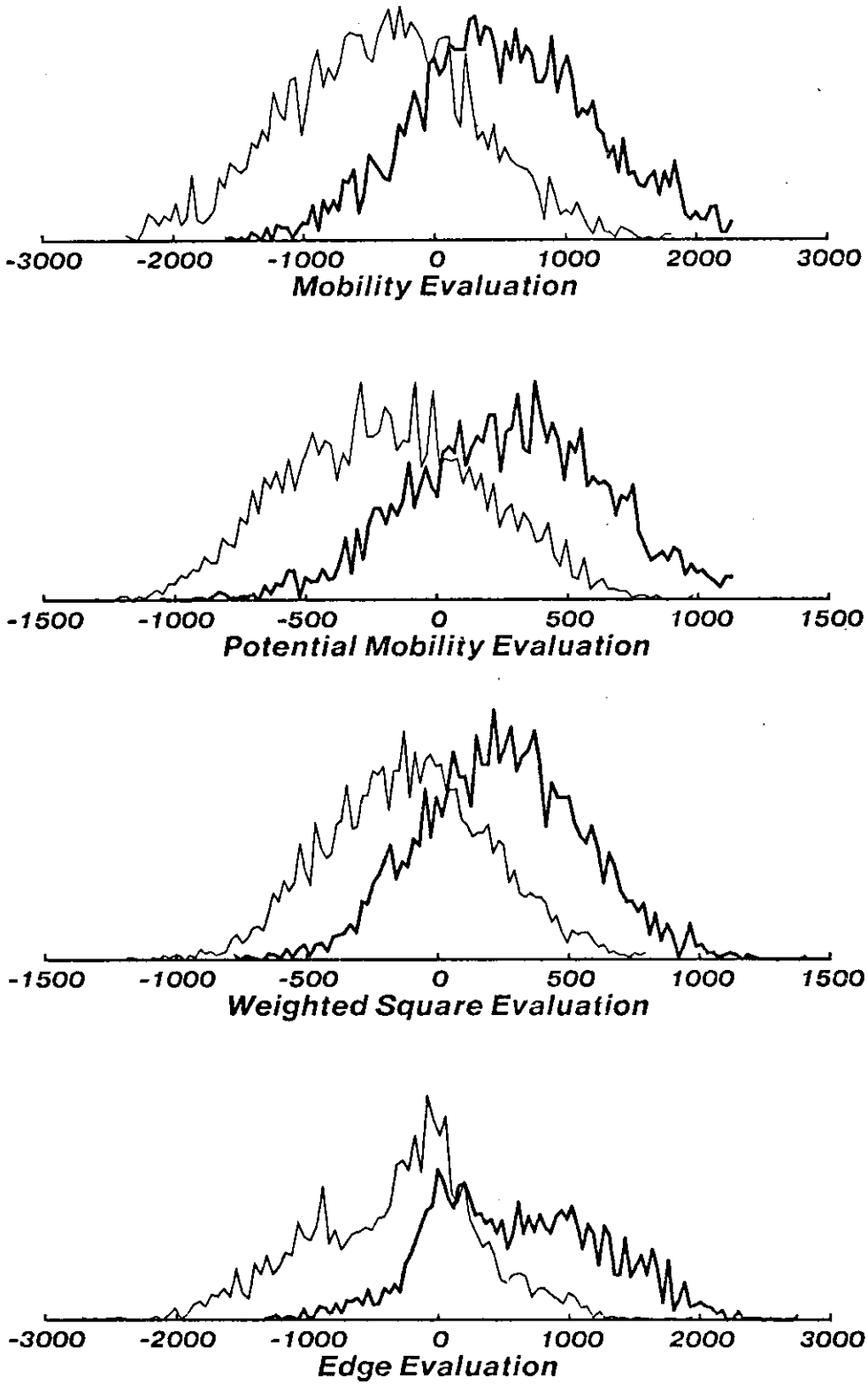


Figure 5-3: Win/Loss distribution of the four feature used.

WIN				LOSS			
Mob.	Pot.	Wtd.	Edge	Mob.	Pot.	Wtd.	Edge
Mean				**Mean**			
533	266	242	599	-419	-200	-112	-334
Covariance Matrix				**Covariance Matrix**			
475871	185298	137911	121288	508045	196432	143986	118982
185298	133714	71658	26931	196432	138063	73963	28984
137911	71658	94977	41957	143986	73963	95262	42341
121288	26931	41957	444184	118982	28984	42341	436256
Correlation Matrix				**Correlation Matrix**			
1.00	0.73	0.65	0.26	1.00	0.74	0.65	0.25
0.73	1.00	0.64	0.11	0.74	1.00	0.64	0.12
0.65	0.64	1.00	0.20	0.65	0.64	1.00	0.21
0.26	0.11	0.20	1.00	0.25	0.12	0.21	1.00

Table 3-1: The mean vector, covariance matrix, and correlation matrix for the classes *win* and *loss* at move 40.)

Players	Win	Tie	Loss	W/L	Avg. Score
7-ply vs. 6-ply	121	7	72	1.68	34.54 - 29.39
8-ply vs. 7-ply	115	6	79	1.46	35.04 - 28.89
7-ply vs. 5-ply	141	10	49	2.88	37.03 - 26.94
8-ply vs. 6-ply	130	13	57	2.28	36.38 - 27.59
Learn vs. Linear	139	6	55	2.53	36.95 - 27.03

Table 4-1: Results between two versions of BILL.

Search Ply	Bayesian	Linear
3	51%	41%
4	51%	43%
5	53%	46%
6	57%	53%
7	61%	53%
8	64%	55%

Table 4-2: Percentage of agreement between two versions of BILL and the move that guarantees the largest winning margin.

	Samuel's Polynomial	Samuel's Signature Table	Bayesian Learning
Non-Linearity	No	Yes	Yes
Can deal with redundancy?	No	Maybe	Yes
Smoothness	Yes	No	Yes
Completely automatic?	No	No	Yes
Optimality	No	No	Yes
General purpose	Probably Not	No	Yes
Concept learned	Good vs. bad features	Strong vs. weak positions	Winning vs. losing positions
Learning Method	Self-play	Book	Games

Table 5-1: Comparison between Samuel's Algorithms and the Bayesian learning algorithm.