Classifying Chess Positions

Christopher De Sa

December 14, 2012

Chess was one of the first problems studied by the AI community. While currently, chess-playing programs perform very well using primarily search-based algorithms to decide the best move to make, in this project I apply machine-learning algorithms to this problem. Specifically, instead of choosing which move is the best to make, I want to produce an function that attempts to determine the probability that a player is likely to win in a given chess position. Note that while searching chess engines also produce a score factor for each position, this score represents the engines own belief (in a Bayesian sense) that it will win the game given the position, whereas our goal is to classify the actual probability of a win given human players.

The main possible application of such a classifying function would be as a heuristic in an A*-like search-based chess engine. Additionally, the structure of the classifier could shed insights on the nature of the game as a whole.

I have acquired training examples from actual games played by humans. I decided to use the FICS games database, which contains over 100 million games played over the internet over a period of years. This dataset consists of games in PGN (portable game notation) format, which encodes the game as a whole rather than as a sequence of positions. Since the goal of this project is to classify positions, I needed to convert these PGN games to position sequences, and used a python script to do so. This presented a technical challenge due to the fact that a sequence of positions is several orders of magnitude larger (in terms of memory consumption) than the PGN-encoded games. By using specialized solvers, such as the stochastic subgradient method, I was able to avoid storing all the positions in memory at once.

Since these positions are played by humans, and humans have a wide distribution of skill levels and play styles, any results from these data will depend on how the data are filtered. For this project, I am only pre-filtering these data by excluding (1) fast games in which the amount of time remaining for each player would be a spoiler factor for the classier, (2) games in which either player forfeited on time, (3) games in which either player forfeited due to network disconnection, (4) extremely short games, and (5) games resulting in a draw. This last exclusion is done in order to use a binary classifier for this problem; however, my approach could be extended to include drawn games.

Formally, we can express this problem as a ML problem as follows: our content $x^{(i)}$ is a legal chess position from a game played by humans, and our annotation $y^{(i)}$ is the outcome of that game (a win or loss by the player- to-move). We are trying to predict the expected value of the outcome of the game given the position. Note here that, due to the fact that humans are playing these games, the result of the game is not a mathematical function of the board state. Furthermore, the nature of chess is such that the vast majority of positions encountered by the algorithm will not necessarily favor either color, so they will both not be useful as training examples for the classifier, and also raise the error rate when the classifier is tested. Because of these factors, it will be impossible for any classifier for this problem to produce a near-zero error rate over this dataset.

The approach will necessitate representing the board state as some multidimensional vector of features. After investigating several possible feature sets, I settled on representing the board

state as a sparse vector, where each entry represents the presence of a particular piece on a particular square.

Let's look in more depth at this representation. A chess position consists of 64 squares, each of which may contain a piece. There are 6 pieces: the **p**awn, the k**n**ight, the **b**ishop, the **r**ook, the **q**ueen, and the **k**ing. Additionally, each piece is one of two colors: white or black. This creates a total of 12 distinct pieces. We can therefore (almost) fully represent a board state as a partial function in:

$$f(s) \in (\{1, 2, \ldots, 8\} \times \{1, 2, \ldots, 8\}) \to (\{\mathbf{P}, \mathbf{N}, \mathbf{B}, \mathbf{R}, \mathbf{Q}, \mathbf{K}\} \times \{\mathbf{White}, \mathbf{Black}\})$$

If this partial function describes our game state, it follows (from the definition of a partial function) that we can equivalently describe the game state as a subset of:

$$(\{1, 2, \ldots, 8\} \times \{1, 2, \ldots, 8\}) \times (\{\mathbf{P}, \mathbf{N}, \mathbf{B}, \mathbf{R}, \mathbf{Q}, \mathbf{K}\} \times \{\mathbf{White}, \mathbf{Black}\})$$

Since this set has magnitude $8 \times 8 \times 6 \times 2 = 768$, it follows that we can represent any subset of it as a vector in $\mathbb{R}^{768}$, where the i-th entry is either 0 or 1 to represent the presence or absense of the i-th element of this set in the subset.

There are some caveats with this representation. The partial function does not include certain non-position-based aspects on the game state, namely: whose turn it is, whether castling is still possible, and whether *en passant* is possible. I have assumed that the latter two things are insignificant enough that they can be safely omitted as features. The former is more difficult to deal with, but it can be resolved by considering the symmetry of the chess board. Because of this symmetry, if we reverse: (1) the board, (2) the colors of all the pieces, and (3) the result, our classifier should produce the same result. Therefore, without loss of generality, we can limit ourselves to only positions in which white is to move, and for these positions, this representation works well. (For all the analysis below, we will use "white" to refer to the player who is to move in the position, and "black" to refer to the player who is not to move.)

We assume that, at any given position, the probability that a human player to move will win the game is a function of the position. This is a reasonable assumption based on the structure of the chess game. If we further constrain this function to be a logistic function of the above board representation with some parameter θ, then the maximum likelihood estimate of this parameter corresponds to logistic regression. Therefore, I trained a logistic classifier using this representation. (I also tried a few other classifiers, such as a linear classifier and a support vector machine; however, these classifiers did not perform well on the given dataset. This is to be expected since the basic properties that we want to have for applying these methods do not hold for this dataset.)

I ran logistic regression over a dataset of about $500,000$ training samples and $500,000$ test samples. My algorithm was written in python using numpy and scipy, and computed the regression using the Newton-Raphson method. It dealt with the relatively-large amount of data both by using sparse matrices whenever possible (in particular, the training matrix X, whose columns are the sparse position vectors, was represented with a sparse matrix), and by avoiding the Hessian matrix inverse step by instead using the conjugate gradient method on the linear system $Hs = g$ (where s is the computed step, g is the logistic gradient, and H is the logistic Hessian). This solver (figure 1) converged in about 7 iterations for most of the data sets tested.

The resulting classifier had:

$$\epsilon_{\text{train}} = 0.316266 \qquad\qquad \epsilon_{\text{test}} = 0.336103$$

The similarity between these numbers suggests that the model is not overfitting the data. While these error rates are not great, the fact that they are significantly lower than 50% for a

```
def train_logistic_regression(X,y):
    theta = numpy.asmatrix([0] * X.shape[1]).T #initialize theta
    for i in range(itermax):
        lgrad = logistic_gradient(X,y,theta) #compute gradient
        if numpy.linalg.norm(lgrad) < tolerance:
            break
        H = logistic_hessian(X,y,theta) #compute Hessian
        step = numpy.linalg.lstsq(numpy.asarray(H), lgrad)[0] #Newton step
        theta = theta - numpy.asmatrix(step) #perform update
    return theta
```

Figure 1: Logistic regression solver code

large dataset of positions suggests that the logistic classifier produced a result that would be a good heuristic for position value.

One advantage that our sparse representation offers us is that it allows us to average the calculated weight parameters for a given piece. This, in turn, allows us to determine how much the logistic classifier "values" a given piece.

When we look at the weights associated with a given piece, averaged over all squares, and normalized to have the weight of the white knight be 3, the linear regression resulted in:

$$w_{\mathbf{WhitePawn}} = 1.122772 \qquad w_{\mathbf{BlackPawn}} = 1.125155$$
$$w_{\mathbf{WhiteKnight}} = 3.000000 \qquad w_{\mathbf{BlackKnight}} = 3.118595$$
$$w_{\mathbf{WhiteBishop}} = 3.379214 \qquad w_{\mathbf{BlackBishop}} = 3.442213$$
$$w_{\mathbf{WhiteRook}} = 4.822891 \qquad w_{\mathbf{BlackRook}} = 4.908966$$
$$w_{\mathbf{WhiteQueen}} = 8.998281 \qquad w_{\mathbf{BlackQueen}} = 9.157469$$

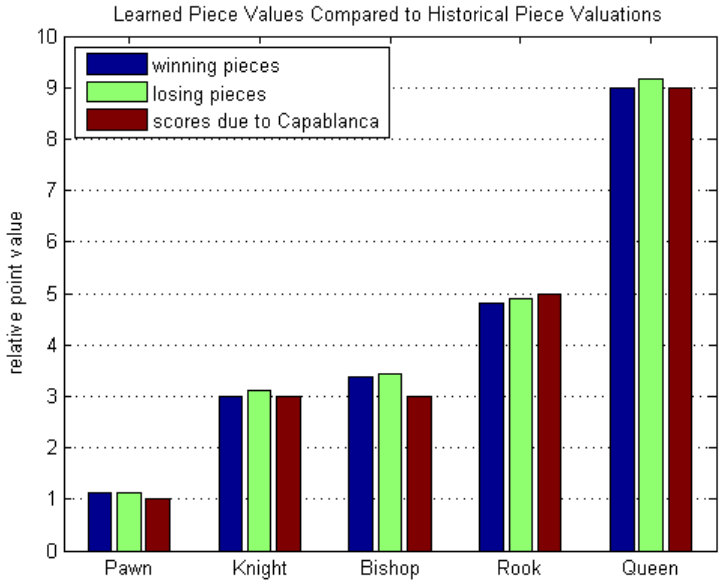

Figure 2: Piece valuations according to the logistic classifier

This is very similar to the well-known system of valuing pieces in chess, which values pawns at 1, knights at 3, bishops at 3, rooks at 5, and queens at 9 (figure 2). The fact that the linear regression independantly reproduced a valuation that was very similar to this system suggests that it is performing some useful classification.

A pawn is a special piece in chess. Once it moves forward, it cannot move backward, and it is generally (bar capturing) restricted to moving within a single file. Since it doesnt move around a lot, it is reasonable to use pawns to study the value of holding a particular square on the board. Here, we cut by both rank and file and look at the resulting valuations of pawns (figure 3).

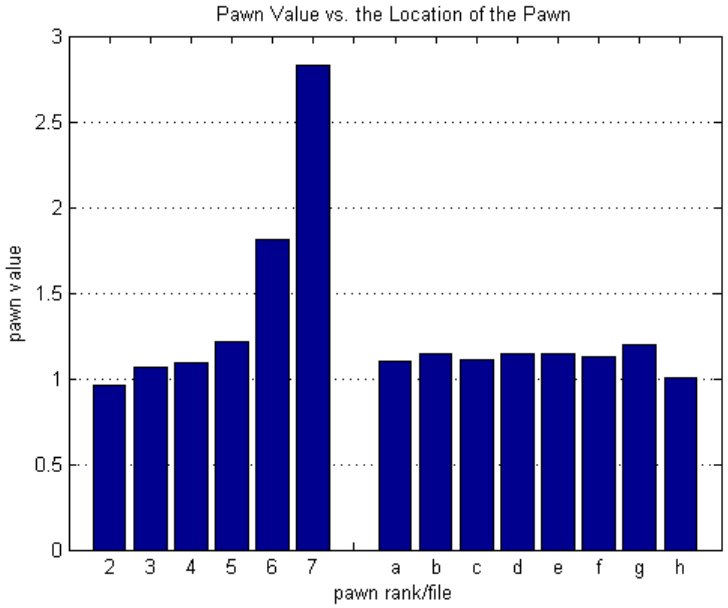

Figure 3: Pawn valuations by rank and file according to the logistic classifier

Notice that, for pawns, their value increases greatly as the move forward on the board, but does not vary much as a function of their file. This makes sense since the further a pawn is along the board, the more likely it is to either be threatening the opponent, or to be promoted to a more valuable piece. It is also interesting that the classifier seems to value b- and g-pawns higher than c- and f- pawns. This is counter to the expected notion that pawns are more valuable the closer they are to the center of the board. One possible reason for this result could be that the absense of a b- or g- pawn indicates damage to the local pawn structure, and suggests the future loss of surrounding pawns. Another possibility is that since c- and f- pawns are routinely sacrificed for other forms of compensation in the opening stages of the game, their loss is valued less by the classifier than pawns in other files.

Now, we still have a relatively large error rate on this classifier. While much of this error is probably explained by player error in the individual games, we might want to ask to what degree this is true. If we assume that player error happens relatively uniformly over time, it follows that the closer a particular position is to the end of the game, the less likely it is that a blunder occured in the time between the position and when the game ended. Such a blunder will cause the position to be "mislabeled", in the sense that it will be labeled with the outcome that was actually less likely to occur. This mislabeling will increase the error rate of the classifier. Therefore, if we filter the original dataset to only include positions from the last n moves of a game, we would expect, as n decreases, for the classifier error to also decrease. Below, we filter the positions in this way, and plot the classifier error as a function of move filtering threshold (figure 4).

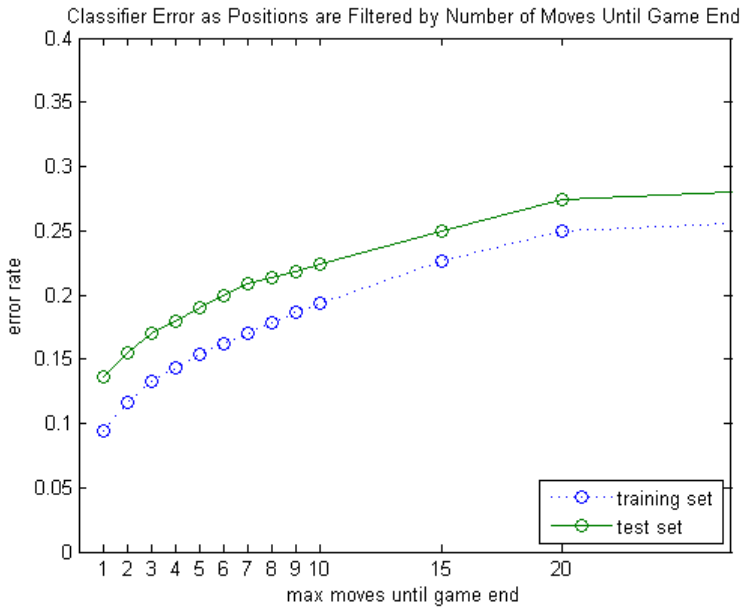

Figure 4: Classifier error as move filtering threshold increases

Since the error rate drops to about 14% within a single move of the end of the game, this suggests that about half of our classifier error in the general case was due to player blunder causing a "mislabeling" of the test data. (Of course, even this figure does not completely rule out the presence of blunders in the test data, since a human error can still occur in the move preceding the end of the game. In fact, this is a relatively common case, where a player who made a bad move will immediately resign as a result of this move.) The fact that our classifier produces this curve is an indication that it is performing well, and not overfitting individual subsections of the data.

We can also intrepret this curve as showing the rate at which players make blunders over time, as a function of distance from the end of the game. One possibly interesting avenue of future research would be to look at how this curve varies for players of different strengths. One would expect stronger players to blunder less frequently, and thus for the curve to be flatter, but in fact this may not be the case.

Even for the general case of positions, this classifier performs relatively well, managing to predict the winner in two-thirds of positions. Considering the large number of drawn or unclear positions, this seems like an impressive feat. From our analysis of human error, it seems like about 10% of the classifier error is due to human error in the test set. The rest is likely due to unclear or sharp positions that aren't easily understood by a position-based classifier.

My algorithm managed to learn, to a high degree of accuracy, a system of piece weights that has been known to players for centuries. Its results for pawn placement values also seem to be interesting, especially in regards to variance by file, where the classifier results run somewhat counter to established wisdom on the subject.

In the future, it could be interesting to try to find novel features for this problem, although I was unable to find anything that produced good results. It also might be interesting to look at the performance of this function as a heuristic in a chess engine, and to then have that engine play and somehow use reinforcement learning to feedback and modify the heuristic based on the results of this play.