The Open University of Israel
Department of Mathematics and Computer Science

Utilizing Raw Moves in Chess Machine Learning Applications

Thesis submitted as partial fulfillment of the requirements
towards a M.Sc. degree in Computer Science
The Open University of Israel
Department of Mathematics and Computer Science

By
Mordechai Goren

Prepared under the supervision of
Prof. Leonid Barenboim and Dr. Mireille Avigal

March 2022

Acknowledgments

I wish to thank my supervisors,
Prof. Leonid Barenboim and Dr. Mireille Avigal
for the fruitful discussions and guidance
throughout the process of writing this thesis.

I wish to thank
Marlaina Freisthler and Andrew Freisthler
for proof-reading this work and for sharing their helpful insights

Contents

Abstract

In this work, we show that raw chess moves can be used for carrying out chess Machine Learning (ML) tasks. The natural resource for ML tasks in chess is the chess position. When playing chess, one plays according to the position and not according to the moves. In addition, position-based features are used in most published works of ML in chess.
In this work we apply usage of move-based features in 4 areas of chess automation: (1) predicting game outcome, (2) predicting human mistakes, (3) determining difficulty of chess puzzles, and (4) determining chess themes of chess puzzles (e.g., determining a chess puzzle involves a *fork* which is a tactical chess theme).
In this work, we show that the raw moves alone can be used to carry out the tasks defined above with a high rate of accuracy. For example, determining whether a puzzle involves a *fork* with an 82.1% accuracy rate, and predicting whether a player would find the correct move with a 78.2% accuracy rate. In addition, we determine which moves and what moves' attributes make the prediction work. For example, to predict the game outcome, we show it is enough to use the last few moves played. We also show that, except for the game outcome-prediction task, a move-based approach is better than a position-based approach. Lastly, we derive beneficial chess lessons, such as exposing which correct move sequences are likely to be missed.

1. Introduction

1.1 Objective - Use Raw Chess Move Features in Machine Learning

The purpose of our work is to show that the raw chess moves contain useful information and can be used as features in *Machine Learning* (ML) chess applications. In this work we apply the usage of move-based features in four types of applications in chess automation: (1) predicting game outcome, (2) predicting human mistakes, (3) determining difficulty of chess puzzles, and (4) determining chess themes of chess puzzles (e.g., determining a chess puzzle involves a *fork* which is a tactical chess theme). For these ML applications we would like to: (1) show that the raw moves alone can be used to carry out the tasks defined, (2) understand which properties of the moves allow the ML application to succeed, (3) compare the performance of the move-based approach to a position-based approach, and (4) derive useful chess lessons that are beneficial for chess study.

Using a move-based approach is innovative. The primary resource for carrying out chess automation tasks is position rather than moves. When playing chess, position is used rather than moves; and, in most documented ML tasks, position-based features are used (See [1]-[14]). Nevertheless, in this work, we would like to show that, for specific ML tasks, a move-based approach is more effective.

We found few works which use move-based features as a primary source for ML tasks. Masud *et al.* [15] describe a move-based model which predicts the game outcome. Presser and Branwen [16], Cheng [17], Noever et al. [18], and Toshniwal *et al*. [19] describe training of move-based models which play chess at a novice level. In this work, we suggest several improvements to the experiment described by Masud *et al.* [15] and expand the usage of move-based features to more areas of chess automation.

1.2 Moves Help in Human Play

Although position contains all information needed to play chess, human players use chess moves as a substantial hint. For instance, in simultaneous chess [20], the players are obligated to inform the exhibitor of their last move. This implies that the exhibitor uses the move to help make a rapid judgment of the position.

Moves can be viewed as a compressed summary of the position. In an ordinary position, many things are occurring: pieces and squares are being attacked, defended, twice attacked, pinned, rayed, and so on. Players select a single move that addresses what the player believes to be the most important theme in the position. The move played may not be the best move, but it reflects how the player sees the position. This means moves add a dynamic aspect, the intent. Moves are not only a transition from position to position, they incorporate intrinsic information. Our goal is to capture and utilize that information in ML algorithms.

1.3 Work Structure and Results

Chapter 2 summarizes documented ML applications and other documented chess automation applications according to the type of features used, positional or move-based.

Chapter 3 explains chess concepts, ML concepts, and middleware used in this work.

Chapter 4 describes models which predict the game outcome. The ability to predict the outcome of terminated games highly depends on the dataset used. When running on the FICS dataset [21], the ML model predicted correctly 94% of games. When running on the PGN-Mentor dataset [22] the model predicted 73.2% of games correctly. These experiments were set up similarly to an experiment reported by Masud *et al.* [15] where an accuracy rate of 65.6% was reported when running on the ChessOK dataset [23]. In another set of experiments, we created models which predict the outcome of live games, reaching an accuracy rate of 57% when running on the FICS dataset and 58.5% when running on the PGN-Mentor dataset.

Chapter 5 describes models predicting whether human players would find the correct move in positions where only one good move exists. The most successful model reached an accuracy rate of 78.2%.

Chapter 6 describes models used to reverse-engineer calculated attributes of the Lichess puzzle database [24]. We describe training of a model which predicts whether a puzzle is a difficult puzzle with an 81.1% success rate. We also explain how we create models which try to tell whether a chess theme is associated with a puzzle. The method was applied for nearly 30 chess themes, with an accuracy rate ranging from 57.8% to 97.3%.

In the chess applications defined in chapters 4, 5, and 6, we show moves can be used as features for carrying out the defined tasks. In addition, as described in these chapters, other experiments were conducted to compare the performance of the move-based models, to those of position-based models, and to understand which attributes of the moves are required for the move-based ML models to perform well.

Chapter 7 summarizes the work, presents conclusions, and suggests ideas for future research.

Chapter 8 lists the references.

Chapter 9 is an appendix, which lists chess findings extracted in Chapter 5 and Chapter 6.

2. Related Work

2.1 Machine Learning Applications

The vast majority of ML applications in chess use position-based features. Furnkranz [1] summarizes 10 of the first ML applications, dating back to 1977, all using position-based features. David-Tabibi *et al.* [2]-[4], present evolutionary algorithm applications used to tune parameters for a chess engine. Hauptman and Sipper [5], [6] describe genetic programming applications creating a chess program that can play endgame positions with less than 3 pieces [5] and solve mate-in-N problems [6]. David-Tabibi *et al.* [7], Oshri *et al.* [8], Sabatelli [9], Sabetelli *et al.* [10], Silver *et al.* [11], and McIlroy-Young *et al.* [12] describe the training of an *artificial neural network* that plays chess. The trained *neural networks* account for *AlphaZero* [11], *Leela* [14] and *Maia* [12], [13], which are known to be the strongest chess-playing chess engines. McIlroy-Young *et al.* [12], also describe training a model which predicts whether humans would blunder in chess. Our work describes an alternative solution.

Only a handful of ML applications in chess use move-based features. Masud *et al.* [15] describe a move-based model which predicts the game outcome. Presser and Branwen [16], Cheng [17], Noever et al. [18], and Toshniwal *et al.* [19] describe training of move-based *transformers* which can play chess. The move-based models play chess at a novice level and occasionally attempt to make illegal moves.

A notable phenomenon is the shift from manually crafted features to raw features. In earlier works [1]-[6] manually crafted are used, such as *Rook attack king file* in [3], and *is my king not stuck* in [5]. In later works raw features are used, such as the location of pieces [7]-[13], or the move attributes [15]-[19]. Our work blends into this trend and uses only the raw attributes incorporated in moves.

From a usefulness perspective, the position-based models are better, as they account for the strongest chess-playing engines. Our work elaborates on the advantages of the move-based approach: (1) Moves consume fewer resources than positions, (2) the move-based approach performs better in other areas of chess-automation.

2.2 Information Retrieval Applications

The field of *Information Retrieval* (IR) deals with retrieving information from data. Two examples of chess IR tasks are (1) searching for positions matching a search criteria and (2) finding similar positions. IR and ML both require defining which features best represent the problem they come to solve.
In IR, in contrast to ML applications, moves are more commonly used as a primary resource. Costeff [25] developed and described a flexible freeware tool called *Chess Query Language* (CQL) that allows retrieving games based on chess themes. The chess themes in CQL include both positional and move-based themes.

Ganguly *et al.* [26] and Bizjak [27] tackle the task of automatically finding similar positions. Ganguly's solution is based on positional themes, while Bizjak's solution uses both position-based and move-based themes.

3. Preliminaries

3.1 Chess Move Notation

The most common method for recording chess moves is the *Standard Algebraic Notation* (SAN). In the SAN method, moves are recorded by indicating the piece moving, the destination square, and the existence of capture or *check*. In English, the uppercase letter K is used for a king move, Q for queen, R for rook, B for bishop, N for knight, while no symbol is used for pawn moves. The coordinates of the squares are marked using a lowercase letter (a-h) for the column, and a number (1-8) for the row (See Figure 3.1). For example, *Bg5* represents a bishop moving to the square on the 7th column and 5th row.

If the move involves a capture, an 'x' sign is used between the piece and destination symbols. For example, *Nxe5* represents a knight capturing a piece on *e5*.

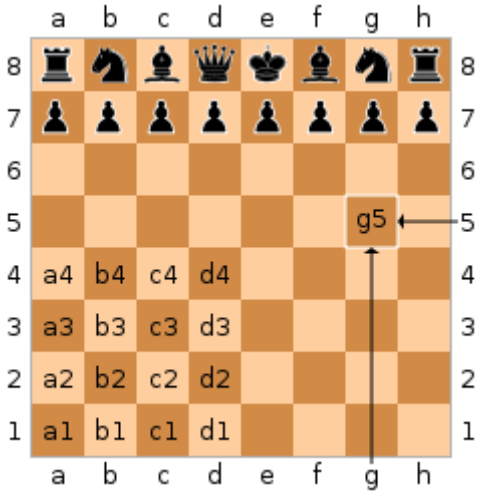
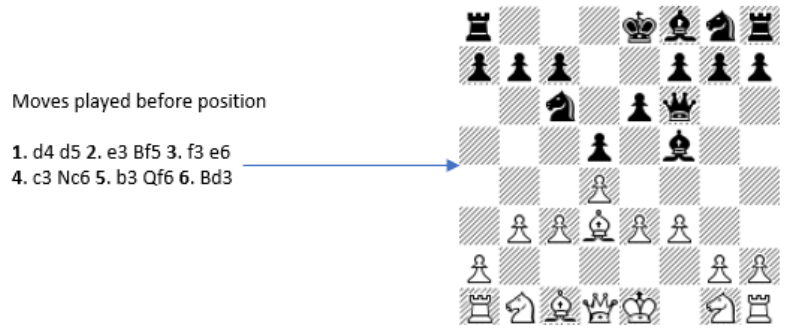


Figure 3.1 - Square coordinates [28]

In the case of a *check*, a '+' sign is appended at the end of the move. For example, *Qh5+* represents a Queen moving to *h5* and "giving *check"*. In the case of a *mate*, a '#' sign is appended at the end of the move. For example, *Rh5#* represents a Rook moving to *h5* and *mating* the opponent king.

Another common method for recording moves is the *Long Algebraic Notation*, also known as the *UCI format*. In the UCI format, the starting and ending squares are specified but not the piece moving, (e.g., *e2e4*). The UCI format is commonly used by Chess Engines like Stockfish [29].



Input Parameters:
Depth=4
MultiPV=2

Output:
Best Move: 6. ... Bxd3
Evaluation: -1.10

Principle Variation (PV)
6. ... Bxd3 7. Qxd3 h5 8. Kf1.
Score: -1.10

Second PV
6. ... Qg5 7. Bxf5 exf5 8. Kf1.
Score: -0.68

Input Parameters:
Depth=14
MultiPV=2

Output:
Best Move: 6. ... Qg6
Evaluation: -2.09

Principle Variation (PV)
6. ... Qg6 7. Ne2 Qxg2 8. Rf1 Bg6 9. Nf4 Qxh2
10. Qe2 Qxe2+ 11. Kxe2 Nf6 12. Nd2 Bd6 13. Nxg6
Score: -2.09

Second PV
6. ... Bxd3 7. Qxd3 Qg6 8. Qxg6 hxg6 9. Ba3 g5
10. Kf2 Nf6 11. Bxf8 Kxf8 12. Ne2 g4 13. Nd2
Score: -0.46

Figure 3.2 - Stockfish input and output for a given position
Top: first moves of the game and position reached.
Bottom: Stockfish input and output.
Input parameters: Depth and MultiPV.
Output: (1) Best Move, (2) Evaluation and (3) 2 best PVs.
Left: Output for depth=4; Right: Output for depth=14.
Negative evaluation represent Black has an advantage.

When counting moves, a ‘move’ usually refers to both a “half-move” by one player and a response by the other player. For example, a “20 move game” means a game where white played 20 moves and black played 20 (or 19) moves. This convention is also reflected in chess notation. When numbering the moves, each pair of white-black moves uses a single number. For example, in Figure 3.2, *1. d4 d5* represents **white** moving a pawn to d4 **followed by black** moving a pawn to d5. When a sequence starts with black's move 3 dots are used to represent the absence of a white move. For example, *6. ... Qg6 7. Ne2* represents **black** moving the queen to g6 **followed by white** moving a knight to e2. In the case described in Figure 3.2, we know the sequence starts at move 6 of the game. If the move number is unknown (e.g. we are given only the position), the numbering starts from '1', and a sequence like *6. ... Qg6 7. Ne2* is written as *1. ... Qg6 2. Ne2*

In our work, we deviate from the move counting convention and refer to each “half move” as a move. A 40-move game in our work is known as a 20-move game in the standard chess convention.[1]

[1] We suggest that the counting standard is a misnomer. In the common standard we can have an inconsistent and strange looking sentence like "this 20-move game contains 30 bad moves", where "20 move" means 40 half-moves, and "30 bad moves" means 30 half-moves. When writing this paper, we tried using the standard counting, but this created many inconsistencies

3.2 Chess Middleware

Stockfish is a free and open-source chess engine [29]. We used Stockfish for computing chess evaluation of positions and plotting the *Principle Variation* (PV) for various *depths.* PV and *depth* are explained below.

3.2.1 Chess Engine Configurable Parameters: depth and multiPV

The *depth* determines the number of half-moves (or *ply)* to look ahead. Stockfish API gives control over the depth parameter. Figure 3.2 demonstrates a case where usage of depth=14 yields a different move than when using depth=4. When selecting a depth, there is a tradeoff to consider: The higher the depth, the more accurate the evaluation, at the cost of more computation time.
The chess evaluation is measured in points and is loosely based on the 1-3-3-5-9 system which assesses the value of a Pawn as 1, a Knight and Bishop each as 3, a Rook as 5, and a Queen as 9. The computed evaluation refers to the resulting position given that both players perform the best moves for a given *depth*.

The PV is a sequence of moves that is considered by the chess engine to be the optimal play by both players for the given position. Stockfish exposes a *Multiple Principle Variation* (multiPV) parameter which gives control over the number of PVs to show.
Figure 3.2 shows the output PVs when using mutiPV=2 for a given position. On the left the 2 best PVs are listed for depth=4. The first PV is evaluated as -1.1 (negative value means black has an advantage) and the second as -0.68. According to *depth=4 output* black should prefer playing *Bxd3* which yields a better evaluation for him. On the right the 2 best PVs are listed for depth=14. The first PV is evaluated as -2.09 and the second as -0.46. According to *depth=14 output* black should play *Qg6*. Per *depth=14 output*, the move suggested when using depth=4 is only the second-best move.

3.2.2 Other Chess Middleware

Other chess-based middleware used were: (1) a Stockfish plugin for python [30], (2) a python-chess [31] library, which was used for extracting position and move attributes, and (3) *fen-to-image* [32], an online tool for image diagrams.

3.3 Methodology

We define in this work, multiple chess classification tasks and describe various models which solve these defined tasks. For all defined tasks, the primary attempt is a ML move-based model. We also describe alternative position-based models, for comparison purposes. In addition, we make some feature refinement attempts to understand what information within the moves is most valuable. For all models we computed the *accuracy rate* and checked whether the initial move-based approach yields a result of *statistical significanc*e.

3.3.1 Statistical Significance

Our hypothesis is that moves contain useful information which can be used to make implications on the chess classification tasks. The *null hypothesis* is that moves cannot be used for chess classification tasks. In all experiments, except the first one, we divide the observations evenly between 2 classes. Using this approach, the expected *null hypothesis* accuracy rate is 50%. We define any result in which the model reaches more than a 55%

accuracy rate as rejecting the *null hypotheses*, thus showing that moves do contain useful information with statistical significance.

3.3.2 Computing Accuracy Rate

For each experiment, we separated the observations between training and testing. The training observations were used as the input for the training phase of the generic ML techniques (described below). Once the learning phase ended, we use the testing observations to compute the *accuracy rate*: the number of correctly predicted testing observations divided by the number of all testing observations.

The testing observations were not used in any way in the training phase. Some ML techniques may separate the input observations, internally, to learning and testing observations. We, however, regarded all these observations as 'training observations'. The observations defined as 'testing observations' were kept externally and used only when the training phase was completed.

3.3.3 Generic Machine Learning Techniques

In all ML tasks described in this work, the first step was to map the input observations to an n-dimensional numeric vector. In the next step, we applied various ML classifier techniques and measured their performance. We used 4 ML classifier techniques: (1) *Support vector machine* (SVM), (2) *random forest* (RF), (3) *naive bayes* (NB), and (4) *artificial neural network* (ANN).

SVM is a supervised binary classification algorithm that seeks to separate observations in an n-dimension domain using an n-1 dimensional hyperplane (See Figure 3.3: Separating observations in 2-dimensional domain using a 1-dimensional hyperplane). In cases where the observations are not separable, the *kernel trick* is applied, mapping the vectors to a higher dimension where they are separable (See Figure 3.4. On the left un-separable observations in a 2-dimensional domain. On the right separable observation in a 3-dimensional domain). We used *scikit-learn* [33] *C-Support Vector Classification* (SVC) implementation of the SVM algorithm.

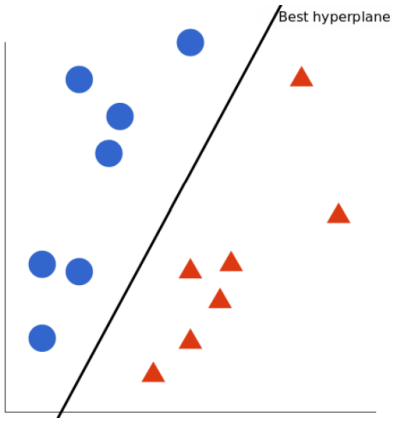


Figure 3.3 Support Vector Machine [34]

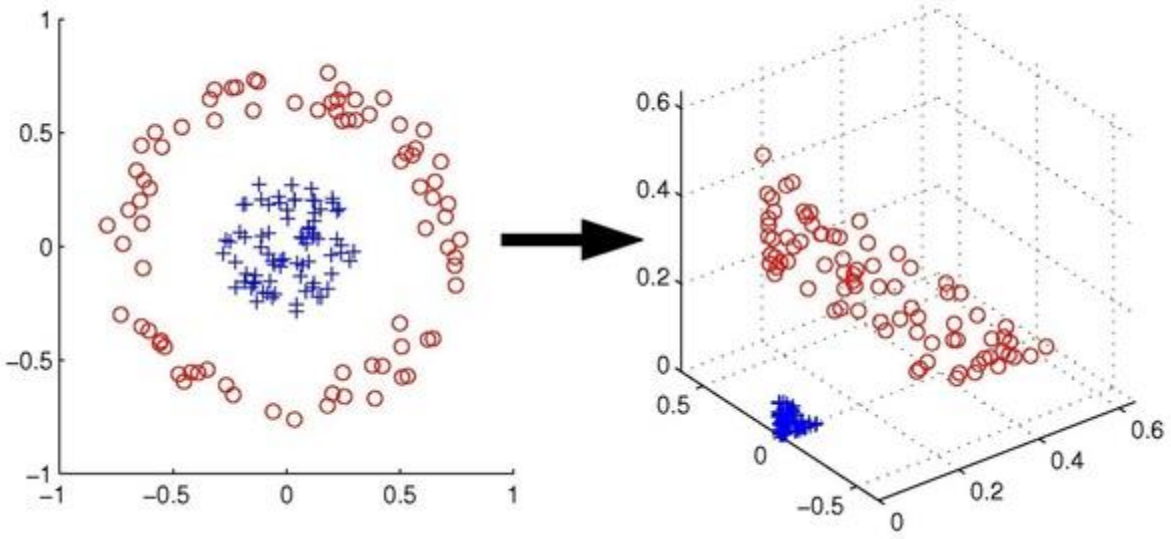


Figure 3.4 - SVM Kernel Trick [35]

RF is an ensemble ML method, combining multiple instances of ML to determine a single optimal solution. The RF model is made up of multiple *Decision Tree* (DT) Classifiers, where each DT acts as a separate model, and the final classification is decided based on a majority. We used the *RandomForestClassifier* implementation of *scikit-learn*.

In this work, we made use of 2 capabilities exposed by the RF implementation of *scikit-learn*: (1) *classification probability* and (2) *feature importance*. The *classification probability* is calculated based on the number of DTs agreeing on the final classification: The more DTs agree, the more confident the classification is. The *feature importance* is computed based on the actual usage of the features in the model. In DTs, some nodes are visited more often than others. For example, in Figure 3.5, the right node in the second layer is visited only if x_2 is more than 5. If in all observations x_2 is equal to or less than 5, the node will never be visited. Using the training data, the *scikit-learn* implementation of a RF checks empirically how often each of the features was involved in the classification. Increasing frequency of involvement increases the importance of the feature. In the last step, feature values are normalized so that all values sum to 1.

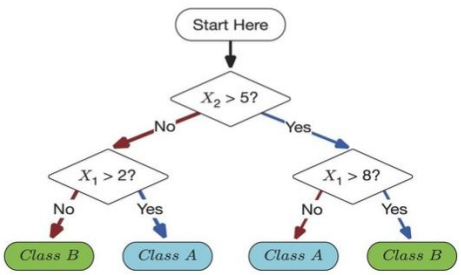


Figure 3.5 - Decision Tree [36]

In NB, the features and possible classifications are considered conditional events. Applying *Bayes theorem*, the *chain rule*, and the "naïve" assumption that the input features are statistically independent, an explicit formula is spelled out. We used the *GaussianNB* implementation of *scikit-learn.* In the *Gaussian NB* model, the assumption is that the feature values associated with each class are distributed according to a normal (or Gaussian) distribution. In the training process, the mean and variance of each conditional event of class and feature are calculated and form a normal distribution.

ANN is a collection of connected nodes called neurons, where each node can transmit signals to other nodes. In the context of ML, the network comprises a layer of input neurons which are activated based on the feature values, an output layer which represents the classes, and one or more hidden layers. We used a 3-layer network where the number of nodes in the input and output layers were determined by the size of the features and the outcome classes. The hidden layer comprised 128 nodes. We used the API of *TensorFlow* [37], in which we had to determine the learning attributes for the learning process. We used the attributes advised in the documentation: *Sparse Categorical Cross Entropy* for the loss function and the *adam* optimizer to determine the learning rate.

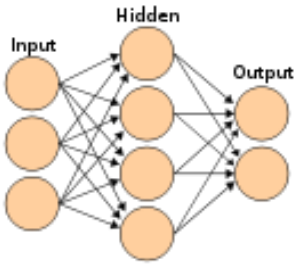


Figure 3.5 - Artificial Neural Network [38]

In all ML models, we utilized either the default configuration or the configurations advised in the documentation. We did experiment with the different optional parameters. For example, we attempted to raise the number of DTs in the RF model and use more nodes and more sophisticated connections for the ANN model (like *Deep Neural Networks* and *Recurrent Neural Networks*). These experiments did not yield better results and were therefore not adopted in the final analysis.

4. Predicting Game Outcome

This chapter discusses the prediction of game outcomes using raw moves as features. In this chapter we: (1) show that raw chess moves contain information that can help predict the game outcome, (2) compare the performance of move-based models to position-based models, and (3) detect which moves of the game are the most significant for predicting the game outcome.

The starting point of this chapter is the work by Masud *et al.* [15], who describe the creation of ML models that use move-based features and predict game outcomes. In this chapter, we focus on the model reported in [15] as most successful. We suggest several improvements to the model and suggest using additional heuristic methods to give a deeper insight regarding the relation between chess moves and game outcome.

4.1 Predicting Game Outcome - Masud *et al.*

Masud *et al.* [15] show how moves of a chess game can be used to predict game outcome. Nearly 300,000 games from the ChessOK [23] database were used to train and test a ML model. An accuracy rate of 65.6% was reached when using *Weka's* [39] *J48* algorithm, which is an implementation of a *Decision Tree* (DT) model.[2]

4.1.1 Feature Definition

The ML model in [15] was set as follows: Each game was regarded as an observation, the labels were the games' outcomes (white win, black win, or draw), and the features were the moves. Each individual move contributed 4 features: (1) the piece moving (2) the destination square (3) an indication whether capture took place, and (4) an indication whether *check* took place. The full feature vector contained all of these features for each move of the game. For instance, a 20-move game[3] contained 4×20=80 features. These 4 features are exposed in the *SAN* chess notation (See Section 3.1) which is the format used by almost all chess databases. A move like *Bxh7+* is trivially mapped to: (1) a *bishop* (**B**) (2) moving to **h7**, (3) capturing a piece (existence of **x**), and (4) "giving check" (existence of **+**). This means computing of move-based features is straightforward, as opposed to position-based features which involve processing of moves to compute the positions.

4.1.2 Distribution to Bins

In [15] the games were divided into categories for training purposes. They were distributed to move-length-range "bins" based on their lengths. For instance, a 15-20 move bin included all games which ended after at least 15 and at most 20 moves. Each bin was trained and tested separately. The maximum game length in each bin determined the number of features for that specific model, and the missing moves were filled with null values. For instance, a bin that contained games with a length of 15-20 moves used 20×4=80 features. For games of 15 moves, the first 60 features were populated based on the moves played, with the last 20 being filled with null values. Bins were used because the ML techniques used require all feature vectors to be of the same size.

---

[2] Masud *et al.* [15] describe 3 strategies for organizing the features and models, and for each of the strategies they applied 4 ML techniques. Here we describe the setup reported as most successful.

[3] To clarify: throughout this work a 'move' refers to a single move by white or a single move by black. A 20-move game contains 10 moves by white and 10 moves by black. (We need this note because an alternative counting convention exists. See Section 3.1).

4.1.3 Outcome

The Masud *et al.* model reached an accuracy rate of 65.6%. The authors report that when the model was applied to *live games* (i.e. games that have not yet ended), in 31% of correctly classified games the correct prediction was achieved 16 moves or more before the end of the game.[4]

We suggest, the experiment and outcome described suffers from two issues: one regarding the statistical evidence, and a second regarding the usefulness of the classifiers.

4.2 Predicting Game Outcome - "Dummy Classifier"

The features used in [15] contain a trivial non-chessic feature that significantly reduces its complexity: it incorporates which player made the last move in the game. Since the feature contains all the moves of the game, it is very easy to tell whether white or black made the last move. Given white made the last move, the chances black won are very low, and given black made the last move, the chances white won are very low.[5]

Consider the following *dummy classifier* and classifying rule: for every even length game, "predict" black won, and for every odd length game "predict" white won. Every game can either be of even length or odd length, and can end up in one of 3 outcomes (white win, black win, or draw). This results in 6 length-outcome combinations (See Table 4.1). The accuracy rate of the *dummy classifier* is the sum of the percent of even-length games won by black and the percent of odd-length games won by white.

We computed the accuracy rate of the *dummy classifier* on two datasets of chess games: Free Internet Chess Server (FICS) [21] and Portable Game Notation-Mentor (PGN-Mentor) [22]. These 2 data collections represent different types of games. The FICS collection comprises games played online by all levels of players, while PGN-Mentor comprises games played by the strongest players in official tournaments. We did not utilize the ChessOK [23] dataset for analysis because it does not offer free access to games.

As listed in Table 4.1, the *dummy classifier* reaches an accuracy rate of 92.1% when run on the FICS dataset and an accuracy rate of 52.9% when run on the PGN-Mentor dataset.[6]

We suggest, that to prove the hypothesis that moves contain information useful for predicting game outcome, we need a model which performs better than the *dummy classifier*. That is, the *null hypothesis* is that ML model ignores the moves, and only uses the odd/even encapsulated feature. To show statistical significance, we need to beat the *dummy classifier* by a clear margin (say 5%; i.e., an accuracy rate of 97.1% for FICS and 57.9% for PGN-Mentor). It is hard to tell whether the 65.6% accuracy rate reported in [15] shows statistical

[4] This whole assertion is convolute; it would be simpler to report that the model predicts the correct winner 16 moves ahead in ~20% of the time (0.655×0.31~0.2)
We re-visit this metric in Section 4.4.1.

[5] The last moving player may lose only in case of resignation. That is, a player makes a move, and before the opponent's response the player forfeits the game.

[6] The difference in the accuracy rate is not surprising. According the *dummy classifier* definition, all drawn games are predicted incorrectly. As shown in Table 4.1: In the FICS dataset less than 4% of games end up in a draw, while in PGN-Mentor the most common outcome of a game is a draw. In the chess world it is a known phenomenon that draw is the most common outcome in formal games.

significance, because we do not have access to the game outcome chances in the ChessOK dataset and therefore cannot compute the *dummy classifier* "accuracy rate". It is likely the ChessOK dataset is similar to that of the PGN-Mentor dataset (and perhaps the results are significant), since they both use games of professional players. The *dummy classifier* exposes the fact that the 65.6% reported in [15] is not particularly significant.

Table 4.1 - Computing *dummy classifier* accuracy rate per dataset
For each dataset there are 6 length-outcome possibilities.
The accuracy rate is computed as the sum of white-odd chances plus black-even chances.

Source	Length	White	Black	Draw	Accuracy Rate
FICS	odd	47.5%	2.0%	1.9%	47.5% + 44.6% = **92.1%**
	even	2.0%	44.6%	2.0%	
PGN-Mentor	odd	31.4%	1.7%	22.4%	31.4% + 21.5% = **52.9%**
	even	2.4%	21.5%	20.6%	

4.3 Predicting Game Outcome - Our Experiment

We ran a similar experiment to the one reported in Masud *et al.* [15], with some technical differences. We ran the experiment on 2 different datasets of games: FICS and PGN-Mentor. We used 4 different ML techniques: (1) *Support vector machine* (SVM), (2) *random forest* (RF), (3) *naive bayes* (NB), and (4) *artificial neural network* (ANN). For the SVM, RF, and NB we used the API of *scikit-learn* [33]. The RF learning technique made use of 10 *decision trees* (DTs) and based the classification on the majority. For the ANN we used the API of *TensorFlow* [37]. The ANN learning technique used 1 hidden layer comprising 128 nodes. The number of nodes in the input and output layers was determined by the size of the features and the outcome classes. See Section 3.3.3 for more information about these techniques.

4.3.1 Feature Definition

The features were defined similarly to those in Masud *et al.* [15] (See Section 4.1.1). Since the ML techniques used require the features to contain numeric values only, the piece feature were mapped to a 6-dimensional Boolean vector, with each dimension representing a different piece. A rook was defined as 100000 (1-True, 0-False), a knight as 010000, a bishop as 001000, a queen as 000100, a king as 000010 and a pawn as 000001. The destination square feature used 2 numeric values representing the row and column of the square. Summarizing the mapping we used, the 4 move attributes map to 10 numeric values: (1) 6 for the piece (2) 2 for the square destination (3) 1 for capture indication, and (4) 1 for *check* indication.

4.3.2 Distribution to Bins

We adapted the strategy in Masud *et al.* [15] of separate bins and created a separate classifier for each different game length. The Masud *et al.* classifier covered a range of game lengths. In our work, each “range” contained only one game length.

We limited the experiment to games that lasted at least 11 moves and no more than 100 moves. This restriction was forced since each bin includes only games of one game length and using too few observations per model causes distortions. The vast majority of game lengths lay within this range of moves, so the model covers almost all games. In addition, games shorter than 10 moves are often spurious (e.g., online games that were accidentally started and dropped immediately); hence, discarding short games also makes sense for "chessic" reasons.

4.3.3 Outcome

For each dataset, 100,000 games were used for training and 10,000 for testing (See Section 3.3.2). The classifiers using RF technique performed the best, with a 94% accuracy rate when running on the FICS dataset and a 73.2% accuracy rate when running on the PGN-Mentor dataset (See Table 4.2 for full results). The difference in accuracy rate is not surprising, it is likely the model makes use of the game length (odd/even) hint, boosting up the accuracy rate of the model which ran on the FICS dataset.
As explained above, the *null hypothesis* for comparison should be the *dummy classifier*. While the FICS trained models are only slightly better than the *dummy classifier* (94% using RF vs. 92.1%), the PGN-Mentor trained models show clear statistical significance (73.2% using RF vs. 52.9%).

Table 4.2 - Predicting game outcome of terminated games using moves as features
Accuracy rate per dataset and ML technique.
Dummy classifier as computed in Table 4.1.

	ANN	RF	SVM	NB	"Dummy Classifier"
FICS	93.9%	94.0%	94.0%	21.4%	92.1%
PGN-Mentor	70.3%	73.2%	71.8%	22.2%	52.9%

4.4 Predicting Outcome of Live Games

4.4.1 Masud *et al.* Results

Masud *et al.* [15] applied the trained models on live games, reporting that in 31% of correctly predicted games, the correct prediction was reached 16 moves before the game ended. Applying the training model on live games is more useful than applying the model on games which had already ended, but the outcome is not very convincing: only 20% ($0.655 \times 0.31 = 0.2$) of games were predicted correctly 16 moves before the end of the game. In addition to the low accuracy rate, the method used to predict the live game outcome is not optimal. The ML model in Masud *et al.* uses terminated games for training and then applies the trained model to live games. The model would likely perform better if it were trained using the same type of observation as tested.

4.4.2 Predicting Outcome of Live Games - Our Experiment

We ran a ML experiment using live-games both for training and for testing. A set of moves is referred to as the "first X moves" of the game, where "X" is the total number of moves that

have occurred up to that point in the game. Each set of "first X moves" of a game was utilized as a separate observation. In this style, a 20-move game contributes 20 observations, with each observation comprising the full set of moves that have been made up to that point in the game: The first observation is the first move, the second observation the first 2 moves, the third observation the first 3 moves, and so on. We dropped the "first-X-moves" observations for which X was lower than 10, as explained below, such that counting started from the 11th move.

4.4.3 Distribution to Bins

For each value of X, first-X-move is represented using a different feature vector size. For instance, a first-12-move uses 120 values (See feature definition in Section 4.3.1) and a first-13-move uses 130 values. The ML techniques used require all feature vectors to be of a set size. To solve this problem (See Section 4.1.2 and Section 4.3.2 where the same solution was applied), we distributed the observations to bins: each first-X-moves were trained and tested using its own first-X-moves classifier. For instance, a first-12-moves of a 20-move game, and a first-12-moves of a 40-move game were trained and tested in the same bin; while the first-12-moves, and the first-13-moves of the same game were separated into different bins. We only created bins of length of at least 10 moves, and at most 100 moves. For example, we did not create first-2-moves or first-110-moves bins. This was done so that results would be consistent with, and therefore comparable, to our previous experiment where this restriction was forced.

4.4.4 Balancing Labels

For the sake of the experiment, draws were removed from the FICS and PGN-Mentor datasets so that there would be only 2 outcome classes (white win and black win) and the observations were balanced evenly between these 2 classes. For example, when we ran on 100,000 observations, white won the game in 50,000 cases and black won the game in 50,000 cases. Because the results were balanced in the resulting dataset, any model that accurately predicted the results with over 50% success performed better than a random guess.

Note that for the live game prediction ML model, only the number of moves played so far is known, but not the final game length. This means the model cannot utilize the trivial, non-chessic, odd/even "hint" in its prediction. (See discussion in Section 4.2)

4.4.5 Experiments and Outcome

In the first 3 experiments 100,000, 200,000, and 400,000 games were used for training and 10,000 games were used for testing. While the SVM framework performed the best, it also took the most processing time (1,450 seconds on the 400K sample, on the PGN-Mentor dataset) and very extensive memory usage (nearly 2GB). For this reason, the SVM framework was not included in the fourth experiment, which used 4,000,000 observations. The last experiment size was driven by the PGN-Mentor dataset size.

As shown in Table 4.3 when using 4M games, the ANN performed the best with a 57% accuracy rate obtained on the FICS dataset and a 58.5% accuracy rate obtained on the PGN-Mentor dataset. The NB performed notably well when considering the time spent in the training process (less than a minute). Reaching a 57% and 58.5% accuracy rate provides statistical evidence moves can help predict the game outcome (See Section 3.3.1).

Table 4.3 - Predicting game outcome of live games using moves as features
Accuracy rate and time taken (in seconds) to train per dataset, ML technique, and training size.

		100,000		200,000		400,000		4,000,000	
	Technique	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time
FICS	ANN	52.7%	35	53.6%	42	55.1%	55	**57.0%**	458
	NB	53.0%	1	52.7%	1	52.9%	3	54.1%	48
	RF	51.8%	3	52.4%	5	52.4%	13	52.8%	258
	SVM	54.4%	45	54.4%	196	55.8%	965		
PGN-Mentor	ANN	53.9%	55	54.1%	56	56.0%	95	**58.5%**	547
	NB	52.2%	1	53.4%	2	54.2%	5	54.2%	42
	RF	52.4%	4	53.0%	7	55.0%	21	58.7%	266
	SVM	56.5%	70	57.6%	259	58.0%	1450		

4.5 Comparison to Position-Based Models

We ran 2 position-based experiments to compare to the move-based approach described above. Experiments were run first by utilizing an existing chess engine for predicting the winner,[7] and second by creating a ML model using the raw position content as features.

4.5.1 Position-Based Approach - Stockfish Model

The question of which player is most likely to win is related to the question of which player is leading. The player who is leading is most likely to win, and the player who is most likely to win is more likely to be leading. We can view the game outcome classifier as a proxy for a position evaluator. This means we can use a chess engine as an outcome classifier and compare the results to the move-trained model.

Using Stockfish [29] with the *depth* parameter set to 10 *ply* (See Section 3.2.1), we analyzed 50,000 games per dataset. For each position analyzed, a white advantage was regarded as a 'white win prediction', and a black advantage as a 'black win prediction'. We then compared the "predictions" to the actual games' outcomes. For the results to be comparable with the previous experiment, we only used evaluation of positions resulting after at least 10 moves into the game and up to 100 moves into the game (See Section 4.3.2 and Section 4.4.3).

[7] Chess engines use the position as an input, and are therefore regarded as position-based model.

Similar to the experiment above, only games that ended with a white or black win were used and the games were balanced evenly between the 2 outcomes (See Section 4.4.4).

The *Stockfish model* reached an accuracy rate of 65.5% when running on the FICS dataset, and 74.1% when running on the PGN-Mentor dataset (See Table 4.4)

Table 4.4 - Predicting game outcome of live games - comparing different models
Accuracy rate per dataset and model.

	FICS	PGN-Mentor
Move-based ML model (ANN)	57.0%	58.8%
Position-based Stockfish model	65.5%	74.1%
Position-based ML model (ANN)	51.9%	60.1%

4.5.2 Position-Based Approach - ML Model

We ran another position-based ML experiment, using the raw data of the positions.
Each position reached was regarded as an observation, and the labels were the games' outcomes. Each square on the board was regarded as a feature. Each square was mapped to a Boolean vector of size 13, representing the 13 potential values a square can get (6 white pieces, 6 black pieces, and an empty square. A white knight for example was mapped to 0100000000000. This is the same mapping technique for pieces as in Section 4.3.1). Since a chessboard comprises 64 squares, each individual observation comprised
$13 \times 64 = 832$ numeric values.

Similar to the experiments above, only games that ended with a white or black win were used and the games were balanced evenly between the 2 possible outcomes (See Section 4.4.4).

As in previous ML models, we kept the distribution to bins (See Section 4.1.2 and Section 4.4.3). The original justification for separate bins was that for each number of moves the feature size is different, and therefore not suitable for the ML techniques used. When dealing with positions, all positions are of the same size. Nevertheless, to be fully comparable with the previous experiment, we kept the distribution to bins. In this setup, all positions resulting after X moves were trained and tested separately, and not mixed with positions resulting after a different number of moves. Similar to previous experiments, we only used evaluation of positions resulting after at least 10 moves into the game and up to 100 moves into the game (See Section 4.3.2 and Section 4.4.3).

100,000 positions were used for training and 10,000 for testing. The ANN framework performed the best with 51.9% accuracy on the FICS dataset and 60% accuracy rate on the PGN-Mentor. (See Table 4.5).

Table 4.5 - Predicting game outcome of live games using positions as features
Accuracy rate per dataset and ML technique.

	ANN	NB	RF	SVM
FICS	51.9%	50.0%	52.1%	50.5%
PGN-Mentor	60.1%	52.8%	56.6%	60.2%

4.5.5 Advantages of Move Based Model

The fact the model which uses Stockfish (which is position-based) outperforms the move-based ML model, shows our model does not offer the world of chess new tools. However, it is also impossible to conclude that more information is implied by position than by moves: Stockfish evaluation is an offspring of 50 years of *Artificial Intelligence* (AI) research of top-notch programmers and chess players, while this move-based model was trained with no special chess knowledge. Perhaps the model which uses Stockfish outperforms the ML model because of the specialized effort which was invested in it.

Though the position-based ML model outperformed the move-based model when running on the PGN-Mentor dataset, some advantages of the move-based model were apparent. Classifying the position-based model was rather resource-consuming, involving much more time and memory to process. Compiling 100,000 positions and mapping them to features took ~450 seconds[8] and consumed 350MB to handle the training feature vectors (before involving the ML techniques) while mapping 100,000 game moves to features (which compares to ~7M positions) took 15 seconds and consumed 315 MB. These values can probably be optimized with some programming effort, yet it demonstrates the built-in advantage of using move-based features over position-based features: When using moves there is no need to process positions and therefore less runtime and coding involved, and the representation of moves is much smaller than positions, thus consuming less memory.

4.6 Feature Refinement

The RF model by *Scikit-learn* API exposes a ‘feature importance’ metric (See Section 3.3.3). Using this capability, we measured the importance of the first X moves and last X moves for values 1 through 10 on the “live game” move-based model of RF (The experiment described in Section 4.4). As shown in Table 4.6, the closer the move is to the given state of the game, the higher the impact the move has. The first move impact is about 0.7%-0.8% and the last move played (so far) has an impact of 3.6%-4.5%. This means the last moves of the game are of the highest importance in the model.

This raises the question as to how well the model performs when using only the last moves of the game. Using the NB framework (which runs fastest) we ran a ‘last X moves’ version of the ‘live game prediction’ experiment. The experiment differed from the original experiment (Described fully in Section 4.4) only in the number of moves used per observation. In the former experiment described above, all moves (played so far) were used, whereas in the “last

[8] The times recorded in Table 4.3 refer to the training time and not to the preparation time. This is because the preparation is done once for all 4 frameworks, and when possible, the data was serialized and re-used in other experiments.

X moves" experiment reported here we used only the X most recent moves played. The experiment was repeated 10 times, iterating X from 1 to 10.

Table 4.6 - Feature Importance as exposed in RF model of "live game" prediction

First Moves Importance				Last Moves Importance		
First	FICS	PGN-Mentor		Last	FICS	PGN-Mentor
1	0.8%	0.7%		-1	3.6%	4.5%
2	1.0%	0.9%		-2	3.4%	4.0%
3	1.6%	1.0%		-3	3.2%	3.7%
4	1.7%	1.3%		-4	3.1%	3.4%
5	1.9%	1.4%		-5	3.0%	3.3%
6	2.1%	1.7%		-6	3.0%	3.1%
7	2.3%	1.9%		-7	2.9%	3.0%
8	2.3%	2.1%		-8	2.8%	2.8%
9	2.5%	2.2%		-9	2.7%	2.7%
10	2.6%	2.3%		-10	2.6%	2.6%

As shown in Table 4.7, the model trained using only the last 3 moves obtained a similar accuracy rate as the model trained using all the moves.

Table 4.7 - Predicting game outcome of live games using last-X-moves as features
The rows are the dataset; the columns are the number of last-X moves used.
ML technique: NB. Accuracy rate per last-X move.

	1	2	3	4	5	6	7	8	9	10	ALL
FICS	52.6%	54.0%	54.6%	55.1%	55.1%	54.4%	55.2%	56.0%	54.9%	55.9%	54.1%
PGN-Mentor	54.0%	54.2%	54.4%	55.8%	55.9%	56.4%	56.3%	57.1%	56.4%	57.4%	54.2%

4.7 Summary

As shown by the results of ML models that solely rely on raw chess moves, we can conclude that (1) Raw chess moves contain implicit information about the game outcome and can help tell which player would probably win. (2) The moves played just before the given position are more significant than those played earlier in the game.
This notation can be utilized for extracting useful chess lessons, such as what move sequences are most associated with white or black winning the game.

A notable phenomenon is that the models consistently performed better on the PGN-Mentor dataset than on the FICS dataset. This may relate to the PGN-Mentor dataset characteristics: The PGN-Mentor dataset contains mature games of professional players playing formal games, while the FICS dataset contains blitz games of all levels of players. In mature games, the correlation between reaching an advantageous position and winning is stronger. This may mean that moves can be used for identifying characteristics of games. In Chapter 7 (Summary Chapter) we suggest this as one opportunity for future research.

In the area of predicting game outcomes, the classic chess engine outperforms the move-based model. This does not detract from the power of the moves. Positions contain more information than moves, and the engine used in prediction is an offspring of over 50 years of *Artificial Intelligence* development. But this means the move-based prediction models have no utility in chess research. There are other chess automation tasks for which moves may be a valuable predictor, such as whether human players would find the correct move. The next chapter shows that in the domain of predicting human play, a move-based model outperforms position-based models.

5. Predicting Whether Players Would Find the Correct Move

This chapter discusses ML models using raw moves as features predicting whether human players would find the correct move, in positions where only one good move exists.

In this chapter we: (1) Show that raw chess moves contain information that can help predict whether a player will find the correct move, (2) compare the performance of move-based models and position-based models, (3) detect which moves and which move attributes are most significant for telling whether the correct move will be found, and (4) compile a useful chess lesson i.e., lists of move-sequences that players tend to miss.

McIlroy-Young *et al*. [12] describe two solutions for the error-predicting task: one using position-based model, and another using position and player metadata (e.g. players' ratings). The best solutions reach 67.7% and 71.7% accuracy rates respectively. The solution in [12] is not fully comparable with the move-based solution described in this chapter since different definitions of mistakes are used.

5.1 Collecting Positions with Only One Good Move

In the experiments described in this chapter, we were interested in positions in which only one good move exists and describe models that can predict whether the only good move would be found.

Using Stockfish [29], we analyzed games downloaded from the FICS database [21] (See Section 4.2). For each position, we requested the 2 best *Principle Variations* (PV) and their evaluation scores using a depth of 14 *ply* (For more information about Stockfish, depth, *ply* and PV see Section 3.2.1). In cases where the difference in scores between the best and second-best PV exceeded 1.5 points, we concluded that the position had only one good move and was suitable for our experiment. In Figure 5.1 the difference in score between the best and second-best moves exceeds 1.5 and therefore we can conclude the position has only one good move.

When calculating the score differences, we considered scores above 5 as a score of 5; this is because it is an implicit psychological assumption that, as long as a crushing advantage is maintained, it does not matter what move is played.[9]

[9] Schematically, in case a player is a queen ahead (9 points), losing a queen would be considered a loss of *min(9, 5) - 0 = 5 - 0 = 5* points and therefore would be considered a mistake. In case a player is a queen ahead, trading it for a rook, (thus converting to a rook ahead advantage; 5 points), would be considered a loss of *min(9, 5) - 5 = 5 - 5 = 0* points and therefore not considered a mistake.

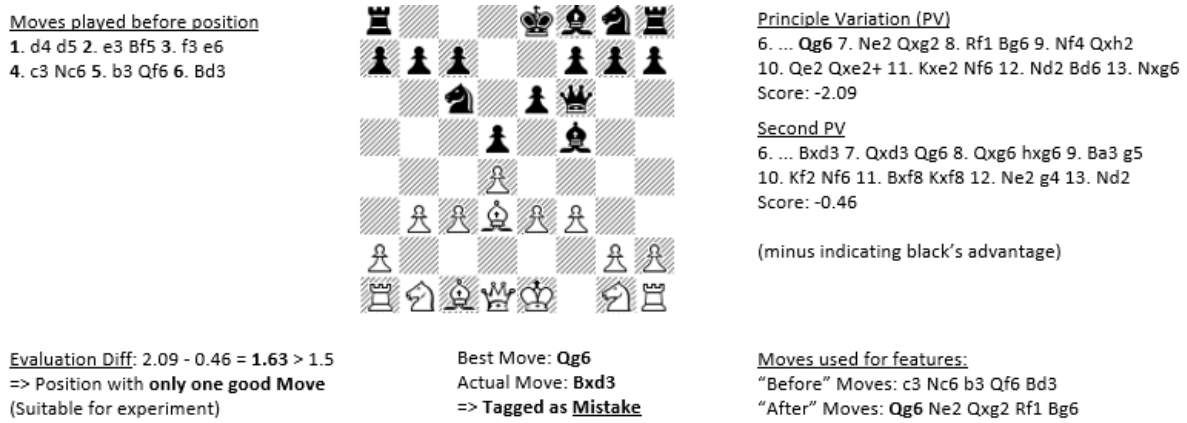


Figure 5.1 - Position with only one good move tagged as "Mistake"
Parameters: depth=14, multiPV=2. See Section 3.2.1.
Top Left: Moves played till the critical position.
Top Middle: Resulting position. Top Right: 2 Principle Variations.
Bottom Left: Difference in score between
best and second-best move exceeds 1.5.
Bottom Middle: Human doesn't play correct move
Observation tagged as "mistake".
Bottom Right: 5 *before* moves and 5 *after* moves used for features.

5.2 Label and Feature Definitions

The actual move played in the critical position was compared to the correct move per Stockfish. If the move played was the same as the move Stockfish suggested, the observation was labeled as 'correct', otherwise, it was labeled as 'mistake'. For example, in Figure 5.1 the actual move differed from the correct move, and therefore the observation was tagged as 'mistake'.

In most positions extracted from the DB, with only one good move, the correct move was found. To keep a balanced ratio between 'correct' and 'mistake' labels, we discarded excess 'correct' observations. After analyzing nearly half a million positions, we gathered 50,000 'correct' observations and 50,000 'mistake' observations.

The features were made of the 5 moves **played before** the critical positions (denoted as *before moves*), and 5 moves which **should be played after** the critical positions (denoted as *after moves*). The correct move is the first *after move* (See Figure 5.1). Similarly to the experiments described in Chapter 4 (See Section 4.1.1 and Section 4.3.1), for each individual move the following features were used: (1) the piece moving (2) the destination square (3) an indication whether capture took place, and (4) an indication whether *check* took place.

5.3 Experiments and Outcome

90,000 observations were used for training and 10,000 for testing (See Section 3.3.2). We tried 4 ML techniques: Random forest (RF), support vector machine (SVM), artificial neural network (ANN), and naive bayes (NB) (See Section 3.3.3).

The ANN performed the best and reached an accuracy rate of 76.8%, and RF with 76.3% accuracy rate. SVM took much longer (nearly half an hour compared to less than 10 seconds using the other techniques) and reached 75.7%. Due to the long run time, we dropped the

usage of SVM for further refinement experiments. The NB model reached only 69.2% accuracy rate but ran very fast. See Table 5.1 for the full results.

Table 5.1 - Predicting whether a human player finds the best move using moves as features Accuracy rate and Time per ML technique.

ML Technique	Accuracy	Time (Seconds)
ANN	76.8%	6.2
RF	76.3%	1.8
SVM	75.7%	1785.9
NB	69.2%	0.1

5.4 Comparison to Position-Based Models

We ran 2 position-based experiments to compare to the move-based approach described above. The first experiment was run by utilizing an existing chess engine for predicting whether the correct move would be found, and a second experiment was run by creating ML models using the raw position content as features.

5.4.1 Position-Based Approach - Customized Stockfish Model

One way to predict whether a human player would find the correct move is to create a customized chess engine that mimics human behavior. Given a customized chess engine we can define a correct-move classifier using the following rule: if the customized engine finds the correct move, predict observation as "correct", otherwise predict observation as "mistake". According to this definition the correct classifications (not to be confused with "correct" move), are the cases where both engine and human find the correct move, or both engine and human fail to find the correct move; all other cases are incorrect classifications.

The Stockfish API exposes the *depth* parameter (See Section 3.2.1) which controls the number of moves to look ahead. We initially used a customized Stockfish instance using depth=14 as an oracle defining which moves are correct. By using a lower depth, we may get a customized chess engine with a more human-like behavior, and this customized engine can be utilized as a correct-move classifier (as defined in the paragraph above).

We used 10 customized stockfish instances: each instance was a Stockfish chess engine with a specific depth, with a varying depth from 1 to 10. For each customized Stockfish instance, we analyzed 2,000 positions, and for each position, we checked whether Stockfish acted like the human player.

For each position the human player either succeeded or failed to find the correct move. Similarly, for each position the customized Stockfish instance either succeeded or failed to find the correct move. This results in 4 combinations: (1) Human-Mistake, Stockfish-Mistake, (2) Human-Correct, Stockfish-Correct, (3) Human-Mistake, Stockfish-Correct, (4) Human-Correct, Stockfish-Mistake. In cases (1) and (2) the human and Stockfish act alike,

and in cases (3) and (4) the human and Stockfish act differently. The accuracy rate of the *customized stockfish model* is the sum of the percent of cases in which Stockfish and the human acted alike, i.e. cases (1) and (2).

Table 5.2 lists the 4 combinations for each customized Stockfish instance according to depth. As shown in Table 5.2 the highest accuracy rate was achieved for depth=1 with a rate of 64.9%, and the accuracy decreased for each additional *ply*.

Table 5.2 - Computing *Customized Stockfish Model* accuracy rate per depth
According to the definition of the *Customized Stockfish Model*, the accuracy rate is computed as the sum of cases (1) *Human-Mistake, Stockfish-Mistake,* and (2) *Human-Correct, Stockfish-Correct*.

	Human-Mistake		Human-Correct		
Depth	Stockfish-Mistake	Stockfish-Correct	Stockfish-Mistake	Stockfish-Correct	Accuracy Rate
1	16.8%	32.9%	2.2%	48.1%	**64.9%**
2	15.1%	34.6%	2.4%	47.9%	**63.1%**
3	13.8%	35.9%	4.0%	46.2%	**60.0%**
4	12.2%	37.5%	5.1%	45.2%	**57.4%**
5	10.3%	39.4%	4.8%	45.5%	**55.8%**
6	8.0%	41.7%	3.9%	46.3%	**54.3%**
7	5.5%	44.2%	2.4%	47.9%	**53.4%**
8	3.2%	46.6%	0.7%	49.6%	**52.8%**
9	1.8%	47.9%	0.4%	49.9%	**51.8%**
10	1.1%	48.7%	0.3%	50.0%	**51.0%**

5.4.2 Position-Based Approach - ML Model

We ran another position-based ML experiment, using the raw information of the positions. The features were defined as in Section 4.5.2. We used 90,000 observations for training and 10,000 for testing (See Section 3.3.2). An accuracy rate of 66.5% obtained when using the ANN ML technique. See full results in Table 5.3.

Table 5.3 - Predicting whether a human player finds the best move using positions as features
Accuracy rate per model.

	NB	RF	ANN
Accuracy	55.7%	59.1%	66.5%

The move-based approach outperformed both position-based approaches. For comparison see Table 5.4.

Table 5.4 - Predicting whether a human player finds the best move - comparing different models
Accuracy rate per model.

	Move based (ANN)	Position based (ANN)	Customized Stockfish (depth=1)
Accuracy	76.80%	66.50%	64.90%

5.5 Feature Refinement

5.5.1 Refining Moves

The initial experiment used 10 moves as features: 5 *before moves* and 5 *after moves*. Using the 'feature importance' metric exposed in the RF API (See Section 3.3.3) we checked which of these moves were most significant. As shown in Table 5.5 the last *before move* (21.1%), and the first *after move* (15.3%) were significantly more important than the other 8 moves involved.

Table 5.5 - Feature Importance as exposed in RF model predicting whether the best move would be found
Marked in **Bold** the 5th *before* and 1st *after* which are most important.

Before-Move Number					After-Move Number				
1	2	3	4	5	1	2	3	4	5
7.2%	7.3%	7.3%	7.8%	**21.1%**	**15.3%**	8.3%	9.8%	7.6%	8.3%

Using the ANN technique (which performed best), we re-ran the experiment using a varying number of *before* and *after* moves with all permutations using 0 moves to 5 moves. (Obviously, we cannot run the experiment with 0 *before moves* and 0 *after moves*). As shown in Table 5.6, *1-before-1-after* permutation reached an accuracy rate of 77.6%, and only 2 permutations passed this rate with an additional insignificant 0.1%. (See bold text in Table 5.6). In addition, when using 0 *after* moves or 0 *before* moves the accuracy rate drops significantly (See italicized text in Table 5.6)
The results strengthen the conclusion of the 'feature importance' experiment above: The last *before move*, and the first *after move* are the most significant, and all other moves do not add to the accuracy rate.

Table 5.6 - Predicting whether a human player finds the best move using a varying number of moves
Accuracy rate per number of moves using ANN ML technique.
Marked in **Bold** the accuracy rates in range 77.5%-77.7%.
Marked in *Italic* permutations with *after*=0 or *before*=0.

		After					
		0	1	2	3	4	5
Before	0		*68.6%*	*70.4%*	*71.1%*	*71.3%*	*71.8%*
	1	*72.1%*	**77.6%**	**77.7%**	77.3%	76.8%	76.9%
	2	*72.3%*	**77.7%**	**77.5%**	**77.5%**	77.0%	76.8%
	3	*73.0%*	77.2%	77.4%	77.2%	**77.5%**	76.6%
	4	*73.0%*	76.9%	77.2%	77.2%	77.0%	77.2%
	5	*72.7%*	77.0%	77.4%	76.3%	77.1%	76.8%

5.5.2 Refining Move Attributes

The raw information of each move comprises the following 4 attributes: (1) the piece moving, (2) the destination square, (3) an indication whether capture took place, and (4) an indication whether *check* took place. These 4 attributes can be denoted as a SAN representation since these are the 4 attributes exposed in the *SAN notation* (See Section 3.1). For example, a move such as *Bxh7+* maps to (1) a *bishop* (**B**) (2) moving to **h7**, (3) capturing a piece (existence of **x**), and, (4) "giving check" (existence of **+**).

To check which of these attributes are most important, we re-ran the ML experiment, each time with different attributes included. Four permutations were of SAN representation without a single attribute (piece, destination, check, and capture). In addition, we ran an ALL permutation, where we added 2 attributes to the SAN representation: (1) the source square (symmetrical to target square), and (2) the captured piece (in case of capture; symmetrical to moving piece). We also ran a UCI permutation, where the raw information was the information exposed in the *UCI Format* (See Section 3.1), including the coordinates of the source and destination squares, but not including all other attributes that exist in SAN.

Table 5.7 summarizes the results. The most important results are that (1) the most significant attribute is the capture existence, (2) the SAN notation is superior to the UCI notation for prediction, and (3) adding more attributes to SAN (i.e. using ALL) is probably better than SAN.

Table 5.7 - Predicting whether a human player finds the best move using different move attributes
Using 1 *before move* and 1 *after move*.
Accuracy rate per permutation and ML technique.
The rows represent the move attributes, the columns represent the ML technique.

	ANN	NB	RF
ALL	78.2%	73.1%	77.3%
SAN	77.6%	71.9%	76.1%
SAN_NO_CHECK	77.6%	72.0%	76.0%
SAN_NO_SQUARE	77.0%	72.0%	77.1%
SAN_NO_CAPTURE	72.0%	61.0%	73.2%
UCI	70.2%	58.8%	73.4%

5.6 Common Move Sequences

Using the probability metric exposed by *scikit-learn* for the RF framework, we extracted all cases where all 10 *Decision trees* (DT) agreed on the classification (See Section 3.3.3). Reviewing the clear-cut predictions, we looked for 2-move sequences that showed up in many observations.

5.6.1 Top-Correct Sequences

Among *top-correct* sequences, the following phenomenon became apparent: there were many repeating 'last *before* move - first *after* move' sequences. (Illustrated example in Figure 5.2). Table 5.8 lists the top 40 *top-correct* sequences.

Moves used for features:
"Before" Moves: c3 Nc6 b3 Qf6 Bd3
"After" Moves: **Qg6** Ne2 Qxg2 Rf1 Bg6

Best Move = First *after move*:
Qg6

Last *before* move - First *after* move:
Bd3 Qg6

First *after* move - Second *after* move:
Qg6 Ne2

First *after* move - skip - Third *after* move:
Qg6 Qxg2

Figure 5.2–Illustration of 2-move sequences derived from features
Left: Initial position. Middle: Moves used for features and best move.
Right: Three examples of derived 2 move sequences.

The fact that almost all the top sequences include a capture and recapture explains why the 'capture' indication is so crucial. The fact that the most common sequence comprises the last *before* move and first *after* move can explain why these are the most important features.

Table 5.8 – *Top-Correct* sequences
Top 40 'last *before m*ove - first *after m*ove' sequences showing up in RF model where all 10 DTs agree the correct move would be found.
See Figure 5.2 illustrating how a last *before m*ove - first *after m*ove is derived.
See Section 3.1 for an explanation about chess notation.

1. … Bxc3+ 2. bxc3	1. Bxd6 Qxd6	1. Nxd5 exd5	1. … Nxd3 2. Qxd3
1. Bxg7 Kxg7	1. … Nxe3 2. fxe3	1. Bxf6 gxf6	1. … Bxf3 2. Bxf3
1. Bxc6+ bxc6	1. … Bxg2 2. Kxg2	1. Bxf6 Bxf6	1. Bxf5 exf5
1. … Bxd3 2. Qxd3	1. … Nxe4 2. Bxe4	1. … Bxe2 2. Qxe2	1. Bxe5 dxe5
1. Nxe5 dxe5	1. … Bxf3 2. gxf3	1. … Bxe3 2. fxe3	1. … Nxd4 2. Qxd4
1. Nxc6 bxc6	1. Bxe7 Qxe7	1. g4 Bg6	1. Nxf7 Kxf7
1. … Nxc3 2. bxc3	1. Nxe6 fxe6	1. … Nxd4 2. exd4	1. … Bxf3 2. Qxf3
1. Bxe6 fxe6	1. Bxf7+ Kxf7	1. … g5 2. Bg3	1. … Nxd4 2. cxd4
1. … Nxe4 2. dxe4	1. Bxc6 bxc6	1. … Bxe4 2. dxe4	1. … b5 2. Bb3
1. … Bxc3 2. bxc3	1. Nxe5 Bxe5	1. Nxe7+ Qxe7	1. … Bxf4 2. exf4

5.6.2 Top-Mistake Sequences

The *top-correct* sequences probably add very little value. What makes them *top-correct* is the fact that even the weakest players find them. In comparison, the *top-mistake* cases are very interesting because they reflect moves which are not found by players, and players should learn to be aware of them.

The most common 2-move sequences in the *top mistake* positions were ‘first *after* move - second *after* move’ sequences as well as many ‘first *after* move - skip - third *after* move’. (Illustrated example in Figure 5.2). Table 5.9 lists the 12 top ‘first *after* move - second *after* move’ *mistake* sequences.

Table 5.9 - *Top Mistake* sequences
Top 12 'first *after m*ove - second *after m*ove' sequences showing up in RF model where all 10 DTs agree the correct move would **not** be found.
See Figure 5.2 illustrating how a first *after m*ove - second *after m*ove is derived.
See Section 3.1 for an explanation about chess notation.

1. Bxf7+ Kxf7	1. … Qh4+ 2. g3	1. Qh5+ Ke7	1. … Bxf2+ 2. Kxf2
1. Qh5+ g6	1. … Qh4+ 2. Ke2	1. … Qa5+ 2. c3	1. Bxf7+ Kd7
1. Qa4+ Nc6	1. … Qh4+ 2. Kd2	1. Bxa6 Nxa6	1. Qh5+ Kd7

We elaborate on these sequences in the appendix (Chapter 9).

5.7 Summary

The fact we successfully trained ML models that solely rely on raw chess moves, shows that raw chess moves contain information about the probability of finding the correct move. We have also determined that the only information that is required for predicting whether the correct move would be found is the moves adjacent to the critical position: the move before the critical position and the correct move itself. We have been able to determine that the move attributes exposed by *SAN notation* are very useful, supporting the decision of FIDE (The International Chess Federation) making *SAN notation* its standard. We have likewise determined that the most important attribute of moves, in regard to finding the correct move, is the indication of capture. Finally, we have determined that the raw moves played are more useful in predicting whether the correct move would be found than the raw data in the position itself.

This chapter and the previous chapters were limited to very specific and well-defined areas of chess: game outcome probability, and positions with only one good move. The purpose of the next chapter is to expand the usage of raw chess moves to a more general and less objective field: predicting difficulty and human defined tactics of chess puzzles.

6. Reverse Engineering Lichess Puzzle Database Attributes

Lichess [40] is a free and open-source internet chess server, used mainly for hosting online chess games. Lichess provides various services and features, one of them is an open database of puzzles [24]. The database contains over 1.7 million puzzles, generated from over 200 million analyzed games from the Lichess games database. We chose to work with Lichess puzzle DB, as it is large, free, and organized in a systematic manner.

In this chapter we describe how, by using raw moves, we trained models to reverse-engineer the difficulty and chess themes associated with the puzzles. This chapter also compares the results to those of a position-based approach and describes how the move-based model can help enhance the database.

6.1 Puzzle Entry Structure and Feature Definition

Each database entry comprises various fields. We were interested in 4 fields: (1) the starting position of the puzzle (2) the solution of the puzzle (i.e. the moves the players are asked to find) (3) the puzzle rating (i.e. difficulty), and (4) the chess themes [41] (e.g., a *fork* is a tactical chess theme). Using the position and solution, we would like to predict the difficulty and chess theme. Figure 6.1 illustrated the puzzle attributes.

The actual puzzle, (i.e. the position for which the solution needs to be found), is the second move in the puzzle entry. Using the terminology of chapter 5 (See Section 5.1), the first move in the database entry is the last ‘before’ move, and the second move and onwards are ‘after’ moves. The position in the puzzle was given using the FEN notation [42], and the moves in the entry were given using the UCI format (See Section 3.1). After converting the moves to SAN format and extracting the position of the puzzle itself (since the position provided is one more before the puzzle itself), we have the same structure as in Chapter 5 for both move and position features, with *1 before move* and several *after moves.* Figure 6.1 can help understand the conversion steps applied to the original DB entry.

Puzzle DB attributes

Position one move before puzzle starts
(Originally in DB given using FEN notation)

Moves
(In SAN format. Originally in DB given in UCI Format):
1. Qxc6 Ne2+ **2**. Kh1 Qxh2+ **3**. Kxh2 Rh5#
actual puzzle starts in second move

Rating:
1580

Themes:
Anastasia Mate, Attraction,
Discovered Attack, Endgame,
Long, Mate, Mate in 3, Sacrifice

Input attributes in experiments

Starting position of puzzle

Before Moves:
Qxc6

After Moves:
Ne2+ Kh1 Qxh2+ Kxh2 Rh5#

Values to predict

Difficulty: rating 1580 > 1500
=> Labeled as **Difficult**

Themes:
Anastasia Mate, Attraction,
Discovered Attack, Endgame,
Long, Mate, Mate in 3, Sacrifice

Figure 6.1 - Puzzle attributes, Input and Output of ML experiments
Each puzzle entry in the puzzle DB include:
(1) The position in FEN notation one move before the actual puzzle
(In figure showing graphical representation).
(2) The moves in UCI format one move before the actual puzzle
(In figure showing in SAN format - see Section 3.1).
(3) The rating and (4) the themes of the puzzle.
The experiments in this chapter use the: Actual starting position of the puzzle
and the *before* moves and *after* moves.
The output of the experiments are the puzzle difficulty and themes.

6.2 Computing Puzzle Difficulty

The puzzle ratings in the Lichess DB were determined using the Glicko rating system [43], where each attempt to solve a puzzle was considered as a Glicko rated game between the player and the puzzle.

In the Glicko system, the rating is calculated per game and the winner "takes" points from the loser based on the difference in rating between the players. In the case where the player with the higher ranking wins the game, few points are transferred, while in the case where the lower ranked player wins, many points are transferred. In the case of a draw, points are transferred from the higher ranked player to the lower ranked player. The Glicko is an improvement over the ELO rating system [44]. Both systems are reliable rating systems and are applied in various areas of competitive sports.

The median ranking in the Lichess DB was 1500. We regarded every puzzle ranked lower or equal to 1500 as 'easy', and every puzzle ranked above 1500 as 'difficult'.

Using ML similar to those applied in previous chapters, we trained models which predict the puzzle difficulty using the moves as features.

6.2.1 Feature Definition

Similar to the experiments described in chapter 4 (See Section 4.1.1), for each individual move the following features were used: (1) the piece moving (2) the destination square (3) an indication whether capture took place, and (4) an indication whether *check* took place.

In the database, the number of *after moves* varies per puzzle. Most puzzles are exactly 3 moves long (70.6%), but some puzzles contain only 1 *after* move (7.8%) and many puzzles contain 5 or more *after* moves. We ran 3 permutations of experiments: (1) 1-*before*, 1-*after*, this permutation was appropriate for all puzzle entries. (2) 1-*before*, 3-*after*, this permutation was not applied to puzzles with only 1-*after* move. (3) 1-*before*, 5-*after* this permutation was applied only to puzzles which comprised at least 5 *after* moves.

For each feature length we ran the experiment using 3 ML techniques: ANN, RF, NB (leaving out SVM which ran too long in previous experiments see Section 4.4.5 and Section 5.3). We used 100,000 observations for training and 10,000 for testing (See Section 3.3.2). The highest accuracy rates were achieved using the ANN framework, with 66% for 1-*after*, 71.6% for 3-*after*, and 81.1% for 5-*after*. See Table 6.1 for full results.

Table 6.1 - Puzzle difficulty using moves as features
Accuracy Rate per ML technique and number of moves used.

	ANN	NB	RF
1-before 1 -after	66.0%	61.2%	63.0%
1-before 3 -after	71.6%	61.1%	70.8%
1-before 5 -after	81.1%	64.8%	80.3%

6.2.2 Comparison with Position-Based Methods

We ran 2 position-based experiments to compare to the move-based approach described above. The first experiment was run by utilizing an existing chess engine which tried solving the puzzles and a second experiment by creating ML models using the raw position content as features.

One way to predict whether a human player with a 1500 rating (see Section 6.2) can solve a puzzle is to create a customized chess engine that mimics a 1500 rated human player (The reasoning and method described in this section are very similar to the reasoning and method described in Section 5.4.1). Given a customized chess engine we can define a difficulty classifier using the following rule: if the customized engine finds the correct move, predict observation as "easy", otherwise predict observation as "difficult". According to this definition the correct classifications are the cases where the engine finds the correct move and the puzzle entry is tagged as easy, or the engine fails to find the correct move and the puzzle entry is tagged as difficult, all other cases are incorrect classifications.

The Stockfish API exposes the *depth* parameter (See Section 3.2.1) which controls the number of moves to look ahead. By using a lower *depth*, we may get a customized chess engine with a more human-like behavior, and this customized engine can be utilized as a difficulty classifier (as defined in the paragraph above).

We used 10 customized stockfish instances: each instance was a Stockfish chess engine with a specific depth, with a varying depth from 1 to 10. For each customized Stockfish instance, we analyzed 2,000 puzzles, and for each puzzle we checked whether Stockfish found the first move of the solution.

Each position entry was considered 'difficult' or 'easy'. And for each position the customized Stockfish instance either succeeded or failed to find the correct move. This results in 4 combinations: (1) Difficult, Stockfish-Mistake, (2) Easy, Stockfish-Correct, (3) Difficult, Stockfish-Correct, (4) Easy, Stockfish-Mistake. In cases (1) and (2) Stockfish successfully acted as a difficulty predictor. Accordingly, the accuracy rate of the *customized stockfish model* is the sum of the percent of cases (1) and (2).

Table 6.2 lists the 4 combinations for each customized Stockfish instance according to depth. As shown in Table 6.2, the highest accuracy rate was achieved for depth=1 with a rate of 61.3%, and the accuracy decreased for each additional *ply*.

Table 6.2 - Computing *Customized Stockfish Model* accuracy rate per depth
According to the definition of the *Customized Stockfish Model*, the accuracy rate is computed as the sum of cases (1) *Difficult, Stockfish-Mistake* and (2) *Easy, Stockfish-Correct.*

	Difficult		Easy		
Depth	Stockfish - Mistake	Stockfish - Correct	Stockfish - Mistake	Stockfish - Correct	Accuracy
1	18.9%	33.1%	5.7%	42.4%	61.3%
2	16.1%	35.9%	3.6%	44.5%	60.6%
3	14.4%	37.6%	2.8%	45.3%	59.6%
4	12.7%	39.3%	2.7%	45.4%	58.1%
5	10.8%	41.2%	2.9%	45.2%	56.0%
6	9.2%	42.8%	2.2%	45.9%	55.0%
7	6.8%	45.2%	1.8%	46.3%	53.0%
8	5.4%	46.6%	1.0%	47.1%	52.5%
9	4.0%	48.0%	0.6%	47.5%	51.5%
10	2.6%	49.4%	0.4%	47.7%	50.3%

We ran another position-based ML experiment using the raw information of the positions. The features were defined as in 4.5.2. We used 100,000 observations for training and 10,000 for testing (See Section 3.3.2). The highest accuracy rate was obtained when using the ANN ML technique: 61.2%. See full results in Table 6.3.

Table 6.3 - Position Based Experiment.
Accuracy rate per ML technique.

	ANN	NB	RF
Accuracy	61.2%	56.1%	51.3%

The move-based approach clearly outperformed both position-based approaches. For comparison see Table 6.4.

Table 6.4 - Position-based vs. Move-Based
Accuracy rate per model.

	Accuracy
Move based Model (ANN) *1-before 1-after*	66.0%
Move based Model (ANN) *1-before 3-after*	71.6%
Move based Model (ANN) *1-before 5-after*	81.1%
Position based Model (ANN)	61.2%
Customized Stockfish Model (depth=1)	61.3%

6.2.3 Possible Usage - Initial Difficulty for New Puzzles

The existing rating was determined based on the attempts of many puzzle-solvers to solve the problem. This means new puzzles have no rating and an easy puzzle can be assigned to a strong player, while a difficult puzzle can be assigned to a newcomer. The capability to predict a difficulty level can be used as a better starting point for new puzzles.

6.3 Computing Puzzle Tags

Each puzzle entry comprises a list of chess themes associated with the puzzle. The themes were initially added programmatically based on precisely defined rules. The themes were later refined based on user feedback. The database comprises 60 different themes. We focused on themes which appeared in at least 2% of all puzzles.

6.3.1 Categorizing Chess Themes

The themes can be categorized as: tactical themes, phase of the game, puzzle length, mating sequences, and player level (of original game). The themes and categories can be referenced in Table 6.5.

6.3.2 Balancing Observations

We ran a move-based and position-based ML experiment for each of the themes. For each model, we kept the number of observations with and without the theme balanced. For each theme we collected up to 50,000 puzzles which were tagged with the theme and 50,000 without. In case fewer than 50,000 tagged puzzles existed, we used all puzzles with the theme and matched them with puzzles without the theme. As in the previous experiment, we used 3 ML techniques and a varying number of *after moves*. In all models, 10,000 puzzles were used for testing and the rest for training (See Section 3.3.2).

6.3.3 Results

The ANN algorithm performed slightly better than the RF, and considerably better than NB. Table 6.5 summarizes the result for the ANN algorithm.

Table 6.5 - Predicting chess themes
Accuracy rate using ANN per number of moves used and position-based model.

Category	Theme	1-*before* 1-*after*	1-*before* 3-*after*	1-*before* 5-*after*	Position
Length	One Move	66.1%			60.9%
Length	Very Long	59.4%	69.4%	85.7%	59.1%
Length	Short	57.1%	68.8%		52.6%
Length	Long	54.8%	67.9%	94.5%	53.9%
Mate	Mate	83.3%	93.6%	94.6%	70.1%
Mate	Mate In 1	80.0%			69.2%
Mate	Mate In 2	78.8%	91.9%		69.4%
Mate	Mate In 3	74.7%	86.6%	97.3%	73.9%
Phase	Opening	75.6%	79.8%	82.4%	98.0%
Phase	Endgame	71.9%	75.6%	78.4%	99.9%
Phase	Middlegame	66.5%	71.2%	74.8%	97.7%
Score	Advantage	68.8%	71.4%	72.0%	66.0%
Score	Equality	63.1%	62.5%	60.6%	59.1%
Score	Crushing	62.7%	67.1%	68.2%	62.9%
Source	Master	55.3%	57.3%	57.8%	65.3%
Tactic	Rook Endgame	89.2%	94.5%	95.1%	99.2%
Tactic	Back Rank Mate	89.1%	95.4%	96.0%	94.6%
Tactic	Hanging Piece	83.6%	90.7%	89.9%	61.7%
Tactic	Quiet Move	80.9%	85.8%	86.0%	66.6%
Tactic	Advanced Pawn	78.6%	92.5%	91.5%	83.4%
Tactic	Kingside Attack	78.2%	84.2%	83.5%	84.8%
Tactic	Attraction	75.8%	90.1%	91.8%	66.3%
Tactic	Skewer	75.7%	88.6%	80.9%	69.6%
Tactic	Defensive Move	75.5%	88.0%	87.5%	66.2%
Tactic	Discovered Attack	75.2%	85.4%	76.7%	62.4%
Tactic	Fork	74.1%	82.1%	76.7%	65.8%
Tactic	Sacrifice	70.3%	89.4%	89.8%	68.1%
Tactic	Deflection	68.8%	82.3%	75.3%	64.3%
Tactic	Pin	67.0%	71.3%	66.0%	65.8%

All puzzles with 'One Move' and 'Mate In 1' themes had only 1 *after* moves in them (See Section 6.2.1), and therefore no value was computed for the 3-*after* and 5-*after* experiments. Similarly, puzzles with 'Mate In 2' had only 3 *after* moves in them, and therefore no value was computed for the 5-*after* experiment. The reason no puzzle length existed for these themes is because of the ad hoc definition of these themes. For example, 'One Move' theme was defined as puzzles with a solution of length 1.

The results are statistically significant (See Section 3.3.1). The move-based features clearly outperform the position-based features for most themes. The exceptions are the ‘phase’ category of themes and the tactical themes of "Rook Endgame" and "Kingside Attack". The 3-*after* and 5-*after* categories are roughly the same, but clearly perform better than the 1-*after* setup. Note that 81.1% of puzzles contain exactly 3 *after* moves (See Section 6.2.1), this means the 5-*after* model cannot be applied to them, thus the 3-*after* model is more useful.

6.4 Possible Usage- Finding Errors in Database

The models described above can be useful in calculating chess themes of puzzles. Seemingly, the models do not offer any added value, since the foundation of the puzzle DB already exists of code which automatically tags the puzzles. Nevertheless, we argue it is almost impossible to define programmatically tactical themes in chess, and therefore the trained ML models may serve as an additional layer for cross validation. In this section we describe how the models can be used to find puzzles where tactical themes were missed (false-negative, a Type 2 error) and puzzles that are accidentally tagged with a chess theme but really are not related to that tactical theme (false-positive, a Type 1 error).

We focus only on tactical themes, because non-tactical themes are usually precisely defined, and therefore there is no added value calculating them in an alternative manner. For example, a puzzle with a theme like 'Mate In One" is precisely defined as a puzzle where the solution is one move which results in a mating positions, this is easy to define programmatically.

The fact the puzzle DB cannot be error free was also known to the puzzle owner. In fact, the DB documentation cites that the themes were refined based on user feedback. We suggest a more efficient method to find errors in the DB.

Using the probability feature of RF, we extracted the probabilities of the classifications (See Section 3.3.3), and manually checked cases where the model strongly leaned in a direction other than the actual tag. We deviated from the previous experiments in 2 ways: (1) We used 100 DTs (instead of default 10) to get a better probability metric. (2) We kept the original ratio between the observations with and without the theme. For example, in this experiment only 2% of observations used had the ‘skewer’ theme, while in the previous experiment the observations were artificially balanced (See Section 6.3.2). The second change was applied because we were interested in a natural prediction as opposed to the previous experiment where we were interested in comparing a 50% random guess to accuracy rate obtained using the model.

For each tactical theme, we computed a ML model, and checked the first 100,000 puzzle entries to see whether 90% of DTs agreed on the same classification. If the DTs agreed, we checked whether the classification matched the actual puzzle-entry tag in the DB.

We distinguished between 5 groups: (1) Cases where the RF did not lean strongly in one direction or another, (i.e., less than 90% of DTs agreed on a specific classification), for these cases we did not compare the model prediction to the actual DB entry value. (2) *True*-

Negative cases: Model predicts "no tag" and in the DB there is "no tag". (3) *True-Positive cases*: Model predicts "tag" and in the DB the puzzle was tagged. (4) *False-Positive* cases: Model predicts "no tag", but in the database it was tagged. (5) *False-Negative* cases: Model predicts "tag" but in the database there was no tag.

Table 6.6 lists the number of incidences, in each of the 5 groups above (for the first 100,000 entries in the DB). As shown in Table 6.6, the contradicting groups, (i.e., groups (4) and (5)), account for less than 1% of all cases.

Table 6.6 - Model with 90% confidence using RF model - Compared to actual tag in database
For each tactical theme, listing the number of puzzle entries of tag/no-tag per model and database, for first 100,000 cases puzzles in the database.

Model Confidence	Less than 90%	Over 90%			
Model Prediction		No Tag	Tag	Tag	No Tag
Puzzle Entry		No Tag	Tag	No Tag	Tag
Tactic					
Back Rank Mate	3375	93740	2755	39	91
Hanging Piece	6073	93591	221	115	0
Quiet Move	16111	82910	662	313	4
Advanced Pawn	8198	87720	3915	167	0
Kingside Attack	17597	80951	1110	289	53
Attraction	5567	88192	6031	201	9
Skewer	3659	95474	625	240	2
Defensive Move	17171	81957	585	285	2
Discovered Attack	12022	86964	653	360	1
Fork	30551	64094	4772	554	29
Sacrifice	19459	70781	9531	208	21
Deflection	12310	86056	1259	375	0
Pin	21228	77735	341	694	2

6.4.1 Manual Observations

For each tactic we manually examined several puzzles in each group. Groups (1), (2) and (3) were as expected: The puzzle theme in the database always seemed to be correct. Among groups (4) and (5), we found many interesting use-cases. Sometimes, we found obvious mistakes of the model, while in other cases it seemed as if there were obvious mistakes in the database, and for still other cases it seemed like a subjective call. Since we did not observe enough puzzles, and since most times the decision was subjective, it was very hard to quantify the number of errors or potential errors found.

In the appendix (Chapter 9), we describe several contradicting cases, and share our subjective insights.

6.5 Summary

Raw chess moves can be used to reverse-engineer some of the calculated attributes of the Lichess puzzle database. This allows assigning an initial difficulty level to brand new puzzles and suggests an efficient method to finding errors in the puzzle tagged themes. For carrying out these tasks, the move-based models are more effective than the position-based models.

7. Summary and Conclusions

The raw moves were successfully used as ML features in 4 different areas of chess automation: (1) predicting game outcome, (2) predicting human mistakes, (3) determining difficulty of chess puzzles, and (4) determining chess themes of chess puzzles. The most important conclusion is that moves contain implicit information about the game. In addition, it showed how the models can be used to extract useful chess lessons.

7.1 Results

For each of the move-based experiments, one or two competing position-based experiments were defined and tried. In the area of predicting the game outcome the position-based experiments performed better, while for all other tasks the move-based approached yielded better results.

Several feature refinement experiments were tried, using fewer moves than in the initial setup, and using attributes of the moves other than the attributes exposed in the *SAN notation* (the commonly used notation). These experiments were deployed to check which moves and which attributes of the moves are most significant for carrying out the specified tasks.

Regarding the question which moves are most significant: the moves closer to the critical position proved to be more useful. In the game-outcome prediction models using the last 3 moves seemed to perform just as well, as when using all the moves of the game. In the mistake-prediction models, the adjacent single *before move* and single *after move* seemed to perform just as well as all 10 surrounding moves.

Regarding the question which attributes of the moves are most significant, our work shows that the attributes used in the *SAN notation*, are more helpful than those used in the *UCI format*, justifying the adaptation of *SAN* to be the standard notation. The most important attribute for predicting whether the correct move would be found, was the capture indication.

7.2 Machine Learning Techniques

We ran multiple ML experiments using 4 different learning techniques: (1) SVM (2) ANN (3) RF, and (4) NB (See Section 3.3.3). When running on small datasets, the SVM performed well, but when moving to larger datasets the SVM ran slowly, consumed a lot of memory, and did not perform better than the RF and ANN models. In terms of accuracy rate, the ANN model usually performed slightly better than the RF model, but also took longer to run. The RF model included valuable additional information about the experiments, such as the probability of the classifications and the 'importance' of the features. The NB performed worse than the other models, but its speed was an advantage compared to other models.

7.3 Future Research

The work can be extended in at least 3 different directions: (1) extracting more chess lessons, (2) extending to more chess-automation areas, and (3) developing better features.

In this work we demonstrated that useful chess lessons can be extracted from the models. In order to truly understand these lessons, time needs to be spent analyzing related positions. (As described in the Appendix, Chapter 9). A closer look would result in more questions to explore, which would require development of more automatic tools, which in turn would require analyzing more phenomena.

In this work, we covered 4 areas of chess automation. There are other areas of chess automation where chess moves can be applied. One example is identifying players, or player

characteristics, using chess moves. The experiments in Chapter 4 were tested on 2 different datasets of chess games: Professional players in official games and online games of all levels of players. The same experiments provided different results on each of these datasets. This may show that we can predict what chess collection a game belongs to based on moves. A practical use is identifying players cheating in online chess games using a chess-engine as an aid. If such behavior can be identified, it could be very helpful.

In this work we assess various features for ML applications. Though our primary focus is move-based features, also position-based features proved useful. It may be even more useful if we combine attributes from both resources and create an improved feature. Such an attempt can be extended by combining information such as human defined features, or information calculated using chess engines. This can help in the quest for creating an ultimate feature which can yield higher accuracy rates than those using any feature-type alone.

8. References

[1] J. Furnkranz, "Knowledge discovery in chess databases: A research proposal", *Austrian Research Institute for Artificial Intelligence, Tech.Rep.OEFAI-TR-97-33,* 1997.

[2] O. David-Tabibi, M. Koppel and N. S. Netanyahu, "Genetic algorithms for mentor-assisted evaluation function optimization", in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation,* 2008.

[3] O. David-Tabibi, H. J. Van Den Herik, M. Koppel and N. S. Netanyahu, "Simulating human grandmasters: Evolution and coevolution of evaluation functions", in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation,* 2009.

[4] O. David-Tabibi, H. J. Van Den Herik, M. Koppel and N. S. Netanyahu, "Genetic Algorithms for Evolving Computer Chess Programs", IEEE Transactions on Evolutionary Computation, Vol. 18, (5), pp. 779-789, 2014.

[5] A. Hauptman and M. Sipper, "Using genetic programming to evolve chess endgame players", in *Proceedings of the 2005 European Conference on Genetic Programming,* 2005.

[6] A. Hauptman and M. Sipper, "Evolution of an efficient search algorithm for the mate-in-N problem in chess", in *European Conference on Genetic Programming,* 2007.

[7] O. E. David, N. S. Netanyahu and L. Wolf, "DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess", *Artificial Neural Networks and Machine Learning - ICANN 2016*, pp. 88-96, 2016.

[8] B. Oshri and N. Khandwala, "Predicting moves in chess using convolutional neural networks", In: Stanford University Course Project Reports - CS231n: Convolutional Neural Networks for Visual Recognition, 2016.

[9] M. Sabatelli, "Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks", Ph.D. thesis, *University of Groningen,* 2017.

[10] M. Sabatelli, F. Bidoia, V. Codreanu and M. A. Wiering, "Learning to evaluate chess positions with deep neural networks and limited lookahead", in *Icpram,* 2018.

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A Guez, M. Lanctot, L. Laurent, S Dharshan, K. Thore, G. Timothy, L. Simonyan, D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play,", in *Science,* vol. 362*, (6419)*, pp. 1140, 2018.

[12] R. McIlroy-Young, S. Sen, J. Kleinberg, A. Anderson. "Aligning superhuman ai with human behavior: Chess as a model system." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining,* 2020.

[13] R. McIlroy-Young, R. Wang, S. Sen, J. Kleinberg, A. Anderson. "Learning personalized models of human behavior in chess.", in *arXiv preprint arXiv:2008.10086,* 2020.

[14] Leela Chess Zero, Open source neural network based chess engine, https://lczero.org, [Accessed March 29, 2022].

[15] M. M. Masud, A. Al-Shehhi, E. Al-Shamsi, S. Al-Hassani, A. Al-Hamoud and, L. Khan, "Online prediction of chess match result", in *Pacific-Asia Conference on Knowledge Discovery and Data Mining,* 2015.

[16] Presser, S. and Branwen, G. A, A Very Unlikely Chess Game, https://slatestarcodex.com/2020/01/06/a-very-unlikely-chess-game, [Accessed February 24, 2022].

[17] Cheng, R. T, GitHub, Transformers Play Chess, https://github.com/ricsonc/transformers-play-chess, [Accessed February 24, 2022].

[18] D. Noever, M. Ciolino and J. Kalin, "The Chess Transformer: Mastering Play using Generative Language Models", in *arXiv preprint arXiv:2008.04057,* 2020.

[19] S. Toshniwal, S. Wiseman, K. Livescu, K. Gimpel, "Learning Chess Blindfolded: Evaluating Language Models on State Tracking", in *arXiv preprint arXiv:2102.13249,* 2021

[20] Wikipedia, Simultaneous exhibition, https://en.wikipedia.org/wiki/Simultaneous_exhibition, [Accessed September 30, 2021].

[21] Free Internet Chess Server (FICS), https://www.freechess.org/, [Accessed September 30, 2021].

[22] PGN Mentor, https://www.pgnmentor.com/, [Accessed September 30, 2021].

[23] *ChessOK,* https://chessok.com, [Accessed September 30, 2021].

[24] Lichess open database, https://database.lichess.org/#puzzles, [Accessed September 30, 2021].

[25] G. Costeff, "The chess query language: Cql", *ICGA Journal,* vol. 27, *(4),* pp. 217-225, 2004.

[26] D. Ganguly, J. Leveling and G. J. Jones, "Retrieval of similar chess positions", in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval,* 2014.

[27] M. Bizjak, "Automatic Recognition of Similar Chess Motifs", Master's. thesis, *University of Ljubljana*, 2020.

[28] Wikipedia, Algebraic notation, https://en.wikipedia.org/wiki/Algebraic_notation_(chess), [Accessed September 30, 2021].

[29] Stockfish, a free and open-source chess engine, https://stockfishchess.org/, [Accessed September 30, 2021].

[30] Stockfish-python, class to integrate Stockfish chess engine with Python, https://github.com/zhelyabuzhsky/stockfish, [Accessed September 30, 2021].

[31] Python-chess, a chess library for Python, https://python-chess.readthedocs.io/en/latest/, [Accessed September 30, 2021].

[32] FEN-to-Image, Chess diagrams, http://www.fen-to-image.com, [Accessed September 30, 2021].

[33] Quora, Support Vector Machine, https://www.quora.com/How-do-support-vector-machines-work, [Accessed September 30, 2021].

[34] Quora, Kernel Trick, https://www.quora.com/What-is-the-kernel-trick, [Accessed September 30, 2021].

[35] Scikit-Learn, a free software machine learning library for Python, https://scikit-learn.org, [Accessed September 30, 2021].

[36] Quora, Decision Tree, https://www.quora.com/What-is-an-intuitive-explanation-of-a-decision-tree, [Accessed September 30, 2021].

[37] TensorFlow, a free and open-source software library for machine learning, https://www.tensorflow.org, [Accessed September 30, 2021].

[38] Wikipedia, Artificial Neural Network https://en.wikipedia.org/wiki/Artificial_neural_network, [Accessed September 30, 2021].

[39] Waikato Environment for Knowledge Analysis (Weka), is a free software for Machine Learning, https://www.cs.waikato.ac.nz/ml/weka/, [Accessed September 30, 2021].

[40] Lichess, https://lichess.org, [Accessed September 30, 2021].

[41] GitHub, puzzle themes, https://github.com/ornicar/lila/blob/master/translation/source/puzzleTheme.xml, [Accessed September 30, 2021].

[42] Wikipedia, Forsyth–Edwards Notation (FEN), https://en.wikipedia.org/wiki/Forsyth–Edwards_Notation, [Accessed September 30, 2021].

[43] Wikipedia, Glicko Rating System, https://en.wikipedia.org/wiki/Glicko_rating_system, [Accessed September 30, 2021].

[44] Wikipedia, Elo rating system, https://en.wikipedia.org/wiki/Elo_rating_system, [Accessed September 30, 2021].

[45] G. Burgess, *The Mammoth Book of Chess.* Hachette UK, 2009.

9. Appendix: Chessic Findings Explained

The purpose of the appendix is to elaborate upon chessic findings resulting from ML experiments described in our work. The reasoning in the appendix is subjective, and not rigorous as in other parts of this work. The discussion in the appendix is chess-oriented, with some references to *The Mammoth book of chess* by *Burgess* [45].

9.1 Critical Moves Sequences

Using the RF 'probability' feature, we extracted lists of popular 2-move sequences strongly associated with players finding or not-finding the correct continuation (Section 5.6). In the *top-correct* 2-move sequences, there were many last-*before*-first-*after* sequences which were popular, and among *top-mistakes* there were many first-*after*-second-*after* sequences (and also many first-*after*-third-*after*, not reported here).
In this section we outline and show the key characteristics of these popular 2-move sequences.

9.1.1 Top-Correct Sequences

In the *top-correct* sequences (Table 5.8), the first move is the move before the critical position. And the model tells us the chances the correct move (i.e. the second move in the sequence) would be found is very high.
From looking at the *top-correct* sequences, it is apparently clear why the correct move is always found. In 37/40 of the sequences the correct move is an obvious re-capture, and in 3/40 the correct move is an obvious bishop retreating move after the bishop was threatened.

Table 5.8 – *Top-Correct* sequences (Copied from Chapter 5)
Top 40 last *before m*ove - first *after m*ove sequences showing up in RF model where all 10 DTs agree the correct move would be found.
See Figure 5.2 illustrating how a last *before m*ove - first *after m*ove is derived.
See Section 3.1 for an explanation about chess notation.

1. … Bxc3+ 2. bxc3	1. Bxd6 Qxd6	1. Nxd5 exd5	1. … Nxd3 2. Qxd3
1. Bxg7 Kxg7	1. … Nxe3 2. fxe3	1. Bxf6 gxf6	1. … Bxf3 2. Bxf3
1. Bxc6+ bxc6	1. … Bxg2 2. Kxg2	1. Bxf6 Bxf6	1. Bxf5 exf5
1. … Bxd3 2. Qxd3	1. … Nxe4 2. Bxe4	1. … Bxe2 2. Qxe2	1. Bxe5 dxe5
1. Nxe5 dxe5	1. … Bxf3 2. gxf3	1. … Bxe3 2. fxe3	1. … Nxd4 2. Qxd4
1. Nxc6 bxc6	1. Bxe7 Qxe7	1. g4 Bg6	1. Nxf7 Kxf7
1. … Nxc3 2. bxc3	1. Nxe6 fxe6	1. … Nxd4 2. exd4	1. … Bxf3 2. Qxf3
1. Bxe6 fxe6	1. Bxf7+ Kxf7	1. … g5 2. Bg3	1. … Nxd4 2. cxd4
1. … Nxe4 2. dxe4	1. Bxc6 bxc6	1. … Bxe4 2. dxe4	1. … b5 2. Bb3
1. … Bxc3 2. bxc3	1. Nxe5 Bxe5	1. Nxe7+ Qxe7	1. … Bxf4 2. exf4

Figure 9.1 demonstrates the sequence *1. ... Bxc3+ 2. bxc3* where black captures a defended knight on *c3* and white finds the only good response: recapture on *c3*. Figure 9.2 demonstrates the sequence *1. g4 Bg6* where white attacks black's bishop, and black finds the correct response: retreat to *g6*.

Figure 9.1
1. ... Bxc3+ 2. bxc3

Figure 9.2
1. g4 Bg6

9.1.2 Top-Mistake Sequences

In the *top-mistake* sequences, the first move is the best move in the critical position (Table 5.9). In all these sequences the first move is a move of seizing an initiative. The player who should play the move is not threatened, and as the model teaches us, the player rarely finds the unusual move, thus missing the opportunity to take the lead.

Table 5.9 – *Top-Mistake* sequences (Copied from Chapter 5)
Top 12 first *after m*ove - second *after m*ove sequences showing up in RF model where all 10 DTs agree the correct move would not be found.
See Figure 5.2 illustrating how a first *after m*ove - second *after m*ove is derived.
See Section 3.1 for an explanation about chess notation.

1. Bxf7+ Kxf7	1. … Qh4+ 2. g3	1. Qh5+ Ke7	1. … Bxf2+ 2. Kxf2
1. Qh5+ g6	1. … Qh4+ 2. Ke2	1. … Qa5+ 2. c3	1. Bxf7+ Kd7
1. Qa4+ Nc6	1. … Qh4+ 2. Kd2	1. Bxa6 Nxa6	1. Qh5+ Kd7

Figure 9.3
1. Bxf7+ Kxf7

Figure 9.4
1. Qh5+ g6

In Figures 9.3 and 9.4 white is not threatened. In Figure 9.3 white needs to "look out of the box" to realize *Bxf7+* generates a deceive attack (at a temporary cost of a bishop). And in Figure 9.4 white needs to realize the not-so-common *Qh5+* would force black to lose the rook in *h8* if black responds with *g6* (as in Figure 9.4; alternatively, black can respond with *Ke7* allowing white a deadly attack on the king).

We need to remember the model was trained using online-blitz games, with players of all levels; The fact the model predicts with 100% certainty that the correct move won't be found, does not reflect the real chance a player would miss the move. It is likely that strong players, with more time to think per move would find the correct move in such situations.

The sequences *1. Qh5+ g6*, *1. Qh5+ Ke7* and *1. Qh5+ Kd7* in Table 5.9, reflect 3 playable continuations for white's *Qh5+* initiative; and *1. … Qh4+ 2. g3*, *1. … Qh4+ 2. Ke2* and *1. … Qh4+ 2. Kd2* represent the same chess-motif, with black's symmetrical *Qh4+* initiative. This chess motif is the principal theme in the *Damiano Defense*, (page 113 in [45]).

Similarly, *1. Bxf7+ Kxf7* and *1. Bxf7+ Kd7* in Table 5.9, reflect 2 playable continuations for white's *Bxf7+* initiative; and *1. … Bxf2+ 2. Kxf2* represents the same chess motif, with black's symmetrical *Bxf2+* initiative. The *Bxf2+* is the underlying plan in the *Wilkes-Barre Countergambit*, (page 145 in [45]). This motif appears often in the opening stage of the game. Another example can be in the *Modern* trap, (see page 167 in [45]).

9.1.3 Lesson Learn

The *top-correct* sequences are not of any added-value. All players find the correct move in these cases. The *top-mistake* sequences are of high value, they reflect commonly missed initiatives. Being conscious of these sequences can help players seize an advantage when the situation appears in real games.

9.2 Model and Puzzle DB Contradictions

In chapter 6 we described how using the 'probability' capability of the RF model, we can extract puzzles with errors (See Section 6.4).

The interesting observations are those where the model is very confident of the classification but disagrees with the actual value in the database (last 2 columns in Table 6.6). We examined multiple puzzles manually and came up with interesting findings. Sometimes, there were obvious mistakes of the model. In other cases it seemed like clear errors in the puzzle. And, for some cases it seemed like a subjective call. Usually the reasoning for the mismatch was clear, in the following sections we would present some examples.

Table 6.6 - Model with 90% confidence using RF model - Compared to actual tag in database
(Copied from Chapter 6)
For each tactical theme, listing the number of puzzle entries of tag/no-tag per model and database, for first 100,000 cases puzzles in the database.

Model Confidence	Less than 90%	Over 90%			
Model Prediction		No Tag	Tag	Tag	No Tag
Puzzle Entry		No Tag	Tag	No Tag	Tag
Tactic					
Back Rank Mate	3375	93740	2755	39	91
Hanging Piece	6073	93591	221	115	0
Quiet Move	16111	82910	662	313	4
Advanced Pawn	8198	87720	3915	167	0
Kingside Attack	17597	80951	1110	289	53
Attraction	5567	88192	6031	201	9
Skewer	3659	95474	625	240	2
Defensive Move	17171	81957	585	285	2
Discovered Attack	12022	86964	653	360	1
Fork	30551	64094	4772	554	29
Sacrifice	19459	70781	9531	208	21
Deflection	12310	86056	1259	375	0
Pin	21228	77735	341	694	2

9.2.1 Clearly Missing Tag

According to the puzzle documentation [41], a King Side Attack is “An attack of the opponent's king, after they castled on the king side”. We found many cases where this was the case, but the tag was missing. For example, the solution of a chess puzzle in Figure 9.5 matches the definition, but the puzzle entry was missing the tag.

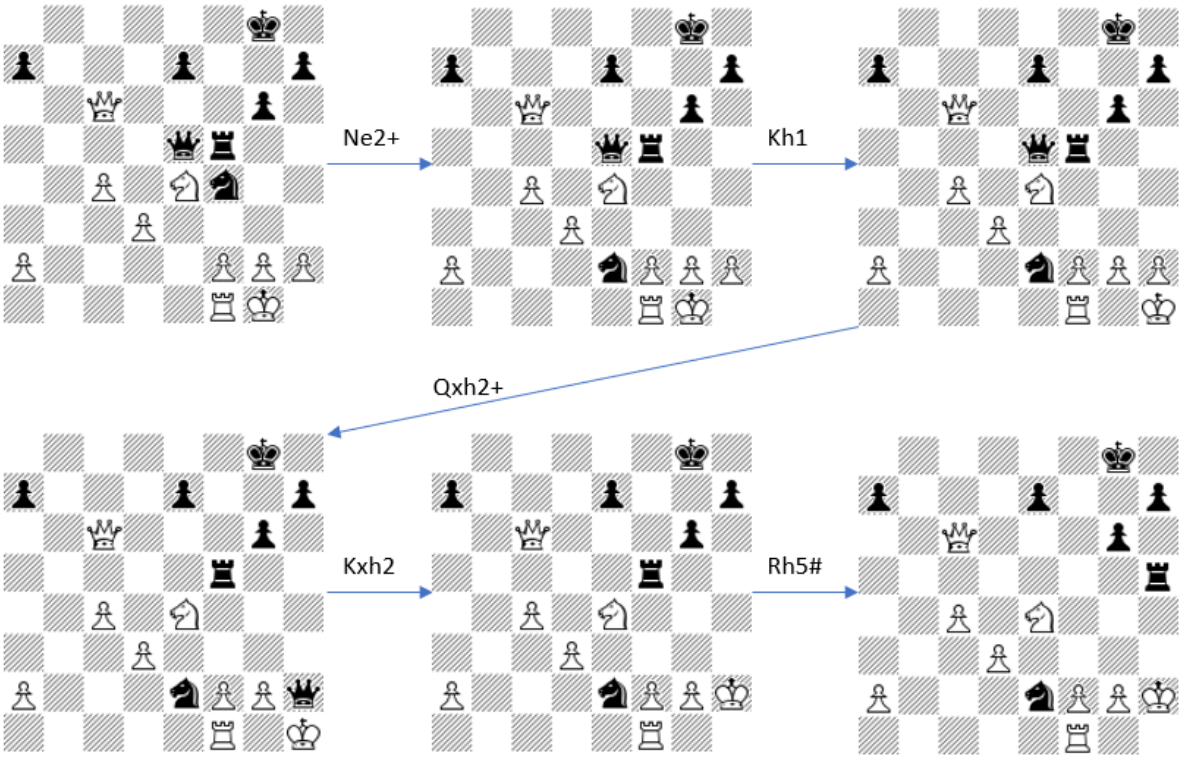


Figure 9.5
1. ... Ne2+ 2. Kh1 Qxh2+ 3. Kxh2 Rh5#
Model: Tagged as King Side Attack
Entry in DB: No tag

9.2.2 Definition Dependent

In the puzzle DB [41], a “Back Rank Mate” is defined as a “Checkmate the king on the home rank, when it is trapped there by its own pieces”. This definition differs from the definition in [45] (page 454): “A simple mating idea in which a rook or queen checks a king along its first rank, and, thanks to the presence of a row of pawns along the second rank, it is also mate”.

Here we show one potential false-positive and one potential true-negative.

Figure 9.6 shows the final state of a puzzle not tagged as ‘Back Rank Mate’. In the mating positions, black rook "gives *check"* along the first rank, and due to the presence of 2 white pawns in the second rank, and the fact the square *h2* is attacked, it is mate. We think this fits with the second definition, but maybe not intended to be covered by the first definition since the king is not fully trapped by its own pieces.

Figure 9.6 - Back Rank Mate
Model: Tagged as Back Rank Mate
Entry in DB: No tag

While the example in Figure 9.6 is debatable, Figure 9.7 demonstrates a clear difference between the definitions. Figure 9.7 is a final state of a puzzle tagged as ‘Back Rank Mate’. In the position, the king is on the home rank and trapped by its own pieces, but it is not checked by a rook or queen.

Figure 9.7 - Back Rank Mate
Entry in DB: Tagged as Back Rank Mate
Model: No tag

We incline toward considering these cases as mistakes in the puzzle DB, since the underlying themes of Figure 9.6 are very similar to most ‘Back Rank Mate’ puzzles, while the underlying themes of Figure 9.7 are not similar to most ‘Back Rank Mate’ puzzles.

9.2.3 Chess Theme Present with no Impact

In the puzzle DB [41] a fork is defined as “A move where the moved piece attacks two opponent pieces at once”. In Figure 9.8 after the first move the black queen is attacking 2 pieces, the king on *g1*, and the loose bishop on *b2*, but this clearly is not the intention of the puzzle. The next move by white neglects the loose bishop and continues with a forced mate sequence (capturing the bishop would actually let white win the game immediately). Since there is no intention of taking advantage of the *fork*, we think this position should not be tagged as a *fork* theme.

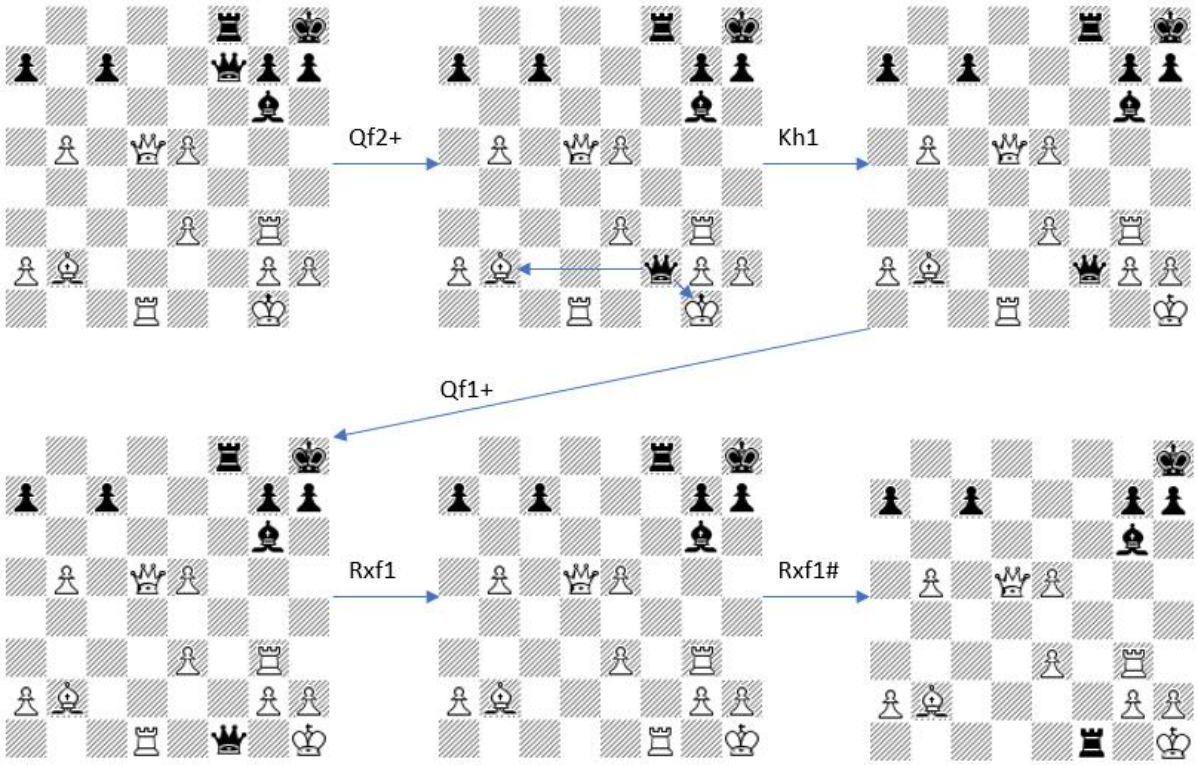


Figure 9.8 - Fork
1. ... Qf2+ 2. Kh1 Qf1+ 3. Rxf1 Rxf1#
Entry in DB: Tagged as Fork
Model: No Fork

9.2.4 Summary

All 4 examples show potential mistakes in the puzzle database. By utilizing the ‘probability’ capability of the RF model it was very easy to find such potential mistakes. In 3 out of 4 examples it also seems clear what caused the error in the DB. These examples also show how difficult it is to define chess themes using code, magnifying the fact that using the RF model based on move-based features as a cross-validation model may be very useful.

תוכן עניינים

תקציר

בעבודה זה אנו מראים שניתן להשתמש במהלכי השחמט הגולמיים בתור מאפיינים במשימות למידה חישובית בשחמט. המשאב הטבעי עבור בעיות למידה חישובית הם מאפיינים המבוססים על עמדות. כעיקרון, כשמשחקים שחמט מתייחסים לעמדה ולא למהלכים. למעשה, כמעט בכל המחקר הקיים בתחום הלמידה החישובית בשחמט, המאפיינים נגזרו מתוך העמדות.
בעבודה זו אנחנו משתמשים במאפיינים המבוססים על מהלכים, בארבעה תחומים של אוטומציה בשחמט: (1) חיזוי תוצאת המשחק, (2) חיזוי טעויות אנושיות, (3) קביעת רמת קושי של חידות, (4) קביעת מוטיבים שחמטיים של חידות (לדוגמא, קביעה שחידה שחמטאית כוללת מוטיב שחמטאי טקטי של *מזלג*).
בעבודה זו אנחנו מדגימים שניתן לבצע את המשימות שהוגדרו לעיל בעזרת המהלכים (בלבד) עם אחוזי דיוק גבוהים. לדוגמה, לזהות עם 82.1% דיוק שחידה שחמטאית כוללת *מזלג*, ולנבא עם 78.2% דיוק ששחקן ימצא את המהלך הנכון. בנוסף בעבודה זו אנחנו מכריעים אילו מהלכים ואילו תכונות של המהלכים מאפשרים את החיזוי. לדוגמה, על מנת לחזות את תוצאת המשחק אנחנו מראים שמספיק להשתמש במהלכים האחרונים ששוחקו. בעבודה גם מראים שחוץ מאשר בתחום חיזוי-תוצאת-משחק, השימוש במאפיינים המבוססים על מהלכים מניב תוצאות טובות יותר מאשר שימוש במאפיינים המבוססים על עמדות. לבסוף, בעבודה גוזרים שיעורי שחמט מועילים, למשל מגלים אילו מהלכים שחקנים נוטים להחמיץ.

תודות

ברצוני להודות למנחים שלי,
פרופ׳ לאוניד ברנבוים וד״ר מיריי אביגל
על הפגישות הפוריות וההנחיה לאורך כתיבת התזה

ברצוני להודות
למרליאנה פריסטלר ואנדרו פריסטלר
על עבודת ההגהה ועל השיתוף בהערות המועילות

האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב

שימוש במהלכים גולמיים בייישומי למידה חישובית בשחמט

עבודת תזה זו הוגשה כחלק מהדרישות
לקבלת תואר M.Sc. במדעי המחשב
האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב

על-ידי
מרדכי גורן

העבודה הוכנה בהדרכתם של
פרופ' לאוניד ברנבוים וד"ר מיריי אביגל

מרץ 2022