Fishnet: Discriminating between Human and AI Chess Players

CS 229 - Final Report

Cristobal Sciutto (*csciutto*) - 06099425
Teammates: Lucas Sato (*satojk*) and Sean Roelofs (*sroelofs*), both enrolled in CS 221.

1 Abstract

We attempt binary classification on Human vs AI chess games, identifying the player that corresponds to the AI. We explore different feature extractions for chess games. We apply Logistic Regression to the dataset to establish a baseline. We then explore SVMs (Linear, Polynomial, and RBF kernels), ANNs, and RNNs for better performance. In the end, we achieved 90% accuracy on a training set of 16.8k games from the FICS server. A key element to our success was Principal Component Analysis on sparse feature-sets. We then explore some implementation details, and briefly discuss where we see humanistic chess AIs going. Code for project available at: `github.com/satojk/fishnet`

2 Motivation

In 1997, Deep Blue defeated Gary Kasparov. Finally, after a long history of computer chess starting in the 60s, artificial intelligence had reached its first public milestone[1]. Despite the initial cynicism, modern chess AIs (e.g. Google's Deep Mind) are well superior to modern Grandmasters. Nevertheless, their mechanics are still "dry". Magnus Carlsen, the current World Champion, remarks that when you play the AIs, not only will you inevitably lose, you will also be bored. In this context, it makes sense to develop a kind of chess Turing test. Can the AI play in a way considered "human?" The definition of that Turing test is the task at hand for this project.

3 Related work

We found little existing literature contrasting Human versus AI chess-playing style from a machine-learning perspective. However, in an attempt to identify cheating chess players, some work has been done to identify AI players. Cheating usually consists of being assisted by an AI to evaluate potential moves. The anti-cheating analyses consist mostly of estimating the ELO rating of the given player's performance, and comparing it to previous performances [2]. Unusual spikes of performance are flagged. Similarly, individual moves are evaluated, looking for unusually good performance that would likely require more foresight than expected even from Grandmasters.

Furthermore, it's incredibly relevant to consider recent research from the DeepMind team on AlphaZero, their *tabula-rasa* Chess reinforcement-learning agent [3]. Stockfish and Fritz, more traditional chess bots, use a series of preset evaluation functions, based on human heuristics, and a catalog of historical openings and end-game maneuvers. In contrast, AlphaZero is given only the rules of the game, and incentivized to learn from there. In a recent showoff against Stockfish, AlphaZero lost no games, winning 28, and drawing 72. Kasparov comments in an article for Science that AlphaZero presents a "dynamic, open style" like his own. Unlike other engines which "reflect priorities and prejudices of programmers", the *tabula-rasa* approach "work[s] smarter, not harder." Thus, AlphaZero is approaching the goal of a human-like AI which motivated us originally.

4 Method

4.1 Model

We structure the problem as a binary classification task on the players of a chess game. Unlike the cheating analyses above, we attempt to analyze the game play of an individual game in an attempt to see if the players are AIs. Longer analyses across several games are outside the scope of this project. Given the constraint to a single game, there are two main ways to classify each game:

1. **Discriminator Model**: given a Human vs AI game, $y = 0$ if the AI is White, and $y = 1$ if the AI is Black.
2. **Classifier Model**: $y \in [0, 1]^2$ indicates respectively if the White is an AI and if Black is an AI. Note that this is equivalent to, given an game and a player (Black or White), output $y = 1$ if the player is an AI.

The central distinction between training these two models is the dataset needed. For the Discriminator model, the data set is a series of Human vs AI games, and a label of which is an AI. For the Classifier model, we would train two models, the Black and White classifier. Each is given the same feature set, and a label corresponding to whether the Black or White player, respectively, is an AI. Note that for a data-set of only Human vs AI games, the Black classifier corresponds to the Discriminator model above, and the White classifier is its inverse. Therefore, the Classifier interpretation only makes sense when treating AI vs AI and Human vs Human data-points. This requires a different dataset from the Discriminator model. For the scope of this project, we focused exclusively on the Discriminator model.

To perform the binary classification, four central methods will be employed, of increasing complexity: Logistic Regression, SVMs, ANNs, and RNN. The Log-Reg establishes our baseline of performance. From there, an SVM is tuned to get compelling results. For the Logistic-Regressions and SVMs, we are finding weights associated to the features that either maximize the likelihood of the data or the margin of separation. The benefit of the SVM's with the RBF and Polynomial kernels is the de-linearization of the boundary.

On the other hand, a neural-network achieves this de-linearization of the boundary via a series of layers, successively reducing the input vector into a final output. Through a final sigmoid node, we output a probability. If this probability is larger than 0.5, we consider the prediction to be that black is an AI. For the recurren unit, an LSTM was used, allowing for the input of time-series. On each step, a vector consisting of a synthesis of previous time-steps is appended to the current timestep. This new vector is then weighted and a value is outputed. Furthermore, the vector is used as the synthesis component for the next time-step. In this way, we iteratively apply the timesteps, extracting a notion of temporality, and are able to output several numbers. Note however, that each output depends only on the short-term, that is timesteps immediately before it.

Beyond the models, performing PCA on the dataset to reduce dimensionality of sparse representations of the chessboard was essential to our performance. PCA maps the dataset into a lower-dimension by finding the optimal hyperplane to project the dataset onto, while maintaining as much variance in data as possible.

4.2 Data

The data we have been working with takes the form PGN (Portable Game Notation) files - a standardized chess representation. The two most important pieces of information in the PGN files are the sequence of moves played in the game (line 18), and which player was an AI (lines 05 and 06). These two pieces of information fully define our dataset (the former being our "X", from which we extract features, and the latter being our label, "y"). Here is an example PGN file (irrelevant lines omitted, and file lines added for reference):

```
05> [White "lichess AI level 5"]
06> [Black "romaaaan"]
18> 1. d4 c6 2. Nc3 d5 3. Nf3 Nf6 4. Ne5 Bf5 5. h3 h6 6. Rb1 b5 7. b4 a5 8. e4 axb4 9. exf5
    bxc3 10. Be2 Rxa2 11. Rb3 Ra1 12. Nf3 Ne4 13. Ne5 f6 14. Bh5+ g6 15. Bxg6# 1-0
```

For simplicity, we have been working only with games played between a human and an AI. The difficulty of adding Human vs Human and AI vs AI games to the dataset is having a proper distribution of types of games. Furthermore, each PGN file has to be partitioned into an analysis of the White and Black player, labelling the games accordingly. We believed that while this would provide a slightly more robust model, the overhead needed for this analysis was not justified.

We started working with data from lichess.org. Scraping data from the server's website gave us our initial 2,878 game dataset, split between training, test, and validation in a 0.6, 0.2, 0.2 fashion. Each of the games is between a human and Stockfish Level 5, stockfish being the most powerful open source chess engine available, which Lichess makes available in levels 1 through 8. We decided on analyzing specifically Stockfish level 5 as this would correspond more accurately to an "average but serious" chess player. This dataset, from now on referenced to as the **Lichess dataset**, served to form our baseline, and perform hyperparameter analyses.

Our second dataset was extracted from the Free Internet Chess Server (https://www.freechess.org). We downloaded 84,033 Human vs AI games from 2017. This dataset, from now on referenced to as the **FICS dataset**, was used to guarantee our model was robust to different kinds of AIs, of varying implementation and level.

4.3 Features

For the features, we considered two approaches: (1) providing specific features, based on human-heuristics, or (2) providing sparse feature-sets from which the algorithm can infer. Together with a regularization term, the second approach was more successful as it is less restrictive. Given a comprehensive model of the game-state, features such as "the number of pawns on the board after turn 25" can be easily computed by the algorithm, if deemed beneficial. The following representations were considered, relating to the entire game or to an individual:

4.3.1 State-representation features

1. Origin and destination coordinates of moves: e.g. $(1,4,2,6)$ correspond to the move $1.Nxc7$, i.e. moving a knight from $b5$ to $c7$, capturing a pawn. All moves are transformed into coordinates and flattened into a 1d-array. To guarantee a fixed size of input, the vector is padded to 35 moves, or $35 \cdot 2 \cdot 4$ coordinates.

2. Board-state vector: each board-state is represented as a 64 x 6 x 2 n-array, corresponding to whether or not a player's piece is present in a given square. Note that there are 64 squares on the board, 6 distinct pieces in Chess, and 2 players. The board-state of several turns are compressed into a single 1d-array. To guarantee a fixed size of input, the vector is also padded to 35 moves. Alternatively, we can consider the first and last 5 turns, or any other combination of turns. As explained in the experiments section, the optimal result used was the first 7 even moves, that is, alternating the first 14 moves.

4.3.2 Player-specific features

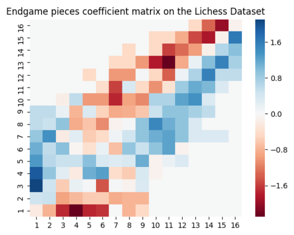
1. Number of pieces left at the end of the game. One-hot encoding of this feature, where we have a single vector with 256 entries all of which but one are zeroes. The single non-zero entry takes the value of 1, and is the $(x \cdot 16 + y)^{th}$ entry of the vector, where x and y are, respectively, how many pieces white and black have at the end of the game.

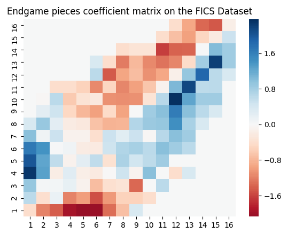
2. Number of controlled squares by each player at a given game state. A controlled square corresponds to a square that can be reached by a piece.

5 Experiments

5.1 Experiment 1: Human Heuristics

We began exploring the problem with human-heuristics of features. Our first approach was the "number of pieces left on the board" encoding model. Run through a Logistic Regression, this gave us our first encouraging result: **Lichess test error:** 0.69, **FICS test error:** 0.66





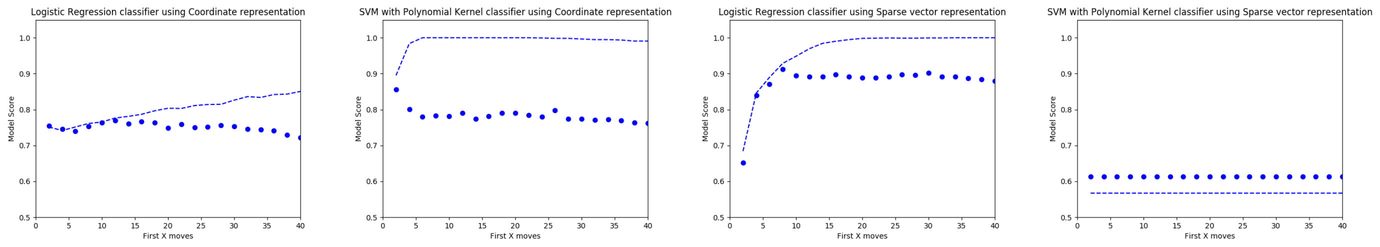
In the figures above, we analyze the weights logistic regression attributes to different outcomes. The x-axis corresponds to the number of white pieces left, y-axis to the number of black pieces left, and the value corresponds to the likelihood that black is the AI. The first figure corresponds to the Lichess dataset, while the second is is the larger FICS dataset. The similarity validates an extrapolation of analyses from the smaller to larger dataset. Furthermore, the anti-symmetrical nature of the heatmap indicates an indifference in playing style of AIs with regards to color.

5.2 Experiment 2: Baseline with Board-state Representations

Impressed with the near 70% accuracy, we doubled down on richer feature combinations, as described in the section above. The primary motivation was attempting to the entirety of the game, rather than just a snap-shot of the final-state. We then ran Logistic Regression and SVM with different kernels on the new representations, leaving default hyper-parameters. The following (train, test) errors were achieved, establishing our baseline of performance.

	Coordinate Representation	Board-state Representation
Logistic Regression	(0.835, 0.741)	(1.000, 0.885)
SVM Linear	(0.847, 0.739)	(1.000, 0.882)
SVM RBF ($\gamma = 0.7$)	(1.000, 0.615)	(1.000, 0.615)
SVM Poly (deg=3)	(0.994, 0.773)	(0.567, 0.619)

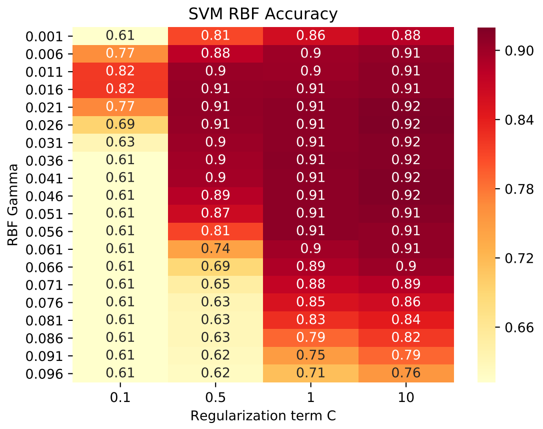
We then focused on both the Logistic Regression and Poly-Kernel, and decided to take a "first X-moves" approach. We tried out different "X"s from 1 to 40. The goal was to attempt to diagnose either over or under-fitting. We got the following accuracies for Lichess dataset, indicating clear over-fitting. In the plots below, the dashed line corresponds to train accuracy, and the dotted to test accuracy:



The analyses above indicated that the first 7 moves were optimal for minimizing overfitting. A similar analysis on the larger FICS dataset gave the value of 14 moves as optimal. Note, however, that we extracted every two moves from the game, given the two-player nature of Chess. This reduces the dimension of the vector without the loss of much information. Furthermore, the polynomial kernel performance was a clear indication that we needed to attempt varying the hyperparameters of the models.

5.3 Experiment 3: PCA and Hyperparamter Optimization

We were confident in the potential of the board-state feature representation. Despite the analysis above, we realized that its sparsity and large dimensionality was the central cause of our overfitting. We opted for performing PCA on the 5376 features extracted (7 moves, 64 squares, 12 values). This reduced the dimension down to around 300 features, preserving 95% of variance in the dataset.



Furthermore, up until this phase of analysis we were relying on default parameters for regularization of our SVMs, and did little experimentation with the γ value of our RBF-kernel or degree of the polynomial kernel. Using the Lichess dataset for validation, we varied over different degrees polynomial kernels with little indication of success. However, when testing for RBF kernels of $\gamma \approx 0.05$, we got great accuracy. This extrapolated well to the FICS dataset:

	SVM $\gamma = 0.04$, $C = 10$	Log-Reg
Accuracy	0.900	0.811

5.4 Experiment 4: Deep and Recurrent Neural Networks

Given the great performance of our SVM with PCA, we decided the next step was to venture into neural networks. Given our 80k samples, we were confident that we had enough data for the task. Furthermore, since the models used up to now compress features into a 1d-array, there is no accounting for the sequentiality of Chess. Thus, a recurrent unit seemed to be a fruitful next approach. This was done via an LSTM implementation from the Keras framework. Given the ease-of-use provided by the framework, we also tried deep neural networks.

For the LSTM, we had to slightly modify our dataset. We continued using the board-state representation, however did not compress the board states into a single vector. Rather, we left each 768 dimensional board-state as a row of a matrix. This was reduced to around 150 features per board-state with PCA. Furthermore, we tested on different subsets of moves, ultimately settling at 15 moves.

We optimized over layer configurations, and early stopping at different epochs. Most of the hyperparameter work was done on the Lichess dataset. We tested the best results on the FICS dataset. Below are our notable results:

	Lichess	FICS
ANN (Layers: 210)	0.904	-
ANN (Layers: 200, 40)	0.890	0.882
ANN (Layers: 60, 50, 12)	-	0.906
LSTM (Output Dim: 10)	0.904	0.860
LSTM (Output Dim: 25)	0.915	-

6 Implementation and Contributions - `github.com/satojk/fishnet`

A major part of the early work was feature extraction and an abstraction for a chess game. This was done with the help of the python-chess library [5]. Due to the large nature of our dataset and computational intensity of our chess abstraction, a caching system was setup to save the games' feature representation. For machine learning, we used the scikit-learn[6] and Keras frameworks[7]. In sum, our pipeline consisted of (1) scrape Lichess or download FICS, (2) read PGN files into chess abstraction, (3) extract feature vectors, (4) PCA, (5) cache features, and (6) perform training and testing with scikit-learn or Keras.

The exact division of labor among team members can be seen on Github. On a higher level, Cristobal Sciutto worked on the Keras/NN modeling, SVM optimization, and caching pipeline. Lucas Sato worked on the Lichess server scraping, PGN-to-feature pipeline, feature-extraction optimization, and data visualizations. Sean Roelofs developed our Chess abstraction class over the python-chess library.

7 Conclusion and next steps

We are very satisfied with our 90% accuracy in identifying an AI chess player using either an SVM with RBF kernel, or a LSTM. Short-term, the following actionable are ways to continue improving our performance:

1. **NN optimization:** Further tweaking the layers of our model, dropoff layers, early stopping, etc
2. **Feature optimizaiton:** Given the 90% accuracy performance of our board-state representation with PCA, we are confident that generic representation of game-states are a fruitful approach. Nevertheless, further work can be done with looking at which subsets of moves are optimal.
3. **Dataset analysis:** An upper-bound of 91% accuracy seems to have been reached. Regardless of the technique used (SVM, ANN, or RNN), we were unable to surpass this limit. Thus, our suspicion is that some inherent randomness exists in the dataset which should be analyzed. On the other hand, a larger dataset might be what is needed for the next level of performance.
4. **Classifier Model:** Begin extrapolating the model to Human vs Human and AI vs AI games.

In the long-run, explainable AI is the goal. With a numerical heuristic for how human-like an AI plays, we can train a new chess-engine that plays human-like. This will enable chess players to learn from the engine, rather than just staring puzzled at a depth-15 max-min optimization choice of move.

References

[1] David W. Smith, *Google Deep Mind's 'alien' chess computer reveals game's deeper truths* (eureka.eu.com/innovation/deep-mind-chess/)

[2] KWRegan, *The Crown Game Afair* (https://rjlipton.wordpress.com/2013/01/13/the-crown-game-affair/)

[3] David Silver et al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, 2017 (https://arxiv.org/pdf/1712.01815.pdf).

[4] Gary Kasparov, *Chess, a Drosophila of reasoning* (http://science.sciencemag.org/content/sci/362/6419/1087.full.pdf)

[5] Python-Chess Library: github.com/niklasf/python-chess

[6] Scikit-Learn Library: github.com/scikit-learn/scikit-learn

[7] Keras Library: github.com/keras-team/keras