Move Ranking in Arimaa

Arzav Jain, Neema Ebrahim-Zadeh, Vasanth Mohan, Vivek Choksi
Stanford University

Abstract

In this paper, we describe our work on ranking moves in the game Arimaa. We first discuss the various models that we used—Naive Bayes, an L2-regularization L2-loss SVM with a linear kernel, and L1 and L2 logistic regression—in order to rank moves based on "expertness". This was done with the goal of pruning an exploration of the game-tree. We found that the SVM model, on average, ranked 67% of expert moves within the top 10% of all possible moves. Other models exhibited comparable performance. While these preliminary results are promising, the models all suffered from high bias, probably due to a limited feature set. The paper ends with future approaches to improve our existing models.

I. Introduction

Arimaa is a two-player board game played on an 8x8 grid, with players controlling 16 pieces each (similar to Chess). For more details on Arimaa gameplay, see the rules.[1] The game was designed to be difficult for computers to play but not (particularly) difficult for humans to learn and play.[2] This human-bot discrepancy stems in part from the game's enormous branching factor. Researchers have estimated an average of 16064 distinct legal moves per turn, as well as an average total of 46 turns per game.[3] The game-tree complexity is estimated to be on the order of 10^{402} (compared to 10^{123} in Chess and 10^{360} in Go).[3] As a result, brute force game-tree traversals are severely limited in terms of depth of exploration.

II. Related Work

Much of the prior work on building AIs for Arimaa finds optimal moves using game-tree search algorithms (for example, alpha-beta search and Monte Carlo Tree Search).[3] However, to date, there has been only one documented major effort to apply machine learning to Arimaa. This work, presented by researcher David Wu as his honors thesis at Harvard, applies machine learning to rank all possible moves given a board state[3] and serves as the foundation for our project.

David Wu's model scores possible moves based on an evaluation function; as such, it implements move *ordering* rather than move *classification*. Moves are ranked by "expertness"—i.e. the likelihood that an expert would play the move—and the model uses this move ranking to prune the exploration of the search tree. So instead of searching the game-tree for all possible moves that can be played, his model does the following:

1. Order all possible moves by expertness
2. Select the most expert (for example, 5%) of all possible moves
3. Search the game-tree only for those top 5% of moves using alpha-beta pruning

While our machine learning approach draws from David Wu's work, our code is built upon an existing Arimaa bot (bot_Clueless) developed in Java by Jeff Bacher.[4] The "starter code" provides functionality to interpret game boards, manage updating boards with moves, and generate all possible moves from a given board state.

III. Data Management

Since 2002, more than 278,000 games have been recorded on the Arimaa website, all in a parse-able text format.[5] This format works well for linear scans of all the data but is cumbersome for more complex queries (such as filtering for games in which both players are experts). For ease of querying, we parsed and loaded the game data into a MySQL database, enabling us to easily execute arbitrary queries.

We hosted the MySQL server on Amazon's RDS so that parallel jobs on multiple machines could freely access the game data.

IV. Scalability

Computation time: We ran multiple evaluations of our model using training sets of different sizes, and each run took around 2 minutes per game. To speed up these evaluations, we parallelized our tasks using Stanford's FarmShare computing resources (corn and barley).

Memory: When training the SVM model, all feature vectors are required to be held in memory. Because of the large branching factor in Arimaa, each trained move corresponds to thousands of training examples and resulting feature vectors: 1 for the expert move and on average 16,000 for non-expert moves. This resulted in about 1 GB of text data for every 10 games, despite a sparse representation of feature vectors. Loading these into memory translated to approximately 3 - 5 GB of RAM for every 10 games when running the C version of LIBLINEAR.[6] Since a typical laptop does not meet these RAM requirements, we used Stanford's barley computing machines to allow us to evaluate training set sizes of up to 100 games.

V. Implementation of Features

The features implemented are inspired by David Wu's work on Arimaa.[3] In order to characterize the "expertness" of a move, the resulting board is mapped onto a feature space composed of features of the board. These features were chosen to correlate with how strong a move is.

In order to reduce redundancy in our feature set, we exploited the left-right *symmetry* of the Arimaa board to cut the number of location-based features in half (giving only 32 mirroring locations instead of 64). Furthermore, we denoted a piece's *type* as the number of stronger opponent pieces on the board (since this is a more meaningful encoding than absolute piece strength). We have implemented the following feature classes:

1. Position and Movements
2. Trap Status
3. Freezing Status
4. Stepping on Traps
5. *Goal Threats
6. *Capture Threats
7. *Correlation with Previous Moves

Due to impractically long computation times, the starred (*) features were eventually not considered when training and testing our learning models. For future work, optimizations (involving significant rewriting) could be made to Bacher's game-logic code to make computing these features feasible.

For details on each feature, please refer to David Wu's paper.[3]

VI. Naive Bayes Model

In order to "quickly" implement a baseline and gain some insight into the problem, we built a variation of the Naive Bayes classifier. We used a multivariate Bernoulli event model coupled with Laplace smoothing.

Methodology

Our implementation slightly modified the traditional Naive Bayes classifier because we are dealing with move *ordering* rather than move *classification*. As such, instead of classifying a given move (denoted by feature vector x) as expert or non-expert ($y = 1$ or 0 respectively), our Naive Bayes model output the log of the posterior odds of y, as shown below. We used this value as a "score" for a move and ordered all possible moves according to this score (in descending order). The ordering function h is given by

$$h(x) = \log\left(\frac{P(y=1|x)}{P(y=0|x)}\right), x \in \{0,1\}^{1776}$$

In order to evaluate our model, we used cross-validation by training on 70% of the data and testing on 30%. In order to train only on expert human moves, we only considered games in which both players' ELO ratings were above 2100. (This still left 5,020 of the 278,000 total games as potential training candidates.)

Training:

During the training phase, we performed the following algorithm (written below in pseudocode):

```
**for** each game in the training data:
  **for** each board position in the game:
    1. *Generate* all possible legal moves, including the
       "expert" move actually played in the game
    2. *Extract* feature vectors for all of these moves
    3. *Update* a table that holds the frequency of
       occurrences for each feature (for $y = 1$ and
       $y = 0$)
```

We then converted these frequencies into log-likelihoods, which we used in the testing phase.

Testing:

We used the following two metrics to evaluate the success of our model on the training and testing data sets:

1. The average percentile of the expert move among all possible moves after ordering. This describes the average percentage of moves that were ranked below the expert move.
2. The proportion of expert moves that rank in the top X percent of all possible moves, for different values of X.

We trained and tested our model on randomized example sets of different sizes. We calculated the first metric on both our training and our testing sets in order to generate learning curves.

Results and Discussion

Figure 1 shows the Naive Bayes model's learning curves for the first of our metrics. We tested our hypothesis four times on each of $\{10, 20, \dots, 150\}$ games to produce the trial averages shown below:

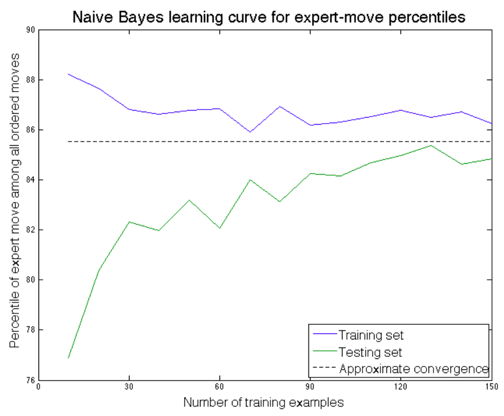

Figure 1: These curves follow the expected behavior for a high bias model. We see that these plots will converge to an average percentile of 86%. We will need to implement more features to decrease bias.

In order to measure not just the average percentile but also the consistency with which the model ranks expert moves highly, we plotted the histogram shown in Figure 2. The histogram shows expert move percentiles along with an added dimension: move number within the game. The plot was generated by running 3 Naive Bayes trials (each of which was trained and tested on 125 games). We see that the Naive Bayes model ranks the expert moves more highly in the beginning of the game (for about the first 10 moves marked in dark green) than it does for the middle- and end-game, where the distribution flattens.

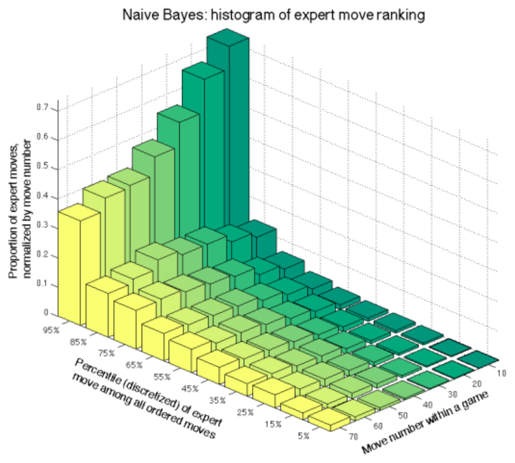

Figure 2: This plot compares the number of moves played within a game and the percentile of the expert moves vs. the proportion of expert moves. We discretize the number of moves played into buckets of 10 moves. We also discretized the percentile in buckets of 10%, where the displayed percentile is the average value of the bucket.

We hypothesize that the features implemented tend to capture the qualities of a good move more for the beginning-game than for the end-game. Features not considered (such as testing for goal and capture threats), which come into play later on in the game, might improve the middle- and end-game performance.

Another possible contributor to the poorer performance on end-game moves could be the underrepresentation of these moves in the data set. As the move number within a game increases, fewer and fewer games actually contain data for these points (many games end before, say, the 50th move). This means our model has more data from which to learn at the beginning of the game; this may partly explain the model's stronger performance on early moves in the game.

VII. LIBLINEAR Models

Given the high bias evident in the Naive Bayes assumption, we wanted to evaluate the performance of other models.

Methodology

We trained SVM and logistic regression models. In particular, we compared 3 different models: L2-regularized L2-loss SVM, L1 logistic regression, and L2 logistic regression.

We also tried to compare SVM with different kernels using LIBSVM[7], but due to our large training set size, the training time for LIBSVM was orders of magnitude longer than for LIBLINEAR (e.g., over 2 days for LIBSVM versus 10 - 20 minutes for LIBLINEAR). As a result, we chose LIBLINEAR.

Training:

In generating SVM and logistic regression models, we addressed the following three concerns.

Performance (memory use and computation time). While our feature extraction and evaluation harness code is in Java, we used the C version of LIBLINEAR for model generation to improve performance.

High ratio of negative to positive training examples. Due to the high branching factor in Arimaa, this ratio is on the order of 16,000 non-expert moves to 1 expert move. To help address this imbalance and to extract features from expert and non-expert moves more equally, we randomly discarded 95% of all non-expert moves. This improved performance and allowed us to train and test on more games.

Tuning model parameters. We cross-validated different values of the SVM parameter $C \in \{0.1, 1, 10, 100, 1000\}$, observing similar results for $C \in \{0.1, 1, 10\}$ and poorer results as C was increased above 10. We also examined different weights for the expert and non-expert classes to further address the disproportion of non-expert moves. Different weights produced very similar results to LIBLINEAR's default weight ratio of 1:1.

Testing:

In computing our ordering of moves, we used the margins for SVM and probabilities of the sigmoid function for logistic regression.

Results and Discussion

We see in Figure 3 that after we randomly discard 95% of the non-expert moves, the models converge to around the 90th percentile after about 250 games. Interestingly, without discarding *any* non-expert moves, the models converge to the same percentile but after 100 games (not shown here).

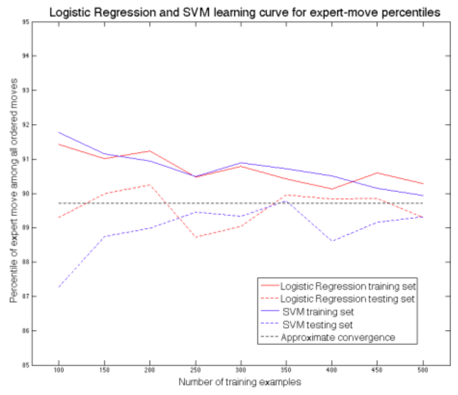

Figure 3: **The above learning curves show that the SVM and logistic regression models similarly suffer from high bias. The average percentiles of the expert moves converge to about 90%, only marginally better than the Naive Bayes average of 86%.**

In terms of the number of feature vectors considered before reaching convergence, discarding 95% requires fewer vectors. This permits shorter computation times to attain similar results, with the caveat of requiring more games in the database.

For analytical purposes, we also plotted the proportion of expert moves that were evaluated above a given percentile. This creates a plot (see Figure 4) that reads very similarly to a CDF. In other words, the y-value corresponding with, for example, $x = 10\%$ represents the percentage of expert moves that our model classified in the top 10% of our move ordering.

One takeaway from this analysis is that, if we limited a search of the game-tree to the top 10% of the move ordering, then we would examine an expert move 67% of the time (when using SVM or logistic regression models).

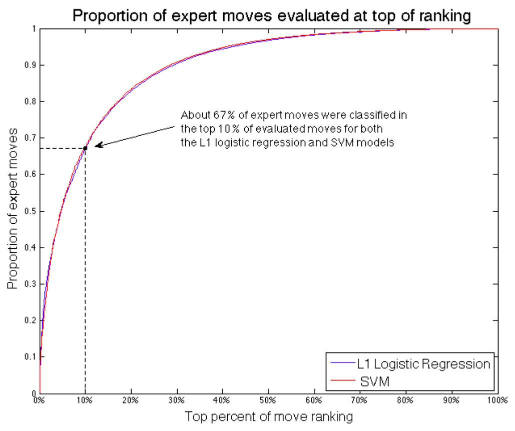

Figure 4: **The SVM and logistic regression models had similar performance, both faring moderately better than the Naive Bayes (not shown) model. They both ranked 67% of expert moves in the top 10%, whereas Naive Bayes ranked 58% of expert moves in the top 10%.**

Figure 5 was generated by running 3 trials, training on 500 games and testing on 250 games. Though the plot was generated and displayed only for the SVM model, it was very similar to that of the logistic regression model.

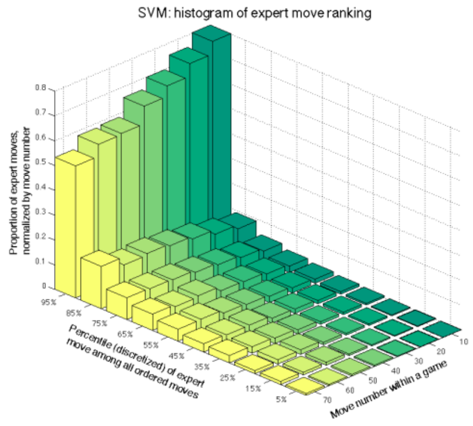

Figure 5: **Similarly to Naive Bayes, there is a trend that expert moves closer to the beginning of the game are ranked more highly. (In general, the SVM model outperforms Naive Bayes).**

Similarly to the Naive Bayes histogram, we see that the proportion of expert moves ranked highly is largest for the beginning moves of the game. This reinforces our hypothesis that the features currently implemented capture essential qualities in early-game moves.

VIII. Future Work

Hard Negative Mining

In deciding our training set of negative examples, we have two objectives:

1. Minimize memory requirements and computation time by considering only a subset of negative examples at a given point in time. (Currently, we are discarding 95% of the data in order to boost SVM training speed.)
2. Retain "useful" information in negative examples.

Hard Negative Mining can potentially achieve these dual objectives. One could first pick a random subset of the negative examples and run a model such as SVM. The model would then be retrained on the negative examples classified most poorly along with an additional random subset of the data. This step-wise training could be repeated until convergence without particularly high memory demands.

Time Based Features

As we hypothesized from the move vs. percentile plots, adding more features to capture qualities of end-game moves might be helpful. However, it could also be interesting to see what features are relevant based on the number of moves currently made in the game. For example, looking for goal threats may not be as insightful at the beginning of the game as it would be at the end of the game. In order to learn which features are important at each time step of a game, one could perform feature selection specific to each time step and choose the most relevant features accordingly.

Acknowledgements

We would like to thank the CS229 teaching staff (and especially Sameep) for their help and guidance throughout the project. In addition, we would like to thank David Wu for his valuable support and insight. We are also grateful for Stanford's FarmShare computing resources, which we used to train and test our learning models.

Works Cited

1. Arimaa.com. Arimaa Game Rules. Retrieved from http://arimaa.com/arimaa/learn/rules.pdf
2. Arimaa Game Archive. (2013). [Data file]. Retrieved from http://arimaa.com/arimaa/download/gameData/
3. Chih-Chung Chang and Chih-Jen Lin. LIBSVM : A library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27. (2011). [Software]. Available at http://www.csie.ntu.edu.tw/~cjlin/libsvm
4. Haskin, B. (2009). Specifications for the Arimaa Engine Interface (AEI). Retrieved from http://arimaa.janzert.com/aei/
5. Syed, O. (2013, May 17). Please say more about the design decisions. [Msg 7]. Message posted to http://arimaa.com/arimaa/forum/cgi/YaBB.cgi?board=talk;action=display;num=1367476894#7
6. Wu, D. J. (2011, 31 March). Move Ranking and Evaluation in the Game of Arimaa. *Harvard College Thesis*. Retrieved from http://arimaa.com/arimaa/papers/DavidWu/djwuthesis.pdf
7. Bacher, J. (2006, 6 March). bot_Clueless: A sample bot written in Java. [Software]. Available from http://arimaa.com/arimaa/bots/
8. R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A Library for Large Linear Classification, Journal of Machine Learning Research 9, 1871-1874. (2008). [Software]. Available at http://www.csie.ntu.edu.tw/~cjlin/liblinear