Game Outcome Analysis

Raymond Barker
Computer Engineering, Columbia University
New York, NY
rjb2150@columbia.edu

Abstract—**The purpose of the Game Outcome Analysis project is to predict the winner of a game using a snapshot of the game state. Though my code and methods are extensible to many games, I focus specifically on the parsing, vectorization, and classification of chess game data**.

Keywords-chess; PGN; classification; Mahout; vectorization

I. INTRODUCTION

The classification of game states and prediction of winners is an important part of game analysis. The importance of game analysis is twofold:

First, games are enjoyable to play and analyze. Many people play games unprofessionally, devoting significant amounts of their leisure time to honing their skills, analyzing effective strategies, etc.

Second, games are increasingly played professionally. For example, the computer game Defense of the Ancients 2 (DotA2) is played by over a million players each day and has active professional leagues; in a recent DotA2 tournament, the total prize money available exceeded $10 million USD.

In this project I provide a framework for doing classification with Mahout more easily, focusing specifically on the task of classifying chess game states. Chess has existed for centuries, been played professionally for decades, and has millions of players globally.

II. DATASET

There is a standard notation for chess games known as Portable Game Notation (PGN). For this project I use the gm2006 dataset[1] of PGN files compiled by Norman Pollock. This dataset includes 74,726 ranked chess games between 1,227 unique players. All players had an ELO rating of at least 2475, and all games were played between 2006 and 2014. Additionally, only untimed, in-person chess games are included; no blitz games or correspondence games are included.

For additional information on PGN files see the PARSING section.

III. SYSTEM OVERVIEW

My library contains two major components: a sub-package for parsing and vectorizing PGN files, and a sub-package that handles classifying input vectors using Mahout. These components are outlined further in the following sections.

Note that I also show other data sources in the system diagram. While I only provide chess data in the uploaded repository, the code can easily be used with any input vectors, including other game data. Some information on how to do this is provided in the PACKAGE DESCRIPTION section.

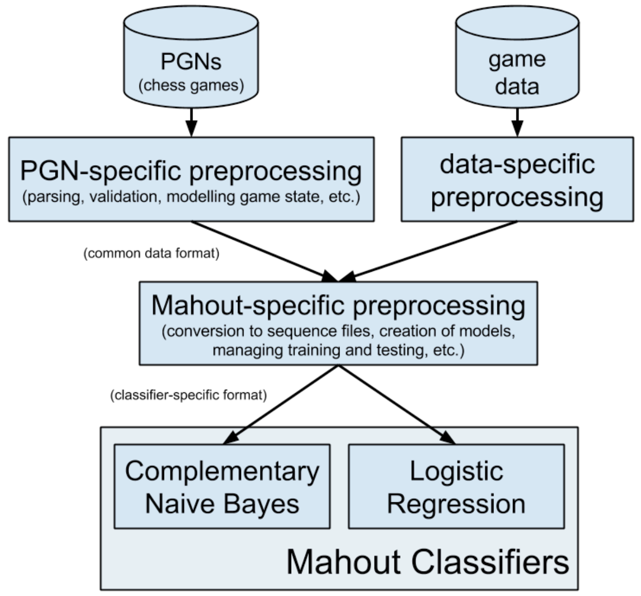
PGNs
(chess games)
game data
PGN-specific preprocessing
(parsing, validation, modelling game state, etc.)
data-specific preprocessing
(common data format)
Mahout-specific preprocessing
(conversion to sequence files, creation of models, managing training and testing, etc.)
(classifier-specific format)
Complementary Naive Bayes
Logistic Regression
Mahout Classifiers

Figure 1. *System overview diagram.*

IV. PARSING

This section describes how to parse PGN files. (Note that I have split the original ALGORITHM section into three sections: PARSING, VECTORIZING and CLASSIFYING.)

A PGN file consists of two major sections: a metadata header, and a body of encoded moves. The PGN specification is available here[2], and a web-based PGN viewer is available here[4]. Here is an example PGN file:

```
[Event "1st Grand Europe Open"]
[Date "2012.06.10"]
[White "Tikkanen, H."]
[Black "Grigoryan, K2"]
[WhiteElo "2566"]
[BlackElo "2517"]
[Result "1-0"]

1. c4 Nf6 2. Nc3 g6 3. d4 Bg7 4. e4 d6 5. Nf3 O-O 6. Be2 e5 7. O-O Nc6
8. d5 Ne7 9. Ne1 Nd7 10. Bd2 f5 11. Nd3 Nf6 12. f3 c6 13. b4 cxd5
14. cxd5 Qb6+ 15. Nf2 Bd7 16. Qb3 Rac8 17. Nd1 f4 18. Nb2 Qd8 19. Rfc1 g5
20. Nc4 Ne8 21. b5 Ng6 22. b6 Ra8 23. bxa7 Rxa7 24. Nb6 h5 25. Nxd7 Qxd7
26. Qb6 Ra8 27. Bb5 Qe7 28. Bxe8 Rfxe8 29. Rc7 Qd8 30. a4 g4 31. fxg4 hxg4
32. Nxg4 Nh4 33. h3 Qg5 34. Qxd6 Nxg2 35. Kxg2 f3+ 36. Kh1 Qxd2 37. Rg1 Ra6
38. Qd7 1-0
```

The metadata is straightforward to parse. However, parsing the moves is more complicated. Each move is encoded in Standard Algebraic Notation (SAN), which looks like this:

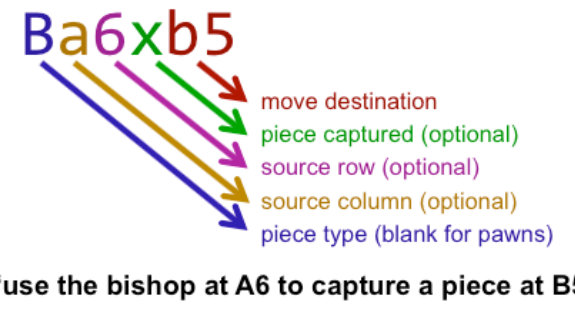
Ba6xb5
move destination
piece captured (optional)
source row (optional)
source column (optional)
piece type (blank for pawns)
"use the bishop at A6 to capture a piece at B5"

Figure 2. *Basic SAN move format.*

Note that most of the fields are optional. If every move included both the source coordinates and the destination coordinates, it would be trivial to parse them. However, SAN is meant to be human-readable as well, and so the specification allows "superfluous" or "obvious" information to be omitted.

For example, suppose the move is "Rxc7". This means, "move a rook to c7 and capture a piece". However, suppose the player moving has two rooks that can reach c7. Normally, some disambiguating source row/column would be provided, such as "Rcxc7", which means "move the rook on file c to c7 and capture a piece". However, if one of the two rooks cannot actually move (e.g., doing so would put the player's king in check), then no disambiguating source row/column must be provided.

The result of this is that the parser must understand the rules of chess, which moves are valid, when a king is in check, and so on. Otherwise, it cannot understand ambiguity in SAN moves. This made creating a PGN parser a non-trivial undertaking.

For more information about the parsing algorithm, see the `PgnParser`, `ChessMove`, and `Chessboard` classes.

V. VECTORIZING

Originally, I wanted to create a generic vectorization approach that could handle any team-based game. For a game where a team is composed of N instances of M pre-defined members, the features are simply the number of instances of each pre-defined member type for both teams:

$$\{N_1^1, N_2^1, \ldots, N_M^1, N_1^2, N_2^2, \ldots, N_M^2\}$$

In the case of chess, the pre-defined members consist of the king (K), queen (Q), rook (R), bishop (B), knight (N), and pawn (P), so the piece-based vector looks like:

$$\{K^1, Q^1, R^1, B^1, N^1, P^1, K^2, Q^2, R^2, B^2, N^2, P^2\}$$

White is team 1, and black is team 2; each feature is simply the count of pieces of that type for that player. (See `PieceCountVectorizer` for the implementation of this vectorizer.)

However, I found that this generic approach was not effective. It does not contain any positional data whatsoever, which is very important in chess; it doesn't matter if you have ten queens, if your king is in checkmate.

Additionally, due to the fact that the games in my dataset were between skilled players, and the fact that good players avoid throwing away their pieces, many game states consisted of the same pieces on each side except for a different number of pawns. This resulted in pawns being over-valued in the model produced by training on these vectors.

Due to these difficulties, I decided to abandon this generic approach and focus specifically on writing a better chess game vectorizer. This vectorizer instead uses common chess evaluation heuristics, including some positional information. (See `HeuristicVectorizer` for the implementation of this vectorizer.)

There are many ways[5] to evaluate chessboards; in this vectorizer I make use of the traditional method[6] of piece valuation described by Claude Shannon. Additionally, I include as features a bit indicating whether each players king is in check. Finally, I include as a features the number of squares occupied by or threatened by each player, intended as a proxy measurement of board control. The final vectors look like this:

{pieceValue^1, kingInCheck^1, threatenedSquares^1, pieceValue^2, kingInCheck^2, threatenedSquares^2}

For the effectiveness of each vectorization method, see the EXPERIMENT RESULTS section.

VI. CLASSIFYING

This project provides two Mahout-based classifiers: a naïve Bayes classifier, and a logistic regression classifier. (For the naïve Bayes classifier, either standard or complementary mode may be selected.)

This report does not go into the details of how each classification algorithm works, since there are much better write-ups available. I suggest either the Big Data Analysis course webpage[9] or the Mahout developer's guide[10].

For a description of how to use the classifier sub-package of this library, see the PACKAGE DESCRIPTION section.

For the effectiveness of each classifier, see the EXPERIMENT RESULTS section.

VII. PACKAGE DESCRIPTION

This section describes the overall structure of this project's `src` directory, and it explains the basics of how to hook up a new data source to a classifier.

The directory `src/data/` contains PGN datasets.

The directory `src/misc/` contains example output from the demonstration.

The directory `src/{main,test}/java/.../rjb/chess/` contains files related to parsing and vectorizing PGN files.

The directory `src/main/java/.../rjb/classifier/` contains files related to setting up Mahout classifiers.

The directory `src/main/java/.../rjb/` contains, in addition to the directories above, example code which demonstrates how to perform classification end-to-end.

In order to hook up a new data source to a classifier, you must be familiar with a few basic classes in the `classifier` sub-package. After parsing your data (or otherwise generating a Java representation of your data), you need to create a `Vectorizer<?>` for your data. `Vectorizer<A>` contains the following methods:

```
// Turns the input <A> into a Vector.
abstract Vector vectorize(A input);

// Returns the list of possible categories.
abstract ImmutableList<String> categories();

// Returns a list containing each feature name.
abstract ImmutableList<String> features();
```

A `Vector` is simply a wrapper around a Mahout vector, except it also includes the category of that data point:

```
public class Vector {
       String category;
       org.apache.mahout.math.Vector vector;
}
```

Next, you have to choose which classifier you want to use. This library provides a `Classifier` wrapper around Mahout classifiers that makes them simpler to train and use. `Classifier` contains the following methods:

```
// Trains the classifier.
abstract void train(List< Vector> vectors);

// Classifies a single vector.
abstract String classify(Vector vector);

// Classifies a group of vectors.
// Summarizes the results.
// Returns the accuracy.
double classifyAndSummarize(
       List<Vector> vectors) { … }
```

The `train` method abstracts away the Mahout set-up process. The `Classifier` abstract class has a naïve Bayes implementation and a logistic regression implementation.

For a more detailed description of the software package, see the README file in this project's GitHub repository[7].

VIII. Experiment Results

This section contains the results of my classification trials. Running the `Demo` code in the repository will reproduce these results. There are three classifiers and two vectorizers, and every combination is shown here. Additionally, the best-performing combination is analyzed further.

For all trials, a split of 80% training data and 20% data was used. Of the 74,726 games, only the 40,199 non-tie games were used.

	standard naïve Bayes piece count vectors	standard naïve Bayes heuristic vectors
total correct:	5496 / 8040 (0.68)	6625 / 8040 (0.82)
white correct:	3481 / 5014 (0.69)	4168 / 5014 (0.83)
black correct:	2015 / 3026 (0.67)	2457 / 3026 (0.81)

	complementary naïve Bayes piece count vectors	complementary naïve Bayes heuristic vectors
total correct:	5732 / 8040 (0.71)	6625 / 8040 (0.82)
white correct:	3616 / 5014 (0.72)	4168 / 5014 (0.83)
black correct:	2116 / 3026 (0.70)	2457 / 3026 (0.81)

	logistic regression piece count vectors	logistic regression heuristic vectors
total correct:	5869 / 8040 (0.73)	6271 / 8040 (0.78)
white correct:	4053 / 5014 (0.81)	4083 / 5014 (0.81)
black correct:	1816 / 3026 (0.60)	2188 / 3026 (0.72)

Figure 3. *Classification accuracies by method.*

The most accurate method was using the complementary naïve Bayes classifier in combination with the chess-specific heuristic vectors.

Both vectorizers are based on looking at a specific game state, and the turn that should be used is configurable. In the above trials, the second-to-last turn is used; this makes classification easier, as we are close to the end of the game. As we get further from the end of the game, the classifier becomes less accurate:

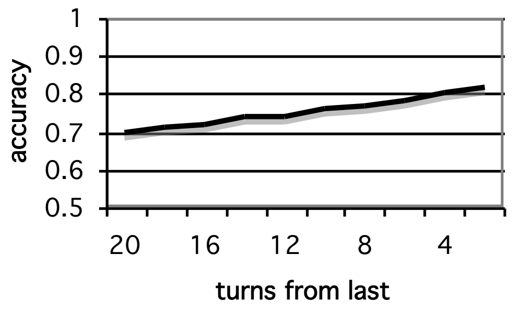

Figure 4. *Classification accuracies by turn from last.*

IX. Conclusion

In this project, I provide a framework for doing classification with Mahout more easily, which reduces the task of performing classification to writing a `Vectorizer`.

I focus specifically on the task of classifying chess game states, and provide feedback on effective and ineffective methods of vectorizing chess games.

Future work can involve either expanding the `HeuristicVectorizer` to include more heuristics, expanding the repository to include other sources of data, or implementing additional classifiers.

This project team had only one member, Raymond Barker.

Acknowledgment

I would like to thank Norman Pollock for collecting a huge amount of chess game data[1], and making it available in a reliable, consistent format.

I would also like to thank Jossef Harush, who provided very useful Mahout example code[3].

Finally, I would like to thank Ching-Yung Lin and the TAs of EECS E6893 (Big Data Analysis) for their hard work in preparing this course[9].

Appendix

To learn more about this project, please visit its repository on GitHub[7] or view its overview video on YouTube[8].

References

[1] N. Pollock, “Norm's PGN-EPD Chess Downloads”, http://hoflink.com/~npollock/chess.html

[2] S. Edwards, “Portable Game Notation Specification”, http://www6.chessclub.com/help/PGN-spec

[3] J. Harush, “Mahout Gist”, http://gist.github.com/Jossef/e6c8fc0c31f0c2bf036a

[4] Chess Tempo, “PGN Viewer”, http://chesstempo.com/pgn-viewer.html

[5] Chess Programming Wiki, “Evaluation”, http://chessprogramming.wikispaces.com/Evaluation

[6] C. Shannon, “Programming a Computer for Playing Chess”, Philosophical Magazine, Ser. 7, Vol. 41, No. 314, 1950

[7] R. Barker, “Game Outcome Analysis Repository”, http://github.com/Sapphirine/Game-Outcome-Analysis

[8] R. Barker, “Game Outcome Analysis Video”, http://youtu.be/7M0_RbmNRfg

[9] C. Lin, “Big Data Course Website”, www.ee.columbia.edu/~cylin/course/bigdata/

[10] Mahout Developers, “Classification”, http://mahout.apache.org/users/classification/