Data Mining With Python and Weka

By Yang Liu & Tyler Nickerson

For this project, a small program was written in Python 2.7 to extract feature data from sets of integers representing Connect-4 game board states (for more info on these strings, click here). These features were then plugged into Weka 3, a Java program by New Zealand's University of Waikato that runs machine learning algorithms on sets of data. In this particular experiment, the feature was data was used to train/build both a neural network and a decision tree.

The Features

As previously stated, each string (line) of input data is interpreted as a Connect-4 board state and loaded into a NumPy matrix. Using NumPy matrix operations, each board state is analyzed and five state features are produced for use in Weka. Those features as follows:

leftCornerPlayer

Returns the number of whichever player is in the left-hand corner of the board. While not entirely useful, this feature acts primarily as a benchmark and does not have a significant impact on the rest of the features.

centerPlayer

Returns the number of the player with the most checkers in the center of the board (no checkers along the left or right edges). This feature is somewhat useful in deducting who has control of the board.

diffMoves

Returns the difference in "open" moves between players, "open" (i.e moves in which another checker may be added to left or right side of the sequence). This feature is useful for deciding how "far ahead" one player is from another player. For example, a higher number would mean that one player has a higher chance of winning, as they have more chances to complete winning moves (or become closer to winning). diffMoves is computed by checking all moves (horizontal, vertical, and diagonal both ways) and counting the strings of same-player checkers that have open spaces on either end. This can be represented by the following pseudocode:

```
function diffMoves
    for PLAYER1, PLAYER2 as player
        playerCount.add(horizontalCount(player))
        playerCount.add(verticalCount(player))
        playerCount.add(diagonalsCount(player))
        count.add(playerCount)
    difference = max(count) - min(count)
    return difference
end
```

blockableMoves

Returns the difference in "blockable" moves between players (i.e. sequences of checkers with gaps in the middle, allowing moves to be blocked). This feature signifies the quality of plays one user is making. If the difference is higher, then we know one player has been making several moves which are not optimal, as they can be easily blocked by the opponent. blockableMoves is calculated by checking all moves (horizontal, vertical, and diagonal both ways), and seeing if there are gaps in any of the move sequences. This can be represented by the following pseudocode:

```
function blockableMoves
    for PLAYER1, PLAÃYER2 as player
        playerCount.add(horizontalBlockCount(player))
        playerCount.add(verticalBlockCount(player))
        playerCount.add(diagonalsBlockCount(player))
        count.add(playerCount)
    difference = max(count) - min(count)
    return difference
end
```

bestMoves

Similar to blockableMoves, bestMoves also helps to define the quality of the plays the leading player is making. However, unlike blockableMoves, where a high value meant a number of low-quality, "blockable" moves, bestMoves does the opposite. bestMoves subtracts blockableMoves from diffMoves to define the difference in optimal moves. This provides a more accurate prediction of how close the game is to ending, as a higher value means one player has made more non-blockable moves, and has a higher chance of winning the game. This feature is provided as an improvement over diffMoves. However, diffMoves is still valuable, as it signifies who has made more valid moves in a single game. This in turn can be used to determine if someone is in the lead or not.

Building a Decision Tree

(will write)