Logical Learning

Jingyu Li

September 10, 2016

1 Introduction

In this project we discuss the application of machine learning to a game called Logic. We compare players trained by supervised learning against a player that makes random decisions as well as a hard coded player that makes decisions similar to a human beginner player. We consider the 3 player version of the game. The rules are as follows.

1.1 Rules of the Game

1. Cards Ace to Queen of two different suits from two different decks are randomly distributed so that each player has 8 cards.
2. The players sort their cards (if two cards have equal value, the player can choose to place them in any order; for our purposes, we just keep the order they were given in) and place them face down. Because the different suits are from two different decks, every player can distinguish between the two suits even when the cards are face down.
3. The players then take turns guessing cards of the other players.
4. If a card is correctly guessed, the card is flipped over. If it is incorrectly guessed, the player who guessed it must flip over one of his/her own cards.
5. The first user to 'claim' – meaning they end the game and guess all the cards on the table – correctly wins. A player can claim at any point in this game. In our code, we check if a player can claim every time their belief on the cards is updated.

2 Setup

A random player, basic player, and AI player play against each other. We set up a player by giving it a hand and initial belief of the other players cards based on its own hand and the fact that the cards are in sorted order and each card is unique. Each of these players uses the same belief update and card reveal strategies. Every turn, a card is revealed and the beliefUpdate method applies logical constraints (for example, two cards cannot be the same card, and a card sorted after another card cannot have a smaller value). If a player guesses incorrectly, the revealCard method returns the card they should flip over. Although not always a good strategy, a large percent of the time, it is desirable to reveal the cards at the two ends (highest and lowest numbers), because they are often already known to the other players. This serves as our revealCard method for all the players. The method that differs between the three players is the chooseMove method, which picks a card on which the player makes a guess on his/her turn.

3 Random Player

The random player picks a random card from those that it is not sure about to guess, and guesses a random value from its set of possible values for that card.

4 Basic Player

The basic player uses a strategy that mimics a beginner human player. It simply chooses a random card out of the cards that have the least number of possible values (excluding those that have only 1 possible value – the cards that are already known) and makes a random guess out of the possible values it can have.

5 AI Player

5.1 Motivation

There are many places in the game where we could use machine learning. Updating beliefs and revealing cards can both be improved. However, training too many parameters is difficult, so we make certain assumptions for viable strategies (the predefined methods described above), and focus primarily on which card a player should make a guess on and what value to guess. For this, we use machine learning.

The most obvious approach to this problem is reinforcement learning, because we want to use the result of the game (win or lose) as feedback for a model that generates moves during the game. This calls for us to use Q-learning (optimize an MDP consisting of game states) or run an evolutionary algorithm on a neural network. However, we found these techniques to be too cumbersome for this particular game. Because there are so many cards in the game and a set of values each card could be, the game state would be a combination of what each card could be, what cards are flipped over, etc. This would result in too many states for an MDP than we'd like to handle. Using a neural network and training with an evolutionary algorithm is also difficult. Parameters such as the number of layers, the number of neurons, mutation rate for the evolutionary algorithm all must be tuned just right for a good model to be produced. Not only this, but evolutionary algorithms often have the problem of getting stuck at local optima and would require further modification of the algorithm to ameliorate this problem.

Thus we explored a different route. Supervised learning is a much easier approach, but would require us to collect data in a creative way. It is not enough to know the result of a game. We need a way of evaluating given the state of the game, what the correct move to make is. If we do this, we can run the game many times to collect data and run a supervised learning algorithm.

We do this by introducing a measure we call 'entropy change'. The entropy change of a certain move for a player is essentially the expected number of card possibilities he/she can eliminate by making that move minus the expected number of card possibilities the opponents can eliminate. Assume that when a player picks a card to guess, he/she chooses a random value out of the remaining possibilities in his/her belief to guess (this assumes uniform probability for all the possible choices, which is not entirely accurate but suffices for our purposes). To be more specific, we calculate the entropy change of choosing a certain card to guess by summing the entropy change over the possible values to guess for that card and dividing by the number of possible guesses. Note that during an actual game, we cannot compute entropy change, because it relies on knowing how the other players beliefs change. However, when we are collecting data, it's okay to 'cheat' to find the correct move to make.

Once we have a set of data, we can train an SVM classifier to determine which card to pick given a set of features representing the current state of the game. We then use this SVM classifier for the AI player's chooseMove method.

5.2 Dataset

Because we don't have human data for this game, we must generate the dataset. We select a set of features representing the state of the game for a player, and assign it a class that represents the optimal card for that player to take a guess on.

5.2.1 Feature Set

We choose the features of a game state to be an array of the number of revealed cards in one's own hand, and for each of the cards in the opponent's hands: the number of possible values of those cards, min possible

value, and max possible value. This is a fairly good representation of the game state, as it changes to reflect the game on every turn (a card is revealed at every turn).

5.2.2 Collecting Data

To collect data, we set up an environment in which we can simulate belief updates to compute entropy change without actually changing players' belief. In this environment, we then iterate through all the guessable cards and compute entropy. We then take the move with greatest entropy, convert it to an integer (i.e. opponent 1, card 3 becomes 3; opponent 2, card 5 becomes $8 + 5 = 13$), and record it as the class, writing it besides the array of features in a data file. We then proceed in the game with that move and continue to collect data at each turn of the game. To generate our training dataset, we collected data for 50 games.

5.3 Training

For the AI player, we train SVMs to classify states in the game as their optimal moves. We train the SVMs on the data we collected. We use Python's sklearn library to train multi-class SVMs. Because our features are 49-dimensional, we cannot visualize our data. Thus we try multiple kernels to separate the data. We trained a linear, polynomial, and Gaussian kernel. Some of the kernels take parameters γ and r. For all our trials, we set $\gamma = 0.001$, $r = 100$. The linear kernel is represented by

$$\kappa(x, x') = x \cdot x'$$

A polynomial kernel of degree d is represented by

$$\kappa(x, x') = (\gamma(x \cdot x') + r)^d \cdot d$$

The Gaussian kernel is represented as

$$\kappa(x, x') = \exp(-\gamma|x - x'|^2)$$

The Sigmoid kernel is represented as

$$\kappa(x, x') = \tanh(\gamma(x \cdot x') + r)$$

6 Results and Analysis

The following table shows the classification accuracy of the SVMs on the training set (the dataset generated above that we used to train the SVMs)

Classification accuracy on dataset	
Linear Kernel	0.585
Polynomial Kernel (degree 2)	0.876
Polynomial Kernel (degree 3)	0.997
Polynomial Kernel (degree 5)	1.0
Gaussian Kernel	0.960
Sigmoid	0.167

As expected, the polynomial kernels have higher accuracy on the training set than the linear kernel. As the degree of the polynomial kernel goes up, accuracy increases (with the risk of overfitting). The Gaussian kernel has high accuracy, accountable by its ability to map data points to infinite dimensional features. The Sigmoid kernel, which seems to be less commonly used, gives a rather poor accuracy on the dataset.

The following table shows the number of wins of each player after the random player, basic player, and AI player play 1000 games against each other.

Number of games won out of 100			
	Random Player	Basic Player	AI Player
Linear Kernel	266	316	418
Polynomial Kernel (degree 2)	279	350	371
Polynomial Kernel (degree 3)	291	332	377
Polynomial Kernel (degree 5)	295	309	396
Gaussian Kernel	264	325	411
Sigmoid Kernel	347	247	406

SVMs using each of the kernels produced logic players that outperformed the basic and random players. While the AI player did not consistently win, we recognize that there is some randomness in the game (the hand that is dealt to a player is important in how the game plays out), as well as the difficulty of this problem. Given the number of possible states and randomness in the game, we feel the AI player is actually quite impressive at making good decisions that can be seen over the course of many games.

It is interesting that although the Sigmoid kernel had poor accuracy on the training set, it still produced a guessing strategy that outperformed the random and basic players. The strange thing here is that while the basic player outperforms the random player on average for the trials using the linear, polynomial, Gaussian kernels, when our AI player uses a Sigmoid kernel, the random player now outperforms the basic player. The random player wins 347 out of 1000 games, while the basic player only wins 247. We hypothesize that this is because the AI player using the Sigmoid kernel has a strategy quite different from the other AI players, and one player's strategy can affect how well opponent strategies work. This is seen when humans play games against each other. For example, often times a person will play a game quite well against a certain opponent or group, but when facing other opponents (whom are not necessarily objectively better), play less well.

In general, all of the kernels perform similarly well. There does not seem to be an added benefit of using a polynomial kernel over a linear kernel, and if anything, hurts performance slightly. This can be accounted to the fact that in a game like this, overfitting to certain game states can lead to quite poor decisions on other previously unseen states.

7 Future Work

There are many aspects we can improve upon for a player's strategy. For one, we assumed we should always reveal cards from the two ends. However, if a player is intelligent enough and able to predict the belief's of the other players, it may sometimes know that the other players already know a card and reveal that card. Furthermore, the entropy change for revealing a certain card card depends on the particular distribution of the player's hand each game. For example, if a player's highest card is a 4, by revealing it, the player essentially reveals all its cards. So in this case, the player would prefer to reveal the earlier cards rather than the card at the end first. As an extension to this project, it would be desirable to implement similar strategies as were used for training the chooseMove method for the revealCard method. Again, we could compute the amount of entropy change for the opponents when a card is revealed. We use the card corresponding to least entropy change for opponents as the optimal class. We would then collect data as we did in this work, and train a classifier to choose the card we should reveal given a set of features representing the state of the game.

Another major area of the game strategy we could improve upon is the belief update. Currently we have a set of rules that eliminate possibilities. However, this game goes a level deeper when players consider what the other players are thinking. For example, when an opponent correctly guesses one of our cards, we can think about what kind of hand the opponent must have that gives them high probability of guessing that particular card correctly. To mimic deeper level thought, we could implement a neural network and use a reinforcement learning algorithm such as an evolutionary algorithm. For the reasons mentioned previously in this paper, we did not implement this here, but think it would be interesting to try for a longer project.

Code including the game environment, generated dataset, and computer players can be found at https://github.com/jingyuli/Logic-Player.git