CS 229 Final Report: RPS Bot
Discerning Human Patterns in a Random Game

Tristan Breeden (tbreeden@stanford.edu)

Introduction

This project focuses on creating a bot that can play the common game of Rock Paper Scissors (RPS) better than its human opponent. The key idea is that humans tend to play in patterns rather than completely at random. Using machine learning techniques, I have trained a bot to learn how I play RPS, thus giving it an advantage over me. For example, if every time I play the sequence of 'Rock', 'Rock', 'Paper' and 'Rock' my next throw is consistently 'Scissors', then my bot learns that and plays 'Rock' in the next game to beat me.

So, for this problem my input is a sequence of n human plays, which I represent as an n-length vector, where each element in the vector is an integer in [1,3]. 'Rock' is represented by 1, 'Paper' is represented by 2, and 'Scissors' is represented by 3. Given this sequence of human plays, my bot uses various classifiers to predict my $n+1^{th}$ play, which is also represented by an integer as described above. Given this play, the bot chooses its winning counterpart (i.e. predicting 'Rock' entails playing 'Paper', predicting 'Paper' entails playing 'Scissors', and predicting 'Scissors' entails playing 'Rock').

As someone who wants to work in the video game industry, this project is of great use to me. AI and bots in video games tend to be surprisingly underdeveloped, and rely on heuristics rather than genuine machine learning algorithms. By starting with the almost trivial game of RPS and nevertheless producing meaningful results, I hope to show that in the more complex environment of a video game, the potential to advance the genre is very significant.

Related Work

RPS strategies:

- Wang, Zhijian, Bin Xu, and Hai-Jun Zhou. "Social cycling and conditional responses in the Rock-Paper-Scissors game." *Scientific reports* 4 (2014).

  This paper describes how if a player has lost two or more times, she is likely to shift her play, and more likely to shift to the play that will beat the one that has just beaten her than the same one her opponent just used to beat her.

- Farber, Neil. "The Surprising Psychology of Rock-Paper-Scissors."Psychology Today. N.p., 26 Apr. 2015. Web.
  This article describes some common RPS strategies and statistics that I will describe later, since my bot seems to be picking up on them.

Current Bots:

- Iocaine Powder: http://ofb.net/~egnor/iocaine.html. This bot was created by Dan Egnor for a RPS bot competition in which bots played against each other. Iocaine Powder uses a combination of three strategies to predict its opponent's next move: guessing randomly, playing against the most

frequently used throw, and history matching (i.e. finding patterns in the opponent's history). However, this bot's edge lies in the fact that it also detects the opponent's meta-strategy (i.e. guessing one ahead, two ahead, etc…). This bot doesn't use a direct application of machine learning as we have studied it in lecture, but I did borrow the overall idea of finding patterns for my project. I considered also using meta-strategies, but in Egnor's case, it was mostly a safeguard against other bots in the competition. Unexperienced players such as myself don't put as much thought into a play, which makes the meta-strategy approach only marginally valuable.

- MegaHAL: http://web.archive.org/web/20020802183909/http://www.amristar.com.au/~hutch/roshambo/. This bot was created by Jason Hutchens for the same RPS competition described above. His approach was to create a simple Markov model that stores frequency information about the opponent's plays for all possible contexts. This allows the bot to predict the next play in the form of a probability distribution over all possible throws. The bot then picks the throw that maximizes the expected value of its score (where a win scores a 1, a tie 0, and a loss -1). It also only tracks the frequency information over a sliding window, in case the opponent changes strategies over time.

Reading the papers and articles about RPS strategies allowed me to understand qualitatively what my bot should be picking up on to win. Analyzing the current bots (which came in 1st and 2nd in the RPS competition), gave me ideas as to how I could build my own bot. Their ideas and methods are extremely clever, but some are based on the knowledge that they would be playing against other bots. Since my project focuses on playing against humans, I had to filter out that information.

Dataset and Features

I constructed my own dataset by playing approximately 500 games against a bot that played randomly between the three possible throws. For every game, I recorded what I chose to play, what the bot played, and the outcome of the game. Each play is represented as either 1, 2, or 3, and each outcome is represented as either -1 (bot loss), 0 (tie), or 1 (bot win). As a result, I ended up with a large sequence of my own throws and embedded in it, my playing patterns in RPS. The next step was to extract features from this sequence in order to train my bot.

I decided that for every single throw in the dataset, I would extract features representing the context for that throw. This means that for any given throw, I would create a feature vector that tracked the last one, two, three, and four plays made previously. For example, if I played 2 (Rock) at a given moment, and the last four plays before that were 1-3-2-1, then the associated feature vector would be [1,0,0,0,1,0,0,0,1,1,0,0], with a label of 2. Each grouping of three elements in the vector represents one of the previous plays (i.e. red is the last play made, green is the one before that and so on). For each play, each of the three elements represents the throw used (first position = rock, second position = paper, third position = scissors). For the first four throws, which don't necessarily have up to four previous throws, all missing values were filled with a 0. I also experimented with features that represented the total amount of plays made, and the total amount of rock/paper/scissors throws made. However, using these tended to worsen my results, as we will see later.

Methods

Once I gathered my dataset and extracted features from it as described above, I used two different classifiers from the scikit learn library (http://scikit-learn.org/).

Support Vector Machine Classifier: Using an SVMs seemed like a good place to start, since we discussed them thoroughly in lecture and they tend to produce very good results. From the lecture notes, we know that the SVM optimization objective is the following:

$$min_{\gamma,w,b} \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i \quad s.t. \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i\,,\ i = 1, \ldots, m$$

$$\xi_i \geq 0,\ i = 1, \ldots, m$$

This allows us to create a hyperplane that separates data into two classes, while maximizing the size of the functional margin from that hyperplane to the nearest data points. Because of the size of my feature vectors, the classifier needs a kernel to operate in the high dimensional feature space. The scikit learn library offers many Kernels, and I am using a Gaussian Radial Basis Kernel:

$$K(\mathbf{x}, \mathbf{x'}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{x'}||^2}{2\sigma^2}\right)$$

At this point, one must note that I am doing a 3-way classification, rather than binary. Via scikit-learn, I am using a “one-against-one” approach in order to account for the additional class. All this does is train 3 * (3 -1)/2 = 3 binary classifiers for distinct combinations of the three original classes. At prediction time, all three classifiers are used, and the one with the highest score for a particular class gets used.

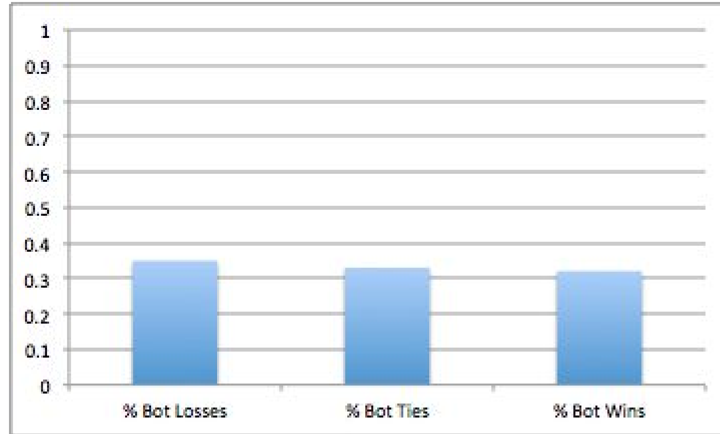
Logistic Regression Classifier: After having used an SVM approach, I decided to use one of the Generalized Linear Models offered by scikit learn, and opted for logistic regression. It minimizes the following cost function:

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

This optimization is done using a coordinate descent algorithm, and allows us to find the best weights associated with the features in order to make predictions. In order to do the multi-class classification however, scikit learn implements another technique: the “one-against-all” approach. In short, this strategy consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes.

Experiments and Results

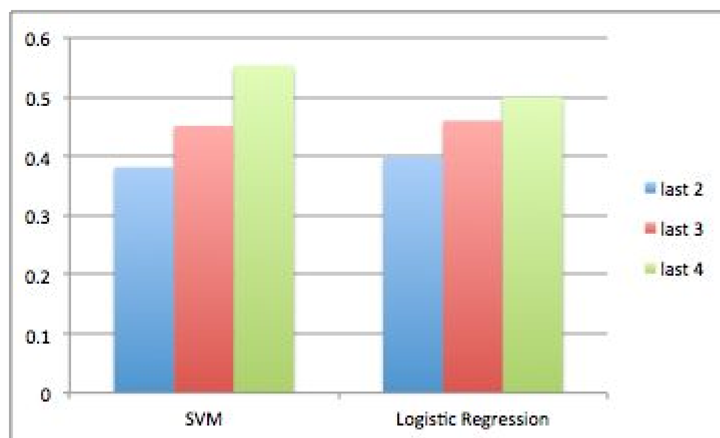
In order to test my bot, I needed a baseline to which I should compare my results. During my initial data gathering stage, I recorded the bot’s performance against me when it used a completely random strategy (i.e. choose a throw randomly).



Bot Performance for Random Strategy

The histogram above shows how the bot performed against me after approximately 500 games. It is clear that the bot is winning, losing, and tying mostly equally. What we're observing here is the Nash Equilibrium for RPS. Playing randomly ensures that, on average, the bot will lose, win, and tie equally. I chose this to be the baseline for my RPS bot.

In order to test the bot after training, I played another series of 500 games for each classifier mentioned above. I also increased my context window for features by increments of 1 in order to get an idea of how useful adding an extra context feature is.



Bot Performance after Training with Varying Features

Quantitatively, the increase in performance is evident. Using the SVM classifier described previously, my RPS bot won against me approximately 55% of the time. My initial observation was that the bot's performance seemed to increase linearly with the size of the context window it was extracting features for. However, increasing the context size beyond 4 did not yield any better results than those displayed above. My interpretation of this is that I most probably tend to play RPS in short patterns. That is, I repeat short sequences of play that don't go beyond ~4 moves in length. That would explain why looking further back in my play history doesn't help uncover my patterns, as they are already evident. Beyond this observation, I

theorize that SVM is performing marginally better than Logistic Regression, as the Kernel Trick is better suited to deal with the large feature space I am using.

Additionally, by looking closely at the data collected from these testing games, it seems that the bot is picking up on some common RPS strategies:

- Play ‘Paper’ as your first throw, since most non-competitive players start with ‘Rock’
- If the opponent plays two throws of the same kind in a row, assume he/she won’t use it a third time (this is a consequence of players wanting to seem random in their playing)
- Play ‘Paper’ more commonly than the other throws, since it is statistically the less used throw

Conclusion and Future Work

To summarize, I have created a RPS bot which looks at the immediate history of a human player’s throw sequence, and uses that context as features. The bot is then trained to use SVM and Logistic Regression classifiers to predict the human opponent’s next play. Using this knowledge, the bot plays accordingly to win. After training, my bot has achieved a best win rate of 55% using this technique, as opposed to a ~33% win rate using a random strategy. In the future, I hope to create better features that the SVM algorithm can exploit. For instance, it seems that winning or losing impacts a player’s strategy and throw patterns (see Related Works section). It would be interesting to include that information when extracting features and see if the bot can push beyond a 55% win rate. One way of doing this would be to interlace the human’s history with the bots in the following manner: $h_0b_0h_1b_1h_2b_2$… where h_i and b_i are the human and bot’s plays at turn i, respectively.

As mentioned in the introduction, the idea behind this project for me is to show how using machine learning algorithms, one can turn a fairly trivial random game into a game of complex pattern identification. I can easily imagine these ideas being ported over to a more elaborate game. For instance, in most video games, the difficulty in a boss fight tends to be either constant or discretized into very intentional sections (e.g. boss starts out easy, then deals twice the damage after 10 minutes). However, using a relatively simple machine learning framework, one can ramp up the difficulty of a boss fight over time by learning the player’s patterns, which is a lot more interesting to play against. Those patterns could be in the form of movement, damage dealt, or any other element of that particular game. The point I’m driving at is that bots in video games are pretty dull nowadays, and a simple change could lead to a much more diverse play experience. For a media/art form that is evolving and expanding so quickly within the technological realm, it would be a shame to fall behind when it comes to machine learning.