1 CSCI 1440/2440 Labs

CSCI 1440/2440 labs are designed to equip students with the skills necessary to develop autonomous trading agents, such as those that might participate in high-frequency trading. To achieve this goal, students design and build agents in lab each week, for simulation environments ranging from simple and deterministic (e.g., the Repeated Prisoners’ Dilemma) to complex and stochastic (Spectrum Auctions and Ad Exchanges). The most successful agent strategies tend to make predictions (e.g., via machine learning) about other agents, individually or collectively, and then optimize (i.e., best-respond) to those predictions.

Most labs are structured such that the students first develop their agents “offline,” meaning in simulation against built-in agents (built by the TAs). Additionally, there is typically a competition, where the students’ agents compete with one another “online.” These competitions comprise multiple runs, with agent scores posted on a LeaderBoard following each lab.

2 Trading Platform

Throughout CS1440/2440, we will simulate auctions and other games using a Java TRADINGPLATFORM that simulates trading in real time by user-designed autonomous agents in user-defined environments. This platform was designed and built by Professor Greenwald in conjunction with past TAs and students of CS1440/2440 (Luke Camery ’17, Andrew Coggins ’18, and Jake Chanan ’20).

TRADINGPLATFORM is a Java program built using a client-server model. One of our primary goals in developing this platform was to provide a system in which users can easily create a diverse array of trading agent games. In CS1440/2440, we use it to simulate a variety of games, from a simple repeated prisoners’ dilemma to a combinatorial auction for wireless spectrum.

In this lab, you will be working with the TRADINGPLATFORM to implement agent strategies for two different 2-player games: the Prisoners’ Dilemma and Rock-Paper-Scissors. The two agent strategies that you will code to play these games are called Fictitious Play and Exponential Weights. Both algorithms are known to converge to Nash equilibrium in repeated zero-sum games.\textsuperscript{1,2}

3 Setup

If you have not already installed the TRADINGPLATFORM, please refer to our Lab Installation/Setup/Handin Guide on the course website. In addition, you can find and download the stencil code for Lab 1 from the course website. Once everything this lab is set up correctly, you should have a project with files for four Java classes, all under src/main/java in the package brown.user.agent.lab01:

- RpsExponentialWeightsAgent.java
- RpsFictitiousPlayAgent.java
- RpsSampleAgent.java
- ChickenAgent.java

4 The Prisoners’ Dilemma

The Prisoners’ Dilemma is one of the most well-known and fundamental problems in game theory. One version of the story goes as follows:

Alice and Bob are suspected of committing the same crime. They are being questioned simultaneously in separate rooms, and cannot communicate with each other. Each prisoner has the option to either cooperate (do not incriminate the other prisoner) or defect (implicate the other prisoner). If one cooperates and one defects, the cooperating prisoner receives a lengthy jail sentence (i.e., a large negative payoff), while the defecting prisoner goes free. Should they both cooperate, they get shorter jail sentences; and should they both defect, they get longer sentences, although shorter than had one prisoner cooperated (the judge goes easier on them, since they both assisted in the prosecution of the other).

The payoff matrix of this game as is shown below:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>-1, -1</td>
<td>-3, 0</td>
</tr>
<tr>
<td>D</td>
<td>0, -3</td>
<td>-2, -2</td>
</tr>
</tbody>
</table>

**Question:** Does this game have an equilibrium? If so, what is it?

5 Rock-Paper-Scissors

Rock-Paper-Scissors, or Rochambeau, can also be represented as a game. (If you are not familiar with the rules of this game, we refer you to Homework 1.)

Rock-Paper-Scissors is an example of a zero-sum game, because one player’s win is the other player’s loss. Its payoff matrix is as follows:

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0, 0</td>
<td>-1, 1</td>
<td>1, -1</td>
</tr>
<tr>
<td>P</td>
<td>1, -1</td>
<td>0, 0</td>
<td>-1, 1</td>
</tr>
<tr>
<td>S</td>
<td>-1, 1</td>
<td>1, -1</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

**Question:** Does this game have an equilibrium? If so, what is it? (Brainstorm about the answer to this question with your partner, but there is no need to derive the solution in lab, as you will do so on Homework 1.)

6 Fictitious Play

Recall from class our analysis of the p-Beauty Contest. Did the class play an equilibrium strategy? They did not, and nor did the experimental subjects in Nagel’s research paper.

If your opponents in a game cannot be “trusted” to play the equilibrium, or if there is more than one equilibrium and none is agreed upon in advance, an alternative is to learn how your opponents are actually playing the game, and to best respond to their behavior. This is the essence of the Fictitious Play strategy.

Fictitious Play collects historical data during a repeated game. It uses these data to build an empirical probability distribution over the opponents’ actions based on their history of play, which it then takes as a

---

prediction of their next action (or action profile, if there are multiple opponents). Finally, it searches among its actions for the one that yields the highest expected payoff, given its prediction.

6.1 Prisoners’ Dilemma

After hundreds of rounds of the Prisoners’ Dilemma, you observe that your opponent defects 80% of the time. What is your best move?

<table>
<thead>
<tr>
<th></th>
<th>C (20%)</th>
<th>D (80%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>-1, -1</td>
<td>-3, 0</td>
</tr>
<tr>
<td>D</td>
<td>0, -3</td>
<td>-2, -2</td>
</tr>
</tbody>
</table>

As the row player:
- Cooperating gives an expected payoff of $0.2(-1) + 0.8(-3) = -2.6$
- Defecting gives an expected payoff of $0.2(0) + 0.8(-2) = -1.6$

Thus, Defect is your best move. Of course, Defect is also the dominant strategy in this game, so no prediction about your opponent’s next move would ever lead you to Cooperate. Fictitious play becomes far more interesting in the absence of a dominant strategy.

6.2 Rock-Paper-Scissors

Imagine that you and your friend have been playing Rock-Paper-Scissors for hours on end. You have been playing each move with equal probability. Meanwhile, your friend has been choosing Rock 25% of the time, Paper 25% of the time, and Scissors, the remaining 50%. It’s time to figure out whether you’ve been playing your best strategy, or if you can do better.

Once again, the Rock-Paper-Scissors payoff matrix is below:

<table>
<thead>
<tr>
<th></th>
<th>R (25%)</th>
<th>P (25%)</th>
<th>S (50%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0, 0</td>
<td>-1, 1</td>
<td>1, -1</td>
</tr>
<tr>
<td>P</td>
<td>1, -1</td>
<td>0, 0</td>
<td>-1, 1</td>
</tr>
<tr>
<td>S</td>
<td>-1, 1</td>
<td>1, -1</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

You are the row player. Your opponent’s move probabilities are shown.

**Question:** What is your expected payoff if you play each move with equal probability?

**Question:** What is your best move according to Fictitious Play? What is its expected payoff? Is it a better strategy than what you’ve been doing?

**Question:** Fictitious Play is not merely a two-player game strategy; it can be extended to any repeated game where the payoff matrices are known and opponents’ actions are observed. What are some of the strengths and weaknesses of this strategy? In which situations does it work well, and in which situations is it limited?

6.3 Simulations

For the first coding section of this lab, you will be implementing a Fictitious Play agent for Rock-Paper-Scissors. Your agent will compete against a TA-built bot in a 100-round simulation.
Task: Implement Fictitious Play in `RpsFictitiousPlayAgent.java`.

To do so, you need to fill in two methods (look for `TODO` in the code):

1. **predict**: Use the opponent’s previous moves to generate a probability distribution over the opponent’s next move. N.B. The opponent’s previous moves are stored in a List, `this.opponentActions`, which is generated from the `GameReports` that your agent receives after each round of the simulation.

2. **optimize**: Use the probability distribution over the opponent’s moves, along with knowledge of the payoff matrix, to calculate the best move according to the Fictitious Play strategy.

Click **Run** and your agent will go head-to-head with a TA bot for 100 rounds. If Fictitious Play has been implemented correctly, your agent should win, earning payoffs of about 10 units more than our bot over the 100 rounds. (N.B. Our bot’s strategy is randomized, so you may not see this outcome every time.)

7 Exponential Weights

Another popular agent strategy for learning in repeated games is **Exponential Weights**. This strategy does not require knowledge of other players’ actions; it only requires that your agent keep track of its own results!

An agent running Exponential Weights keeps track of its average payoff over time from playing each of its actions. Using these average payoffs, the agent builds a probability distribution, from which its next action is sampled. This strategy works under the assumption that you should continue to choose actions that have been strong historically, but at the same time, you should not stop exploring other actions with at least some small probability, in case the environment changes (which happens when your opponent is also learning).

Here is a more formal description of the strategy. Given a set of available actions \( A \), and a vector of historical average payoffs \( r \in \mathbb{R}^{|A|} \), the probability of choosing action \( a \in A \) is:

\[
p(a) = \frac{e^{r_a}}{\sum_{a' \in A} e^{r_{a'}}}
\]

For example, in a game where choosing action \( x \) has provided an average payoff of 2 and choosing action \( y \) has provided an average payoff of 1.5, your next move is sampled from:

\[
p(x) = \frac{e^2}{e^2 + e^{1.5}} \approx 62%
\]

\[
p(y) = \frac{e^{1.5}}{e^2 + e^{1.5}} \approx 38%
\]

**Question**: Compared to Fictitious Play, what are some benefits and drawbacks of Exponential Weights?

**Question**: There are a few variations of Exponential Weights. For examples, some versions assign higher weights to more recent moves based on the assumption that these moves are more relevant. When would you expect a version like this to work well?

7.1 Simulations

Next, you will be implementing an Exponential Weights agent for Rock-Paper-Scissors, just as you did for Fictitious Play.

---

\(^4\)Payoffs are also referred to as rewards; hence, the letter \( r \).
Task: Implement Exponential Weights in `RpsExponentialWeightsAgent.java`.

To do so, you only need to fill in one method (again, look for `TODO:` in the code!):  

1. `calcMoveProbabilities`: Use your historical average payoffs to generate a probability distribution over your next move using the Exponential Weights strategy.

Note: The code handles the sampling for you; all you need to do is return a distribution.

Click **Run** and your agent will once again face a TA bot for 100 rounds. Once again, if your implementation is correct, your agent should win, earning payoffs of about 10 units more than our bot over the 100 rounds.

Question: Does one of the two strategies perform much better than the other against our bot?

8 Class Competition

Having implemented two agent strategies, and run two simulations against TA bots, you should have a pretty good idea of how the platform works by now. More importantly, you may also have some good ideas for strategies that can be used to play different games.

To conclude this lab, you will be implementing an agent to play the game of **Chicken**. Your agent will compete not with another TA bot, but rather with an agent developed by another pair of students in this lab, as usual for 100 rounds. In this competition, you are free to use any strategy you want, whether it is inspired by the ideas reviewed today, or something completely original.

8.1 Chicken

Chicken, like the Prisoners’ Dilemma, is a symmetric two-player, two-action, non-zero-sum game.

The premise is that two daredevil stuntmen are trying to impress a casting director in order to be chosen for *Fast and Furious 12*. The two stuntmen are driving in opposite directions on the road and are about to collide, head-on. Each has the option to **Swerve** or **Continue** going straight. If both players continue, they will crash, and receive a massive negative payoff in the form of injuries. If they both swerve, neither is rewarded with the part, as the casting director is left unimpressed. But if one swerves and one continues, the swerving player loses face, while the player who continued is rewarded handsomely.

Chicken is defined by the following payoff matrix:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0, 0</td>
<td>-1, 1</td>
</tr>
<tr>
<td>C</td>
<td>1, -1</td>
<td>-5, -5</td>
</tr>
</tbody>
</table>

The only way to win is to continue while the other player swerves. But are you willing to take the risk?

8.2 Implementing your Competition Agent

Task: Implement an agent that plays Chicken in `ChickenAgent.java`.

To do so, you again only need to fill in one method (as usual, look for `TODO:` in the code!):  

1. `nextMove`: Execute your strategy, and then return either **SWERVE** or **CONTINUE**.
Note that your agent must return its move within 1 second. If your `nextMove` method takes longer than 1 second, your action will not register, and neither agent will accrue any payoffs that round.

Because you may want to use some strategies from the previous simulations, we have included the helper methods they use in `ChickenAgent.java`. These include methods to get your average payoff for a given action, a list of your opponent’s previous moves, a list of your own previous moves, and the hypothetical payoff from a pair of actions. We have also included a method to sample a probability distribution. However, you are encouraged to expand your strategies beyond what you have already implemented today.

For our class competition leaderboard, your agent will need a name. You should name your agent by filling in the `NAME` variable in `ChickenAgent.java`.

### 8.3 Testing your Competition Agent Locally

Before submitting your agent code, you can test it against in a competition against itself.

To test your agents, simply run your competition agent file, `RpsExponentialWeightsAgent.java` or `RpsFicticiousPlayAgent.java`, in Eclipse. Doing so will launch a local competition in which your agent plays 100 rounds of the game against itself. The results of learning in self-play can be interesting in and of themselves, but here we are primarily interested in making sure your program does not crash, thereby ensuring that your agent will run smoothly in the class competition.

Note that competitions take longer to run than simulations, as they involve message passing over a network. Likewise, these tests of your agent in a simulated competition environment will take longer to run than the simulations, but they shouldn’t take more than a couple of minutes to complete.

The actual competition between you and your classmates is launched via `ChickenAgent.java`, which you are free to read of course, but which should not edit.

### 9 Submitting your Lab

*Before submitting your code, please make sure to name your agent!* Also, we haven’t thoroughly debugged the naming restrictions, but it is probably safest not to use any white space in your name.

In order to submit your code, please follow the instructions in the Lab Installation/Setup/Handin Guide.

### A Simulation Details; TLDR

Your task in this lab is to implement Fictitious Play and Exponential Weights agents, specifically in the methods `predict()` and `optimize()` for the former, and `calcMoveProbabilities()` for the latter. To provide context for these methods, we explain how the simulation proceeds, first in terms of a more generic `nextMove()` method, and then in terms of the methods you will implement in this lab.

At a high-level, the TRADINGPLATFORM simulates a game repeatedly as follows:

1. Your agent is paired against another agent (or set of agents, as the game rules require).
2. In each round of the simulation:
   (a) `GameSimulator` calls each agent’s `nextMove()` method, which returns the agents’ next moves.
   (b) `GameSimulator` executes these moves and calculates payoffs.
   (c) `GameSimulator` then broadcasts the results back to the agents in a sanitized `GameReport`. The
The calls to the `nextMove()` method are implemented differently for Fictitious Play agents and Exponential Weights agents. For the former, the `GameSimulator` calls the agent’s `predict()` method, so that it can build its probability distribution, and then `optimize()` to solicit its next move. For the latter, the `GameSimulator` calls `calcMoveProbabilities()`, and then samples from this distribution to arrive at the agent’s next move.