1 Trading Platform

Throughout CS1951k/2951z, we will simulate auctions and other games using a trading platform (TRADING-PLATFORM; TP) that simulates trading in user-defined environments by user-designed autonomous agents in real time. This platform was designed and built by Professor Greenwald in conjunction with past TAs and students of CS1951k/2951z, most notably Luke Camery ('17) and Andrew Coggins ('18).

TP is a Java program, built using a client-server model to support the creation of trading environments for autonomous agents. One of our primary goals in developing this platform is to provide a system in which users can easily create a consistent, yet diverse and robust, array of trading agent games. We are testing out its viability in CS1951k/2951z by using it to create various games, from a simple repeated prisoners' dilemma to a combinatorial auction for wireless spectrum.

In this lab, we will be working with TP to implement agent strategies for two different 2-player games: **The Prisoner's Dilemma** and **Rock-Paper-Scissors**. The two agent strategies that you will be implementing 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.[1,2]

2 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 reward), 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 follows:

	C	**D**
C	-1, -1	-3, 0
D	0, -3	-2, -2

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

3 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:

[1] Julia Robinson. An iterative method of solving a game. *Annals of Mathematics*, 54(2):296—301, 1951.

[2] Yoav Freund & Robert Schapire. Game theory, on-line prediction and boosting. Proceedings of the 9th Annual Conference on Computational Learning Theory, pp. 325–332, 1996.

	R	**P**	**S**
R	0, 0	-1, 1	1, -1
P	1, -1	0, 0	-1, 1
S	-1, 1	1, -1	0, 0

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.)

4 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.[3]

If your opponents in a game cannot be "trusted" to play the equilibrium, or if there is more than one equilibrium, 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 empirical data during a **repeated game**. It then builds an empirical probability distribution over the opponents' actions based on their play history, 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.

4.1 Prisoner's Dilemma

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

	C (20%)	**D** (80%)
C	-1, -1	-3, 0
D	0, -3	-2, -2

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.

4.2 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:

[3] Rosemarie Nagel. Unraveling in guessing games: An experimental study. *American Economic Review*, 85(5):1313–26, 1995.

	R (25%)	**P** (25%)	**S** (50%)
R	0, 0	-1, 1	1, -1
P	1, -1	0, 0	-1, 1
S	-1, 1	1, -1	0, 0

You are the row player. You 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 is 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?

4.3 Simulation: Implementing Fictitious Play

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.

4.3.1 Installing the Game Simulator

In order to install the Game Simulator and the agent stencil code:

1. Open a terminal and type `eclipse &` to open Eclipse.
2. Either create a new workspace, or use a pre-existing one.
3. Create a new project (**New** → **Java Project**). Eclipse will ask if you would like to create "module-info.java". Choose **Don't Create**.
4. Right click on your project, then choose **Build Path** → **Configure Build Path**.
5. Under the **Libraries** tab, click **Add External JARs** on the right hand side, and import the following file:
   - `/course/cs1951k/pub/2020/game_simulator/GameSimulator.jar`
6. Expand your project directory, right click on **src**, and create a new package to put your agents in.
7. Download the template agent files from the course website, then drag and drop them into your package. Change the package declaration in the template agents (line 1) to match your package in Eclipse.

4.3.2 Implementing your Agent

You will be implementing Fictitious Play in `RpsFictitiousPlayAgent.java`.

To do so, you will need to fill in two methods:

1. `predict` uses the opponent's previous moves to generate and return a probability distribution over the opponent's next move.
2. `optimize` a 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.

Note: Check out `RpsSampleAgent.java` to see a sample implementation of an agent that always plays the opponent's previous move. This should give you a good idea of how to write agent code for our platform.

4.3.3 Running the Simulation

Once your methods are filled in, simply click the **Run** button in Eclipse. This should launch a simulation, in which your agent will compete against our bot. After 100 rounds of Rock-Paper-Scissors, the results of the game will be displayed. 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 (although our bot's strategy is randomized, so you may not see this outcome every time).

Note: Because these simulations rely on launching a server, you will need to make sure you **Stop** the program once the game is over—it does not terminate on its own. If you try to stop the program, and still get a "port already in use" error when running it again, try right-clicking on the console and selecting **Terminate/Disconnect All**. If that does not work, please ask a TA for help.

5 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 using Exponential Weights simply keeps track of its average reward over time from playing each of its actions. Using these average rewards, 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 historically strong for you, but at the same time, you should continue to explore other actions with at least some small probability, in case the environment changes.

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

$$p(a) = \frac{\boldsymbol{e}^{R_a}}{\sum_{a' \in A} \boldsymbol{e}^{R_{a'}}}$$

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

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

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

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?

5.1 Simulation: Implementing Exponential Weights

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

You will be implementing Exponential Weights in `RpsExponentialWeightsAgent.java`.

To do so, you will need to fill in one method:

1. `calcMoveProbabilities` uses your historical average rewards 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.

If you completed the Fictitious Play section, you should not have do any addition work to install and run the simulation. Simply **Run** the agent file and your agent will once again face a 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?

6 Competition

Having implemented two agent strategies, and run the first two simulations, 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 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.

6.1 Chicken

Chicken, like the Prisoner's 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 in 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:

	S	**C**
S	0, 0	-1, 1
C	1, -1	-5, -5

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

6.2 Implementing your Agent

You will implement your agent that plays Chicken in `ChickenAgent.java`.

To do so, you will only need to fill in the `nextMove` method. This method should 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 any longer than that, your action will not register.

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 reward 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 we have already used today.

6.3 Running Your Agent

In order to enter the competition, you will need to give your agent a name. You will also need to input the correct host. Please wait for TA instructions on how to do this.

Once your agent is ready, the TAs will begin the competition by launching a server. Once that has happened, you will have a short window of time to **Run** your agent file. After the registration window is over, your agent will be paired up with another agent for 100 rounds, after which you will receive your game results.