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.

Labs are run using a simulator in which user-designed autonomous agents trade in real time in user-defined environments. The TAs have defined all the environments, and they have also designed some sample agents, against which you can test your own agent ideas. Most labs are structured such that the students first develop their agents "offline," meaning locally, against the TA's built-in agents. Then, at the end, there is typically a competition, where the students' agents compete with one another "online." These competitions comprise multiple runs, with agent scores projected at the end of the lab.

In this lab, you will be implementing two different agent strategies, and playing three different two-player games: the **Prisoners' Dilemma**, **Rock-Paper-Scissors**, and **Chicken**. 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.[1,2]

2 Setup

The stencil code for Lab 1 is available here. It includes two Python files in which you will implement fictitious play and exponential weights:

- `exponential_stencil.py`
- `fictitious_play_stencil.py`
- `competition_agent.py`

Please be sure to read the `README.md`. Here is an abridged version of the setup guide described in there:

1. Clone the repository with the command
   `git clone https://github.com/brown-agt/lab01-stencil.git`
2. Create a python virtual environment and activate it
3. Run `pip install agt_server`

3 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

[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.

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:

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

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

4 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:

	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.)

5 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 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.

5.1 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?

As the row player:

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

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

- 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.

5.2 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:

	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 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?

5.3 Simulations

For the first coding section of this lab, you will be implementing a Fictitious Play agent. Your agent will compete against a TA-built bot in a 1000-round simulation of Rock-Paper-Scissors.

Task: Implement Fictitious Play in `fictitious_play_stencil.py`

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, `self.opp_action_history`, which is updated after each round of the simulation by the server.
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.
   N.B. `self.calculate_utility(a1, a2)` returns the utility if player 1 plays $a1$ and player 2 plays $a2$ in the form $[u1, u2]$ where $u1$ is player 1's utility and $u2$ is player 2's utility

Click **Run** (or run `fictitious_play_stencil.py` in your terminal) and your agent will go head-to-head with a TA bot for 1000 rounds. If Fictitious Play has been implemented correctly, your agent should win, earning payoffs of about 500–600 units more than our bot over the 1000 rounds. (N.B. Our bot's strategy is randomized, so you may not see this outcome every time.)

6 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[4] $r \in \mathbb{R}^{|A|}$, 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 payoff of 2 and choosing action y has provided an average payoff 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?

6.1 Simulations

Next, you will be implementing an Exponential Weights agent, and you will again be competing with a TA bot in a 1000-round simulation of Rock-Paper-Scissors.

Task: Implement Exponential Weights in `exponential_stencil.py`.

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

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

Note: The support code handles samples actions from this probability distribution for you; all you need to do is return a distribution.

[4] Payoffs are also referred to as rewards; hence, the letter r.

Click **Run** (or run `exponential_stencil.py` in your terminal) and your agent will once again face off against a TA bot. As above, if your implementation is correct, your agent should win, earning payoffs of about 150-200 units more than our bot over the 1000 rounds.

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

7 Class Competition

Having implemented two agent strategies, and run two simulations against TA bots, you should have a pretty good idea of how the simulation environment 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 playing the game of **Chicken** repeatedly not against the TA bots, but against other students' bots. You are free to use *any strategy you want* in this competition, whether it is inspired by the ideas reviewed today, or something completely original, but your strategy should *not* be uniform random, as that would make for a boring competition!

7.1 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:

	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 take the risk?

7.2 Implementing your Competition Agent

Task: Implement an agent that plays **Chicken** in `competition_agent.py`.

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

1. `setup`: Initializes the agent for each new game they play.
2. `get_action`: Returns your agent's next action
3. `update`: Updates your agent with the current history, namely your opponent's choice and your agent's utility in the last game

Class competitions take longer to run than local simulations, as they involve message passing over a network. Consequently, **your agent must return its move within 1 second.** If your `get_action` method takes

longer than 1 second, your action will not register, and neither agent will accrue any payoffs that round. That said, you are encouraged to implement strategies beyond what you have already learned about today.

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

7.3 Testing your Competition Agent Locally

Before participating in the class competition, you can test your agent locally, in a mock competition against a TA bot. To do so, simply run your competition agent file, `competition_agent.py`, in your terminal of choice. We implemented this functionality so that you can make sure your agent does not crash in competition mode, thereby making it more likely your agent will run smoothly in the class competition.

The class competition is also launched via `competition_agent.py`, using the `join_server` boolean, along with the appropriate IP address and port. This latter information will be announced during lab.

8 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 `calc_move_probs()` for the latter.

To provide context for these methods, we explain how the simulation proceeds, first in terms of a more generic `get_action()` method, and then in terms of the methods you will implement in this lab.

At a high-level, the AGT_SERVER simulates a repeated game tournament as follows:

1. Your agent is paired against another agent (or set of agents, as the game rules require) in a round robin style. Each agent will face off against every other agent in a repeated game *twice*, once as player 1 and once as player 2.
2. Before each repeated game simulation begins, the AGT_SERVER calls each agent's `setup()` method.
3. Then, during each round of the simulation:
   (a) AGT_SERVER requests an action from each agent. Upon receiving this request, each agent calls its `get_action()` method, and then sends its action to AGT_SERVER.
   (b) AGT_SERVER executes these moves and calculates payoffs.
   (c) AGT_SERVER then broadcasts the results back to the agents in a *sanitized* json report summarizing the round's results. (The report is sanitized, as not all information is necessarily broadcast to all agents.) Upon receipt, your agent's history is automatically updated with the information it receives, which it can then retrieve using methods like `self.get_action_history()`, `self.get_util_history()`, etc..
   (d) Once the round has concluded, AGT_SERVER calls the agent's `update()` method, which gives it a chance to update its strategy.
4. After the simulation (i.e., all rounds of the repeated game) conclude, AGT_SERVER resets the game history and calls the agents `setup()` methods to allow them to reset for the next simulation.

The calls to the `get_action()` method are implemented differently for Fictitious Play and Exponential Weights agents. For the former, the AGT_SERVER 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 AGT_SERVER calls `calc_move_probs()`, and then samples from this distribution to arrive at the agent's next move.