Online Learning

February 10, 2023

1 Introduction

Within this project, we explored how various competing no regret algorithms could and will converge to various Nash equilibriums. We tested how various factors could affect this convergence such as learning rate, and tested these algorithms within multiple scenarios such as First Price Auction (FPA) and Generalised Second Price Auction (GSPA). We also tested how certain online learning algorithms could be manipulated within those scenarios to increase payoff for a competitor that knows what algorithm is being used.

2 Preliminaries

2.1 Learning Algorithms

Within this project, we utilized three various learning algorithms. The first is the exponential weights algorithm which utilizes the following equation to determine the probability of an action j will be chosen on round i.

$$EW_j^i(w) = \frac{(1+\epsilon)^{\frac{w_j^i}{h}}}{\sum_{w' \in w}(1+\epsilon)^{\frac{w'^i}{h}}}$$

The second algorithm, the hedge algorithm, we chose was similar to exponential weights in that it is a multiplicative weights formula but utilizes a slightly different calculation that allows for more exploration.

Finally the third algorithm we chose was Follow the Perturbed Leader (FTPL) with slight variations. The FTPL we implemented adder perturbation generated from a Gumbel distribution every round. We wanted to compare how adding perturbation only once at the beginning could possibly lead to poor predictions due to the decay over many rounds.

2.2 Pure Nash Equilibrium

Since we first wanted to compare how no regret algorithms learned over time while competing with other algorithms, we started off with a basic bimatrix setup where there are mirrored actions.

$$\begin{bmatrix} 3/3 & 0/5 \\ 5/0 & 1/1 \end{bmatrix}$$

Where the left number represents the row players payoffs and the right number representing the column players payoffs. Within this case, there is only a pure nash equilbria. Consider the case for the probability, p, of the column player choosing action 0, or the first column. Given that there are only two columns then the probability of choosing the second column is $(1-p)$. Then the mixed Nash equilibria equation says that according to the row players payoffs:

$$3p + 0(1-p) = 5p + 1(1-p)$$

$$3p = 5p + 1 - p$$

$$-p = 1$$

Given that the probability is negative, this means that there are no other nash equilibrium that exists besides the pure nash equilibria. Intuitively this also makes sense given that if the column player chooses action 0, then in a full information setting, the row player will always choose action 1, which will equate in the highest payoff for the row player and the lowest payoff for the column player. Since this is mirrored, the same can be said about the row player choosing action 0.

2.3 Multiple Nash Equilibrium

After we introduced a single pure nash equilibria, we wanted to see how learning algorithms converge in the cirucmstance where there are multiple nash equilbria. As such we ran the learning algorithms against each other in this new setup:

$$\begin{bmatrix} 2/2 & -5/-5 \\ -5/-5 & 1/1 \end{bmatrix}$$

This is similar to the original pure nash equilibria in that there are mirrored outcomes. This was intentional to view how it would split decisions between the equilbrium that exists both at 2/2 and 1/1.

2.4 Manipulating No Regret Learning Algorithms

The specific learning algorithm we targeted was the exponential weights algorithm. This is because it utilizes previous results to determine probabilities for current results. If it is possible to influence previous results as to lower the probability of a result that is theoretically better, then that can be exploited to influence future results.

The setup of this experiment is a bidder with value 90 and a bidder with value 30 in a first price auction. The 90 value bidder is utilizing an exponential weights algorithm while the 30 value bidder will be exploiting that. In a full information context, the 90 value bidder will always win if they know the 30 value bidder's bid. However within this scenario, we are able to manipulate the 90 value bidder's exponential weights algorithm to win the bid as the 30 value bidder.

Since the 90 value bidder is utilizing exponential weights which is nondeterministic, we can assume that every possible action a nonzero chance of occurring. As such, for small bids such as 10 where the 30 value bidder has a chance of winning, the 30 value bidder could purposefully lose to increase the chance of 90 value bidder to continually bid 10. This will allow the 30 value bidder to eventually change their bid and take advantage of this heightened chance to bid 10 by the 90 value bidder. Since the 90 value bidder is using an exponential weights learning algorithm that will eventually incorporate this change, and will adjust, in which the 30 value bidder could repeat the same process.

2.5 Generalized Second Price Auction

Generalized Second Price Auction is similar in the First Price Auction in that a learning algorithm could be manipulated by another bidder. Specifically in the case of two bidders, there is a delicate balance between wanting to be the lower bidder and pricing your bid. This is because the lower bidder will be paying 0 for a lesser weighted position, but lowering a bid too much would cause the other bidder to win out on the higher weighted position for less of a price. Specifically, this equation must be satisfied otherwise the bidder will be losing out on expected utility if paying above b_{i+1}:

$$(v - b_{i+1}) * w_i = v * w_1$$

In other words, the other bidder's bid could determine if you are losing expected utility. Bidding significantly below b_{i+1} could also prove to unviable as if the other bidder bids slightly above your bid, they would maximize their expected utility at the detriment of your expected utility.

These strategies allow for learning algorithms to be exploited. By overbidding for higher bids produced by the learning algorithm, we can force learning algorithms such as exponential weights to favor

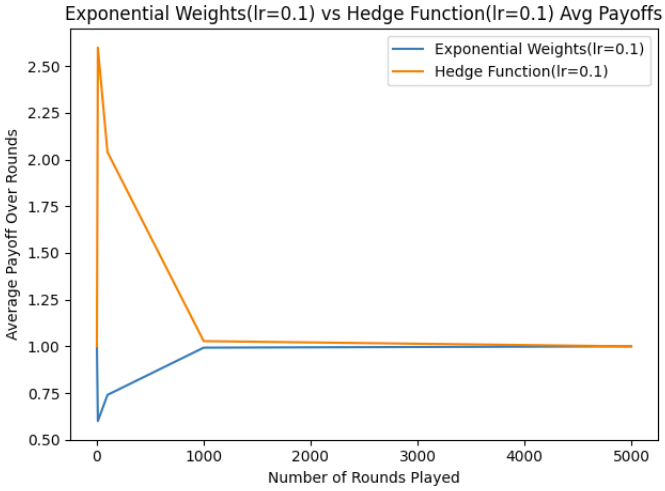
lower bids in order to balance the equation shown above. Similarly in the first price auction, we can then utilize this learnt behavior to slightly beat out their bid in order to increase expected utility.

3 Results

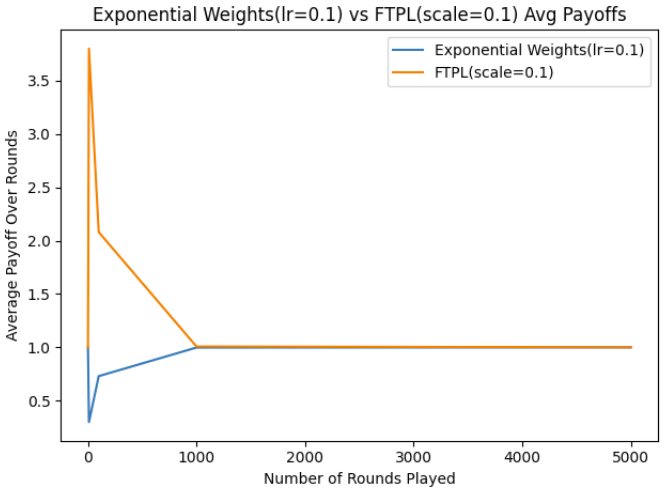
3.1 Comparisons Across Learning Algorithm in One Pure Equilibria

We tested various exponential weights learning algorithms with different learning rates against each other and found that within this scenario, there was negligible differences since there was only one equilbria. However, within the First Price Auction experiment, we noticed different trends.

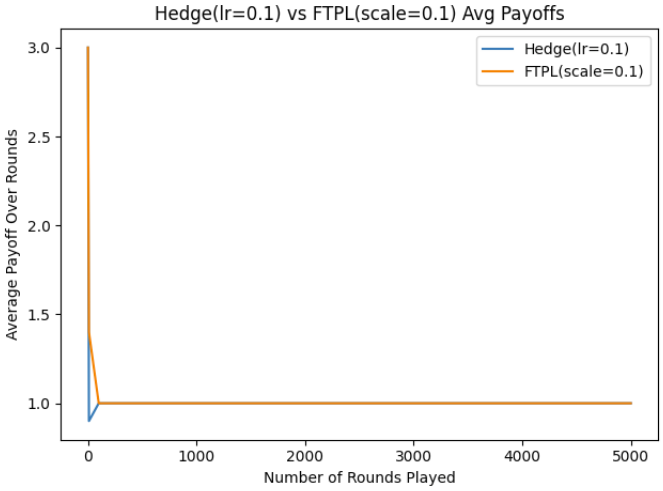
Furthermore we also tested the exponential weights algorithm versus the hedge algorithm which was a modified multiplicative weights that utilized regret and number of rounds in part of their calculation.



Additionally, we tested exponential weights against follow the perturbed leader (FTPL)

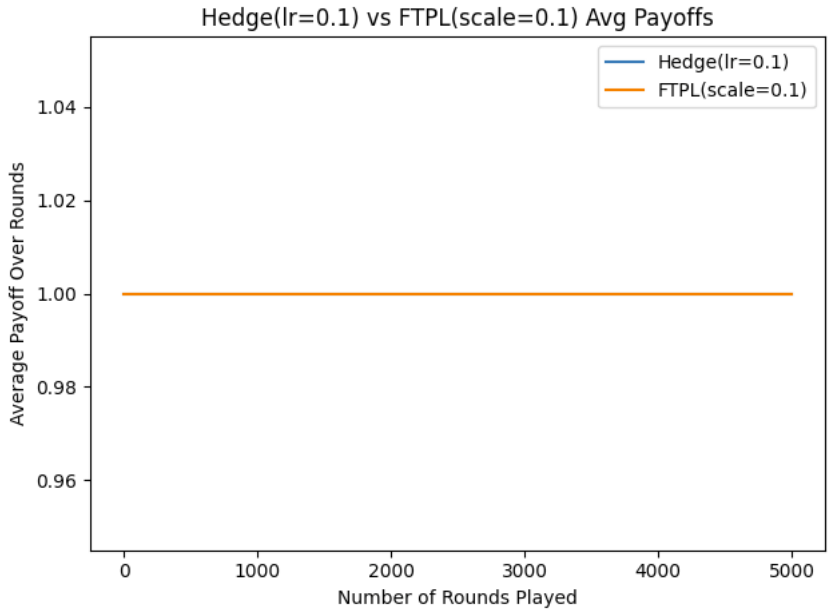


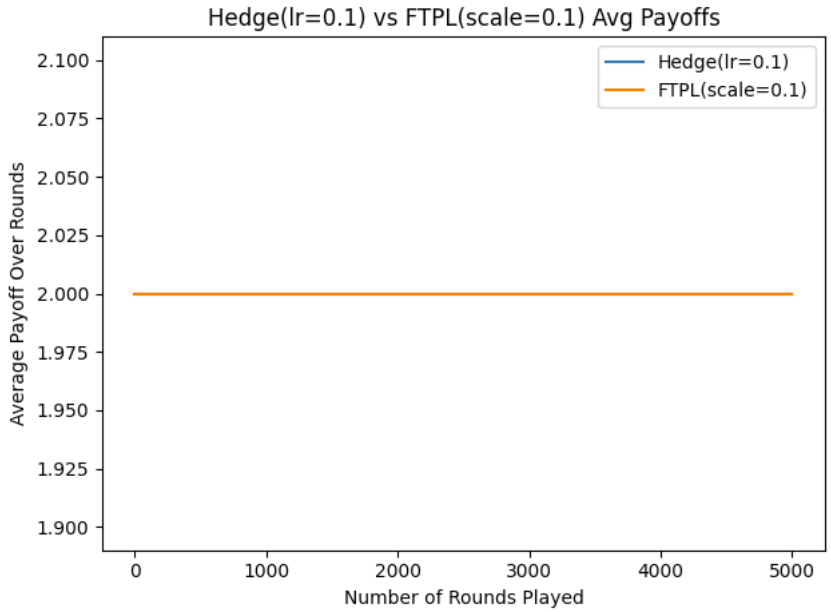
Finally we tested FTPL agains the hedge algorithm:



3.2 Multiple Nash Equilibrium

Additionally, we ran multiple the setup with multiple nash equilibrium to see which equilbria would be converged upon. We found interesting results that differed across which combination of learning algorithms that we used. Below are two graphs produced by running the same algorithm against each other.





Despite running on the same bimatrix payoffs, they converged to different equilbria. As such, we calculated how often this deviation was for the three combinations of learning algorithms against each other.

Table 1: Probability of Converging to the 1/1 Nash

(player1, player2)	(EW, FTPL)	(Hedge, FTPL)	(EW, Hedge)
Optimal	0.08	0.29	0.30

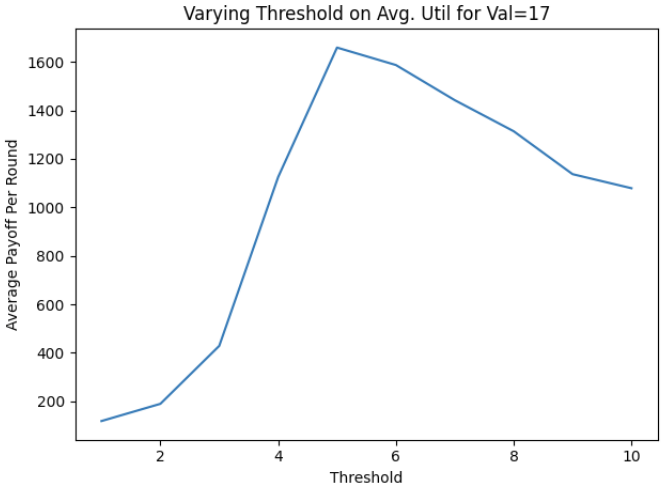
3.3 Manipulability on First Price Auction

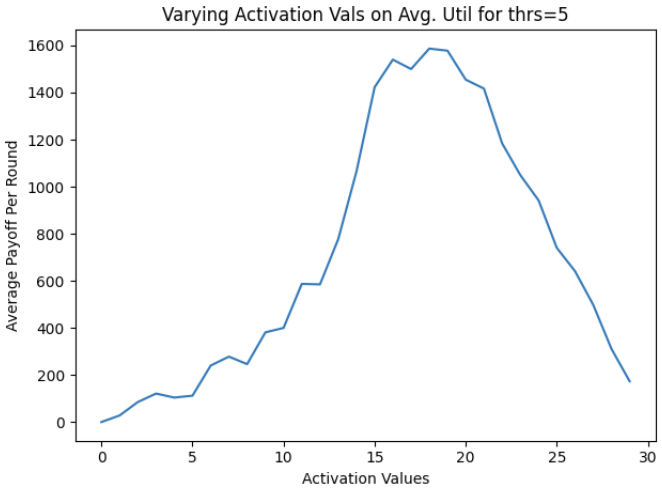
We were curious about the manipulability of the first price auction given that multiple nash equilibrium exist and that learning algorithms behavior could be easily predicted given that the opponent knows what learning algorithm is being used.

We first ran a control group between player 1 with value 30 which used FTPL and player 2 with value 90 which used EW and obtained that the converged nash equilibria was (29, 30) with payoffs (0, 60.31) respectively.

These results produced vanishing payoff for player 1. Our goal was to beat that result, and so we utilized the method as mentioned in the Preliminary. Conditioning the player 2 to bid low while occasionally beating out their bid, proved to be a successful strategy.

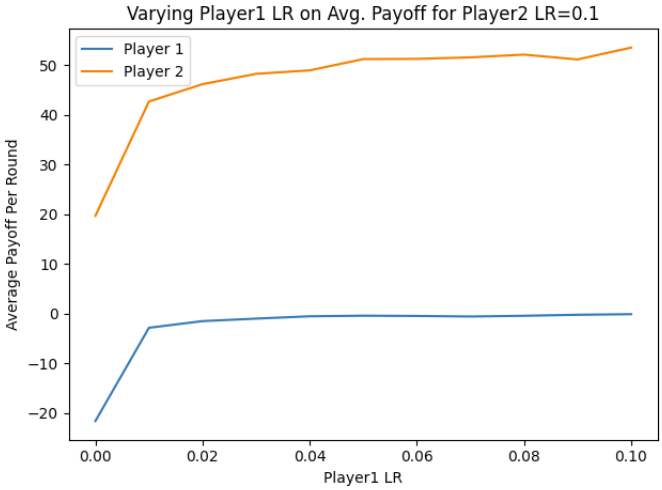
There was two factors that we could control. One was how many times should the exponential weights be conditioned before a mix up, and the second was the activation value to do the mix up. We decided to empirically test these values:

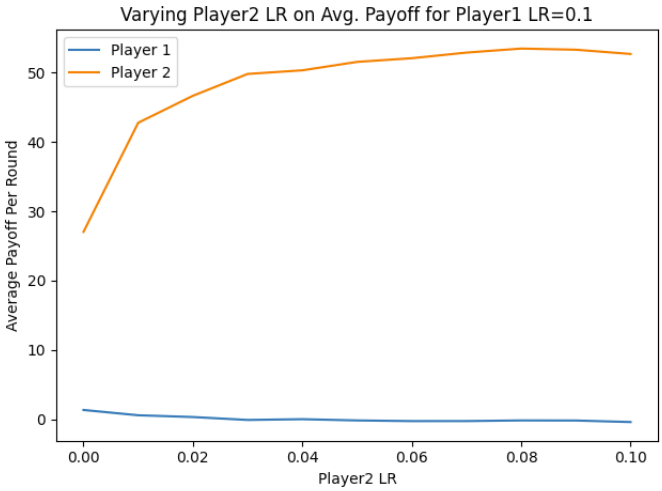




Note that for these graphs, the average payoff is represented over 1000 trials, such that at any given round, there is around a range of (0-1.6) payoff for player1, the exploiter. There similarly was a range of (59, 64) for player2, the online learning algorithm).

Regarding learning rates and exponential rates, the following are some trends for varying lrs for player 1 nad player 2:





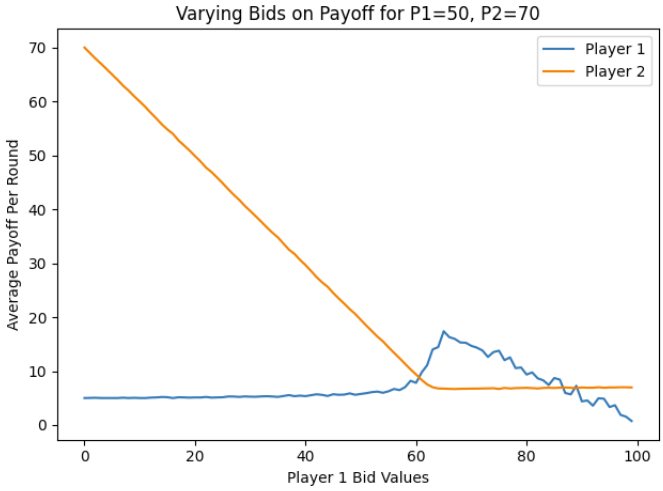
Interestingly enough, if two identical EW was ran given enough trials and a large enough learning rate, the nash equilbria was (54, 55).

3.4 Generalised Second Price Auction

We wanted to similarly prove that manipulability was not unique only the FPA, but that it could be occur in various other auctions as well. As such, we looked at the Generalised Second Price Auction.

Within this experiment, we utilized the values 50 for player1 and 70 for player2. Player1 would be the our methodology to exploit player2 which is still using an exponential weights learning algorithm. Furthermore, we utilized weights of 1 and 0.1 for the higher and lower weight respectively. For these values, we found this trend for player1 bids.

We ran a control group that pitted the FTPL, player1, against EW, player2, and found that the average payoff per round converged to (5, 27) while two EW with high enough leanring rates con-



verged to (5, 37).

As seen from our graph after, overbidding at value 64, the expected value converged to around (18, 7).

4 Conclusions

In our project, we compared various no regret algorithms and noticed how some combinations between no regret algorithms converged to different Nash equilbrium. We also demonstrated the possible manipulability of learning algorithms through two different approaches in the FPA and GSPA.

4.1 Comparisons of No Regret Learning Algorithms

Although, it is a bit difficult to say which no regret learning algorithm is generally better since they could perform differently across various situations. It was interesting to note the different convergence to different Nash Equilibirium in Double Nash example and within the FPA. For example, within the FPA scenario, the FTPL vs. Exponential Weights converged to what humans might have naturally converged to, that being the (29, 30) equilibria which is comparable to the (30, 31) equilbria with only slight differences in how draws are handled. Whereas, two competing exponential weights functions had various nash equilibria that was witnessed which differed based on learning rate. Increasing the learning rate enough for both players and over significant trials showed that it converged on (54, 55) which is inherently an unique number that might not make sense to a human, yet it continually converged up to that limit.

More on exponential weights, there was a slight yet noticeable trend of increasing learning rate that led to better payoffs and a better equilbria. This might only be in the case of repeated rounds of constant payoffs, where as increasing learning rates in more of a random bimatrix could actually have a negative effect. However, increasing learning does make sense in that exponential weights will strive to force an equilbria with higher learning rates.

It was also interesting to note the convergence speed for the single Pure Nash Equilbria example. Over multiple trials, the exponential weights seems to always perform a bit worse at the beginning which could possibly be alleviated with a higher learning rate. Hedge and FTPL both demonstrated faster convergence speed.

4.2 Manipulability of FPA

The control of FPA, as previously mentioned, made pratical sense with its convergence to (29, 30). This produces almost no payoff for the lower bid which makes sense intuitively that the higher value bidder could always just bid one higher than lower bidder's value. However, within an online learning context, the lower bidder could deceive the higher value bidder about what their true value is.

We utilized this approach to "trick" the 90 value bidder, player2, into believing that our value was 0. Repeating the 0 bid, encouraged player2 to maximize their return by slightly beating out 0 with a 1 bid. Since exponential weights takes heavy influence over successive bids about what decisions to make (multiplicative growth), even a few repeated behavior could continually trick the 90 value bidder into putting a poor bid.

This ended up being a question of balancing conditioning and payoff where conditioning is how many attempts to increase the probability of putting a bid of 1 and payoff being the actual mix up of inserting a bid at the true value. Since each condition takes away at an opportunity of a mixup, this maximization problem was not inherently apparent how to solve. Through our empirical results, we can see very similar trends across threshold and various activation values. According to our data, it seems that around 3-5 steps of conditioning as well as an activation value of 17 increased the payoff the most, with almost a polynomial increase and drop off after the max values.

With this glaring manipulability, we sought to consider how dangerous this problem really was. In terms of payoff, we saw that the actual payoff for both players increased (+1, +4) respectively per round. This was because player1 put in a bid of 0, player2 was able to win the auction for a majority of a time with a bid less than other equilbrium. Despite losing occasionally (in this case around once every four rounds), the profited utility makes up for that loss. We came to the conclusion that we would need further exploration to see if it was possible that this is an equilbria or if there were even more possibilities to further exploit EW within this context.

4.3 Manipulability of GSPA

The GSPA had a much more straightforward and direct way of exploiting learning algorithm. As mentioned in the preliminary, player1, the exploiter, can overbid at a specific such that they force player2, the learning algorithm, to bid lower than their bid. This is because if player2 beats out player1 with a higher bid, they would lose utility since the lower position would cost less (in this case free).

This was apparent in our graph as the moment player1's bid increased to 64, forcing an inequality in $(V_2 - b_1) * w_2 < V_2 * w_1$, player1's average payoff greatly increases. This is because player2 will not start bidding below b_1 which can be any number between (0, $b_1 - 1$). Given how close player1 and player2's true values are, this means that player1 will likely force a repeated behavior in player2's learning algorithm where they will continually reward a lower bid of player1's true value.

Unlike FPA, player2 is actively hurt by this behavior. This is because of the direct correlation of player1's bid to the utility unlike FPA where player2 paid utility based on their bid if they won. This is demonstrated in our results with the linear drop of player2's utility until it hits the limit of the previously mentioned inequality.

Again, we questioned whether this behavior was detrimental to the overall mechanism. We found that this strategy is possibly within reason and that there are not many more ways to exploit GSPA. Despite there being overbidding and thus the lack of truthful reporting, the amount changed is not egregious, and could be accounted as expected behavior. In fact, this equilibrium could be calculated and converted into a truthful mechanism if wanted. Additionally, if player2's algorithm was changed to mask their true value while attempting to track player1's true value, they could implement the same strategy.

One quick method of counteracting could just implementing a strategy to bid one below the overbid value. Since the overbidding strategy involves just repeating this one bid, this could force an adaptation of strategy of player1 as they would lose utility when forced to pay above their true value.
Sources: https://parameterfree.com/2019/09/11/online-gradient-descent/

https://www.cs.princeton.edu/ rlivni/cos511/lectures/lect18.pdf
https://inst.eecs.berkeley.edu/ ee290s/fa18/scribe$_n$otes/EE290S$_L$ecture$_N$ote$_7$.pdf

Code for project:

```
import numpy as np
import matplotlib.pyplot as plt


class BimatrixGame:
    def __init__(self, p1, p2):
        self.player1_payoffs = p1
        self.player2_payoffs = p2
        self.n_actions = len(p1)

    def play(self, p1_action, p2_action):

        return self.player1_payoffs[p1_action, p2_action], self.player2_payoffs[p1_actio


class LearningAlgorithm:

    def chooseAction(self):
        pass

    def updateProbs(self, action_payoffs, round_number):
        pass


class Hedge(LearningAlgorithm):
    # run choose action for each round
    # afterwards update with updateProbs
    # optimal lr =  (2kTlog(n)/2)^(2/3) i think?
    def __init__(self, actions_len, lr):
        self.probs = np.divide(np.ones(actions_len), actions_len)
        self.regret = 0
        self.lr = lr
        self.actions_len = actions_len
        self.best_hindsight_matrix = []

    def chooseAction(self):
        action = np.random.choice(self.actions_len, p=self.probs)
        return action

    def updateProbs(self, action_payoffs, round_number):
        self.best_hindsight_matrix.append(action_payoffs)
        best_hindsight_val = best_hindsight(self.best_hindsight_matrix)
        self.regret += best_hindsight_val[0] - np.max(action_payoffs)
        sum = 0
        for action, action_payoff in enumerate(action_payoffs):
            if self.regret == 0:
                self.probs[action] = self.probs[action] * (action_payoff/(round_number+1
            else:
                self.probs[action] = self.probs[action] * (action_payoff / (1+ self.regr
            sum += self.probs[action]

        self.probs = self.probs/sum

class FollowPerturbedLeader(LearningAlgorithm):
    def __init__(self, actions_len, scale):
```

```
        self.actions_len = actions_len
        self.weights = np.divide(np.ones(actions_len), actions_len)
        self.scale = scale
        self.weights = np.add(self.weights, np.random.gumbel(scale=self.scale, size=self
    def chooseAction(self):
        # print(np.argmax(self.weights))
        return np.argmax(self.weights)

    def updateProbs(self, action_payoffs, round_number):
        self.weights = np.add(self.weights, action_payoffs)
        self.weights = np.add(self.weights, np.random.gumbel(scale=self.scale, size=self

class ExponentialWeights(LearningAlgorithm):
    def __init__(self, actions_len, lr, h):
        self.probs = np.divide(np.ones(actions_len), actions_len)
        self.lr = lr
        self.actions_len = actions_len
        self.summed_weights = np.zeros(actions_len)
        self.h = h
    def chooseAction(self):
        action = np.random.choice(self.actions_len, p=self.probs)
        return action

    def updateProbs(self, action_payoffs, round_number):
        summed_weights = self.calculateSummedWeights(action_payoffs)
        out = []
        sum = 0
        for ele in summed_weights:
            numerator = np.power((1 + self.lr), ele / self.h)
            sum += numerator
            out.append(numerator)

        self.probs = np.divide(out, sum)

    def calculateSummedWeights(self, round):
        self.summed_weights = np.add(self.summed_weights, round)
        # print(self.summed_weights, self.probs)
        return self.summed_weights


def best_hindsight(matrix):
    summed = np.sum(matrix, axis=0)
    argmax_payoff = np.argmax(summed)
    max_payoff = np.max(summed)

    return max_payoff, argmax_payoff


best_hindsight_matrix = []


def runGame(p1, p2, num_rounds):
    # Play the game num_rounds times
    avg_p1_payoff = 0
    avg_p2_payoff = 0
    for i in range(num_rounds):
```

```
        player_1 = p1.chooseAction()
        player_2 = p2.chooseAction()

        payoffs = game.play(player_1, player_2)


        possible_payoffs_1 = game.player1_payoffs[:, player_2]
        possible_payoffs_2 = game.player2_payoffs[player_1, :]
        # print(player_1, player_2)
        best_hindsight_matrix.append(possible_payoffs_2)

        p1.updateProbs(possible_payoffs_1, i)
        p2.updateProbs(possible_payoffs_2, i)

        avg_p1_payoff += payoffs[0]
        avg_p2_payoff += payoffs[1]
        # print(f"Play {i}: Player 1 payoff = {payoffs[0]}, Player 2 payoff = {payoffs[1
    return avg_p1_payoff/num_rounds, avg_p2_payoff/num_rounds
# x_history_avg, y_history_avg = run_simulation(num_rounds, learning_rate1, learning_rate
# plot_results(x_history_avg, y_history_avg)

# def runFPA(p1, p2, num_rounds):
#     # Play the game num_rounds times
#     avg_p1_payoff = 0
#     avg_p2_payoff = 0
#     for i in range(num_rounds):
#         player_1 = p1.chooseAction()
#         player_2 = p2.chooseAction()
#         payoffs = game.play(player_1, player_2)
#
#
#         possible_payoffs_1 = game.player1_payoffs[:, player_2]
#         possible_payoffs_2 = game.player2_payoffs[player_1, :]
#         for idx, ele in enumerate(possible_payoffs_1):
#             if idx > player_2:
#                 pass
#             elif idx < player_2:
#                 possible_payoffs_1[idx] = 0
#             elif idx == player_2:
#                 possible_payoffs_1[idx] /= 2
#
#         for idx, ele in enumerate(possible_payoffs_2):
#             if idx > player_1:
#                 pass
#             elif idx < player_1:
#                 possible_payoffs_2[idx] = 0
#             elif idx == player_1:
#                 possible_payoffs_2[idx] /= 2
#
#         best_hindsight_matrix.append(possible_payoffs_2)
#
#         p1.updateProbs(possible_payoffs_1, i)
#         p2.updateProbs(possible_payoffs_2, i)
#
#         avg_p1_payoff += payoffs[0]
#         avg_p2_payoff += payoffs[1]
```

```
#         # print(f"Play {i}: Player 1 payoff = {payoffs[0]}, Player 2 payoff = {payoffs
#     return avg_p1_payoff/num_rounds, avg_p2_payoff/num_rounds

def plotAlgorithm(p1, p2, p1_name, p2_name):
    rounds = [1, 10, 100, 1000, 5000]
    p1_arr = []
    p2_arr = []
    for num_rounds in rounds:
        p1 = Hedge(len(player2_payoffs[0]), 0.1)
        # p1= ExponentialWeights(len(player1_payoffs[0]), 0.1, h=np.max(player1_payoffs)
        p2 = FollowPerturbedLeader(len(player2_payoffs[0]), 0.1)
        p1_payoff, p2_payoff = runGame(p1, p2, num_rounds)
        p1_arr.append(p1_payoff)
        p2_arr.append(p2_payoff)

    plt.plot(rounds, p1_arr, label=p1_name)
    plt.plot(rounds, p2_arr, label=p2_name)
    plt.title(f"{p1_name} vs {p2_name} Avg Payoffs")
    plt.legend()
    plt.xlabel("Number of Rounds Played")
    plt.ylabel("Average Payoff Over Rounds")
    plt.show()

def countMultipleNash(p1, p2):
    num_rounds = 100
    iterations = 100
    count = 0
    for iter in range(iterations):
        # p1 = Hedge(len(player2_payoffs[0]), 0.1)
        p1= ExponentialWeights(len(player1_payoffs[0]), 0.1, h=np.max(player1_payoffs))
        p2 = FollowPerturbedLeader(len(player2_payoffs[0]), 0.1)
        p1_payoff, p2_payoff = runGame(p1, p2, num_rounds)
        print(p1_payoff)
        if np.abs(1-p1_payoff) < 0.5:
            count += 1

    return count/iterations

def generatePayoffs(player1_val, player2_val, step, auction):
    if auction == "fpa":
        player2_payoffs = []
        player1_payoffs = []

        for i in range(0, 100, step):
            player2 = np.arange(0, 100, step)
            for idx, ele in enumerate(player2):
                if ele == i:
                    player2[idx] = (player2_val-ele)/2
                elif ele < i:
                    player2[idx] = 0
                elif ele > i:
                    player2[idx] = player2_val-ele

            player2_payoffs.append(player2)
            player1 = np.arange(0, 100, step)
            for idx, ele in enumerate(player1):
```

```
                if ele == i:
                    player1[idx] = (player1_val-ele)/2
                elif ele > i:
                    player1[idx] = 0
                elif ele < i:
                    player1[idx] = player1_val - i

            player1_payoffs.append(player1)
        player1_payoffs = np.array(player1_payoffs)
        player2_payoffs = np.array(player2_payoffs)
        return player1_payoffs, player2_payoffs
    if auction == "gspa":

        player2_payoffs = []
        player1_payoffs = []
        for i in range(0, 100, step):
            player2 = np.arange(0, 100, step)
            for idx, ele in enumerate(player2):
                if ele == i:
                    player2[idx] = (player2_val*w_0 + (player2_val-i)*w_1)/2
                elif ele < i:
                    player2[idx] = player2_val*w_0
                elif ele > i:
                    player2[idx] = (player2_val-i)*w_1

            player2_payoffs.append(player2)
            player1 = np.arange(0, 100, step)
            for idx, ele in enumerate(player1):
                if ele == i:
                    player1[idx] = (player1_val * w_0 + (player1_val - ele)*w_1)/2
                elif ele > i:
                    player1[idx] = player1_val * w_0
                elif ele < i:
                    player1[idx] = (player1_val - ele)*w_1

            player1_payoffs.append(player1)
        player1_payoffs = np.array(player1_payoffs)
        player2_payoffs = np.array(player2_payoffs)
        return player1_payoffs, player2_payoffs
w_0 = 0.1
w_1 = 1
player1_payoffs, player2_payoffs = generatePayoffs(30, 90, 1, "fpa")
# print(player1_payoffs)

# player1_payoffs = np.array([[2, 0.1], [0.1, 1]])
# player2_payoffs = np.array([[2, 0.1], [0.1, 1]])
# Create a repeated bimatrix game
# player1_payoffs, player2_payoffs = generatePayoffs(50, 70, 1, "gspa")
game = BimatrixGame(player1_payoffs, player2_payoffs)

num_rounds = 1000
hedge = Hedge(len(player2_payoffs[0]), 0.1)
lrs = np.arange(0, 0.11, 0.01)
ftpl = FollowPerturbedLeader(len(player2_payoffs[0]), 0.1)
util_1 = []
util_2 = []
```

```
for lr in lrs:
    exp_weights = ExponentialWeights(len(player1_payoffs[0]), lr, h = np.max(player2_pay
    exp_weights_2 = ExponentialWeights(len(player1_payoffs[0]), 0.1, h = np.max(player1_


    payoff1, payoff2 = runGame(exp_weights_2, exp_weights, num_rounds)
    print(payoff1, payoff2)
    util_1.append(payoff1)
    util_2.append(payoff2)
plt.plot(lrs, util_1, label="Player 1")
plt.plot(lrs, util_2, label="Player 2")
plt.legend()
plt.xlabel("Player2 LR")
plt.ylabel("Average Payoff Per Round")
plt.title("Varying Player2 LR on Avg. Payoff for Player1 LR=0.1")
plt.show()
# print(countMultipleNash(hedge, ftpl))
# runGame(hedge, ftpl, 100)
# plotAlgorithm(hedge, ftpl, "Hedge(lr=0.1)", "FTPL(scale=0.1)")

# print(player2_payoffs)
def searchexploitLearning(p2, num_rounds):
    avg_p1_payoff = 0
    avg_p2_payoff = 0
    count = 0
    thresholds = [5]
    vals = range(0, 30, 1)
    cur_max = -1
    best_val_threshold = None
    util = []
    for val in vals:
        for threshold in thresholds:
            count = 0
            avg_p1_payoff = 0
            avg_p2_payoff = 0
            p2 = ExponentialWeights(len(player1_payoffs[0]), 0.1, h = np.max(player2_pay
            for i in range(num_rounds):
                if count == threshold:
                    count = 0
                    player_1 = val
                else:
                    player_1 = 0
                    count+=1

                player_2 = p2.chooseAction()

                payoffs = game.play(player_1, player_2)
                possible_payoffs_1 = game.player1_payoffs[:, player_2]
                possible_payoffs_2 = game.player2_payoffs[player_1, :]
                # print(player_1, player_2, possible_payoffs_1, possible_payoffs_2)


                p2.updateProbs(possible_payoffs_2, i)

                avg_p1_payoff += payoffs[0]
                avg_p2_payoff += payoffs[1]
```

```
            util.append(avg_p1_payoff)
            if avg_p1_payoff/num_rounds > cur_max:
                print(avg_p1_payoff/num_rounds)
                best_val_threshold = (val, threshold)
                cur_max = avg_p1_payoff/num_rounds
                # print(f"Play {i}: Player 1 payoff = {payoffs[0]}, Player 2 payoff = {p
    plt.plot(vals, util)
    plt.xlabel("Activation Values")
    plt.ylabel("Average Payoff Per Round")
    plt.title("Varying Activation Vals on Avg. Util for thrs=5")
    plt.show()
    return best_val_threshold, cur_max
def exploitLearning(p2, num_rounds):
    avg_p1_payoff = 0
    avg_p2_payoff = 0
    count = 0
    for i in range(num_rounds):
        if count == 3:
            count = 0
            player_1 = 17
        else:
            player_1 = 0
            count+=1

        player_2 = p2.chooseAction()

        payoffs = game.play(player_1, player_2)
        possible_payoffs_1 = game.player1_payoffs[:, player_2]
        possible_payoffs_2 = game.player2_payoffs[player_1, :]
        # print(player_1, player_2, possible_payoffs_1, possible_payoffs_2)


        p2.updateProbs(possible_payoffs_2, i)

        avg_p1_payoff += payoffs[0]
        avg_p2_payoff += payoffs[1]
        print(f"Play {i}: Player 1 payoff = {payoffs[0]}, Player 2 payoff = {payoffs[1]}
    return avg_p1_payoff/num_rounds, avg_p2_payoff/num_rounds
# print(searchexploitLearning(exp_weights, 1000))
# print(exploitLearning(exp_weights, 100))

def exploitGSPA(p2, num_rounds):
    avg_p1_payoff = 0
    avg_p2_payoff = 0
    count = 0
    for i in range(num_rounds):
        if count == 10000:
            count = 0
            player_1 = 2
        else:
            player_1 = 64
            count+=1

        player_2 = p2.chooseAction()

        payoffs = game.play(player_1, player_2)
```

```
            possible_payoffs_1 = game.player1_payoffs[:, player_2]
            possible_payoffs_2 = game.player2_payoffs[player_1, :]
            print(player_1, player_2)


            p2.updateProbs(possible_payoffs_2, i)

            avg_p1_payoff += payoffs[0]
            avg_p2_payoff += payoffs[1]
            # print(f"Play {i}: Player 1 payoff = {payoffs[0]}, Player 2 payoff = {payoffs[1

        return avg_p1_payoff/num_rounds, avg_p2_payoff/num_rounds

    def searchexploitGSPA(p2, num_rounds):
        avg_p1_payoff = 0
        avg_p2_payoff = 0
        vals = range(0, 100)
        util = []
        util_2 = []
        for val in vals:
            avg_p1_payoff = 0
            avg_p2_payoff = 0
            p2 = ExponentialWeights(len(player1_payoffs[0]), 0.1, h=np.max(player2_payoffs))
            for i in range(num_rounds):
                player_1 = val
                player_2 = p2.chooseAction()

                payoffs = game.play(player_1, player_2)
                possible_payoffs_1 = game.player1_payoffs[:, player_2]
                possible_payoffs_2 = game.player2_payoffs[player_1, :]


                p2.updateProbs(possible_payoffs_2, i)

                avg_p1_payoff += payoffs[0]
                avg_p2_payoff += payoffs[1]
                # print(f"Play {i}: Player 1 payoff = {payoffs[0]}, Player 2 payoff = {payof
            print(avg_p1_payoff/num_rounds)
            util.append(avg_p1_payoff/num_rounds)
            util_2.append(avg_p2_payoff / num_rounds)
        plt.plot(vals, util, label="Player 1")
        plt.plot(vals, util_2, label="Player 2")
        plt.legend()
        plt.xlabel("Player 1 Bid Values")
        plt.ylabel("Average Payoff Per Round")
        plt.title("Varying Bids on Payoff for P1=50, P2=70")
        plt.show()
        return avg_p1_payoff/num_rounds, avg_p2_payoff/num_rounds

    # searchexploitGSPA(exp_weights, 1000)
    # print(exploitGSPA(exp_weights, 1000))
```