

Bimatrix games, different equilibria - Generate list of matrices (m1 = round 1) - Pure nash - Mixed nash - Prisoners' dilemma - RPS - Skip coarse-correlated equilibriums

FTL, OL, FTRL (regularized based on how recent the feedback was - $\frac{1}{i}$) A B X AX BX Y AY BY Online Learning - Given opponent took action X, we give alg AX, BX

MAB - Given opponent took action X and we took action A, we give MAB just AX

```
In [ ]: 1 import matplotlib.pyplot as plt
        2 import seaborn as sns
```

```

In [ ]: 1 import sys
2
3 import random
4 import nashpy as nash
5 import numpy as np
6
7 def rand_decimal():
8     return random.randrange(0, 99)/100
9
10 def find_max_payoffs(playoff_matrix):
11     max_row_payoff, max_col_payoff = 0, 0
12     for row in payoff_matrix:
13         for payoffs in row:
14             row_payoff = payoffs[0]
15             col_payoff = payoffs[1]
16             if row_payoff > max_row_payoff: max_row_payoff = row_payoff
17             if col_payoff > max_col_payoff: max_col_payoff = col_payoff
18     return max_row_payoff, max_col_payoff
19
20 def generate_dominant_strategy(num_actions=2, num_rounds=1):
21     row_dominant, col_dominant = random.randrange(0, num_actions), random.randrange(0, num_actions)
22     #print(row_dominant, col_dominant)
23     #generate randomized payoff matrix
24     payoff_matrix = [[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
25
26     #overwrite payoffs of dominant row and col with 'dominant' payoffs (random values that are higher than the max payoff)
27     max_row_payoff, max_col_payoff = find_max_payoffs(payoff_matrix)
28     for row in payoff_matrix:
29         row[col_dominant][1] = random.randrange(int(max_col_payoff*100), 100)/100
30     for payoff in payoff_matrix[row_dominant]:
31         payoff[0] = random.randrange(int(max_row_payoff*100), 100)/100
32
33     return payoff_matrix
34
35 def is_pure_nash(row, col, payoff_matrix, num_actions):
36     row_player_val, col_player_val = payoff_matrix[row][col][0], payoff_matrix[row][col][1]
37     for i in range(num_actions):
38         if payoff_matrix[row][i][1] > col_player_val: return False
39         if payoff_matrix[i][col][0] > row_player_val: return False
40     return True
41
42 def add_pure_nash(payoff_matrix, num_actions):
43     #print('pre-added')
44     #print(payoff_matrix)
45     pnash_row, pnash_col = random.randrange(0, num_actions), random.randrange(0, num_actions)
46     old_row_val, old_col_val = payoff_matrix[pnash_row][pnash_col][0], payoff_matrix[pnash_row][pnash_col][1]
47     row_max, col_max = 0, 0
48     row_max_index, col_max_index = None, None
49     for i in range(num_actions):
50         if payoff_matrix[pnash_row][i][1] > col_max:
51             col_max = payoff_matrix[pnash_row][i][1]
52             col_max_index = i
53
54         if payoff_matrix[i][pnash_col][0] > row_max:
55             row_max = payoff_matrix[i][pnash_col][0]
56             row_max_index = i
57
58     col_max_loc = payoff_matrix[pnash_row][col_max_index]
59     row_max_loc = payoff_matrix[row_max_index][pnash_col]
60     col_max_loc[1], payoff_matrix[pnash_row][pnash_col][1] = old_col_val, col_max
61     row_max_loc[0], payoff_matrix[pnash_row][pnash_col][0] = old_row_val, row_max
62     #print('added')
63     return [pnash_row, pnash_col]
64
65
66 def generate_pure_nash(num_actions=2, num_rounds=1):
67     payoff_matrix = [[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
68     pure_nash_list = []
69     for row in range(num_actions):
70         for col in range(num_actions):
71             if is_pure_nash(row, col, payoff_matrix, num_actions): pure_nash_list.append([row, col])
72     # if no pure nash randomly generated, recreate one
73     if pure_nash_list == []:
74         new_nash = add_pure_nash(payoff_matrix, num_actions)
75         pure_nash_list.append(new_nash)
76
77     #print(payoff_matrix)
78     #print(pure_nash_list)
79     return payoff_matrix
80
81 def generate_mixed_nash(num_actions=2, num_rounds=1):
82     pure_nash_list = None
83     while pure_nash_list != []:
84         payoff_matrix = [[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
85         pure_nash_list = []
86         for row in range(num_actions):
87             for col in range(num_actions):
88                 if is_pure_nash(row, col, payoff_matrix, num_actions): pure_nash_list.append([row, col])
89     return payoff_matrix
90
91 def generate_any_nash(num_actions=2, num_rounds=1):
92     #generate randomized payoff matrix, may have pure or mixed nash equilibrium(s)
93     payoff_matrix = [[rand_decimal(), rand_decimal()] for i in range(num_actions)] for i in range(num_actions)]
94     return payoff_matrix
95

```

```

96 def generate_prisoners():
97     row_cooperate_payoff, col_cooperate_payoff = random.randrange(3, 6), random.randrange(3, 6)
98     row_betray_payoff, col_betray_payoff = random.randrange(10, 20), random.randrange(10, 20)
99     row_double_betray_payoff, col_double_betray_payoff = random.randrange(0, 3), random.randrange(0, 3)
100     payoff_matrix = [
101         [[row_cooperate_payoff, col_cooperate_payoff], [0, col_betray_payoff]],
102         [[row_betray_payoff, 0], [row_double_betray_payoff, col_double_betray_payoff]]
103     ]
104     return payoff_matrix
105
106 def generate_rps():
107     rock_win_payoff = random.randrange(10, 20)
108     paper_win_payoff = random.randrange(10, 20)
109     scissors_win_payoff = random.randrange(10, 20)
110     tie_payoff = random.randrange(0, 3)
111     rock_loss_payoff = random.randrange(5, 10)
112     paper_loss_payoff = random.randrange(5, 10)
113     scissors_loss_payoff = random.randrange(5, 10)
114     payoff_matrix = [
115         [[tie_payoff, tie_payoff], [rock_loss_payoff, paper_win_payoff], [rock_win_payoff, scissors_loss_payoff]],
116         [[paper_win_payoff, rock_loss_payoff], [tie_payoff, tie_payoff], [paper_loss_payoff, scissors_win_payoff]],
117         [[scissors_loss_payoff, rock_win_payoff], [scissors_win_payoff, paper_loss_payoff], [tie_payoff, tie_payoff]]
118     ]
119
120     return payoff_matrix
121
122 generate_any_nash()
123 generate_prisoners()
124 generate_rps()

```

In []: 1 ## Multi-Armed Bandit Online Learning Algorithm

In []: 

```
1 class MAB:
2
3     def __init__(self, epsilon, num_actions=2):
4         self.weights_vector = [((1 / num_actions) * 100) for i in range(num_actions)]
5         self.totals_by_round = []
6         self.partial_totals_by_round = []
7         self.payoffs_by_round = []
8         self.choices_by_round = []
9         self.pi_tilda = []
10        self.actions_list = [i for i in range(num_actions)]
11        self.epsilon = epsilon
12        self.num_actions = num_actions
13
14    def reset_instance(self, epsilon=None, num_actions=2):
15        self.weights_vector = [((1 / num_actions) * 100) for i in range(num_actions)]
16        self.totals_by_round = []
17        self.partial_totals_by_round = []
18        self.payoffs_by_round = []
19        self.choices_by_round = []
20        self.pi_tilda = []
21        self.actions_list = [i for i in range(num_actions)]
22        self.num_actions = num_actions
23        if epsilon == None:
24            self.epsilon = self.epsilon
25        else:
26            epsilon = None
27
28    def choose_action(self, max_payoff):
29        # find weights
30        current_weights = [None for i in range(self.num_actions)]
31        for action in range(self.num_actions):
32            if self.choices_by_round == []:
33                #print(self.choices_by_round)
34                current_weights = self.weights_vector[0]
35            else:
36                #print(self.weights_vector)
37                #print(self.choices_by_round)
38                total_weights = sum(self.weights_vector[-1])
39                V_last = self.partial_totals_by_round[-1][action]
40                exp = V_last / max_payoff
41                current_weights[action] = (pow(1 + self.epsilon, exp) / total_weights) * 100
42        #convert probabilities to new MAB distribution
43        mab_weights = []
44        for i in range(len(current_weights)):
45            mab_weights.append(((1 - self.epsilon) * (current_weights[i] / 100) +
46                               (self.epsilon / self.num_actions)) * 100)
47
48        # randomly select from actions using weights from MAB
49        selected_action = random.choices(self.actions_list, weights=mab_weights, k=1)[0]
50        self.pi_tilda.append(mab_weights[selected_action])
51        self.weights_vector.append(current_weights)
52        self.choices_by_round.append(selected_action)
53
54        return selected_action
55
56    def process_payoff(self, selected_payoff, payoff_list):
57        # add new payoffs to totals, add payoff choice this round to payoffs matrix
58        #self.payoffs_by_round.append(selected_payoff/self.pi_tilda[-1])
59        self.payoffs_by_round.append(selected_payoff)
60
61        if self.totals_by_round == []:
62            temp_totals = []
63            for i in range(self.num_actions):
64                if i == self.choices_by_round[-1]:
65                    temp_totals.append(selected_payoff/self.pi_tilda[-1])
66                else:
67                    temp_totals.append(0)
68            self.partial_totals_by_round.append(temp_totals)
69            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
70        else:
71            last_round_totals = self.totals_by_round[-1]
72            curr_payoffs = []
73            for i in range(self.num_actions):
74                if i == self.choices_by_round[-1]:
75                    curr_payoffs.append(selected_payoff/self.pi_tilda[-1])
76                else:
77                    curr_payoffs.append(0)
78            self.partial_totals_by_round.append([(last_round_totals[i] + curr_payoffs[i]) for i in
79                                                range(self.num_actions)])
80            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])
81
82        #print(self.totals_by_round)
83        #print(self.payoffs_by_round)
84        #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

```

In [ ]: 1 class FTLRegularization:
2
3     def __init__(self, num_actions=2):
4         self.weights_vector = [1 for i in range(num_actions)]
5         self.totals_by_round = []
6         self.payoffs_by_round = []
7         self.choices_by_round = []
8         self.all_payoffs_by_round = []
9         self.actions_list = [i for i in range(num_actions)]
10        self.epsilon = 1000
11        self.num_actions = num_actions
12
13    def reset_instance(self, epsilon=None, num_actions=2):
14        self.weights_vector = [1 for i in range(num_actions)]
15        self.totals_by_round = []
16        self.payoffs_by_round = []
17        self.choices_by_round = []
18        self.all_payoffs_by_round = []
19        self.actions_list = [i for i in range(num_actions)]
20        self.num_actions = num_actions
21        if epsilon == None:
22            self.epsilon = self.epsilon
23        else:
24            epsilon = None
25
26    def find_ftlr_vector(self):
27        vector = [0 for i in range(self.num_actions)]
28        for index in range(len(self.all_payoffs_by_round)):
29            for action in range(self.num_actions):
30                #print(action, index, self.all_payoffs_by_round)
31                vector[action] += self.all_payoffs_by_round[index][action] * (index / len(self.all_payoffs_by_round))
32        return vector
33
34
35    def choose_action(self, max_payoff):
36        # find weights
37        current_weights = [None for i in range(self.num_actions)]
38        ftlr_vector = self.find_ftlr_vector()
39        for action in range(self.num_actions):
40            if self.totals_by_round == []:
41                V_last = 0
42            else:
43                V_last = ftlr_vector[action]
44            exp = V_last / max_payoff
45            current_weights[action] = pow(1 + self.epsilon, exp)
46            # randomly select from actions using weights as probabilities
47            selected_action = random.choices(self.actions_list, weights=current_weights, k=1)[0]
48            self.choices_by_round.append(selected_action)
49            self.weights_vector.append(current_weights)
50        return selected_action
51
52    def process_payoff(self, selected_payoff, payoff_list):
53        # add new payoffs to totals, add payoff choice this round to payoffs matrix
54        self.payoffs_by_round.append(selected_payoff)
55        self.all_payoffs_by_round.append(payoff_list)
56        if self.totals_by_round == []:
57            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
58        else:
59            last_round_totals = self.totals_by_round[-1]
60            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])
61
62        #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'

```

Algorithm Classes

```

In [ ]: 1 class ExponentialWeights:
2
3     def __init__(self, epsilon, num_actions=2):
4         self.weights_vector = [1 for i in range(num_actions)]
5         self.totals_by_round = []
6         self.payoffs_by_round = []
7         self.choices_by_round = []
8         self.actions_list = [i for i in range(num_actions)]
9         self.epsilon = epsilon
10        self.num_actions = num_actions
11
12    def reset_instance(self, epsilon=None, num_actions=2):
13        self.weights_vector = [1 for i in range(num_actions)]
14        self.totals_by_round = []
15        self.payoffs_by_round = []
16        self.choices_by_round = []
17        self.actions_list = [i for i in range(num_actions)]
18        self.num_actions = num_actions
19        if epsilon == None:
20            self.epsilon = self.epsilon
21        else:
22            epsilon = None
23
24    def choose_action(self, max_payoff):
25        # find weights
26        current_weights = [None for i in range(self.num_actions)]
27        for action in range(self.num_actions):
28            if self.totals_by_round == []:
29                V_last = 0
30            else:
31                V_last = self.totals_by_round[-1][action]
32            exp = V_last / max_payoff
33            current_weights[action] = pow(1 + self.epsilon, exp)
34            # randomly select from actions using weights as probabilities
35            selected_action = random.choices(self.actions_list, weights=current_weights, k=1)[0]
36            self.choices_by_round.append(selected_action)
37            self.weights_vector.append(current_weights)
38            return selected_action
39
40    def process_payoff(self, selected_payoff, payoff_list):
41        # add new payoffs to totals, add payoff choice this round to payoffs matrix
42        self.payoffs_by_round.append(selected_payoff)
43        if self.totals_by_round == []:
44            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
45        else:
46            last_round_totals = self.totals_by_round[-1]
47            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])
48
49
50    #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'

```

```

In [ ]: 1 class FTL:
2
3     def __init__(self, num_actions=2):
4         self.totals_by_round = []
5         self.payoffs_by_round = []
6         self.choices_by_round = []
7         self.actions_list = [i for i in range(num_actions)]
8         self.num_actions = num_actions
9
10    def reset_instance(self, num_actions=2):
11        self.totals_by_round = []
12        self.payoffs_by_round = []
13        self.choices_by_round = []
14        self.actions_list = [i for i in range(num_actions)]
15        self.num_actions = num_actions
16
17    def choose_action(self, max_payoff):
18        # randomly select from actions using highest total payoff so far
19        if self.totals_by_round != []:
20            selected_action = self.totals_by_round[-1].index(max(self.totals_by_round[-1]))
21            self.choices_by_round.append(selected_action)
22            return selected_action
23        else:
24            selected_action = random.randrange(0, self.num_actions)
25            return selected_action
26
27    def process_payoff(self, selected_payoff, payoff_list):
28        # add new payoffs to totals, add payoff choice this round to payoffs matrix
29        self.payoffs_by_round.append(selected_payoff)
30        if self.totals_by_round == []:
31            self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
32        else:
33            last_round_totals = self.totals_by_round[-1]
34            self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])
35
36
37    #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'

```

Matchup Simulator

```

In [ ]: 1 # helpers to find regret of an algorithm
2 def sum_to_round_i(alg_payoffs, current_round):
3     total = 0
4     for i in range(current_round):
5         total += alg_payoffs[i]
6     return total
7
8 def individual_regrets(alg_payoffs, round_totals):
9     final_payoffs = round_totals[-1]
10    opt_action = final_payoffs.index(max(final_payoffs))
11    #print(opt_action)
12    individual_regrets = [0 for i in range(len(alg_payoffs))]
13    for round in range((len(alg_payoffs))):
14        individual_regrets[round] = (round_totals[round][opt_action] - sum_to_round_i(alg_payoffs, round))
15        / (round + 1)
16    return individual_regrets
17
18 #takes two instantiations of algorithm classes as inputs
19 def matchup_simulator(alg1, alg2, payoff_matrix, num_rounds, max_payoff):
20     num_actions = len(payoff_matrix)
21     for round in range(num_rounds):
22         # determine which action each algorithm picks
23         alg1_action = alg1.choose_action(max_payoff)
24         alg2_action = alg2.choose_action(max_payoff)
25
26         # determine the payoffs and payoff lists for the algorithm combination
27         payoff_cell = payoff_matrix[alg1_action][alg2_action]
28         alg1_payoff, alg2_payoff = payoff_cell[0], payoff_cell[1]
29         alg1_payoff_list, alg2_payoff_list = [], []
30         for i in range(num_actions):
31             alg1_payoff_list.append(payoff_matrix[i][alg2_action][0])
32             alg2_payoff_list.append(payoff_matrix[alg1_action][i][1])
33
34         # process the payoffs for the algorithm combination to prep alg1, alg2 for the next round
35         alg1.process_payoff(alg1_payoff, alg1_payoff_list)
36         alg2.process_payoff(alg2_payoff, alg2_payoff_list)
37         #print(alg1.choices_by_round)
38         #print(alg2.choices_by_round)
39         # find the regret at each round, return the regret List for each algorithm
40         alg1_regrets = individual_regrets(alg1.payoffs_by_round, alg1.totals_by_round)
41         alg2_regrets = individual_regrets(alg2.payoffs_by_round, alg2.totals_by_round)
42         #print(alg2.payoffs_by_round)
43         #print(alg2.totals_by_round)
44         return alg1_regrets, alg2_regrets
45
46 payoff_matrix = generate_dominant_strategy()
47 alg1 = MAB(0.5)
48 alg2 = MAB(0.1)
49 #alg2 = FTLRegularization()
50 #print(alg2.weights_vector)
51 #print(alg2.choose_action(1))
52 #alg2.choose_action(1)
53 matchup_simulator(alg1, alg2, payoff_matrix, 100, 1)

```

```

In [ ]: 1 ## Delete contents of result file ###
2
3 # DO NOT RUN INDIVIDUALLY #
4
5 file = open("results.txt", "r+")
6 file.truncate(0)
7 file.close()

```

Visualization of Regrets

```

In [ ]: 1 def visualize_regret(alg_results, rounds, lr_1, lr_2, plot_title, alg_1_name, alg_2_name, trial_type):
2
3     file_name = trial_type + '_' + alg_1_name + alg_2_name + "_" + f'{lr_1}' + "_" + f'{lr_2}' + '.png'
4
5     x = np.array(list(range(0, rounds)))
6     y_1 = np.array(alg_results[0])
7     y_2 = np.array(alg_results[1])
8     plt.plot(x, y_1, label=f'{alg_1_name}, learning rate = {lr_1}'.format(alg_1_name=alg_1_name, lr_1 = lr_1),
9             linewidth=1)
10    plt.plot(x, y_2, label=f'{alg_2_name}, learning rate = {lr_2}'.format(alg_2_name=alg_2_name, lr_2 = lr_2),
11            linewidth=1)
12    plt.xlabel("Round")
13    plt.ylabel("Average Regret Per Round")
14    plt.title(plot_title)
15    plt.legend(loc='best', prop={'size': 7})
16
17    plt.savefig(file_name)
18
19    plt.show()
20
21    file1 = open("results.txt", "a") # append mode
22    file1.write(file_name + ", alg1" + ": " + f'{alg_results[2]}' + "\n")
23    file1.write(file_name + ", alg2" + ": " + f'{alg_results[3]}' + "\n")
24    file1.close()

```

Matchup Trials


```

In [ ]: 1 # matchup trial helpers
2 def update_avg_regrets(alg1_avg_regret_per_round, alg2_avg_regret_per_round, n, new_alg1_regrets, new_alg2_regrets):
3     if alg1_avg_regret_per_round == None:
4         alg1_avg_regret_per_round = new_alg1_regrets
5     else:
6         for i in range(len(alg1_avg_regret_per_round)):
7             alg1_avg_regret_per_round[i] = ((n * alg1_avg_regret_per_round[i]) + new_alg1_regrets[i]) / (n + 1)
8
9     if alg2_avg_regret_per_round == None:
10        alg2_avg_regret_per_round = new_alg2_regrets
11    else:
12        for i in range(len(alg2_avg_regret_per_round)):
13            alg2_avg_regret_per_round[i] = ((n * alg2_avg_regret_per_round[i]) + new_alg2_regrets[i]) / (n + 1)
14
15 def find_bimatrix_equilibria(payoff_matrix):
16     row_player_payoffs = []
17     col_player_payoffs = []
18     for row in payoff_matrix:
19         new_cplayer_row = []
20         new_rplayer_row = []
21         for payoff in row:
22             new_cplayer_row.append(payoff[1])
23             new_rplayer_row.append(payoff[0])
24         row_player_payoffs.append(new_rplayer_row)
25         col_player_payoffs.append(new_cplayer_row)
26
27     A = np.array(row_player_payoffs)
28     B = np.array(col_player_payoffs)
29     game = nash.Game(A, B)
30     equilibria = game.support_enumeration()
31     return equilibria
32
33 # calculate what percent deviation alg1 and alg2 had from the closest nash equilibrium to their decisions
34 def dev_from_nash(alg1_last_choices, alg2_last_choices, payoff_matrix):
35     num_actions = len(payoff_matrix)
36     equilibria = find_bimatrix_equilibria(payoff_matrix)
37     alg1_choice_averages = [0 for i in range(num_actions)]
38     for action in range(num_actions):
39         for choice in alg1_last_choices:
40             if choice == action: alg1_choice_averages[action] += 1
41     alg2_choice_averages = [0 for i in range(num_actions)]
42     for action in range(num_actions):
43         for choice in alg2_last_choices:
44             if choice == action: alg2_choice_averages[action] += 1
45
46     for index in range(len(alg1_choice_averages)):
47         alg1_choice_averages[index] = alg1_choice_averages[index] / len(alg1_last_choices)
48     for index in range(len(alg2_choice_averages)):
49         alg2_choice_averages[index] = alg2_choice_averages[index] / len(alg2_last_choices)
50
51     alg1_min_diff = float('inf')
52     alg2_min_diff = float('inf')
53     for eq in equilibria:
54         alg1_eq, alg2_eq = eq[0], eq[1]
55         alg1_curr_diff = abs(alg1_eq[0] - alg1_choice_averages[0]) + abs(alg1_eq[1] - alg1_choice_averages[1])
56         alg2_curr_diff = abs(alg2_eq[0] - alg2_choice_averages[0]) + abs(alg2_eq[1] - alg2_choice_averages[1])
57         if alg1_curr_diff < alg1_min_diff: alg1_min_diff = alg1_curr_diff
58         if alg2_curr_diff < alg2_min_diff: alg2_min_diff = alg2_curr_diff
59
60     return alg1_min_diff, alg2_min_diff
61
62
63
64 def matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds):
65     alg1_avg_regret_per_round, alg2_avg_regret_per_round = None, None
66     alg1_dev_from_nash_list, alg2_dev_from_nash_list = [], []
67
68     for payoff_matrix in payoff_matrix_list:
69         # find which trial number we are on
70         n = payoff_matrix_list.index(payoff_matrix)
71
72         # find max payoff (h)
73         max_payoff = 0
74         for row in payoff_matrix:
75             for payoff in row:
76                 if payoff[0] > max_payoff: max_payoff = payoff[0]
77                 if payoff[1] > max_payoff: max_payoff = payoff[1]
78
79         # run matchup and find regret lists
80         new_alg1_regrets, new_alg2_regrets = matchup_simulator(alg1, alg2, payoff_matrix, num_rounds, max_payoff)
81
82         # update average regret lists with new regret lists
83         # update_avg_regrets(alg1_avg_regret_per_round, alg2_avg_regret_per_round, n, new_alg1_regrets, new_alg2_regrets)
84         if alg1_avg_regret_per_round == None:
85             alg1_avg_regret_per_round = new_alg1_regrets
86         else:
87             for i in range(len(alg1_avg_regret_per_round)):
88                 alg1_avg_regret_per_round[i] = ((n * alg1_avg_regret_per_round[i]) + new_alg1_regrets[i]) / (n + 1)
89
90         if alg2_avg_regret_per_round == None:
91             alg2_avg_regret_per_round = new_alg2_regrets
92         else:
93             for i in range(len(alg2_avg_regret_per_round)):
94                 alg2_avg_regret_per_round[i] = ((n * alg2_avg_regret_per_round[i]) + new_alg2_regrets[i]) / (n + 1)
95         #TODO: take final stored nash values, check if they are nash equilibrium, update average deviation from nash

```

```

96     alg1_last_actions = alg1.choices_by_round[-(int(num_rounds/10)):]
97     alg2_last_actions = alg2.choices_by_round[-(int(num_rounds/10)):]
98     alg1dev, alg2dev = dev_from_nash(alg1_last_actions, alg2_last_actions, payoff_matrix)
99     alg1_dev_from_nash_list.append(alg1dev)
100    alg2_dev_from_nash_list.append(alg2dev)
101
102    # reset alg1 and alg2 internally stored values
103    alg1.reset_instance()
104    alg2.reset_instance()
105
106    # calculate average deviation from nash equilibria
107    alg1_avg_nash_dev = sum(alg1_dev_from_nash_list) / len(alg1_dev_from_nash_list)
108    alg2_avg_nash_dev = sum(alg2_dev_from_nash_list) / len(alg2_dev_from_nash_list)
109
110    return [alg1_avg_regret_per_round, alg2_avg_regret_per_round, alg1_avg_nash_dev, alg2_avg_nash_dev]
111
112
113 payoff_matrix_list = []
114 for i in range(1000):
115     payoff_matrix_list.append(generate_dominant_strategy())
116 alg1 = ExponentialWeights(0.5)
117 alg2 = ExponentialWeights(1.0)
118 num_rounds = 500
119 matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
120

```

Run Trials on Payoff Matrix Types

```

In [ ]: ▶ 1 # Constants
2 NUM_TRIALS = 1000
3 NUM_ROUNDS = 500
4
5 #
6 # Trials for payoff matrices with RPS
7 #
8 payoff_matrix_list = []
9 for i in range(NUM_TRIALS):
10     payoff_matrix_list.append(generate_rps())
11 alg1 = ExponentialWeights(1.0)
12 alg2 = ExponentialWeights(1.0)
13 num_rounds = NUM_ROUNDS
14 rps_result_array = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
15
16 print(rps_result_array[0])
17 print(rps_result_array[1])

```

Dominant Strategy EW Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with dominant equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_dominant_strategy())
8     alg1 = ExponentialWeights(0.5)
9     alg2 = ExponentialWeights(0.5)
10    num_rounds = NUM_ROUNDS
11    ew_dominant_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(ew_dominant_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
14                    'EW', 'EW', 'DomStr')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_dominant_strategy())
19        alg1 = ExponentialWeights(0.1)
20        alg2 = ExponentialWeights(1.0)
21        num_rounds = NUM_ROUNDS
22        ew_dominant_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(ew_dominant_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
25                        'EW', 'EW', 'DomStr')
26
27    payoff_matrix_list = []
28    for i in range(NUM_TRIALS):
29        payoff_matrix_list.append(generate_dominant_strategy())
30        alg1 = ExponentialWeights(1.0)
31        alg2 = FTL()
32        num_rounds = NUM_ROUNDS
33        ew_dominant_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
34
35        visualize_regret(ew_dominant_result_array3, num_rounds, 1.0, 'inf (FTL)',
36                        'Round vs. Average Regret for EW Algorithm & FTL', 'EW', 'FTL', 'DomStr')
37
38    payoff_matrix_list = []
39    for i in range(NUM_TRIALS):
40        payoff_matrix_list.append(generate_dominant_strategy())
41        alg1 = ExponentialWeights(0)
42        alg2 = ExponentialWeights(1.0)
43        num_rounds = NUM_ROUNDS
44        ew_dominant_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
45
46        visualize_regret(ew_dominant_result_array4, num_rounds, 0, 1.0,
47                        'Round vs. Average Regret for EW Algorithms', 'EW', 'EW', 'DomStr')
48
49    #print(ew_dominant_result_array4[2])
50    #print(ew_dominant_result_array4[3])
51    #print(ew_dominant_result_array4[0])
52    #print(ew_dominant_result_array4[1])
53    #print(num_rounds)
54
55
56
57

```

Pure Nash EW Trials

In []: ▶

```
1 #
2 # Trials for payoff matrices with Pure Nash equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_pure_nash())
8     alg1 = ExponentialWeights(0.5)
9     alg2 = ExponentialWeights(0.5)
10    num_rounds = NUM_ROUNDS
11    ew_pure_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(ew_pure_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
14                    'EW', 'EW', 'Pure Nash')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_pure_nash())
19        alg1 = ExponentialWeights(0.1)
20        alg2 = ExponentialWeights(1.0)
21        num_rounds = NUM_ROUNDS
22        ew_pure_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(ew_pure_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
25                        'EW', 'EW', 'Pure Nash')
26
27    payoff_matrix_list = []
28    for i in range(NUM_TRIALS):
29        payoff_matrix_list.append(generate_pure_nash())
30        alg1 = ExponentialWeights(1.0)
31        alg2 = FTL()
32        num_rounds = NUM_ROUNDS
33        ew_pure_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
34
35        visualize_regret(ew_pure_result_array3, num_rounds, 0.1, 'inf (FTL)',
36                        'Round vs. Average Regret for EW Algorithm & FTL', 'EW', 'FTL', 'Pure Nash')
37
38    payoff_matrix_list = []
39    for i in range(NUM_TRIALS):
40        payoff_matrix_list.append(generate_pure_nash())
41        alg1 = ExponentialWeights(0)
42        alg2 = ExponentialWeights(1.0)
43        num_rounds = NUM_ROUNDS
44        ew_pure_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
45
46        visualize_regret(ew_pure_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
47                        'EW', 'EW', 'Pure Nash')
```

Mixed Nash EW Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with Mixed Nash equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_mixed_nash())
8     alg1 = ExponentialWeights(0.5)
9     alg2 = ExponentialWeights(0.5)
10    num_rounds = NUM_ROUNDS
11    mn_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(mn_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
14                    'EW', 'EW', 'Mix Nash')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_mixed_nash())
19        alg1 = ExponentialWeights(0.1)
20        alg2 = ExponentialWeights(1.0)
21        num_rounds = NUM_ROUNDS
22        mn_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(mn_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
25                        'EW', 'EW', 'Mix Nash')
26
27    payoff_matrix_list = []
28    for i in range(NUM_TRIALS):
29        payoff_matrix_list.append(generate_mixed_nash())
30        alg1 = ExponentialWeights(1.0)
31        alg2 = FTL()
32        num_rounds = NUM_ROUNDS
33        mn_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
34
35        visualize_regret(mn_result_array3, num_rounds, 1.0, 'inf (FTL)', 'Round vs. Average Regret for EW Algorithm & FTL',
36                        'EW', 'FTL', 'Mix Nash')
37
38
39    payoff_matrix_list = []
40    for i in range(NUM_TRIALS):
41        payoff_matrix_list.append(generate_mixed_nash())
42        alg1 = ExponentialWeights(0)
43        alg2 = ExponentialWeights(1.0)
44        num_rounds = NUM_ROUNDS
45        mn_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
46
47        visualize_regret(mn_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
48                        'EW', 'EW', 'Mix Nash')
49

```

Any Nash EW Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with Any Nash Equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_any_nash())
8     alg1 = ExponentialWeights(0.5)
9     alg2 = ExponentialWeights(0.5)
10    num_rounds = NUM_ROUNDS
11    an_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(an_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms',
14                    'EW', 'EW', 'Any Nash')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_any_nash())
19        alg1 = ExponentialWeights(0.1)
20        alg2 = ExponentialWeights(1.0)
21        num_rounds = NUM_ROUNDS
22        an_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(an_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
25                        'EW', 'EW', 'Any Nash')
26
27
28    payoff_matrix_list = []
29    for i in range(NUM_TRIALS):
30        payoff_matrix_list.append(generate_any_nash())
31        alg1 = ExponentialWeights(1.0)
32        alg2 = FTL()
33        num_rounds = NUM_ROUNDS
34        an_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
35
36        visualize_regret(an_result_array3, num_rounds, 1.0, 'inf (FTL)', 'Round vs. Average Regret for EW Algorithm & FTL',
37                        'EW', 'FTL', 'Any Nash')
38
39    payoff_matrix_list = []
40    for i in range(NUM_TRIALS):
41        payoff_matrix_list.append(generate_any_nash())
42        alg1 = ExponentialWeights(0)
43        alg2 = ExponentialWeights(1.0)
44        num_rounds = NUM_ROUNDS
45        an_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
46
47        visualize_regret(an_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
48                        'EW', 'EW', 'Any Nash')
49

```

Prisoners' Dilemma EW Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with Prisoners' Dilemma
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_prisoners())
8     alg1 = ExponentialWeights(0.5)
9     alg2 = ExponentialWeights(0.5)
10    num_rounds = NUM_ROUNDS
11    p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(p_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW Algorithms', 'EW',
14                    'EW', 'Pr Dil')
15
16
17    payoff_matrix_list = []
18    for i in range(NUM_TRIALS):
19        payoff_matrix_list.append(generate_prisoners())
20        alg1 = ExponentialWeights(0.1)
21        alg2 = ExponentialWeights(1.0)
22        num_rounds = NUM_ROUNDS
23        p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
24
25        visualize_regret(p_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW Algorithms',
26                        'EW', 'EW', 'Pr Dil')
27
28    payoff_matrix_list = []
29    for i in range(NUM_TRIALS):
30        payoff_matrix_list.append(generate_prisoners())
31        alg1 = ExponentialWeights(1.0)
32        alg2 = FTL()
33        num_rounds = NUM_ROUNDS
34        p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
35
36        visualize_regret(p_result_array3, num_rounds, 1.0, 'inf (FTL)', 'Round vs. Average Regret for EW Algorithm & FTL',
37                        'EW', 'FTL', 'Pr Dil')
38    payoff_matrix_list = []
39    for i in range(NUM_TRIALS):
40        payoff_matrix_list.append(generate_prisoners())
41        alg1 = ExponentialWeights(0)
42        alg2 = ExponentialWeights(1.0)
43        num_rounds = NUM_ROUNDS
44        p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
45
46        visualize_regret(p_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for EW Algorithms',
47                        'EW', 'EW', 'Pr Dil')

```

Dominant Strategy MAB Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with dominant equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_dominant_strategy())
8     alg1 = MAB(0.5)
9     alg2 = MAB(0.5)
10    num_rounds = NUM_ROUNDS
11    mab_dominant_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(mab_dominant_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
14                    'MAB', 'MAB', 'Dom Str')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_dominant_strategy())
19        alg1 = MAB(0.1)
20        alg2 = MAB(1.0)
21        num_rounds = NUM_ROUNDS
22        mab_dominant_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(mab_dominant_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
25                        'MAB', 'MAB', 'Dom Str')
26
27        payoff_matrix_list = []
28        for i in range(NUM_TRIALS):
29            payoff_matrix_list.append(generate_dominant_strategy())
30            alg1 = MAB(1.0)
31            alg2 = FTL()
32            num_rounds = NUM_ROUNDS
33            mab_dominant_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
34
35            visualize_regret(mab_dominant_result_array3, num_rounds, 1.0, 'inf (FTL)',
36                            'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Dom Str')
37
38            payoff_matrix_list = []
39            for i in range(NUM_TRIALS):
40                payoff_matrix_list.append(generate_dominant_strategy())
41                alg1 = MAB(0)
42                alg2 = MAB(1.0)
43                num_rounds = NUM_ROUNDS
44                mab_dominant_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
45
46                visualize_regret(mab_dominant_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for MAB Algorithms',
47                                'MAB', 'MAB', 'Dom Str')

```

Pure Nash MAB Trials


```

In [ ]: 1 #
2 # Trials for payoff matrices with pure nash equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_pure_nash())
8     alg1 = MAB(0.5)
9     alg2 = MAB(0.5)
10    num_rounds = NUM_ROUNDS
11    mab_pn_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(mab_pn_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
14                    'MAB', 'MAB', 'Pure Nash')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_pure_nash())
19        alg1 = MAB(0.1)
20        alg2 = MAB(1.0)
21        num_rounds = NUM_ROUNDS
22        mab_pn_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(mab_pn_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
25                        'MAB', 'MAB', 'Pure Nash')
26
27    payoff_matrix_list = []
28    for i in range(NUM_TRIALS):
29        payoff_matrix_list.append(generate_pure_nash())
30        alg1 = MAB(1.0)
31        alg2 = FTL()
32        num_rounds = NUM_ROUNDS
33        mab_pn_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
34
35        visualize_regret(mab_pn_result_array3, num_rounds, 1.0, 'inf (FTL)',
36                        'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Pure Nash')
37
38
39    payoff_matrix_list = []
40    for i in range(NUM_TRIALS):
41        payoff_matrix_list.append(generate_pure_nash())
42        alg1 = MAB(0)
43        alg2 = MAB(1.0)
44        num_rounds = NUM_ROUNDS
45        mab_pn_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
46
47        visualize_regret(mab_pn_result_array4, num_rounds, 0, 1.0,
48                        'Round vs. Average Regret for MAB Algorithms', 'MAB', 'MAB', 'Pure Nash')
49

```

Any Nash MAB Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with pure nash equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_any_nash())
8     alg1 = MAB(0.5)
9     alg2 = MAB(0.5)
10    num_rounds = NUM_ROUNDS
11    mab_an_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(mab_an_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
14                    'MAB', 'MAB', 'Any Nash')
15
16    payoff_matrix_list = []
17    for i in range(NUM_TRIALS):
18        payoff_matrix_list.append(generate_any_nash())
19        alg1 = MAB(0.1)
20        alg2 = MAB(1.0)
21        num_rounds = NUM_ROUNDS
22        mab_an_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
23
24        visualize_regret(mab_an_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
25                        'MAB', 'MAB', 'Any Nash')
26
27
28    payoff_matrix_list = []
29    for i in range(NUM_TRIALS):
30        payoff_matrix_list.append(generate_any_nash())
31        alg1 = MAB(1.0)
32        alg2 = FTL()
33        num_rounds = NUM_ROUNDS
34        mab_an_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
35
36        visualize_regret(mab_an_result_array3, num_rounds, 1.0, 'inf (FTL)',
37                        'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Any Nash')
38
39
40    payoff_matrix_list = []
41    for i in range(NUM_TRIALS):
42        payoff_matrix_list.append(generate_any_nash())
43        alg1 = MAB(0)
44        alg2 = MAB(1.0)
45        num_rounds = NUM_ROUNDS
46        mab_an_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
47
48        visualize_regret(mab_an_result_array4, num_rounds, 0, 1.0, 'Round vs. Average Regret for MAB Algorithms',
49                        'MAB', 'MAB', 'Any Nash')
50

```

Prisoners' Dilemma Trials

```

In [ ]: 1 #
2 # Trials for payoff matrices with pure nash equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_prisoners())
8     alg1 = MAB(0.5)
9     alg2 = MAB(0.5)
10    num_rounds = NUM_ROUNDS
11    mab_p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(mab_p_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for MAB Algorithms',
14                    'MAB', 'MAB', 'Pris Dil')
15
16
17    payoff_matrix_list = []
18    for i in range(NUM_TRIALS):
19        payoff_matrix_list.append(generate_prisoners())
20        alg1 = MAB(0.1)
21        alg2 = MAB(1.0)
22        num_rounds = NUM_ROUNDS
23        mab_p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
24
25        visualize_regret(mab_p_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for MAB Algorithms',
26                        'MAB', 'MAB', 'Pris Dil')
27
28
29    payoff_matrix_list = []
30    for i in range(NUM_TRIALS):
31        payoff_matrix_list.append(generate_prisoners())
32        alg1 = MAB(1.0)
33        alg2 = FTL()
34        num_rounds = NUM_ROUNDS
35        mab_p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
36
37        visualize_regret(mab_p_result_array3, num_rounds, 1.0, 'inf (FTL)',
38                        'Round vs. Average Regret for MAB Algorithm & FTL', 'MAB', 'FTL', 'Pris Dil')
39
40    payoff_matrix_list = []
41    for i in range(NUM_TRIALS):
42        payoff_matrix_list.append(generate_prisoners())
43        alg1 = MAB(0)
44        alg2 = MAB(1.0)
45        num_rounds = NUM_ROUNDS
46        mab_p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
47
48        visualize_regret(mab_p_result_array4, num_rounds, 0, 1.0,
49                        'Round vs. Average Regret for MAB Algorithms', 'MAB', 'MAB', 'Pris Dil')
50

```

EW vs. MAB Trials

```

In [*]: 1 #
2 # Trials for payoff matrices with pure nash equilibria
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_any_nash())
8     alg1 = ExponentialWeights(0.5)
9     alg2 = MAB(0.5)
10    num_rounds = NUM_ROUNDS
11    ew_mab_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12
13    visualize_regret(mab_p_result_array1, num_rounds, 0.5, 0.5, 'Round vs. Average Regret for EW & MAB Algorithms',
14                    'EW', 'MAB', '')
15
16
17    payoff_matrix_list = []
18    for i in range(NUM_TRIALS):
19        payoff_matrix_list.append(generate_any_nash())
20        alg1 = ExponentialWeights(0.1)
21        alg2 = MAB(1.0)
22        num_rounds = NUM_ROUNDS
23        ew_mab_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
24
25        visualize_regret(mab_p_result_array2, num_rounds, 0.1, 1.0, 'Round vs. Average Regret for EW & MAB Algorithms',
26                        'EW', 'MAB', '')
27
28
29    payoff_matrix_list = []
30    for i in range(NUM_TRIALS):
31        payoff_matrix_list.append(generate_any_nash())
32        alg1 = ExponentialWeights(1.0)
33        alg2 = MAB(0.1)
34        num_rounds = NUM_ROUNDS
35        ew_mab_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
36
37        visualize_regret(mab_p_result_array3, num_rounds, 1.0, 0.1, 'Round vs. Average Regret for EW & MAB Algorithms',
38                        'EW', 'MAB', '')
39

```

Part 2

```
In [ ]: 1 def generate_asymmetric_prisoners():
2     row_cooperate_payoff, col_cooperate_payoff = random.randrange(3, 6), random.randrange(3, 6)
3     row_betray_payoff, col_betray_payoff = random.randrange(10, 20), random.randrange(10, 20)
4     row_double_betray_payoff, col_double_betray_payoff = random.randrange(0, 3), random.randrange(0, 3)
5     payoff_matrix = [
6         [row_cooperate_payoff, 10*col_cooperate_payoff], [0, col_betray_payoff]],
7         [[row_betray_payoff, 0], [row_double_betray_payoff, col_double_betray_payoff]]
8     ]
9     return payoff_matrix

In [ ]: 1 class EWPrisonersExploitation:
2     def __init__(self, num_actions=2):
3         self.totals_by_round = []
4         self.payoffs_by_round = []
5         self.choices_by_round = []
6         self.actions_list = [i for i in range(num_actions)]
7         self.payoff_matrix = [None for i in range(num_actions)]
8         self.confess = None
9         self.deny = None
10        self.opponent_confess_vals = None
11        self.opponent_deny_vals = None
12        self.num_actions = num_actions
13
14        def reset_instance(self, num_actions=2):
15            self.totals_by_round = []
16            self.payoffs_by_round = []
17            self.choices_by_round = []
18            self.actions_list = [i for i in range(num_actions)]
19            self.payoff_matrix = [None for i in range(num_actions)]
20            self.confess = None
21            self.deny = None
22            self.opponent_confess_vals = None
23            self.opponent_deny_vals = None
24            self.num_actions = num_actions
25
26            def choose_action(self, max_payoff):
27
28                # if within first 3 actions of game, or have not yet built our payoff matrix, guess randomly
29                if len(self.payoffs_by_round) <= self.num_actions or None in self.payoff_matrix:
30                    selected_action = random.randrange(0, self.num_actions)
31                    self.choices_by_round.append(selected_action)
32                    return selected_action
33
34                # If for the last 2 rounds the opponent confessed, deny
35                if self.payoffs_by_round[-1] in self.opponent_confess_vals and self.payoffs_by_round[-2] in
36                    self.opponent_confess_vals:
37                    selected_action = self.deny
38                    self.choices_by_round.append(selected_action)
39                    return selected_action
40
41                # otherwise, confess to bait opponent into higher probability of confessing
42                selected_action = self.confess
43                self.choices_by_round.append(selected_action)
44                return selected_action
45
46            def process_payoff(self, selected_payoff, payoff_list):
47                # find selected action
48                selected_action = payoff_list.index(selected_payoff)
49                if selected_action not in self.payoff_matrix:
50                    self.payoff_matrix[selected_action] = payoff_list
51
52                # if payoff matrix is full, find which action is confess, which action is deny
53                if self.confess == None or self.deny == None:
54                    if payoff_matrix[0][0] > payoff_matrix[1][1]:
55                        self.confess = 0
56                        self.deny = 1
57                    self.opponent_confess_vals = [payoff_matrix[0][0][0], payoff_matrix[1][0][0]]
58                    self.opponent_deny_vals = [payoff_matrix[1][1][0], payoff_matrix[0][1][0]]
59                else:
60                    self.confess = 1
61                    self.deny = 0
62                    self.opponent_confess_vals = [payoff_matrix[1][1][0], payoff_matrix[0][1][0]]
63                    self.opponent_deny_vals = [payoff_matrix[0][0][0], payoff_matrix[1][0][0]]
64
65                # add new payoffs to totals, add payoff choice this round to payoffs matrix
66                self.payoffs_by_round.append(selected_payoff)
67                if self.totals_by_round == []:
68                    self.totals_by_round.append([payoff_list[i] for i in range(self.num_actions)])
69                else:
70                    last_round_totals = self.totals_by_round[-1]
71                    self.totals_by_round.append([last_round_totals[i] + payoff_list[i] for i in range(self.num_actions)])
72
73                #NOTE: totals_by_round[-1] at the end of the simulation will help find 'OPT'
```

```

In [ ]: 1 #
2 # Trials against EQ
3 #
4
5 payoff_matrix_list = []
6 for i in range(NUM_TRIALS):
7     payoff_matrix_list.append(generate_asymmetric_prisoners())
8     alg1 = EWPrisonersExploitation()
9     alg2 = ExponentialWeights(0.1)
10    num_rounds = NUM_ROUNDS
11    mab_p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
12    print(mab_p_result_array1)
13
14 visualize_regret(mab_p_result_array1, num_rounds, 'Pris. Exploit.', 0.1,
15                 'Round vs. Average Regret for EW & Exploitative Exponential Weights (EEW)', 'EEW', 'EW', 'Pris Dil')
16
17
18 payoff_matrix_list = []
19 for i in range(NUM_TRIALS):
20     payoff_matrix_list.append(generate_asymmetric_prisoners())
21     alg1 = EWPrisonersExploitation()
22     alg2 = ExponentialWeights(0.5)
23     num_rounds = NUM_ROUNDS
24     mab_p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
25
26 visualize_regret(mab_p_result_array2, num_rounds, 'Pris. Exploit.', 0.5,
27                 'Round vs. Average Regret for EW & Exploitative Exponential Weights (EEW)', 'EEW', 'EW', 'Pris Dil')
28
29
30 payoff_matrix_list = []
31 for i in range(NUM_TRIALS):
32     payoff_matrix_list.append(generate_asymmetric_prisoners())
33     alg1 = EWPrisonersExploitation()
34     alg2 = ExponentialWeights(1.0)
35     num_rounds = NUM_ROUNDS
36     mab_p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
37
38 visualize_regret(mab_p_result_array3, num_rounds, 'Pris. Exploit.', 1.0,
39                 'Round vs. Average Regret for EW & Exploitative Exponential Weights (EEW)', 'EEW', 'EW', 'Pris Dil')
40
41
42 payoff_matrix_list = []
43 for i in range(NUM_TRIALS):
44     payoff_matrix_list.append(generate_asymmetric_prisoners())
45     alg1 = EWPrisonersExploitation()
46     alg2 = FTL()
47     num_rounds = NUM_ROUNDS
48     mab_p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
49
50 visualize_regret(mab_p_result_array4, num_rounds, 'Pris. Exploit.', 'inf (FTL)',
51                 'Round vs. Average Regret for Exploitative Exponential Weights (EEW) & FTL', 'EEW', 'FTL', 'Pris Dil')
52
53
54 #
55 # Trials against MAB
56 #
57 payoff_matrix_list = []
58 for i in range(NUM_TRIALS):
59     payoff_matrix_list.append(generate_asymmetric_prisoners())
60     alg1 = EWPrisonersExploitation()
61     alg2 = MAB(0.1)
62     num_rounds = NUM_ROUNDS
63     mab_p_result_array1 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
64     print(mab_p_result_array1)
65
66 visualize_regret(mab_p_result_array1, num_rounds, 'Pris. Exploit.', 0.1,
67                 'Round vs. Average Regret for MAB & Exploitative Exponential Weights (EEW)', 'EEW', 'MAB', 'Pris Dil')
68
69
70 payoff_matrix_list = []
71 for i in range(NUM_TRIALS):
72     payoff_matrix_list.append(generate_asymmetric_prisoners())
73     alg1 = EWPrisonersExploitation()
74     alg2 = MAB(0.5)
75     num_rounds = NUM_ROUNDS
76     mab_p_result_array2 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
77
78 visualize_regret(mab_p_result_array2, num_rounds, 'Pris. Exploit.', 0.5,
79                 'Round vs. Average Regret for MAB & Exploitative Exponential Weights (EEW)', 'EEW', 'MAB', 'Pris Dil')
80
81
82 payoff_matrix_list = []
83 for i in range(NUM_TRIALS):
84     payoff_matrix_list.append(generate_asymmetric_prisoners())
85     alg1 = EWPrisonersExploitation()
86     alg2 = MAB(1.0)
87     num_rounds = NUM_ROUNDS
88     mab_p_result_array3 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
89
90 visualize_regret(mab_p_result_array3, num_rounds, 'Pris. Exploit.', 1.0,
91                 'Round vs. Average Regret for MAB & Exploitative Exponential Weights (EEW)', 'EEW', 'MAB', 'Pris Dil')
92
93
94 payoff_matrix_list = []
95 for i in range(NUM_TRIALS):

```

```

96     payoff_matrix_list.append(generate_asymmetric_prisoners())
97     alg1 = EWPrisonersExploitation()
98     alg2 = FTL()
99     num_rounds = NUM_ROUNDS
100     mab_p_result_array4 = matchup_trial(alg1, alg2, payoff_matrix_list, num_rounds)
101
102     visualize_regret(mab_p_result_array4, num_rounds, 'Pris. Exploit.', 'inf (FTL)',
103                    'Round vs. Average Regret for Exploitative Exponential Weights (EEW) & FTL', 'EEW', 'FTL', 'Pris Dil')
104

```

Prisoner's Dilemma EW Exploit. Trial Payoff Visualization Function

```

In [ ]: 1 def visualize_payoff(payoff1, payoff2, rounds, lr_1, lr_2, plot_title, alg_1_name, alg_2_name, trial_type):
2
3     file_name = trial_type + '_' + alg_1_name + alg_2_name + "_" + f'{lr_1}' + "_" + f'{lr_2}' + '.png'
4
5     x = np.array(list(range(0, rounds)))
6     y_1 = np.array(payoff1)
7     y_2 = np.array(payoff2)
8     plt.plot(x, y_1, label='{alg_1_name}, learning rate = {lr_1}'.format(alg_1_name=alg_1_name, lr_1 = lr_1),
9              linewidth=1)
10    plt.plot(x, y_2, label='{alg_2_name}, learning rate = {lr_2}'.format(alg_2_name=alg_2_name, lr_2 = lr_2),
11             linewidth=1)
12    plt.xlabel("Round")
13    plt.ylabel("Payoff Per Round")
14    plt.title(plot_title)
15    plt.legend(loc='best', prop={'size': 7})
16
17    plt.savefig(file_name)
18
19    plt.show()

```

Prisoner's Dilemma EW Exploitation Sample Trial

```

In [ ]: 1 payoff_matrix = generate_asymmetric_prisoners()
2     alg1 = EWPrisonersExploitation()
3     alg2 = ExponentialWeights(0.5)
4     num_rounds = 100
5     max_payoff = 0
6     for row in payoff_matrix:
7         for payoff in row:
8             if payoff[0] > max_payoff: max_payoff = payoff[0]
9             if payoff[1] > max_payoff: max_payoff = payoff[1]
10    regret1, regret2 = matchup_simulator(alg1, alg2, payoff_matrix, num_rounds, max_payoff)
11    payoffs1, payoffs2 = alg1.payoffs_by_round, alg2.payoffs_by_round
12    for row in payoff_matrix:
13        print(row)
14
15    visualize_payoff(payoffs1, payoffs2, num_rounds, 'Pris. Exploit.', 0.5,
16                   'Round vs. Payoffs for EW & Exploitative Exponential Weights (EEW) Algorithms', 'EEW', 'EW',
17                   'Pris Dil')
18
19    #print(payoffs1)
20    #print(payoffs2)

```