Exponential Weights Online Learning Algorithm Analysis

Northwestern University,
CS-396: Online Markets

1 Introduction

This analysis evaluates the performance of the exponential weights online learning algorithm on sets of generative and real-world data to determine where and how it can be most valuable. While there is a theoretical optimal epsilon value ($\epsilon = \sqrt{\ln(k)/n}$), the results showed that optimal fittings of the algorithm varied greatly by method of data generation. More specifically, the exponential weights algorithm was not well suited to certain randomized adversarial models and either unable to produce vanishing regret or less efficient than follow-the-leader. It performed well and produced vanishing regret in respect to a distribution with fixed-probability success rates and a stock market sample. The analysis concluded that exponential weights is an effective alternative to follow-the-leader in that it improves on FTL's exceedingly low worst-case regret by being non-deterministic at the cost of being outperformed by FTL on less adversarial generative models.

2 Preliminaries

This analysis evaluates the performance of the exponential weights algorithm with various epsilon values, random guessing, and follow-the-leader in respect to their performance on three generative data sets and one real-world data set by conducting Monte Carlo trials. Random guessing is the approach of randomly selecting an action each round. Follow-the-leader is the approach of always selecting the action with the best cumulative payoff in hindsight. Exponential weights is the approach of using cumulative payoffs in hindsight to give each value an exponential weight, where the weights grow proportionate to the epsilon value used with the algorithm. The exponential weights algorithm behaves as follows: given learning rate ϵ, the cumulative utility of action j in round i is $V_j^i = \sum_{r=1}^{i} v_j^r$. Using that, in round i choose action j with probability proportional to $(1+\epsilon)_J^{V_j^{i-1}/h}$. An algorithm's success is gauged by the regret it produces, where an algorithm's regret in round $n = Regret_n = \frac{1}{n}[OPT - ALG]$ and OPT is the cumulative payoff in hindsight of the best action, and ALG is the cumulative payoff in hindsight of the algorithm's choice.

The 'adversarial fair payoffs' data sets were generated by each round drawing a payoff p from the uniform distribution on the interval $[0, 1]$, then assigning p to the action with the smallest total cumulative payoff so far and giving all other actions a payoff of 0. The 'Bernoulli payoffs' data sets were generated by fixing a probability for each action within $[0, 1/2]$ and each round assigning each action a payoff of 1 or 0 based off the action's fixed probability.

The 'alternating random payoffs' data sets were generated by drawing $n = \#actions$ payoffs from the uniform distribution on the interval $[0, 1]$ taking the maximum payoff p from that distribution, and assigning it from a randomly selected action a_i from set of all actions A. Each round, the payoff p would be assigned to either a_i or a_{i+1}, alternating back and forth between the actions, and all other actions would have a payoff of 0. The optimal solution in hindsight will either be a_i or a_{i+1}, whichever has cumulative payoff $p * \frac{\#\ rounds}{2}$. The total payoff over all actions each round will be p, and random guessing has a $\frac{1}{\#\ actions}$ chance of selecting the payoff, so it's cumulative payoff will be $\frac{p*\#\ rounds}{\#\ actions}$, and $Regret_{random\ guessing} = \frac{1}{\#\ rounds}(\frac{p*\#\ rounds}{2} - \frac{p*\#\ rounds}{\#\ actions}) = \frac{p}{\#\ actions} \neq 0\ if \#\ actions > 2$. Therefore the regret will not decrease towards 0 and random guessing does not have vanishing regret as long as $\#\ actions > 2$. Follow-the-leader will always pick the action a_i when action a_{i+1} has the payoff and vice versa, so it will have a cumulative regret of $\frac{p*\#\ rounds}{2}$, and $Regret_{FTL} = \frac{1}{\#\ rounds}(\frac{p*\#\ rounds}{2}) = \frac{p}{2} \neq 0$. Thus the regret will not decrease towards 0, and it does not have vanishing regret.

The 'real life S&P stock data' represents an extensive data set of roughly 500 different S&P companies stock prices retrieved from the online source Kaggle *(Can be accessed here)*. There is data representing stock prices of these companies from 2013-2018 throughout the entire year, however our analysis focuses on 2017. The following data table is a preview of the data set:

	index	date	open	high	low	close	volume	Name	Day	payoff
0	982	2017-01-03	47.28	47.340	46.135	46.30	6737752	AAL	1	0
1	983	2017-01-04	46.63	47.435	46.350	46.70	5859604	AAL	2	1
2	984	2017-01-05	46.52	46.930	45.610	45.89	6825316	AAL	3	0
3	985	2017-01-06	45.85	46.720	45.470	46.21	7260197	AAL	4	1
4	986	2017-01-09	46.01	47.340	45.780	47.08	4739142	AAL	5	1
...	...	...	...	...	...	...	...	...	...	...
126027	619009	2017-12-22	72.30	72.370	71.790	71.99	1345683	ZTS	247	0
126028	619010	2017-12-26	72.40	72.550	71.900	72.34	792134	ZTS	248	0
126029	619011	2017-12-27	72.59	72.690	72.250	72.45	1159771	ZTS	249	0
126030	619012	2017-12-28	72.49	72.600	72.140	72.39	710499	ZTS	250	0
126031	619013	2017-12-29	72.55	72.760	72.040	72.04	1704122	ZTS	251	0

The stock market data utilized in the report only included a sample of 10 stocks (*Ticker symbols: AAL, AAPL, AAP, ABBV, ABC, ABT, ACN, ADBE, ADI, ADM*) and the 251 calendar dates in 2017 where stock market data was accessible (January-December). This was accomplished by performing data cleaning on the imported data set using Python *numpy* and *pandas* packages. The actions are defined by the 10 distinct stocks that a player can invest any amount into and the rounds are the different calendar dates when the player invests. The payoffs for each stock or action are dependent on if the company's stock price increased, remained equal, or decreased between the start and end of the day ($closing price - opening price$). More specifically, a payoff of 1 is assigned for increased price or equal price ($closing price - opening price \geq 0$) and a payoff of 0 is assigned for decreasing price ($closing price - opening price < 0$). These payoffs were chosen to simulate a day-trading investor with the primary interest of making money every day rather than losing it. For the sake of the analysis, a player in this scenario must invest some amount of money every day.

Each data set was analyzed using Monte Carlo trials, so N data sets from each generation method were generated and analyzed across a set of learning rates $\epsilon = \{0, 0.25, 0.5, 0.75, 1, \sqrt{\ln(k)/n}, 1000\}$. The N chosen was sufficiently large that repeat trials produced consistent estimates of algorithm payoff and regret.

We hypothesized that the exponential weights algorithm would perform better than follow-the-leader or random guessing on adversarial models like adversarial fair payoffs and alternating random payoffs, but on non-adversarial data like the Bernoulli payoffs and the stock market payoffs, it would perform worse than follow-the-leader and better than random guessing. Our reasoning was that exponential weights acts as a less consistent version of follow-the-leader to be able to adapt to models that take advantage of follow-the-leader's deterministic nature.

3 Results

Average Regret Calculations	rand. guess	theor. opt. learn rate					FTL
	$\epsilon = 0$	$\epsilon = \sqrt{(lnk/n)}$	$\epsilon = 0.25$	$\epsilon = 0.5$	$\epsilon = 0.75$	$\epsilon = 1$	$\epsilon = 100$
Adversarial fair payoffs	**0.0015**	0.0053 ($\epsilon \approx 0.13$)	0.0095	0.0124	0.0165	0.0188	0.0807
Bernoulli payoffs	0.1766	0.1006 ($\epsilon \approx 0.13$)	0.0739	0.0562	0.0493	0.0470	**0.0361**
2017 S&P stock data	0.052	0.036 ($\epsilon \approx 0.1$)	0.048	0.036	0.056	0.04	**0.024**
Random alternating payoffs	0.2570	0.0905 ($\epsilon \approx 0.13$)	0.0638	**0.0633**	0.0704	0.0750	0.2062

The above table provides a informative summary of the results when simulating the exponential weights online learning algorithm on all 4 data sets. It is important to note that the learning rate influences algorithm with the intuition that a smaller learning rate takes longer to make good decisions but is more accurate overall, while a larger learning rate can commit to good decisions earlier in the simulation but in the long run its decisions are not as accurate. No learning ($\epsilon = 0$) is equivalent to random guessing or always picking a uniform random action, theoretical optimal learning rate ($\epsilon = \sqrt{(lnk/n)}$), and a very high learning rate ($\epsilon = 100$) represents follow the leader algorithm. The values bolded in the table represent the value with the lowest average regret compared to the other learning rates.

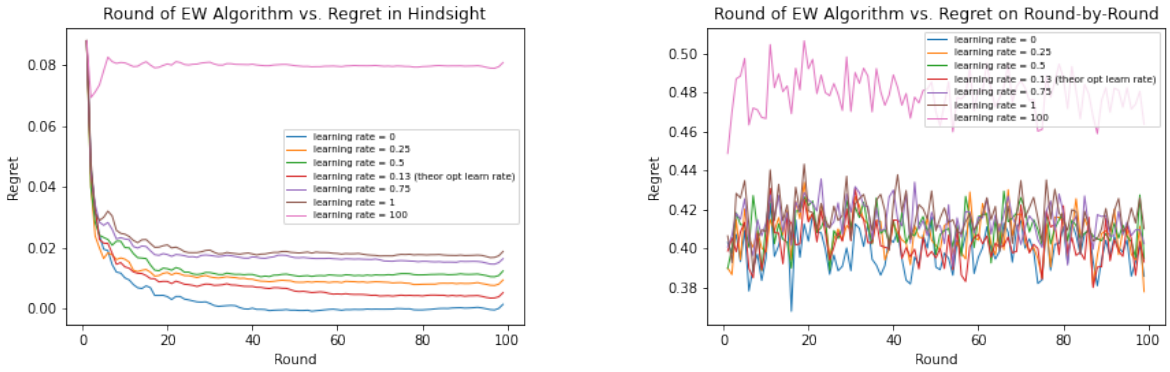

Figure 1: Exponential weights algorithm applied to **adversarial fair payoffs** (data generating model) using Monte Carlo trials

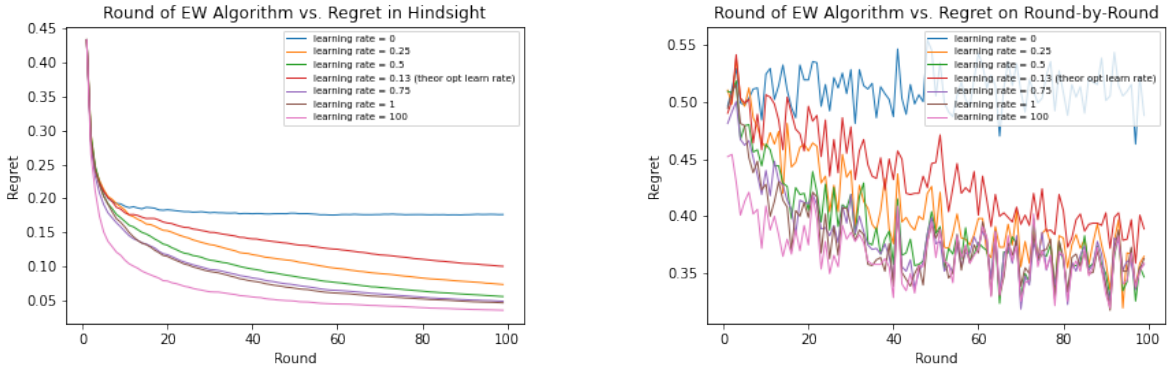

Figure 2: Exponential weights algorithm applied to **Bernoulli payoffs** (data generating model) using Monte Carlo trials

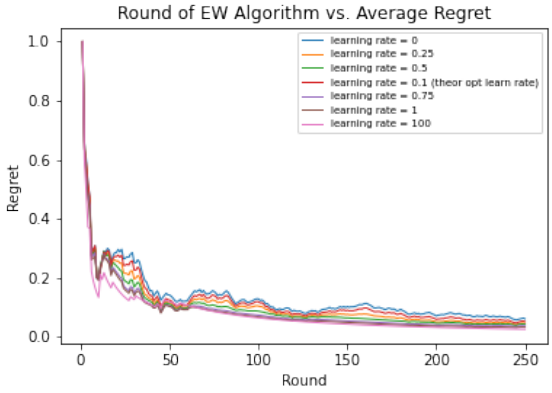

Figure 3: Exponential weights algorithm applied to **real life S&P stock data from 2017** using Monte Carlo trials, based off regret in hindsight

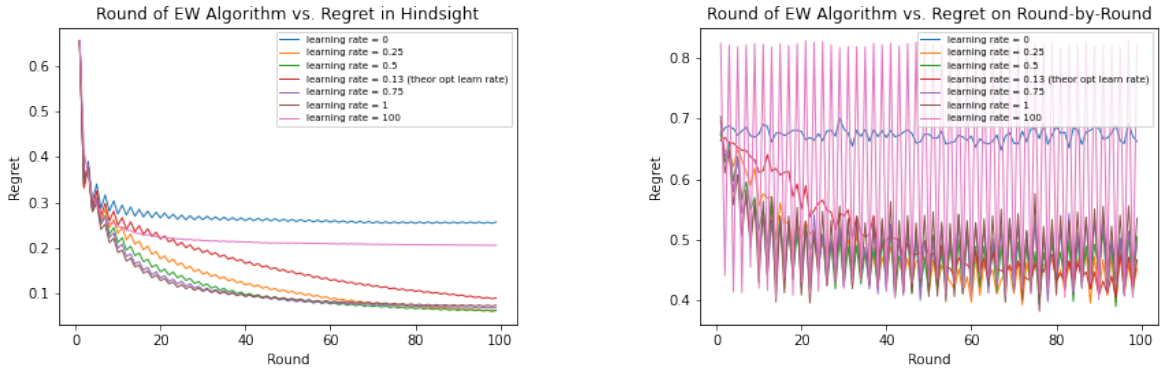

Figure 4: Exponential weights algorithm applied to **random alternating payoffs** (data generating model) using Monte Carlo trials

In the above plots, the average regret is plotted in line graphs for all 4 data sets as the simulation is ran through multiple rounds. The adversarial fair payoffs, Bernoulli payoffs, and random alternating payoffs are simulated through 100 rounds with 5 actions while the S&P stock data is simulated through 251 rounds with 10 actions. It is important to note that some data sets include 2 plots. The plot on the left represents average regret as it is calculated in hindsight based on the optimal action in respect to the final round. In contrast, the plot on the right represents average regret based on which actions was optimal on a round-by-round basis (*OPT* of each individual round i in n).

4 Conclusions

In part 1, after conducting an empirical study of the learning rates for the exponential weights algorithm on both the Bernoulli and adversarial fair payoffs data generating models, we concluded that exponential weights was not the best choice for either model on its own, but would be the best option if you did not know whether your data would be adversarial or not. On the adversarial fair payoffs data set, the lowest average regret was 0.0015, which was achieved when $\epsilon = 0$ which is essentially guessing randomly. As the learning rate was increased, the regret of the algorithm increased as well. This aligns with the generation method's adversarial nature: the higher the epsilon value, the less likely we are to guess the action with the lowest cumulative payoff and the less likely we are to guess the only action with any payoff in the next round. This result did not align with the original hypothesis that the exponential weights algorithm would outperform on all adversarial models. This is because different adversarial

models can target different flaws in follow-the-leader, exponential weights, and random guessing algorithms. In the case of the adversarial fair payoffs method, it targeted a flaw that is shared between follow-the-leader and exponential weights, but has a much worse worst-case outcome on follow-the-leader. So, random guessing was the most successful, followed by exponential weights, both with vanishing regret, while follow-the-leader did not have vanishing regret.

Regarding the Bernoulli payoffs data generation model, follow-the-leader performed best, followed by exponential weights, both accomplishing vanishing regret, while random guessing performed the worst, unable to produce vanishing regret. As the learning rate increased, the rate of regret reduction increased. Follow-the-leader acts as a faster, more decisive exponential weights algorithm, and because the Bernoulli payoffs generation method exhibited no adversarial behavior, follow-the-leader acted as a more efficient version of exponential weights. Therefore, if your goal is to simply accomplish vanishing regret, exponential weights is the best choice, while if you want to fit your algorithm to perform more efficiently on a certain dataset, follow-the-leader and random guessing may be better options their own respective use cases. This refuted our hypothesis, as random guessing outperformed the exponential weights algorithm on the adversarial fair payoffs data set.

In part 2, we first worked with a subset of S&P 500 data. The S&P 500 stock-price data presented an interesting perspective by considering real-world decisions. Considering that stock prices depend on the economic supply and demand of a company's shares, it was not surprising that the follow the leader algorithm performed the best with 0.024 average regret, supporting the original hypothesis. The reasoning behind this conclusion could be related to the fact that if a company performs well, the market price tends to increase over the long-term. Moreover, the 10 companies were selected alphabetically and not by any specific criteria. This could result in certain stocks standing out in overall stock price performance. Let us take Apple Inc. (AAPL), for example. The expected weights algorithm is dependent on the competition between the actions presented, and in 2017 Apple Inc. was high-performing with the launching of products such as iPhone X and iMac Pro. Therefore, it would make sense that "following the leader" (algorithm chooses the company stock that has consistently performed well in previous days) seems the most accurate. However, according to both the table of regret calculations and figure 4, the relative disparity between the average regret of different learning rates is relatively minimal.

Also in part 2, we ran a third Monte Carlo trial on an original adversarial generative case, random alternating payoffs, designed so that follow-the-leader and random guessing could not produce vanishing regret. Considering that in the preliminary we motivated why follow-the-leader and random guessing did not produce vanishing regret, it was not surprising that our trial results and plots indicated that as well. The exponential weights algorithm, however, produced vanishing regret. Because the two actions that were passing the single payoff back and forth had similar total cumulative payoffs, the exponential weights algorithm gave both those actions high weights without deterministically choosing one or the other and being misled, resulting in vanishing regret. This further supports our conclusion from part 1, that the exponential weights algorithm may act slower than random guessing or follow-the-leader on specific data sets but it is better equipped to handle unknown data or data with varied patterns that may or may not be adversarial. Overall, the results has led to the following motivated questions:

(1) Instead of assigning payoffs of 0 and 1 for the possible company stocks to invest in, would the results be different if the payoffs were proportional to the increase or decrease in price? Would negative payoffs affect the feasibility of the algorithm simulation?
(2) What if the player can choose multiple stocks to invest in at a time rather than just one, how would this change the results?
(3) What if the number of stocks to choose from (actions) increased? If there were 500 stocks to choose to invest in, Would this change the best algorithmic method?
(4) What if we selected company stocks that tend to be the top performers from previous years?
(5) How would exponential weights handle data where the best action would remain dominant for a significant period of rounds, then another action could arise as the best action to take while it still may not have the best cumulative payoff in hindsight? (i.e. a winstreak model, where only one action has a payoff each round and the action that won last round is 10x as likely to have the payoff as any other round) That pattern would be easy for a human to pick up on, but if exponential weights fell short, what other algorithms or modifications could better

handle that sort of behavior?

5 Appendix

Code for conducting relevant analysis and simulations were completed using the Jupyter Notebook platform and are attached on the following page.

Part 1

In [ ]:
```
import random
import numpy
import math
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

Generate Distribution Functions

In [ ]:
```
def sorted_indexes(payoffs):
    vals_indexes=[]
    ind_by_val = []

    for i in range(len(payoffs)):
        vals_indexes.append([payoffs[i],i])

    vals_indexes.sort(reverse=True)
    for x in vals_indexes:
        ind_by_val.append(x[1])
    return ind_by_val

def find_min_index(payoffs):
    min_value = min(payoffs)
    min_index = payoffs.index(min_value)
    return min_index


def generate_adversarial_payoffs(num_actions, num_rounds):
    rounds_list = []
    totals_by_round = []
    initial_payoff = round(random.random(), 2)
    first_payoffs = [0 for i in range(num_actions)]
    first_payoffs[random.randrange(num_actions)] = initial_payoff
    total_payoffs = [first_payoffs[i] for i in range(num_actions)]
    min_index = find_min_index(total_payoffs)
    rounds_list.append(first_payoffs)
    totals_by_round.append([total_payoffs[i] for i in range(num_actions)])

    for i in range(num_rounds - 1):
        new_payoff = round(random.random(), 2)
        adversarial_payoffs = [0 for i in range(num_actions)]
        adversarial_payoffs[min_index] = new_payoff
        for i in range(num_actions):
            total_payoffs[i] += adversarial_payoffs[i]
            total_payoffs[i] = round(total_payoffs[i], 2)

        min_index = find_min_index(total_payoffs)
        new_totals = [total_payoffs[i] for i in range(num_actions)]
        totals_by_round.append(new_totals)
        rounds_list.append(adversarial_payoffs)

    #print("utility at each round: \n", rounds_list)
    #print("totals by round: \n", totals_by_round)
    #print("final payoffs: \n", total_payoffs)
    return rounds_list, totals_by_round


#generate_adversarial_payoffs(10, 10)
```

In [ ]:
```
#when generating the bernoulli payoffs, generate the payoffs of each action at each round and the
#total payoffs up to that point for each action. i.e. list of lists of payoffs/round & list of lists of aggregated payoffs.
#uncomment the last line of the generate_adversarial_payoffs section for an example
def find_payoff(success_chance):
    comparison_val = random.random()
    return int(success_chance > comparison_val)

def generate_bernoulli_payoffs(num_actions, num_rounds):
    rounds_list = []
    totals_by_round = []
    total_payoffs = [0 for i in range(num_actions)]
    totals_by_round = []
    action_success_chances = [round(random.random() / 2,2) for i in range(num_actions)]

    for i in range(num_rounds):
        new_payoffs = [find_payoff(action_success_chances[j]) for j in range(num_actions)]

        for i in range(num_actions):
            total_payoffs[i] += new_payoffs[i]
            total_payoffs[i] = round(total_payoffs[i], 2)

        new_totals = [total_payoffs[i] for i in range(num_actions)]
        totals_by_round.append(new_totals)
        rounds_list.append(new_payoffs)

    #print("utility at each round: \n", rounds_list)
    #print("totals by round: \n", totals_by_round)
    #print("final payoffs: \n", total_payoffs)
    return rounds_list, totals_by_round

#generate_bernoulli_payoffs(3, 3)
```

In [ ]:
```
def rotate_action_payoffs(action_payoffs):
    copy_payoffs = [action_payoffs[i] for i in range(len(action_payoffs))]
    for i in range(0, len(action_payoffs)):
        action_payoffs[i] = copy_payoffs[i-1]
    return action_payoffs

def generate_rotational_random_payoffs(num_actions, num_rounds):
    rounds_list = []
    totals_by_round = []
    action_payoffs = [round(random.random(), 2) for i in range(num_actions)]
    max_index = action_payoffs.index(max(action_payoffs))
    secondary_max_index = max_index - 1
    max_payoff = action_payoffs[max_index]
    if max_index == 0:
        action_payoffs[-1] = 0
    else:
        action_payoffs[max_index - 1] = 0
    total_payoffs = [0 for i in range(num_actions)]
    action_payoffs = [0 for i in range(num_actions)]
    action_payoffs[max_index] = max_payoff

    for i in range(num_rounds):
        #if random.random() > 0.9:
        #action_payoffs = rotate_action_payoffs(action_payoffs)
        action_payoffs[max_index], action_payoffs[secondary_max_index] = action_payoffs[secondary_max_index], action_payoffs[max_index]

        for i in range(num_actions):
            total_payoffs[i] += action_payoffs[i]
            total_payoffs[i] = round(total_payoffs[i], 2)
        new_totals = [total_payoffs[i] for i in range(num_actions)]
        rounds_list.append([action_payoffs[i] for i in range(num_actions)])
        totals_by_round.append(new_totals)

    return rounds_list, totals_by_round
```

Simulate Algorithm Behavior Functions

In [ ]:
```
def simulate_exponential_weights(rounds_list, totals_by_round, epsilon, max_payoff):
    num_rounds = len(rounds_list)
    num_actions = len(rounds_list[0])
    choices_made = []
    action_weights = []
    action_probabilities = [(1/num_actions) for i in range(num_actions)]
    current_weights = [1 for i in range(num_actions)]
    action_weights.append(current_weights)
    alg_payoffs = []
    opt_payoffs = []

    for round in range(1, num_rounds):
        last_round = round - 1
        current_weights = [None for i in range(num_actions)]
        for action in range(num_actions):
            V_last = totals_by_round[last_round][action]
            exp = V_last / max_payoff
            current_weights[action] = pow(1 + epsilon, exp)
        #randomly select from actions using weights as probabilities
        selected_payoff = random.choices(rounds_list[round], weights=current_weights, k=1)[0]
        alg_payoffs.append(selected_payoff)
        opt_payoffs.append(max(rounds_list[round]))
        action_weights.append(current_weights)

    return alg_payoffs, totals_by_round, opt_payoffs
```

Regret Visual Analysis Function

In [ ]:
```
def visualize_regret(avg_regret_per_round, rounds, learning_rates, plot_title, file_name):

    add_str = ''
    for i in range(len(learning_rates)):
        if i == 3:
            each_lr = round(learning_rates[i], 2)
            add_str = '(theor opt learn rate)'
        else:
            each_lr = learning_rates[i]
            add_str = ''
        #print(each_lr)
        x = np.array(list(range(1, rounds)))
        y = np.array(avg_regret_per_round[learning_rates[i]])
        plt.plot(x, y, label='learning rate = {each_lr} {add_str}'.format(each_lr=each_lr, add_str = add_str), linewidth=1)
    plt.xlabel("Round")
    plt.ylabel("Regret")
    plt.title(plot_title)
    plt.legend(loc='best', prop={'size': 7})

    plt.savefig(file_name)

    plt.show()
```

Monte Carlo Trials

- Declare size of inputs

- Generate payoffs
- For each learning rate $\{\epsilon_1, \ldots, \epsilon_n\}$
  - For each input
    - Simulate the algorithm
    - calculate OPT (best in hindsight payoff)
    - calculate the algorithm's regret
  - Calculate the average regret for this learning rate ϵ

In [ ]:
```
def sum_to_round_i(alg_payoffs, current_round):
    total = 0
    for i in range(current_round):
        total += alg_payoffs[i]
    return total

def individual_regrets(alg_payoffs, round_totals):
    final_payoffs = round_totals[-1]
    opt_action = final_payoffs.index(max(final_payoffs))
    #print(opt_action)
    individual_regrets = [0 for i in range(len(alg_payoffs))]
    for round in range((len(alg_payoffs))):
        individual_regrets[round] = (round_totals[round][opt_action] - sum_to_round_i(alg_payoffs, round)) / (round + 1)
    return individual_regrets
```

In [ ]:
```
rounds = 100
actions = 5
N = 1000
# ADD OPTIMAL LEARNING RATE EPSILON
opt_lr_eps = math.sqrt(numpy.log(actions)/rounds)
learning_rates = [0, 0.25, 0.5, opt_lr_eps, 0.75, 1, 100]
```

In [ ]:
```
#adversarial monte carlo trial
max_payoff = 1
avg_lr_payoffs = dict()
all_opt_payoffs = []
avg_regret_per_round = dict()
avg_regret_per_round_1 = dict()
for n in range(N):
    adversarial_payoffs, adversarial_totals = generate_adversarial_payoffs(actions, rounds)
    for epsilon in learning_rates:
        adv_payoffs, adv_round_totals, adv_opt = simulate_exponential_weights(adversarial_payoffs, adversarial_totals, epsilon, max_payoff)
        adv_regrets = individual_regrets(adv_payoffs, adv_round_totals)
        adv_avg_regrets = sum(adv_regrets) / len(adv_regrets)
        adv_final_regret = adv_regrets[-1]
        adv_regrets_1 = []
        for i in range(len(adv_payoffs)):
            adv_regrets_1.append(adv_opt[i] - adv_payoffs[i])

        if epsilon not in avg_regret_per_round_1:
            avg_regret_per_round_1[epsilon] = adv_regrets_1
        else:
            for i in range(len(avg_regret_per_round_1[epsilon])):
                avg_regret_per_round_1[epsilon][i] = ((n * avg_regret_per_round_1[epsilon][i]) + adv_regrets_1[i]) / (n + 1)

        if epsilon not in avg_regret_per_round:
            avg_regret_per_round[epsilon] = adv_regrets
        else:
            for i in range(len(avg_regret_per_round[epsilon])):
                avg_regret_per_round[epsilon][i] = ((n * avg_regret_per_round[epsilon][i]) + adv_regrets[i]) / (n + 1)

        if epsilon not in avg_lr_payoffs:
            avg_lr_payoffs[epsilon] = [sum(adv_payoffs)]
        else:
            avg_lr_payoffs[epsilon].append(sum(adv_payoffs))
    all_opt_payoffs.append(max(adv_round_totals[-1]))
for key, val in avg_regret_per_round.items():
    print("Average ALG regret for epsilon =", key, "on adversarial distribution =", val[-1])

for key, val in avg_lr_payoffs.items():
    print("Average ALG payoff for epsilon =", key, "on adversarial distribution =", sum(val) / len(val))
print("Average OPT payoff for adversarial distribution =", sum(all_opt_payoffs) / len(all_opt_payoffs))
```

In [ ]:
```
visualize_regret(avg_regret_per_round, rounds, learning_rates,'Round of EW Algorithm vs. Regret in Hindsight','adv_plot_1.png')
visualize_regret(avg_regret_per_round_1, rounds, learning_rates, 'Round of EW Algorithm vs. Regret on Round-by-Round', 'adv_plot_2.png')
```

In [ ]:
```
#bernoulli monte carlo trial
max_payoff = 1
avg_lr_payoffs = dict()
all_opt_payoffs = []
avg_regret_per_round = dict()
avg_regret_per_round_1 = dict()

for n in range(N):
    bernoulli_payoffs, bernoulli_totals = generate_bernoulli_payoffs(actions, rounds)
    for epsilon in learning_rates:
        bern_payoffs, bern_round_totals, bern_opt = simulate_exponential_weights(bernoulli_payoffs, bernoulli_totals, epsilon, max_payoff)
        bern_regrets = individual_regrets(bern_payoffs, bern_round_totals)
        bern_avg_regrets = sum(bern_regrets) / len(bern_regrets)
        bern_final_regret = bern_regrets[-1]
        bern_regrets_1 = []
        for i in range(len(adv_payoffs)):
            bern_regrets_1.append(bern_opt[i] - bern_payoffs[i])
```

```
        if epsilon not in avg_regret_per_round_1:
            avg_regret_per_round_1[epsilon] = bern_regrets_1
        else:
            for i in range(len(avg_regret_per_round_1[epsilon])):
                avg_regret_per_round_1[epsilon][i] = ((n * avg_regret_per_round_1[epsilon][i]) + bern_regrets_1[i]) / (n + 1)

        if epsilon not in avg_regret_per_round:
            avg_regret_per_round[epsilon] = bern_regrets
        else:
            for i in range(len(avg_regret_per_round[epsilon])):
                avg_regret_per_round[epsilon][i] = ((n * avg_regret_per_round[epsilon][i]) + bern_regrets[i]) / (n + 1)

        if epsilon not in avg_lr_payoffs:
            avg_lr_payoffs[epsilon] = [sum(bern_payoffs)]
        else:
            avg_lr_payoffs[epsilon].append(sum(bern_payoffs))

    all_opt_payoffs.append(max(bern_round_totals[-1]))
for key, val in avg_regret_per_round.items():
    print("Average ALG regret for epsilon =", key, "on bernoulli distribution =", val[-1])
for key, val in avg_lr_payoffs.items():
    print("Average ALG payoff for epsilon =", key, "on bernoulli distribution =", sum(val) / len(val))
print("Average OPT payoff for bernoulli distribution =", sum(all_opt_payoffs) / len(all_opt_payoffs) )
```

In [ ]:
```
visualize_regret(avg_regret_per_round, rounds, learning_rates, 'Round of EW Algorithm vs. Regret in Hindsight', 'bern_plot_1.png')
visualize_regret(avg_regret_per_round_1, rounds, learning_rates, 'Round of EW Algorithm vs. Regret on Round-by-Round', 'bern_plot_2.png')
```

In [ ]:
```
# rotational generation monte carlo trial
generate_rotational_random_payoffs
max_payoff = 1
avg_lr_payoffs = dict()
all_opt_payoffs = []
avg_regret_per_round = dict()
for n in range(N):
    rotational_payoffs, rotational_totals = generate_rotational_random_payoffs(actions, rounds)
    for epsilon in learning_rates:
        rot_payoffs, rot_round_totals, rot_opt = simulate_exponential_weights(rotational_payoffs, rotational_totals, epsilon, max_payoff)
        rot_regrets = individual_regrets(rot_payoffs, rot_round_totals)
        rot_avg_regrets = sum(rot_regrets) / len(rot_regrets)
        rot_final_regret = rot_regrets[-1]
        if epsilon not in avg_regret_per_round:
            avg_regret_per_round[epsilon] = rot_regrets
        else:
            for i in range(len(avg_regret_per_round[epsilon])):
                avg_regret_per_round[epsilon][i] = ((n * avg_regret_per_round[epsilon][i]) + rot_regrets[i]) / (n + 1)

        rot_regrets_1 = []
        for i in range(len(adv_payoffs)):
            rot_regrets_1.append(rot_opt[i] - rot_payoffs[i])

        if epsilon not in avg_regret_per_round_1:
            avg_regret_per_round_1[epsilon] = rot_regrets_1
        else:
            for i in range(len(avg_regret_per_round_1[epsilon])):
                avg_regret_per_round_1[epsilon][i] = ((n * avg_regret_per_round_1[epsilon][i]) + rot_regrets_1[i]) / (n + 1)

        if epsilon not in avg_lr_payoffs:
            avg_lr_payoffs[epsilon] = [sum(rot_payoffs)]
        else:
            avg_lr_payoffs[epsilon].append(sum(rot_payoffs))

    all_opt_payoffs.append(max(rot_round_totals[-1]))
for key, val in avg_regret_per_round.items():
    print("Average ALG regret for epsilon =", key, "on rotational random distribution =", val[-1])
for key, val in avg_lr_payoffs.items():
    print("Average ALG payoff for epsilon =", key, "on rotational random distribution =", sum(val) / len(val))
print("Average OPT payoff for rotational random distribution =", sum(all_opt_payoffs) / len(all_opt_payoffs) )
```

In [ ]:
```
visualize_regret(avg_regret_per_round, rounds, learning_rates, 'Round of EW Algorithm vs. Regret in Hindsight', 'rot_plot_1.png')
visualize_regret(avg_regret_per_round_1, rounds, learning_rates, 'Round of EW Algorithm vs. Regret on Round-by-Round', 'rot_plot_2.png')
```

Part 2C. Data in the wild

Data Cleaning

In [ ]:
```
#Stock dataset taken from https://www.kaggle.com/datasets/camnugent/sandp500?select=all_stocks_5yr.csv

df = pd.read_csv('all_stocks_5yr.csv')

df = df.query('date.str.startswith("2017")',
engine="python")

# resetting index
df.reset_index(inplace = True)

df['Day']=df.groupby(['Name']).cumcount()+1

#Add payoffs of each stock each day as column

df['payoff'] = np.where(df['close'] / df['open'] >= 1, 1, 0)
```

```
#get the number of total stocks
df.drop_duplicates(subset = ["Name"]).shape[0]

#get the number of total stocks
df.drop_duplicates(subset = ["Name"]).shape[0]

#Max payoff value

df['payoff'].max()

#query stocks by date or day
df[df['Day'] == 1]

df

#identify stocks that do not have full year stock prices data

temp = df[df['Day'] == 1]
temp = temp['Name'].tolist()

temp_2 = df[df['Day'] == 251]
temp_2 = temp_2['Name'].tolist()

print('Stocks that do not have data for all dates throughout the year: ', ','.join(set(temp).difference(temp_2)))

#drop outlier stocks with not enough data

df = df.drop(df[df.Name == 'BHF'].index)
df = df.drop(df[df.Name == 'DXC'].index)
df = df.drop(df[df.Name == 'HLT'].index)
df = df.drop(df[df.Name == 'APTV'].index)
df = df.drop(df[df.Name == 'DWDP'].index)
df = df.drop(df[df.Name == 'BHGE'].index)

df
```

In [ ]:

```
#identify range of all stocks starting with A

range_ds = df

# get the unique values (rows)
range_ds = range_ds.drop_duplicates(subset = ["Name"])

#First 11 stocks starting with A
range_ds = range_ds.head(11)

range_ds

#first stock
range_ds.head(1)['Name']

#last stock
range_ds.iloc[-1]['Name']

#index of last day of 10th stock

index = range_ds.index[-1]

index

#slice first 10 stocks

df = df[:index]

df

df.drop_duplicates(subset = ["Name"])
```

Applying EW algorithm to stock data with payoffs

In [ ]:

```
def generate_stock_payoffs(dataset, num_actions):
    rounds_list = []
    totals_by_round = []
    total_payoffs = [0 for i in range(num_actions)]
    day = 1
    list_of_dates = []
    while day <= 251:
        temp_ds = dataset[dataset['Day'] == day]
        #print(len(temp_ds['payoff'].tolist()))
        if len(temp_ds['payoff'].tolist()) != num_actions:
            day += 1
            #print(day)
            continue
        else:
            list_of_dates.append(temp_ds['date'].iloc[0])
            action_payoffs = temp_ds['payoff'].tolist()
            #print(len(action_payoffs),'action payoffs')
            #print(len(total_payoffs),'total_payoffs')
            #print(day)
            for j in range(num_actions):
                total_payoffs[j] += action_payoffs[j]
            new_totals = [total_payoffs[i] for i in range(num_actions)]
            rounds_list.append([action_payoffs[i] for i in range(num_actions)])
            totals_by_round.append(new_totals)
            day += 1
```

```
    return rounds_list, totals_by_round, list_of_dates

#generate_stock_payoffs(df, 10)
```

In [ ]:

```
actions = 10
N = 1000
rounds = 251

#optimal learning rate epsilon ε = √(ln k / n)
opt_lr_eps = math.sqrt(numpy.log(actions)/rounds)

learning_rates = [0, 0.25, 0.5, opt_lr_eps, 0.75, 1, 100]

max_payoff = 1 #max payoff of all stocks
#max_payoff = 11.678064176749078
avg_lr_payoffs = dict()
all_opt_payoffs = []
avg_regret_per_round = dict()
avg_regret_per_round_1 = dict()
list_of_dates = []
for n in range(N):
    stock_payoffs, stock_totals, date_list = generate_stock_payoffs(df, actions)
    for epsilon in learning_rates:
        sto_payoffs, sto_round_totals, sto_opt_payoffs = simulate_exponential_weights(stock_payoffs, stock_totals, epsilon, max_payoff)
        sto_regrets = individual_regrets(sto_payoffs, sto_round_totals)
        sto_avg_regrets = sum(sto_regrets) / len(sto_regrets)
        sto_final_regret = sto_regrets[-1]
        if epsilon not in avg_regret_per_round:
            avg_regret_per_round[epsilon] = sto_regrets
        else:
            for i in range(len(avg_regret_per_round[epsilon])):
                avg_regret_per_round[epsilon][i] = ((n * avg_regret_per_round[epsilon][i]) + sto_regrets[i]) / (n + 1)

        sto_regrets_1 = []
        for i in range(len(sto_payoffs)):
            sto_regrets_1.append(sto_opt_payoffs[i] - sto_payoffs[i])

        if epsilon not in avg_regret_per_round_1:
            avg_regret_per_round_1[epsilon] = sto_regrets_1
        else:
            for i in range(len(avg_regret_per_round_1[epsilon])):
                avg_regret_per_round_1[epsilon][i] = ((n * avg_regret_per_round_1[epsilon][i]) + sto_regrets_1[i]) / (n + 1)

        if epsilon not in avg_lr_payoffs:
            avg_lr_payoffs[epsilon] = [sum(sto_payoffs)]
        else:
            avg_lr_payoffs[epsilon].append(sum(sto_payoffs))
    list_of_dates = date_list

all_opt_payoffs.append(max(sto_round_totals[-1]))
for key, val in avg_regret_per_round.items():
    print("Average ALG regret for epsilon =", key, "on stock data =", val[-1])
for key, val in avg_lr_payoffs.items():
    print("Average ALG payoff for epsilon =", key, "on stock data =", sum(val) / len(val))
print("Average OPT payoff for on stock data =", sum(all_opt_payoffs) / len(all_opt_payoffs) )
print("The list of dates where you are investing are the following...",  list_of_dates)
print("The number of days that you are investing is ",  len(list_of_dates))
```

In [ ]:

```
visualize_regret(avg_regret_per_round, rounds, learning_rates, 'Round of EW Algorithm vs. Average Regret', 'stock_plot.png')
visualize_regret(avg_regret_per_round_1, rounds, learning_rates, 'Round of EW Algorithm vs. Average Regret', 'stock_plot.png')
```