Online Learning

January 27, 2023

1 Introduction

Exponential Weights is a strategy used to minimize regret by utilizing previous payoffs and scaling those values into probabilities through exponents. This project reports how we investigated various learning rates (epsilon) across different datasets to draw conclusions as to what learning rates should be used when approaching new datasets. Overall we found that various strategies may falter when occurring different datasets, such as those that include patterns, and that exponential weights if given a proper epsilon will produce vanishing regret performance (performance similar to best in hindsight over many rounds).

2 Preliminaries

2.1 Adversarial Fair Payoffs

Adversarial Fair is a scenario where the action with the least payoff prior to the round is assigned a new uniformly generated payoff. In order to construct the payoff matrix for this scenario, we first generated random values uniformly from 0 to 1 into a 1D array through numpy.random.rand($round_{len}$). These represent the payoffs for the round corresponding to the index that each payoff lives in. Since we do not know which corresponding action and index each payoff should be assigned to at the moment of creation, we elected to keep the payoffs within a one dimensional array.

At round i, $round_i$ payoff is going into the smallest total payoff action prior to the round. In order to keep track of what action is the smallest total payoff action, we chose to utilize Python's heapq to heapify an array of the minimum total payoff action. When we needed the smallest total payoff at each round, we heapq.pop() which results in minimum value of the heap and its corresponding int index that represents its action. We then added the new payoff to its total payoff and heapq.push() it back which will determine if its still the minimum or not.

For example, the min heap might look like this for k=10:

$$\text{if TotalPayoffs}(round_i) = [[0,2],[0,3],[0,5],[0,7],[0,4],[0.763,0],[0,6],[0,9],[0,8],[0.792,1]]$$

At each round, this would produce an array of weights which could be plugged into the exponential weights formula to produce probabilities of each action.

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

Furthermore, we can utilize the second element of the popped value to find the j index of where the payoff is assigned to for that round. As such, we can create the payoff array for that round and append it to the matrix to form the payoff matrix for Adversarial Fair Payoffs.

2.2 Bernoulli Payoffs

We wanted to compare the differences between various utilizations and datasets of exponential weights, so we conducted experiments on a Bernoulli Payoff matrix as well. Bernoulli is the probability whether an action would receive a payoff of 1, or the case of it not happening, where it receives a payoff of 0.

To set up a Bernoulli Payoff Matrix, we simply ran the same numpy.random.rand() but with the input of $round_{len}$ and $action_{len}$. Then by dividing the array by 2 via numpy.divide(), the payoff matrix is already completed.

We treated each payoff that lived in i, j as its payoff because its a Bernoulli distribution meaning that its each values expected payoff is:

$$E[\text{Payoff}]_j^i = p * 1 + (1 - p) * 0$$

$$E[\text{Payoff}]_j^i = p$$

As such, we could sum the previous rounds' arrays together to form the current array for the weights. Once again, we can then input that weight array for round i into the exponential formula to determine the probabilities for that round.

2.3 Adversarial Generative Model

Since the Adversarial Fair and Bernoulli Payoff were random, we wanted to introduce a dataset that utilized some patterns to test exponential weight's performance. As such, we created a dataset that alternated increasing values across only a few actions.

The way this data was generated was through assigning increasing payoffs ($1/round_{len}$ to ensure an $h = 1$) to only half the actions of $action_{len}$ while keeping all other actions at 0 payoff for that round. The purpose of this was to change the leader every single round since assigning increasing payoffs will set the $i - 1$ rounds best action as the new leader. Given that the $i - 1$ action will never occur on i, this guarantees that the follow the leader strategy will have not have vanishing regret.

Furthermore, since we only utilize half the actions, this means that for uniform guesses, half the time the expected payoff is 0. This further guarantees that the regret will not be zero, but will be around 1/2.

An example of the produced data for $n = 4$ $k = 5$:

$$\text{arr} = [[0, 0, 0, 0.25, 0], [0, 0, 0, 0, 0.5], [0, 0, 0.75, 0, 0], [0, 0, 0, 1, 0]]$$

The first two actions are not used, and the payoff increases with each round until they reach 1.

2.4 Real World Dataset on Stocks

We wanted to apply exponential weights onto a real dataset, so we chose to utilize stock data. We obtained the dataset from Kaggle[Nug18], which contains the historical dataset of S&P stock markets in the past 5 years. We treated each day as a new round, ignoring after-hour and premarket trading, meaning that changes between two days were not accounted for. Only changes within the day were calculated as payoff with the following equation:

$$\text{Payoff} = [endOfDayPrice]/[beginningOfDayPrice]$$

Within this dataset, we had 1259 days of data for 500 companies. We tested exact and emperical estimates of best epsilons as well as tested for vanishing regret for the following strategies: follow the leader, theoretical best epsilon, and random guessing.

In this project specifically, we will be testing on 10 stocks chosen randomly from 1000+ stock pool, calculating its payoffs and draw conclusions on whether following the leader or empirical epsilons leads to a higher payoff and lower regrets.

2.5 Experimentation

We were curious how well exponential weights would perform over different random conditions and across different epsilons. In order to make this comparison, we first implemented the optimal algorithm, which in this case is the best in hindsight algorithm, to compare with.

The best in hindsight algorithm finds the maximum payoff produced by choosing the same action every time. This refers to the hindsight part of the name, as it would be impossible to know which action would have netted the maximum payoff compared to the other options.

To implement this, we summed up the payoff matrix on the column axis to form 1xk array of total payoffs corresponding to action. We then then took argmax of that array to determine the best in hindsight action.

Algorithm 1 Best in Hindsight

```
arr = numpy.sum(matrix, axis=0)
best_action = numpy.argmax(arr)
best_payoff = numpy.max(arr)
```

With the optimal, OPT, values. We can then calculate how far our exponential weights expected payoff was off by subtracting the two. The resulting value is called Regret, which when divided by the round length, n, gives the per round Regret:

$$\text{Regret}_n = \frac{1}{n}(OPT - ALG)$$

This allows us to compare future changes in datasets, epsilons, and methodologies with a set metric.

One experiment we tried was to empirically test what epsilon would perform the best. In order to do this, we used the generated Adversarial Fair and Bernoulli payoff matrices and weights to test various epsilons that have a tiny step between each. We compared the best performing epsilon out of these with the theoretical best epsilon which was found using this proof:

$$\text{Given theorem stating:E[EW]} \geq (1-\epsilon)OPT - \frac{h}{\epsilon}\log k$$

$$\text{E[EW]} \geq OPT - OPT * \epsilon - \frac{h}{\epsilon}\log k$$

$$OPT <= hn$$

$$\text{E[EW]} \geq OPT - hn\epsilon - \frac{h}{\epsilon}\log k$$

To maximize this: take the derivative in regards to ϵ and find extrema

$$\text{E[EW]} = OPT - hn\epsilon - \frac{h}{\epsilon}\log k$$

$$\frac{d}{d\epsilon}\text{E[EW]} = \frac{d}{d\epsilon}(OPT - hn\epsilon - \frac{h}{\epsilon}\log k)$$

$$\frac{d}{d\epsilon}\text{E[EW]} = -hn - (-\frac{h}{\epsilon^2}\log k$$

$$\frac{d}{d\epsilon}\text{E[EW]} = -hn + \frac{h}{\epsilon^2}\log k$$

$$0 = -hn + \frac{h}{\epsilon^2}\log k$$

$$hn = \frac{h}{\epsilon^2}\log k$$

$$hn(\epsilon^2) = h \log k$$
$$\epsilon^2 = \frac{\log k}{n}$$
$$\epsilon = \sqrt{\frac{\log k}{n}}$$

Another experiment we conducted was to compare various algorithms to exponential weights algorithm. We compared the following algorithms: uniform guessing, follow the leader, and linear weights. Uniform guessing and follow the leader consists of setting the epsilon to 0 and ∞ respectively, while linear weights is the following adaptation:

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

By simply removing the power element and replacing it with a multiplication, we introduce linearity which slightly differs from the prior uniform guessing. The logic behind this change was that for smaller weights, multiplication would net larger numbers. This means that this algorithm would be a bit more biased towards repeating values then the exponential weights. Another interesting thing is the deterministic behavior of this algorithm before encountering a payoff at action j. This is because the w_j^i would be 0 before encountering, and would then multiply the numerator to 0. It is only after encountering all actions with a payoff at least once that this algorithm no longer is deterministic.

3 Results

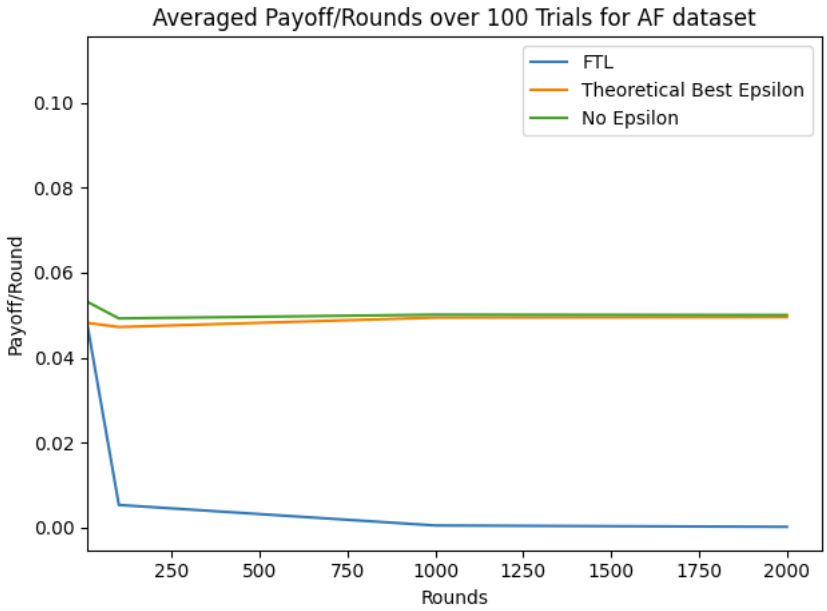
Given the theoretical epsilon formula, we obtained the result of trial counts to action counts in the following.

Table 1: Theoretical Epsilon for each Trial total(n), Action count(k)

(row = n, col = k)	k = 5	k = 10	k = 100	k = 500
n = 10	0.4011780044361979	0.47985259121880813	0.6786140424415112	0.7883278568224132
n = 100	0.12686362411795196	0.15174271293851463	0.21459660262893474	0.2492911570517934
n = 1000	0.04011780044361979	0.04798525912188081	0.06786140424415112	0.07883278568224132
n = 4000	0.020058900221809894	0.023992629560940407	0.03393070212207556	0.03941639284112066

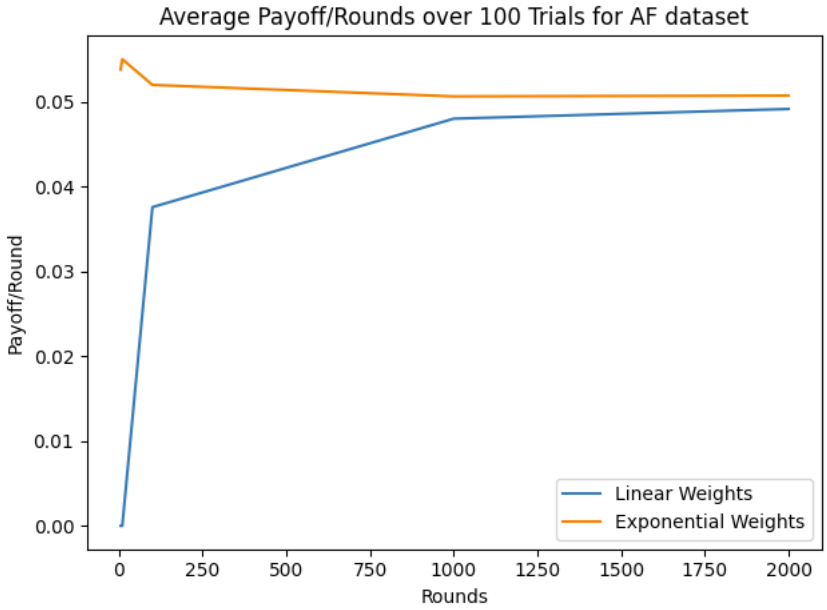
3.1 Experiments on Adversarial Fair

We tested various epsilon values for the adversarial fair methodology. These include $\epsilon = 0$ which results in uniform random actions and $\epsilon = \infty$ which creates a follow-the-leader algorithm.



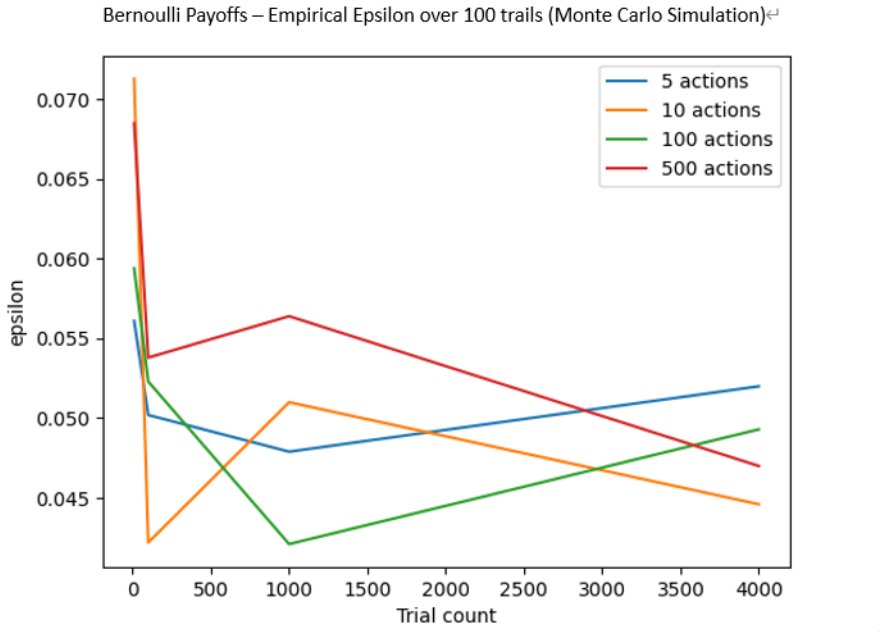
These are the tested expected payoff for various epsilon values. The y axis is the average payoff divided by the round count times h over 100 trials. Since $h = 1$ within this case, it is just the payoff divided by the round count. We believe a round length of 2000 was sufficient to determine if some strategies had vanishing regret.

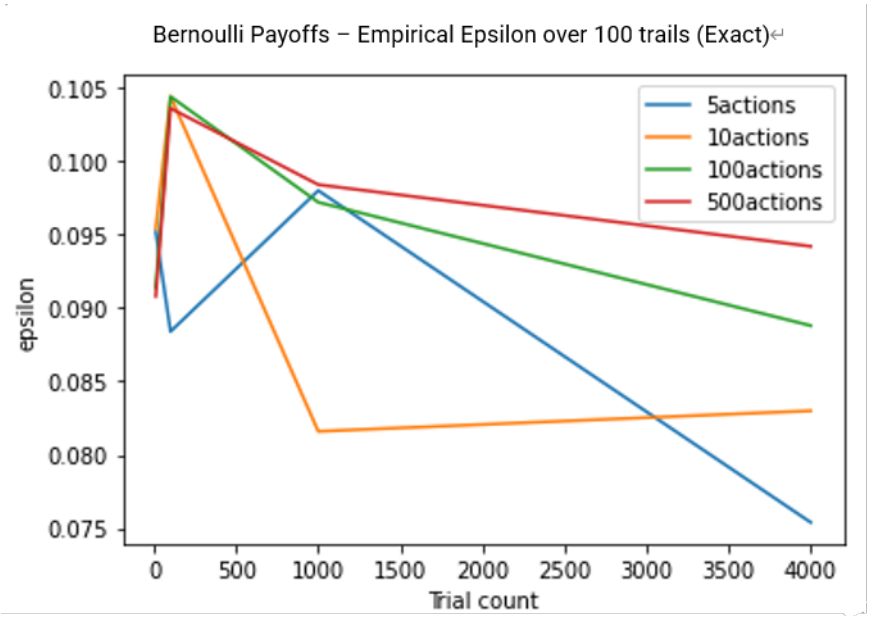
We additionally tested to see if linear weights could perform similarly to exponential weights. The following is our results.



3.2 Various Epsilon Values on Bernoulli Payoffs

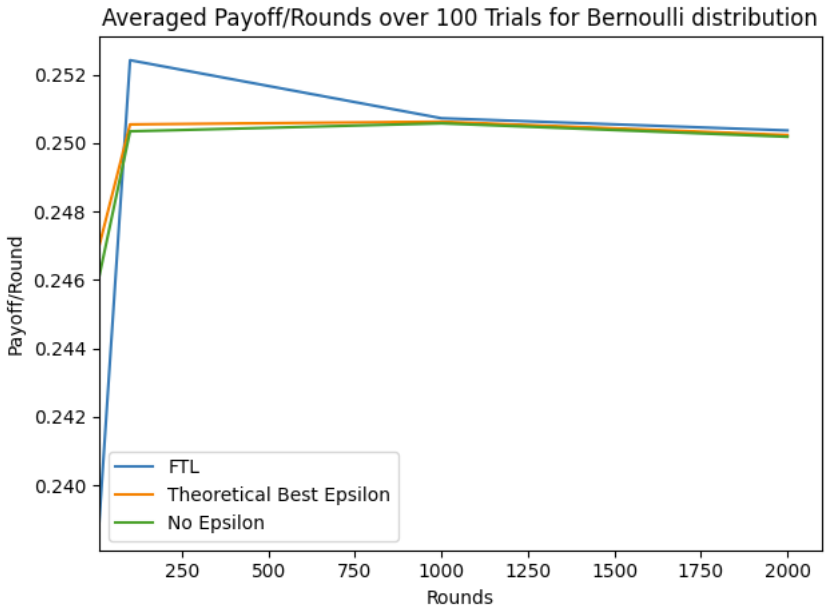
Furthermore, we tested the epsilon values under Monte Carlo simulation for the Bernoulli Payoffs. In order to average the epsilon values of total trail counts (denoted as n) = [10, 100, 1000, 4000] respective to total action count (denoted as k) = [5, 10, 100, 500].





In order to determine an algorithm that accounts for the size of the data sample to ensure no fluctuation and random results, the calculation was tested by an epsilon tested across 100 trials.

Additionally, we tested the same expected payoff as Adversarial Fair for the strategies previously mentioned: FTL, Theoretical Best Epsilon, and No Epsilon.

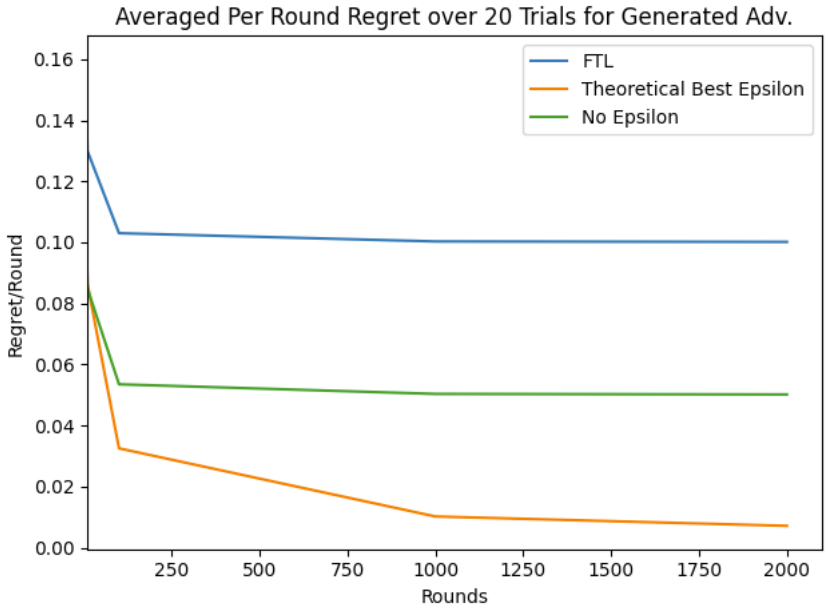


We also ran the linear weights vs. exponential weights, and found no noticeable different trends than the Adversarial Fair's comparison.

3.3 Experiments on Adversarial Generative Model

We ran the same calculations of expected payoff and empirically tested best epsilon values onto the Adversarial Generative dataset. Unless stated otherwise, these graphs all have an action length of 10.

This graph represents the per round regret between OPT and various strategies as labeled for 10 actions:



Furthermore we tested Monte Carlo Simulations and exact calculations for best epsilons. Although there were some slight differences between epsilons between the exact and Monte Carlo Simulations, the expected payoff was about the same. This follow graph depicts the Monte Carlo Simulation em-

pirically tested epsilon values compared to theoretical best at k=10:

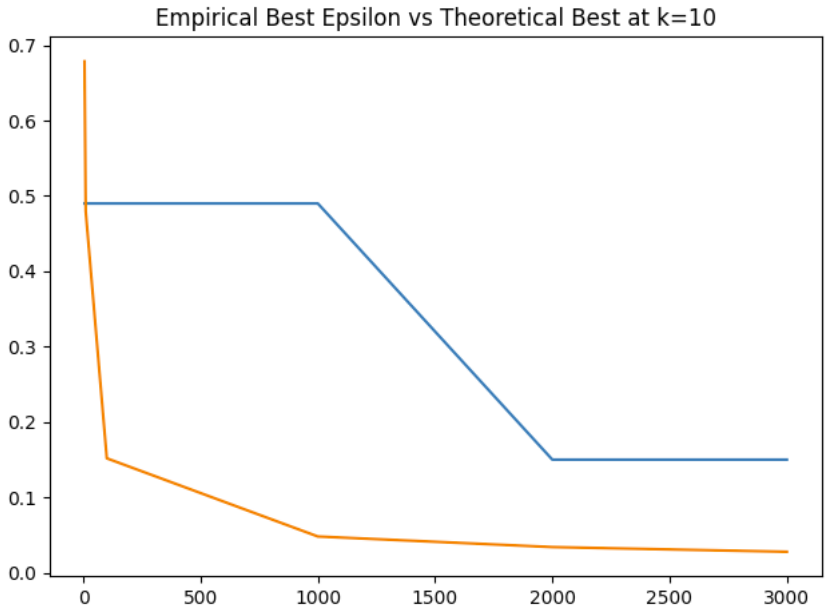
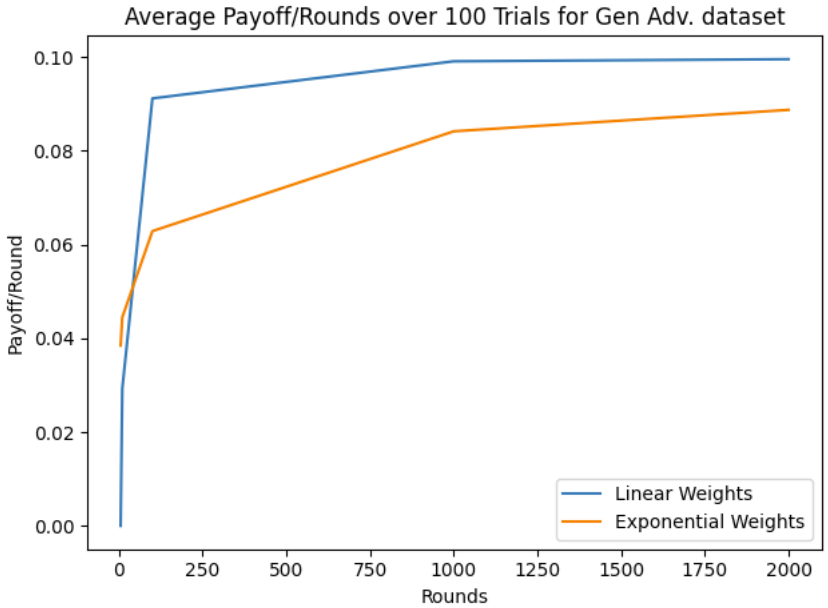


Table 2: Payoffs Based on Epsilon Values of the Above Graph

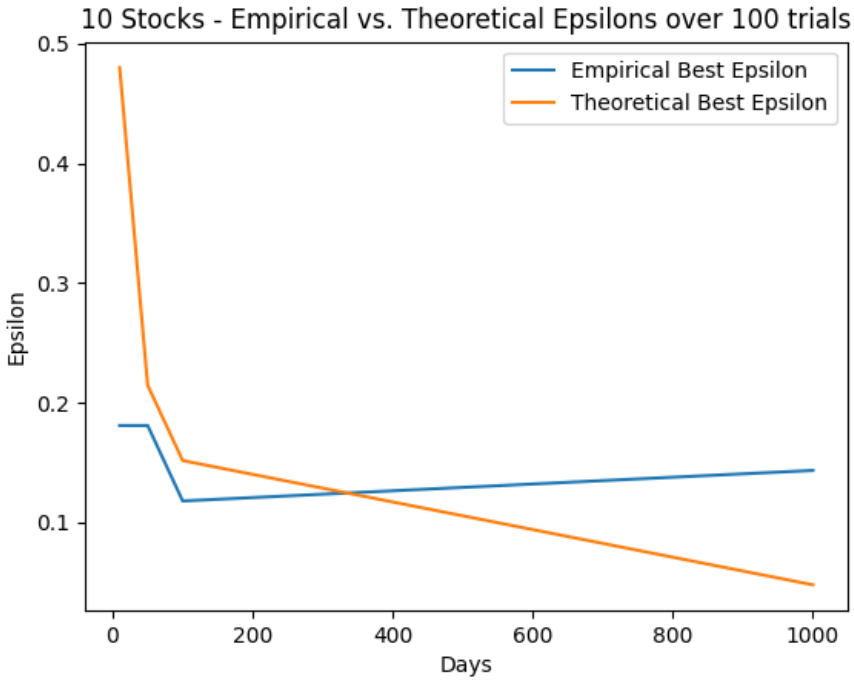
(Epsilons, Rounds)	5	10	100	1000	2000	3000
Optimal	0.8	1.3	10.3	100.3	200.3	300.3
Empirical Best	0.167	0.48	10.09	98.8	192.50	283.51
Theoretical Best	0.19	0.44	6.29	84.16	177.47	272.44

We also ran a the linear weights comparison to the exponential weights and found interesting results:

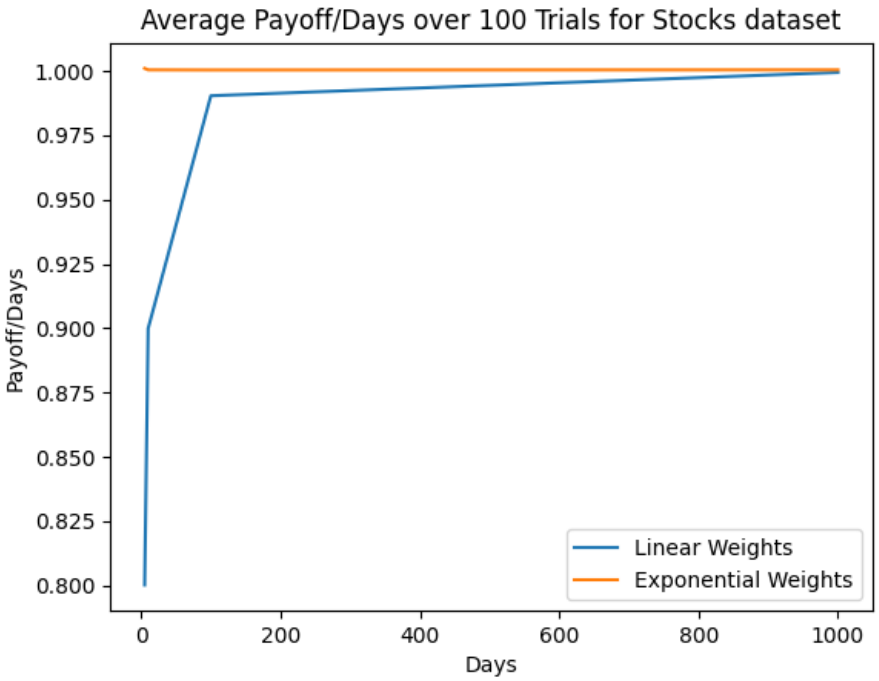


3.4 Result of real dataset in stocks

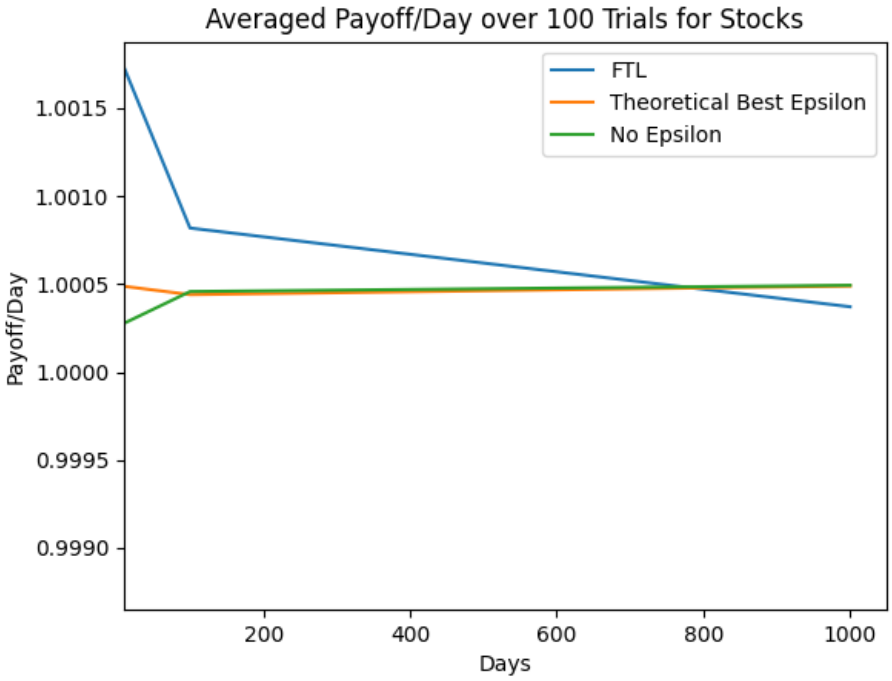
On the other hand, we also tested the exponential change through the data of stocks, we compared the results of our monte carlo simulated epsilon results with the theoretical epsilons per stock numbers of [10, 20, 50, 100] over 1259 days.



We saw a trend of increasing k (number of stocks) leading to a smaller epsilon, which is as expected as the increase of k is rather similar to the real world of stocks. Furthermore, we also ran the comparison of the linear weight to the exponential weights and found a similar result.



This graph represents the per-day payoff between OPT and strategies as labeled for 10 stocks:



As the graph suggests, in the long term, following the leader might not be a good strategy when it comes to investment since it's continuously declining over the days. As for the Regret = 1/n [OPT - ALG], the resulting graph would be the following:

4 Conclusions

In our project, we compared various epsilon values across different generations of data and real-world data. We analyzed how different epsilons could create different strategies, and furthermore, we tried to empirically maximize the expected payoff by changing epsilon values and methodologies.

4.1 Adversarial Fair

Adversarial Fair is an interesting generation of data that assigns a payoff to the action with the lowest payoff as of that round. Even though this result is in a uniform distribution payoff across all actions at the end, this does discourage certain online learning strategies such as following the leader.

Given that values are uniformly generated (and thus will be generally equal across all actions) and added to the lowest payoff as of that round, it is unlikely that an action that recently received a payoff will receive another one in the next round. Knowing that following the leader will attempt to repeat the highest payoff action prior to that round, this means that following the leader will continually predict a wrong action given that the payoff will be assigned to the least payoff while following the leader will predict the highest. Thus, despite the uniform nature of the data, the following the leader online strategy has poor performance without modifications such as a regularizer.

4.2 Bernoulli Distribution

Bernoulli and Adversarial Fair both have the innate property of producing uniform distributions over many trials. However, as previously mentioned, the manner of generation as a stream is important for online learning algorithms that can generalize the trend as they receive data.

Whereas Adversarial Fair had a set method of producing payoffs, Bernoulli is more standard to true randomness for payoffs. As such, we saw that doing any guess could result in about the same expected payoff. This means that for any epsilon within $[0, \infty)$, a generally decent payoff could be expected. This is proven by how the best-performing epsilon across various amounts of actions and rounds remained overall unpredictable with very slight changes in the expected payoff. In fact, even just guessing one value could net the expected payoff close to the best in hindsight. We call this strategy best in foresight. However, the best algorithm to generate the highest expected value was the uniform random draw.

Uniform random draws occur when epsilon is 0. This means that the chance of drawing an action depends solely on the payoff that it has produced prior to that round without any learning. Since we can expect the payoffs to be uniformly distributed, every action has about the same chance to be drawn. This proved to be the best-performing algorithm by a slight margin likely due to the uniform nature of payoffs and guesses aligning.

4.3 Adversarial Generative Model

As expected, the performance for the following the leader strategy and uniform guessing performed poorly for this dataset. Since this dataset was made specifically to increase regret for those datasets, it was clear that exponential weights would outperform those two strategies.

Specifically, the per-round regret for the exponential weights could be seen as approaching 0 for larger Ns. In other words, exponential weights have vanishing regret while following the leader and uniform guessing maintains their regret at a stable rate.

It was also interesting to view the comparison between the best empirically tested epsilon and the theoretical best epsilon. There is the apparent trend of epsilon decreasing with N, however the difference between epsilon is noticeable as well as the produced payoffs. In fact, the empirical best epsilon occasionally outperformed the theoretical best epsilon albeit by a small margin.

Furthermore, it was interesting to note that the empirical best epsilon produced via Monte Carlo and via exact calculations differed where the exact calculations had slightly lower epsilons, yet produced similar payoffs.

4.4 Real World Data on Stocks

Real World Data mirrored our Bernoulli Distribution in that all three strategies proved to be viable and to have vanishing regret. Furthermore for this case where the range of the data was not extremely large (less than 1), we saw that the regret for all three strategies essentially reach 0 at even lower amounts of rounds.

This might hint that during day, even if the stock had a lot of movement (with lower lows and higher highs), overall the stock's "true" value does not change much. Additionally, when taking a look at overall changes in afterhours and premarket, there are noticeable changes within the previous days close and the open of a new day. This might suggest that news that happen afterhours could greatly affect the price of the stock on a new day -take for example quarterly reports.

In general, we noticed that the expected value of simple day-trading is about the dollar that was put in, meaning that new significant profit were to be made if only a dollar was spent on the open and sold at the close over many days. This did not imply the stock didn't increase over time likely previously mentioned, but implies that it is difficult to predict the true value of a stock within a small timeslice.

4.5 Comparison of Linear and Exponential Weights

Across all datasets, we saw comparable results for linear and exponential weights in terms of expected payoff over larger number of rounds. However for lower number of rounds such as rounds consisting only of 10 rounds, the expected payoff was lower than expected. This makes sense given that actions without any values are given no probability of occurring.

However, if you add a regularizer to linear weights or even just a probability for previously unoccurred actions, the linear weights increases its performance for small Ns to around the same performance as exponential weights. Despite this, linear weights across all trials, still performed worse than exponential weights even if it was a diminishing difference across larger Ns.

This might suggest that for larger Ns across the datasets we used, the distributions average out to around uniform where decisions could be made a bit more haphazardly, or possibly that linear weights could generalize some patterns with better or equal success as exponential weights. Specifically for the Adversarial Generative dataset, the linear model actually outperformed the exponential dataset interestingly enough. This might be because the manner of payoff was similarly linearly scaled.

5 Appendix

[Nug18] Cam Nugent. *SP 500 stock data - Historical stock data for all current SP 500 companies.* 2018. URL: https://www.kaggle.com/datasets/camnugent/sandp500?resource=download.

Code for project:

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



def calculateTheoreticalEpsilon(n, k):
    return np.sqrt(np.log(k)/n)

def calculateEmpiricalEpsilonMonte(arr, trials, h, consecutive=1):
    if len(arr) > 1000:
```

```
        epsilons = np.arange(0, 0.3, 0.05)
    else:
        epsilons = np.arange(0, 0.5, 0.01)
    best_epsilon = -1
    best_payoff = -1
    random_trials = []
    for i in range(trials):
        cur_trial = random.randint(0, len(arr) - 1)
        end = cur_trial + consecutive
        while cur_trial < end:
            random_trials.append(arr[cur_trial])
            cur_trial += 1

    for epsilon in epsilons:
        weights = [[0 for i in range(len(arr[0]))]]
        weights_idx = 1
        for i in range(len(random_trials)): #number of monte carlo trials
            payoffs = random_trials[i]
            payoffs = np.add(payoffs, weights[weights_idx-1])
            weights.append(payoffs)
            weights_idx += 1

        cur_payoff = calculateExpectedPayoff(random_trials, weights, epsilon, h)

        if cur_payoff > best_payoff:
            best_payoff = cur_payoff
            best_epsilon = epsilon

    return best_epsilon, best_payoff

def calculateEmpiricalEpsilonExact(arr, weights, h, type="exp"):
    if len(arr) > 1000:
        epsilons = np.arange(0, 0.3, 0.05)
    else:
        epsilons = np.arange(0, 0.5, 0.01)
    best_payoff = -1
    best_epsilon = -1
    for epsilon in epsilons:
        payoff = calculateExpectedPayoff(arr, weights, epsilon, h, type)
        if payoff > best_payoff:
            best_payoff = payoff
            best_epsilon = epsilon

    return best_epsilon, best_payoff

def generateAdversarial(round_len, action_len):
    type_a = []
    cur = action_len
    i = 0
    while cur > 0:
        copy = 0
        while copy < cur:
            new = [0 for i in range(action_len)]
            new[i] = 1
            type_a.append(new)
            copy += 1
```

```
            i += 1
            cur -= 1

        type_b = []
        cur = action_len
        i = action_len - 1
        while cur > 0:
            copy = 0
            while copy < cur:
                new = [0 for i in range(action_len)]
                new[i] = 1
                type_b.append(new)
                copy += 1
            i -= 1
            cur -= 1

        arr = []
        i = 0
        a = True
        while len(arr) < round_len:
            if a:
                if i == len(type_a):
                    i = 0
                    a = False
                    continue
                arr.append(type_a[i])
            else:
                if i == len(type_b):
                    i = 0
                    a = True
                    continue
                arr.append(type_b[i])

            i+=1
        return arr

    def generateAdversarial2(round_len, action_len):

        type_a = []
        val = 1/round_len
        count = -1
        i = int(action_len/2)
        while len(type_a) < round_len:

            if i == action_len:
                i = int(action_len/2)
                continue
            count += 1
            new = [0 for j in range(action_len)]
            new[i] = val * count
            i += 1
            type_a.append(new)
        return type_a


    def calculateWeights(arr):
```

```
    weights = [[0 for i in range(len(arr[0]))]]
    weights_idx = 1
    for i in range(len(arr)):
        payoffs = arr[i]
        payoffs = np.add(payoffs, weights[weights_idx - 1])
        weights.append(payoffs)
        weights_idx += 1
    return weights

def calculateExpectedPayoff(arr, weights, epsilon, h, type="exp"):
#calculates by previously produced weights
    payoff = 0
    i = 0
    avg = 0
    while i < len(arr):
        if type == "exp":
            cur_weight = exponentialWeights(weights[i], epsilon, h)
        elif type == "linear":
            cur_weight = linearWeights(weights[i], epsilon, h)
        cur_arr = arr[i]
        payoff = np.multiply(cur_weight, cur_arr)
        i+=1
        avg += np.sum(payoff)

    return avg

def followTheLeader(arr):
    weights = [[0 for i in range(len(arr[0]))]]
    expected_payoff = 0
    args = []
    for idx, row in enumerate(arr):
        maxarg = np.argmax(weights[idx])
        args.append(maxarg)
        expected_payoff += row[maxarg]
        weights.append(np.add(weights[idx], row))

    return expected_payoff, args

def exponentialWeights(arr, epsilon, h):
    out = []
    sum = 0
    for ele in arr:
        numerator = np.power((1+epsilon), ele/h)
        sum += numerator
        out.append(numerator)

    return np.divide(out, sum)


def linearWeights(arr, epsilon, h): #for comparison with linear weights
    sum = 0
    out = []
    for ele in arr:
        numerator = ((1+epsilon) * ele /h)
        sum += numerator
        out.append(numerator)
```

```
    if sum==0:
        return out
    return np.divide(out,np.sum(out))

def generateLuckyStreak(round_len, action_len):
    last_chance = 0
    last_index = -1
    lucky_matrix = []
    used = []
    for i in range(round_len):
        arr = [0 for i in range(action_len)]
        chance = random.random()
        if chance < last_chance:

            last_chance -= 0.1
            arr[last_index] = 1
        else:
            last_index = random.randint(0, action_len-1)
            while last_index in used:
                last_index = random.randint(0, action_len - 1)
            used.append(last_index)
            if len(used) == action_len:
                used = []
            last_chance = 0.8
            arr[last_index] = 1
        lucky_matrix.append(arr)

    return lucky_matrix

def generateStrictLuckyStreak(round_len, action_len):
    lucky_matrix = []
    n = action_len
    n_copy = action_len
    sign = -1
    count = 0
    for i in range(round_len):
        arr = [0 for i in range(action_len)]
        if n_copy == 0:
            n += sign * 1
            if n == 0 and sign == -1:
                n = 1
                sign = sign * -1
                count += 1
            n_copy = n
        arr[n-1] = 1
        n_copy += sign * 1
        lucky_matrix.append(arr)
    return lucky_matrix

def optimal(arr):
    best_payoff = -1
    best_action = -1
    for k in range(len(arr[0])):
        cur_payoff = 0
        for n in range(len(arr)):
            cur_payoff += arr[n][k]
```

```
            if cur_payoff > best_payoff:
                best_payoff = cur_payoff
                best_action = k

    return best_action, best_payoff
round_len = 1000 #n
action_len = 10 #k

def generateAdversarialFair(round_len, action_len):
    actions_total_payoff = [[0, i] for i in range(action_len)]
    actions_total_payoff_stable = [0 for i in range(action_len)]


    heapq.heapify(actions_total_payoff)


    random_payoffs = np.random.rand(round_len, 1).squeeze()
    # print(random_payoffs)

    round_payoff_matrix = np.zeros((round_len, action_len))
    weight_matrix = []
    exp_weight_matrix = []
    weight_matrix.append(np.copy(actions_total_payoff_stable))
    for i in range(round_payoff_matrix.shape[0]):
        new_payoff = random_payoffs[i]
        min_payoff = heapq.heappop(actions_total_payoff)
        idx = min_payoff[1]
        actions_total_payoff_stable[idx] += new_payoff
        min_payoff[0] += new_payoff
        heapq.heappush(actions_total_payoff, min_payoff)
        # print(actions_total_payoff)
        exp_weight_matrix.append(exponentialWeights(actions_total_payoff_stable,0.05,1))
        weight_matrix.append(np.copy(actions_total_payoff_stable))
        round_payoff_matrix[i][idx] = new_payoff

    return round_payoff_matrix, weight_matrix

#calculates the theoretical epsilon of our lucky streak matrix
total = 0
monte_total = 0
for i in range(20):
    lucky_streak = generateLuckyStreak(round_len, action_len)
    weights = calculateWeights(lucky_streak)
    monte_total += calculateEmpiricalEpsilonMonte(lucky_streak, 100, 1, 1)[0]
    total += calculateEmpiricalEpsilonExact(lucky_streak, weights, 1)

print(calculateTheoreticalEpsilon(round_len, action_len))
print(total/20)
print(monte_total/20)

#function comparing linear and exponential weights
#plotting the graphs afterwards
def compareLinearExp(data="fair"): #fair, bernoulli, lucky
    rounds = [5, 10, 100, 1000, 2000]
    trials = 100
```

```
    linear = []
    exp = []
    for round_len in rounds:
        linear_avg = 0
        exp_avg = 0
        for i in range(trials):
            arr = []
            weights = []
            h = -1
            if data == "fair":
                arr, weights = generateAdversarialFair(round_len, action_len)
                h = 1
            elif data == "bernoulli":
                arr = np.random.rand(round_len, action_len)
                arr = np.divide(arr, 2)
                weights = calculateWeights(arr)
                h = 0.5
            elif data == "lucky":
                arr = generateAdversarial2(round_len, action_len)
                weights = calculateWeights(arr)
                h = 1

            theoretical_epsilon = calculateTheoreticalEpsilon(round_len, action_len)
            linear_avg += calculateExpectedPayoff(arr, weights, theoretical_epsilon, h,
            exp_avg += calculateExpectedPayoff(arr, weights, theoretical_epsilon, h, "ex

        linear.append(np.divide(linear_avg, trials))
        exp.append(np.divide(exp_avg, trials))

    plt.plot(rounds, np.divide(linear, rounds), label = "Linear Weights")
    plt.plot(rounds, np.divide(exp, rounds), label = "Exponential Weights")
    plt.title(f"Average Payoff/Rounds over 100 Trials for Gen Adv. dataset")
    plt.ylabel("Payoff/Round")
    plt.xlabel("Rounds")
    plt.legend()
    print(linear, exp)
    plt.show()

# lucky_streak = generateStrictLuckyStreak(round_len, action_len)
# print(lucky_streak)
# weights = calculateWeights(lucky_streak)
# print(lucky_streak)
# print(weights)
# print(calculateExpectedPayoff(lucky_streak, weights, 1, 1))
# trials = 100
# print(optimal(lucky_streak))
# print(f"Theoretical Epsilon: {calculateTheoreticalEpsilon(round_len, action_len)}")
# epsilons = np.arange(0.01, 0.2, 0.005)
# best_epsilon = -1
# best_payoff = -1
#
# for epsilon in epsilons:
#     weights = calculateWeights(lucky_streak)
#     payoff = calculateExpectedPayoff(lucky_streak, weights, epsilon, 1)
#     if payoff > best_payoff:
#         best_payoff = payoff
```

```
#         best_epsilon = epsilon
#
# print(best_epsilon, best_payoff)
# print(optimal(lucky_streak))
# print(followTheLeader(lucky_streak)[0])
# print(f"Empirical Epsilon with {trials} trials: {calculateEmpiricalEpsilonExact(lucky_

#calculating the average payoff per rounds for Bernoulli distribution
#plotting it compared in three ALG ftl, no epsilon, and theoretical_best
rounds = [5, 10, 100, 1000, 2000]
trials = 20
ftl= []
theoretical_best = []
random = []
for round_len in rounds:
    ftl_avg = 0
    tb_avg = 0
    random_avg = 0
    tb = calculateTheoreticalEpsilon(round_len, action_len)
    for i in range(trials):
        arr = np.random.rand(round_len, action_len)
        arr = np.divide(arr, 2)
        weights = calculateWeights(arr)
        ftl_avg += followTheLeader(arr)[0]
        tb_avg += calculateExpectedPayoff(arr, weights, tb, 0.5)
        random_avg += calculateExpectedPayoff(arr, weights, 0, 0.5)
    ftl.append(ftl_avg/trials)
    theoretical_best.append(tb_avg/trials)
    random.append(random_avg/trials)

plt.plot(rounds, np.divide(ftl, rounds), label="FTL")
plt.plot(rounds, np.divide(theoretical_best, rounds), label = "Theoretical Best Epsilon"
plt.plot(rounds, np.divide(random, rounds), label = "No Epsilon")
plt.title("Averaged Payoff/Rounds over 100 Trials for Bernoulli distribution")
plt.ylabel("Payoff/Round")
plt.xlabel("Rounds")
plt.xlim(xmin = 10)
plt.legend()
plt.show()


#calculating the regret for three ALGS in Adversarial Generative Model
#plotting the regret for comparison
rounds = [5, 10, 100, 1000, 2000]
trials = 20
ftl= []
theoretical_best = []
random = []
opt = []
for round_len in rounds:
    ftl_avg = 0
    tb_avg = 0
    random_avg = 0
    opt_avg = 0
    tb = calculateTheoreticalEpsilon(round_len, action_len)
    for i in range(trials):
```

```
            arr = generateAdversarial2(round_len, action_len)
            weights = calculateWeights(arr)
            ftl_avg += followTheLeader(arr)[0]
            tb_avg += calculateExpectedPayoff(arr, weights, tb, 0.5)
            random_avg += calculateExpectedPayoff(arr, weights, 0, 0.5)
            opt_avg += optimal(arr)[1]
        opt.append(opt_avg/trials)
        ftl.append(ftl_avg/trials)
        theoretical_best.append(tb_avg/trials)
        random.append(random_avg/trials)

opt = np.divide(opt, rounds)
ftl = np.divide(ftl, rounds)
theoretical_best = np.divide(theoretical_best, rounds)
random = np.divide(random, rounds)

plt.plot(rounds, opt-ftl, label="FTL")
plt.plot(rounds, opt-theoretical_best, label = "Theoretical Best Epsilon")
plt.plot(rounds, opt-random, label = "No Epsilon")
plt.title("Averaged Per Round Regret over 20 Trials for Generated Adv.")
plt.ylabel("Regret/Round")
plt.xlabel("Rounds")
plt.xlim(xmin = 10)
plt.legend()
plt.show()

#Comparisons between Empirical Best Epsilon vs Theoretical Best for Adversial Gen.
actions = [3, 5, 10, 20]
tb = []
eb = []
eb_p = []
tb_p = []
opt = []
trials = 10
for action_len in actions:
    e = 0
    tb_e = calculateTheoreticalEpsilon(round_len, action_len)
    tb.append(tb_e)
    for i in range(trials):
        arr = generateAdversarial2(round_len, action_len)
        weights = calculateWeights(arr)
        coupled = calculateEmpiricalEpsilonMonte(arr, round_len, 1)
        e += coupled[0]
        eb_p.append(coupled[1])
        tb_p.append(calculateExpectedPayoff(arr, weights, tb_e, 1))
        opt.append(optimal(arr)[1])
    eb.append(e/trials)

print(opt)
print(eb_p)
print(tb_p)
plt.plot(actions, eb)
plt.plot(actions, tb)
plt.title("Empirical Best Epsilon vs Theoretical Best at k=10")
plt.show()
```

```
#stocks, real dataset
data = []
with open('all_stocks_5yr.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        data.append(row)

data.pop(0)

k = 0
stock_name_arr = []
row = []
stock_amount = 10

def generateRealData(data, round_len, action_len):
    new_data = np.zeros((round_len, 10))
    for j in range(action_len): #number of companies
        for i in range(round_len): #rounds
            new_data[i][j] = float(data[i+(j*1259)][4]) / float(data[i+(j*1259)][1])
    max = np.max(new_data)
    min = np.min(new_data)

    return new_data, max-min

#Empirical vs. Theoretical Epsilons over 100 trials
days = [10, 50, 100, 1000]
empirical_epsilon_monte = []
theoretical_epsilon = []
trials = 20
for day_len in days:
    # print(stock_name_arr)
    new_data,h  = generateRealData(data, day_len, 10)
    new_data_weights = calculateWeights(new_data)
    # print("calculateEmpiricalEpsilonExact = ",calculateEmpiricalEpsilonExact(new_data,
    e = 0
    for i in range(trials):
        e+=calculateEmpiricalEpsilonMonte(new_data, day_len, h, consecutive=1)[0]/20
    empirical_epsilon_monte.append(e)
    theoretical_epsilon.append(calculateTheoreticalEpsilon(day_len, 10))


pyplot.title("10 Stocks - Empirical vs. Theoretical Epsilons over 100 trials")
pyplot.xlabel("Days")
pyplot.ylabel("Epsilon")
pyplot.plot(days, empirical_epsilon_monte, label = "Empirical Best Epsilon")
pyplot.plot(days, theoretical_epsilon ,label = "Theoretical Best Epsilon")
pyplot.legend()
pyplot.show()


#linear and exponential comparison
days = [5, 10, 100, 1000]
trials = 100
linear = []
exp = []
```

```
for days_len in days:
    linear_avg = 0
    exp_avg = 0
    new_data,h  = generateRealData(data, days_len, 10)
    new_data_weights = calculateWeights(new_data)
    for i in range(trials):
        theoretical_epsilon = calculateTheoreticalEpsilon(new_data.shape[0], new_data.sh
        linear_avg += calculateExpectedPayoff(new_data, new_data_weights, theoretical_eps
        exp_avg += calculateExpectedPayoff(new_data, new_data_weights, theoretical_epsilo
    linear.append(np.divide(linear_avg, trials))
    exp.append(np.divide(exp_avg, trials))

pyplot.plot(days, np.divide(linear, days), label = "Linear Weights")
pyplot.plot(days, np.divide(exp, days), label = "Exponential Weights")
pyplot.title(f"Average Payoff/Days over 100 Trials for Stocks dataset")
pyplot.ylabel("Payoff/Days")
pyplot.xlabel("Days")
pyplot.legend()
print(linear, exp)
pyplot.show()


#Averaged Payoff/Day over 100 Trials for Stocks
rounds = [5, 10, 100, 1000]
trials = 20
ftl= []
theoretical_best = []
random = []
for round_len in rounds:
    ftl_avg = 0
    tb_avg = 0
    random_avg = 0
    tb = calculateTheoreticalEpsilon(round_len, 10)
    for i in range(trials):
        new_data,h  = generateRealData(data, round_len, 10)
        new_data_weights = calculateWeights(new_data)
        ftl_avg += followTheLeader(new_data)[0]
        tb_avg += calculateExpectedPayoff(new_data, new_data_weights, tb, h)
        random_avg += calculateExpectedPayoff(new_data, new_data_weights, 0, h)
    ftl.append(ftl_avg/trials)
    theoretical_best.append(tb_avg/trials)
    random.append(random_avg/trials)

pyplot.plot(rounds, np.divide(ftl, rounds), label="FTL")
pyplot.plot(rounds, np.divide(theoretical_best, rounds), label = "Theoretical Best Epsilo
pyplot.plot(rounds, np.divide(random, rounds), label = "No Epsilon")
pyplot.title("Averaged Payoff/Day over 100 Trials for Stocks")
pyplot.ylabel("Payoff/Day")
pyplot.xlabel("Days")
pyplot.xlim(xmin = 10)
pyplot.legend()
pyplot.show()
```