Online Markets Project 2

Online Learning

April 2022

1 Introduction

For this project we implemented exponential weights (EW) algorithm. We then conducted an empirical study of learning rates for the EW algorithm using Monte Carlo trials on three data generating models. In addition, we created an extended version of EW, dubbed the Exponentially Weighted Voting algorithm (EWV), and evaluated its performance on daily Ethereum data over the past year. In particular, we evaluated its ability to adapt to market fluctuations.

2 Preliminaries

In this study we utilized three data generation models and a data set we pulled from Coinbase's publicly available Markets API. Links to the documentation of the API and data set can be found in the Appendix.

Before we get into the details of our models and set, we will describe our base implementation of an online learning algorithm. The exponential weights algorithm is an algorithm which chooses an action from a set of actions based on certain probabilities. As the algorithm runs, it adjusts those probabilities based on the following three parameters. The first is a list of payoffs for each possible action, which was generated by the models to be described below. The second is a learning rate, which determines how fast/slow the algorithm learns, with higher learning rates heavily weighting actions that have been observed to be successful so far, and lower learning rates doing the opposite. The third is the maximum possible expected payoff, and it makes the algorithm learn slower the higher payoffs are in general. The algorithm takes these parameters and performs the following steps for each round of actions:

1. Choose an action based on the probabilities for each
2. Take that action
3. Obtain payoff from action taken and update payoffs
4. Given learning rate, update probabilities of the actions, giving each weight $= (1+\epsilon)^{V_j^{i-1}/h}$ where $V_j^i = \sum_{r=1}^{i} v_j^i$

For the three generative models, we sought to empirically compare certain learning rates. Specifically, we wanted to pay particularly close attention to the following rates:

1. $\epsilon = 0$ (equivalent to random guessing)
2. $\epsilon = \sqrt{\frac{\ln(k)}{n}}$ (theoretically optimal learning rate
3. $\epsilon = \infty$ (equivalent to Follow-The-Leader (FTL) algorithm)

To do this, we conducted Monte Carlo Trials, which go as follows: Given a learning algorithm, a distribution of inputs for that algorithm, and a set of learning rates:

1. Draw N inputs at random from the distribution of inputs
2. For each learning rate $\epsilon \in \{\epsilon_1, ..., \epsilon_d\}$ and each input $1, ..., N$
   2.1. Simulate the algorithm
   2.2. Calculate OPT
   2.3. Calculate the algorithm's regret
3. Calculate the average regret for each learning rate
4. Plot the average regret as a function of learning rate

In general, we found that our average regrets did not change significantly after $N = 100$, so this is the value we used.
In addition, we decided our data would be more accurate if we simulated $\epsilon = \infty$ by instead running the FTL algorithm. In order to make our graphs readable, we decided to construct our list of learning rates as follows:

1. Calculate theoretical learning rate
2. Set max learning rate to 2 times this value
3. Add learning rates $\frac{i* \text{ max learning rate}}{20}$ for $i \in [0, 20]$
4. Add learning rate equal to three time theoretical learning rate (algorithm knows to conduct FTL when it recieves this value)

We did this because it will include our learning rates of interest and allow us to see how the average regret changes with linear increases of the learning rate.
We will now introduce the three generative models we implemented.

Adversarial Fair Payoffs Model To generate payoffs according to this model, for each round i:

1. Draw a payoff x $U[0, 1]$
2. Assign this payoff to the action $j^* = argmin_j V_j^{i-1}$ where $V_j^i = \sum_{r=1}^{i} v_j^i$

3. Assign all other actions a payoff of 0

Bernoulli Payoffs Model To generate payoffs according to this model, first fix a probability in $[0, \frac{1}{2}]$ for each action $p_1, ..p_k$.
Then, for each round i:

1. Draw the payoff of each action j as v_j^i $B(p_j)$ where $B(p_j)$ is the Bernoulli distribution according to probability p_j

Increasing Cumulative Payoff Model To generate payoffs according to this model, for each round i:

1. Draw a payoff x $U[0, i]$
2. Assign to action $i \mod k$ the last payoff of action i plus i (set last payoff to zero if first round)
3. Leave all other payoffs unchanged (setting them to zero if first round)

This model was designed in the hopes that neither uniform guessing nor FTL would have vanishing regret.
In addition to these models, we wanted to test an online learning algorithm on a set of daily open and close data for Ethereum over the past year. Cryptocurrency prices are inherently volatile, and initial experiments proved that our standard exponential weights algorithm had trouble adapting to rapidly shifting market conditions. Accordingly, we developed an expansion of our EW algorithm that we predicted would more readily be able to adapt to rapidly shifting market conditions. We will refer to this algorithm as the Exponentially Weighted Voting Algorithm (EWV).

Exponentially Weighted Voting Algorithm Before we formally describe this algorithm, we will give some context to complement the soon-to-follow definition. The idea of this algorithm is that it is a composition of multiple standard EW algorithms running in parallel, but created in different rounds. Specifically, this algorithm will create one new instance of an EW algorithm each round and allow it to learn from scratch. In any round, each algorithm will pick an action according to what it's learned so far. They will then each "vote" on their pick by increasing the probability the overall algorithm picks their choice. The catch is that each algorithms vote will be weighted, in an exponentially increasing fashion, according to how recently they were created: the latest algorithm will have the largest weight. Keep in mind that the latest algorithm will also have no context and will be picking an action uniformly in the round it was created. This will allow the algorithm a reasonable ability to select a so far unprofitable action (we want this given cryptocurrency's propensity to bubbles). In addition, while the second most recent algorithm has less weight than the latest, it still has exponentially more than the ones before it. This will allow the algorithm to more easily adapt to sudden shifts in the market. In addition, it is important to note that, traditionally, quantitative trading algorithms rely on context to

predict market moves. Our algorithm will not have any context outside of the payoffs it observes. Accordingly, we do not expect this algorithm to be profitable, but we do expect it quickly adapt to market fluctuations.
Given this context, we can formally define an EWV algorithm as one that, for each round i:

1. Create a new EW algorithm
2. Set its probability vector to a uniform distribution
3. Set its birth round to i
4. Have each algorithm pick according to their current probability vectors and weight their choices according to $(birth\_round + 1)^2$
5. Normalize the distribution and pick an action, noting the received payoffs for each action and adding to the algorithms payoff the payoff of its choice
6. Set the learning rates of each algorithm according to the payoffs each has observed while setting $\epsilon = \sqrt{\frac{\ln(k)}{i - birth\_round + 1}}$ (theoretical learning rate according to how old the algorithm is.

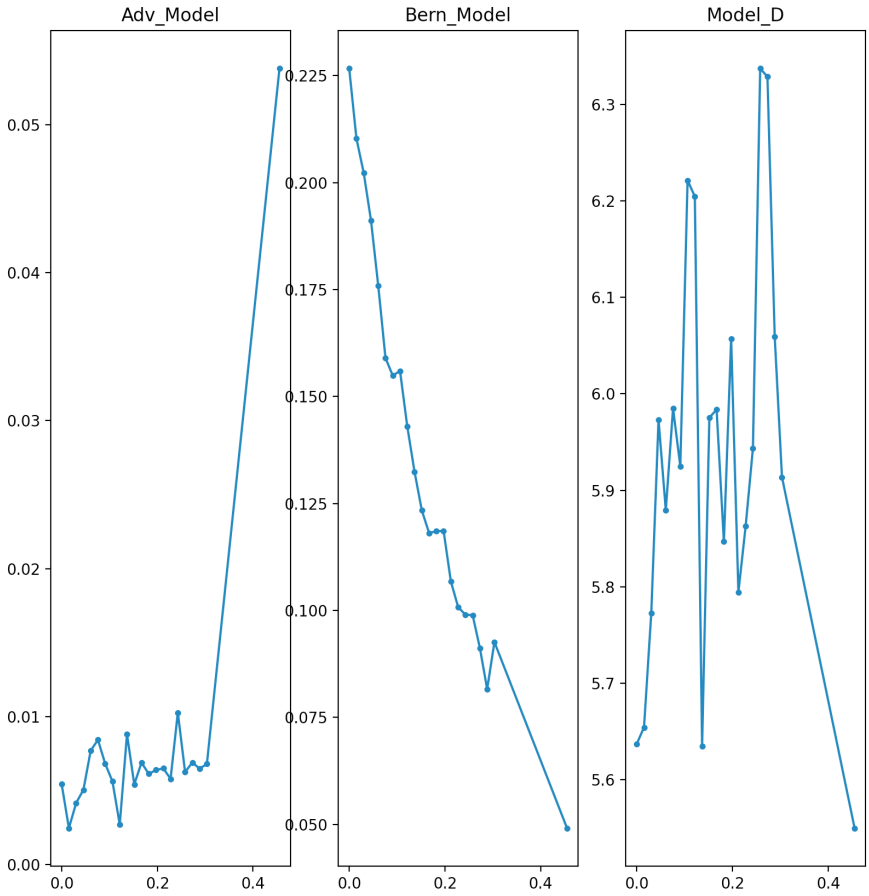
Note that we could have decided to keep the learning rates the same for each algorithm, but we decided that letting n be smaller for newer algorithms will increase their learning rate and allow the EWV algorithm to adapt faster.
In addition, because we do not vary the way the learning rate is determined across trials, we needed to determine a different mechanism to evaluate this algorithms success. To do this, we ran the algorithm N times, $(N = 100)$ and we averaged the algorithms average regret at each round. That is, we plotted the average of the algorithms average regret for each round. Formally, this means that the value plotted for round i will be

$$\frac{1}{100} \sum_{0}^{100} \left(\frac{\sum_{0}^{i} OPT_i(j) - ALG_i(j)}{i + 1} \right)$$

This measure will allow us to get a good understanding of how the algorithm adapts to market fluctuations.

3 Results

For the three generative models, we set $k = 10$ and $n = 100$ arbitrarily. This means our theoretically optimal learning rate is $\epsilon = 0.1$. Then, following the procedure outlined in the previous section, plotted the algorithm's final average regret as a function of learning rate. Here are the three plots we generated:



Let's begin with the Adversarial Model. Clearly, our EW algorithm was not very successful with an arbitrarily large learning rate. This is to be expected. The adversarial model works by assigning the payoff it produces to the least profitable action so far, whereas FTL is programmed to pick the most successful action so far, so it will nearly always get a payoff of zero each round. In fact, because EW is designed to favor actions that were more favorable in the past, it should be the case that the optimal learning rate in this case is zero, and this is the case. We can see that the theoretically optimal learning rate's average regret is only slightly larger than the algorithm's average regret when $\epsilon = 0$, and this follows considering the theoretically optimal learning rate is fairly small given our values for k and n.

Now we examine the Bernoulli Model. It is clear that our EW's average regret on this model follows an inverted relationship compared to its average regret on the Adversarial Model. This makes sense because in the long run, the action with the largest payoff so far is likely to be the action with the highest likelihood of having a payoff. Thus, favoring actions that have previously been successful is the best strategy in this case. This clearly follows given given the algorithms

success with an arbitrarily large learning rate. In addition, it makes sense that our algorithm's theoretically optimal learning rate beat a learning rate of zero by a significant margin.

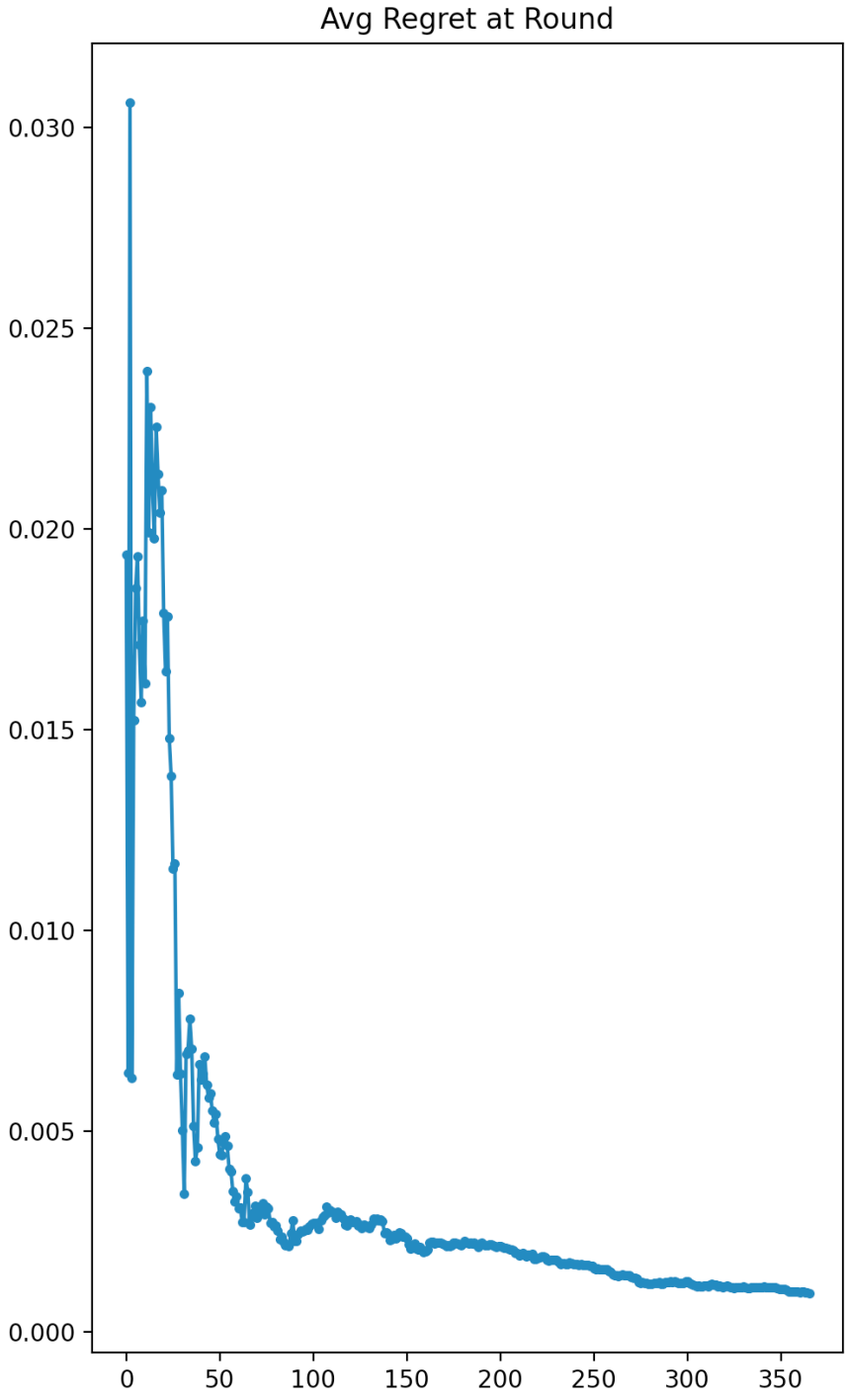
We now examine the Increasing Cumulative Payoff Model (Model_D). This model was designed so that neither uniform guessing ($\epsilon = 0$) nor FTL ($\epsilon = \infty$) would have final average regret near zero. As we can see, this model achieves this as all observed final average regrets were above 5.5. Interestingly, this minimum value is achieved with FTL. Furthermore, note that a learning rate of zero was relatively successful as well, while our theoretical learning rate resulted in an average regret of about 5.92. To see why this happened, let's break this model down a bit.

In the first i rounds, with $i < k$, we can see that the first i actions will have equal total payoffs (except for the first which will have zero payoff), while all other actions will have zero payoff. In round $k + 1$, action 0's payoff will be highest, but it's total payoff will be relatively low. In fact, this will always be the case. In every round, the highest payoff next round will be an action with a relatively lower total payoff so far. Accordingly, it makes sense that all learning rates were unsuccessful; the trend EW was trying to learn is based on the round, whereas EW is designed to determine trends according to actions.

Now we evaluate our ETH model. First, we determined that the three actions available each round are

1. invest a fixed amount x: payoff equal to percent change over the day plus one
2. do nothing: payoff equal to one
3. short a fixed amount x: payoff equal to percent change over the day

Setting up the payoffs this way allows us to expect the payoffs to be in the range $[0, 2]$. This ensures payoffs will not be negative, as this proved to complicate our algorithm. Accordingly, we can think of a payoff of 1 as realistically being a payoff of 0. Thus, $k = 3$, and we evaluated this data on a year long set of daily data, so $n = 366$ (we included April 22 2021 and April 22 2022). We then ran EWV according to the proceduce outlined in the previous section. Here is the plot we generated:



For comparison, here is a graph of Ethereum's price over the same time period.

Note that our payoffs were measured in percent change, so the graph above is not directly transferable to our situation, but it is a decent enough proxy.
Recall that we are seeking to analyze EWV's ability to adapt to market fluctuations. First note that it was very successful (which we think is pretty cool), in terms of average regret, at the end of 366 rounds. In addition, note that the degree to which the average regret oscilatted decreased exponentially as time went on, to the point where the algorithm was able to fairly seamlessly adapt to market changes. In fact, we concluded that our algorithm's average regret approaches about 0.1 percent in the long run, and while this is not exactly zero, it's pretty good. In general, we can fairly reasonably conclude that our algorithm was able to adapt to market fluctuations well.

4 Conclusions

There are many ways in which we could've improved on this empirically study. In the future, we would like to more formally compare EW's performace to EWV's performance in terms of adapting to market fluctuations and overall success. In addition, we would like to examine how changing the number of actions and trials of our models, where appropriate, changes the results we observed. Furthermore, we would like to see how an extension of EWV or EW that incorporates context would perform on market data, and we also think it would be interesting to expand the algorithm to also decide which cryptocurrencies to invest in. Another thing we could do is try to figure out how we could get the algorithm to decide how much to invest or short; in our study we assumed a fixed investment.

5 Appendix

5.1 ETH Daily Data

Because our ETH daily data set is fairly large and not that interesting in itself, you can view it if you wish by clicking here.
Here is the link to Coinbase Pro's REST API documentation.

5.2 API script

The API only lets you get so many values at a time, so split your requests into sections around 6 months long if you'd like to get a full year's worth of data.

```
import pandas as pd
import requests
import json

def fetch_daily_data(symbol, start, end, granularity):
    pair_split = symbol.split('/')
    symbol = pair_split[0] + '-' + pair_split[1]
    url = 'https://api.pro.coinbase.com/products/'+symbol+'/candles?start='+star
    response = requests.get(url)
    if response.status_code == 200:  # check to make sure the response from serv
        data = pd.DataFrame(json.loads(response.text), columns=['unix', 'low', '
        data['date'] = pd.to_datetime(data['unix'], unit='s')
# convert to a readable date
        if data is None:
            print("Did not return any data from Coinbase for this symbol")
        else:
            data.to_csv('coinbase.csv', index=False)
    else:
        print("Did not receieve OK response from Coinbase API")

pair = 'ETH/USD'
start = '2021-10-10T12:00:00'
end = '2022-04-22T12:00:00'
granularity = '86400'
fetch_daily_data(pair, start, end, granularity)
```

5.3 Libraries for Source Code

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

```
import pandas as pd
```

5.4 Generative Models Source Code

```
rand.seed()

def Generate_Adv_Payoff(payoffs, k):
    payoff = rand.uniform(0,1)
    V_list = []
    for i in range(0, k):
        action_payoffs = payoffs.get(i, [0])
        action_payoff = sum(action_payoffs)
        V_list.append([action_payoff, i])
    V_list.sort()
    j_star = V_list[0][1]
    cur = 0
    next = 1
    while (next < len(V_list) and V_list[cur][0] == V_list[next][0]):
        j_star = V_list[cur][1]
        cur = cur + 1
        next = next + 1
    new_payoffs = {}
    for i in range(0, k):
        action_payoffs = payoffs.get(i,[])
        if i == j_star:
            action_payoffs.append(payoff)
        else:
            action_payoffs.append(0)
        new_payoffs[i] = action_payoffs

    return new_payoffs

def Generate_Bern_Payoff(payoffs, probs, k):
    new_payoffs = {}
    for i in range(0, k):
        action_payoffs = payoffs.get(i, [])
        prob = probs[i]
        x = rand.uniform(0,1)
        if x <= prob:
            action_payoffs.append(1)
        else:
            action_payoffs.append(0)
        new_payoffs[i] = action_payoffs
    return new_payoffs
```

```
def Generate_D_Payoff(payoffs, round, k):
    new_payoffs = {}
    x = rand.uniform(0, round)
    idx = round % k
    for i in range(0, k):
        action_payoffs = payoffs.get(i, [0])
        last_payoff = action_payoffs[-1]
        new = last_payoff + round
        if i == idx:
            action_payoffs.append(new)
        else:
            action_payoffs.append(last_payoff)
        new_payoffs[i] = action_payoffs
    return new_payoffs

def Choose_Action(probs):
    x = rand.uniform(0,1)
    cur = 0
    next = probs[0]
    for i in range(0,len(probs)):
        if i == (len(probs) - 1):
            return i
        elif x >= cur and x <= next:
            return i
        else:
            cur = cur + probs[i]
            next = next + probs[i+1]
    return -1

def Convert_Date_To_FileDate(date):
    filedate = date.replace(':', '/')
    return filedate

def Update_Choice_Probs(payoffs, probs, round, e, k, h):
    prob_sum = 0
    for i in range(0, k):
        action_payoffs = payoffs[i]
        sum = 0
        for j in range(0,round):
            sum = sum + action_payoffs[j]
        exp = sum / h
        prob = ((1+e)**exp)
        probs[i] = prob
        prob_sum = prob_sum + prob
    for i in range(0, k):
        probs[i] = probs[i] / prob_sum
```

```
    return probs

class Exponential_Weights:
    def __init__(self, k, n, h):
        self.k = k
        self.n = n
        self.h = h
        e_list = []
        e = np.sqrt(np.log(k)/n)
        self.theoretical_e = e
        e_max = 2*e
        e_step = e_max / 20
        for i in range(0,20):
            if i != 10:
                e_i = i*e_step
                e_list.append(e_i)
            else:
                e_list.append(e)
        ftl_e = e*3
        e_list.append(e_max)
        e_list.append(ftl_e)
        self.learning_rates = e_list


    def Create_Adv_Payoffs(self):
        payoffs = {}
        for i in range(0,self.n):
            payoffs = Generate_Adv_Payoff(payoffs, self.k)
        return payoffs

    def Create_Bern_Payoffs(self):
        probs = {}
        for i in range(0, self.k):
            probs[i] = rand.uniform(0, 0.5)
        payoffs = {}
        for i in range(0, self.n):
            payoffs = Generate_Bern_Payoff(payoffs, probs, self.k)
        return payoffs

    def Create_D_Payoffs(self):
        payoffs = {}
        for i in range(0,self.n):
            payoffs = Generate_D_Payoff(payoffs, i, self.k)
        return payoffs
```

```
def Run(self, payoffs, e):
    ftl_e = self.learning_rates[-1]
    if e == ftl_e:
        payoff = self.FTL(payoffs)
    else:
        payoff = 0
        initial_prob = 1/self.k
        choice_probs = [initial_prob] * self.k
        k = self.k
        h = self.h
        for i in range(0, self.n):
            choice = Choose_Action(choice_probs)
            assert(choice != -1)
            payoff_list = payoffs.get(choice, [0])
            payoff = payoff + payoff_list[i]
            round = i + 1
            choice_probs = Update_Choice_Probs(payoffs, choice_probs, round,
    return payoff
@classmethod
def Calc_OPT(cls, payoffs):
    total_payoffs = []
    for action in payoffs:
        action_payoffs = payoffs[action]
        total = sum(action_payoffs)
        total_payoffs.append(total)
    total_payoffs.sort(reverse=True)
    OPT = total_payoffs[0]
    return OPT
@classmethod
def Cur_Winner(cls, payoffs, round):
    total_payoffs = []
    for action in payoffs:
        action_payoffs = payoffs[action]
        sum = 0
        for i in range(0, round):
            sum = action_payoffs[i] + sum
        total_payoffs.append([sum, action])
    total_payoffs.sort(reverse=True)
    cur = total_payoffs[0]
    cur_idx = cur[1]
    return cur_idx

def FTL(self, payoffs):
    payoff_list = []
    initial_prob = 1/self.k
    choice_probs = [initial_prob] * self.k
```

```
        first_choice = Choose_Action(choice_probs)
        choice_payoffs = payoffs.get(first_choice, [0])
        payoff = choice_payoffs[0]
        payoff_list.append(payoff)
        for i in range(1, self.n):
            cur = self.Cur_Winner(payoffs, i)
            choice_payoffs = payoffs.get(cur, [0])
            payoff = choice_payoffs[i]
            payoff_list.append(payoff)
        payoff = sum(payoff_list)
        return payoff



figure, axis = plt.subplots(1,3)
# model == 0 --> adv, model == 1 --> bern, model == 2 --> D
def Monte_Carlo(EW, trials, model):
    inputs = []
    for i in range(0,trials):
        if model == 0:
            input = EW.Create_Adv_Payoffs()
        elif model == 1:
            input = EW.Create_Bern_Payoffs()
        elif model == 2:
            input = EW.Create_D_Payoffs()
        inputs.append(input)
    e_list = EW.learning_rates
    avg_regret_list = []
    for i in range(0, len(e_list)):
        e = e_list[i]
        regrets = []
        for j in range(0, trials):
            payoffs = inputs[j]
            ALG = EW.Run(payoffs, e)
            OPT = Exponential_Weights.Calc_OPT(payoffs)
            regret = OPT - ALG
            regret = regret / EW.n
            regrets.append(regret)
        avg = sum(regrets)
        avg = avg / len(regrets)
        avg_regret_list.append(avg)
    axis[model].plot(e_list, avg_regret_list, '.-')
    if model == 0:
        axis[model].set_title("Adv_Model")
    if model == 1:
```

```
        axis[model].set_title("Bern_Model")
    if model == 2:
        axis[model].set_title("Model_D")


EW = Exponential_Weights(10, 100, 1)
EW_D = Exponential_Weights(10, 100, 100)

Monte_Carlo(EW, 100, 0)
Monte_Carlo(EW, 100, 1)
Monte_Carlo(EW_D, 100, 2)

plt.show()
```

5.5 ETH Model Source Code

```
def create_coinbase_payoffs():
    data = pd.read_csv('coinbase.csv')
    data['change'] = (data['close'] - data['open']) / data['open']

    payoffs = {}
    data_length = len(data['change'])
    idx_list = []
    for idx, row in data.iterrows():
        delta = row['change']
        invest_payoff = 1 + delta
        short_payoff = 1 - delta
        pidx = data_length - idx - 1
        idx_list.append(idx)
        payoffs[pidx] = [invest_payoff, 1, short_payoff]

    return payoffs, idx_list

class EW_ALGS:
    instances = []
    payoffs = {}
    action_count = 0
    round = 0
    payoff = 0
    OPT_actions = []
    ALG_actions = []
    e = 0

    def __init__(self, i, k):
        self.birth_round = i
```

```
        self.observed_payoffs = {}
        self.time_ran = 0
        self.probs = []

    @classmethod
    def initialize(cls, payoffs, k):
        cls.action_count = k
        cls.round = 0
        cls.payoff = 0
        cls.payoffs = payoffs
        cls.instances.clear()
        cls.OPT_actions.clear()
        cls.ALG_actions.clear()

    @classmethod
    def create_ALG(cls):
        ALG = cls(cls.round, cls.action_count)
        ALG.probs = cls.init_probs()
        cls.instances.append(ALG)


    @classmethod
    def init_probs(cls):
        initial_prob = 1/cls.action_count
        choice_probs = [initial_prob] * cls.action_count
        return choice_probs

    def choose_action(self, k):
        probs = self.probs
        x = rand.uniform(0,1)
        cur = 0
        next = probs[0]
        for i in range(0, k):
            if i == (k - 1):
                return i
            elif x >= cur and x <= next:
                return i
            else:
                cur = cur + probs[i]
                next = next + probs[i+1]
        return -1

    @classmethod
    def pick_choice(cls, probs):
        x = rand.uniform(0,1)
        cur = 0
```

```
        next = probs[0]
        for i in range(0, k):
            if i == (k - 1):
                return i
            elif x >= cur and x <= next:
                return i
            else:
                cur = cur + probs[i]
                next = next + probs[i+1]
        return -1

    @classmethod
    def run(cls, payoffs, k):
        cls.initialize(payoffs, k)
        n = len(cls.payoffs)
        for i in range(0, n):
            cls.run_step()

    @classmethod
    def run_step(cls):
        cls.create_ALG()
        choices = {}
        for alg in cls.instances:
            choice = alg.choose_action(cls.action_count)
            birth = alg.birth_round
            adj_birth = birth + 1
            choices[choice] = choices.get(choice, 0) + (adj_birth**2)

        vote_sum = 0
        for i in range(0, cls.action_count):
            vote_sum = vote_sum + choices.get(i, 0)
        prob_choices = []
        for i in range(0, cls.action_count):
            prob = choices.get(i, 0) / vote_sum
            prob_choices.append(prob)

        agg_choice = cls.pick_choice(prob_choices)
        round_payoffs = cls.payoffs[cls.round]
        cls.payoff = cls.payoff + round_payoffs[agg_choice]
        cls.update_ALGS()
        cls.round = cls.round + 1
        cls.ALG_actions.append([agg_choice, cls.payoff])
        OPT_idx_payoff = cls.calc_OPT()
        OPT_choice = OPT_idx_payoff[0]
        OPT_payoff = OPT_idx_payoff[1]
        cls.OPT_actions.append([OPT_choice, OPT_payoff])
```

```
    @classmethod
    def update_ALGS(cls):
        round_payoffs = cls.payoffs[cls.round]
        for alg in cls.instances:
            observed_payoffs = alg.observed_payoffs
            for i in range(0, cls.action_count):
                observed_payoffs[i] = observed_payoffs.get(i, 0) + round_payoffs
            birth = alg.birth_round
            time_ran = alg.time_ran
            time_ran = time_ran + 1
            #cls.update_e()
            e = cls.theoretical_e(time_ran)
            alg.probs = alg.update_probs(e, cls.action_count)

    @classmethod
    def calc_OPT(cls):
        first_ALG = cls.instances[0]
        observed_payoffs = first_ALG.observed_payoffs
        idx_payoff = []
        for i in range(0, cls.action_count):
            idx_payoff.append([i, observed_payoffs.get(i, 0)])
        return max(idx_payoff,key=lambda x:x[1])

    #@classmethod
    #def update_e(cls):
    #    cls.e = np.sqrt(np.log(cls.action_count)/(cls.round+1))
    @classmethod
    def theoretical_e(cls, time_ran):
        return np.sqrt(np.log(cls.action_count)/time_ran)

    def update_probs(self, e, k):
        probs = []
        for i in range(0, k):
            observed_payoffs = self.observed_payoffs
            payoff = observed_payoffs.get(i, 0)
            exp = payoff / 2
            top = (1+e)**exp
            probs.append(top)
        total = sum(probs)
        for p in probs:
            p = p / total
        return probs

coinbase_payoffs, idx_list = create_coinbase_payoffs()
data_length = len(idx_list)
```

```
k = 3
trials = 100
regrets = {}
payoffs = {}
for i in range(0, trials):
    EW_ALGS.run(coinbase_payoffs, k)
    for j in range(0, data_length):
        ALG = EW_ALGS.ALG_actions[j]
        ALG_choice = ALG[0]
        ALG_cum = ALG[1]
        OPT = EW_ALGS.OPT_actions[j]
        OPT_choice = OPT[0]
        OPT_cum = OPT[1]

        regret = (OPT_cum-ALG_cum)/(j+1)
        payoff = ALG_cum
        if j > 0:
            prev_ALG = EW_ALGS.ALG_actions[j-1]
            payoff = payoff - prev_ALG[1]

        regrets[j] = regrets.get(j, 0) + regret
        payoffs[j] = payoffs.get(j, 0) + (payoff - 1)

regret_list = []
payoff_list = []

for i in range(0, data_length):
    regret = regrets.get(i, 0) / trials
    payoff = payoffs.get(i, 0) / trials
    print(regret)
    regret_list.append(regret)
    payoff_list.append(payoff)


def EX_Payoff_From(payoffs, i):
    payoff_list = payoffs[i:]
    sum_payoffs = sum(payoff_list)
    n = data_length - i
    return (sum_payoffs / n)

EX_payoff_list = []
for i in range(0, data_length):
    ex = EX_Payoff_From(payoff_list, i)
    EX_payoff_list.append(ex)

figure, axis = plt.subplots(1,3)
```

```
axis[0].plot(idx_list, regret_list, '.-')
axis[1].plot(idx_list, payoff_list, '.-')
axis[2].plot(idx_list, EX_payoff_list, '.-')
axis[0].set_title('Avg Regret at Round')
axis[1].set_title('Avg Payoff at Round')
axis[2].set_title('EX Payoff from Round to End')

plt.show()
```

Note that this code includes some extra plots. These were simply to see if the algorithm would've actually made a profit, and are not involved in the study. We've included them so you can run our code and see them yourself.