Student: Krishnarupa Sewsunker

Professor: Dr. Jie Wu

Course: CIS 4282 – Independent Study

Semester: Summer 2022

Final Report

The Multi-Armed Bandit and Combinatorial Multi-Armed Bandit Problem

The Multi-Armed Bandit (MAB) is a problem studied within probability theory, statistics and machine learning which can be understood as a machine (or "bandit") with many levers (or "arms"). These arms each have a pre-defined distribution of reward values that they can provide to the person pulling the arms, i.e., the user or player. In other words, for each arm, there is a set of possible rewards a user can achieve from pulling it, and each item within that set of rewards can be achieved with a certain likelihood. Furthermore, due to each arm's defined probability distribution, there is an expected value for the reward of each arm, which is the mean reward for each arm based on its distribution.

A user does not know in advance what the probability distribution is for each arm, nor does the user know the mean reward of each arm. A round or turn consists of the user pulling one arm out of the k available arms, thus achieving a certain reward from the selected arm. The user's goal is to yield the greatest possible total reward within the available number of rounds. The overarching strategy for achieving this goal is for the user to use the available number of rounds in such a way that they can effectively discover the best-performing arms (i.e., the arms that consistently provide the highest rewards, in other words, the arms with the highest means) in a swift and reliable manner so that they can go on to pull those best-performing arms as much as possible for the remaining rounds. Hence, the user sets out to both explore (learn about all arms' performance to identify the best arms) and exploit (repeatedly pull the best arms) the set of k arms over t rounds, where k and t are non-negative by necessity.

The realization of this goal of optimizing the exploration-exploitation tradeoff in order to maximize cumulative rewards has been studied in several works and contributions, which has led

to various algorithms being designed which each have different approaches to addressing the learning/exploration phase and the exploitation phase. The most widely used algorithms are the Upper Confidence Bound (UCB), Epsilon-Greedy, and Thompson Sampling algorithms. Other well-known algorithms are the Softmax (Boltzmann Exploration), Pursuit and Reinforcement Comparison algorithms. [1] [2] Based on the nature of the specific problem which is being set in the framework of MAB, one or more of the algorithms above may be better-suited for solving the problem and may outperform the other algorithms.

We now define a crucial term in understanding algorithms' performance in solving the MAB problem. How well an algorithm performs can be measured by the *regret* yielded by the algorithm. Over a certain number of rounds, say *t* rounds, the regret is the difference between the cumulative reward achieved by pulling the optimal arm (arm with highest mean reward) *t* times and the cumulative reward achieved by pulling arms according to the specific algorithm's routine.

$$R_T = T\mu^* - \sum_{t=1}^{T} \hat{r}_t$$

Equation 1: Regret – the difference between the reward sum from playing the optimal arm for all rounds and the reward sum from playing the arm selected in each round according to a given algorithm.

The UCB algorithm is often described as "optimism in the face of uncertainty" seeing as the strategy takes into account not only the empirical mean of each arm up until the present round but also an added term that favors arms that have been selected less frequently. Each arm must be initialized (i.e., played once) and thereafter, each arm is given a UCB score which is updated at each round. The added term mentioned above necessitates that each arm's UCB score is slightly higher the respective arm's empirical mean. This added term has a larger magnitude for arms that have been pulled a smaller number of times, as can be seen in Equation 2 below, since the number of times an arm *i* has been selected up until round *t* is denoted by $n_{i,t}$ which is found in the denominator. Since the arm with the current highest UCB score is selected in each round, the fact that the added term is larger for arms that have been selected less signifies that the added term encourages the exploration of those less-selected arms. The factor in the added term

impacts how much the exploration is to be weighted: a larger factor will encourage more exploration by attributing a higher UCB score to those arms less selected; and a smaller factor will discourage exploration and put more emphasis on exploitation, since there won't be a substantial amount added to the UCB scores of the arms that are less selected.

$$j(t) = \arg\max_{i=1\ldots k} \left(\hat{\mu}_i + \sqrt{\frac{2 \ln t}{n_i}} \right)$$

Equation 2: The UCB algorithm (also referred to as UCB1 to distinguish it from the various modifications on this original algorithm)

In our work in implementing the UCB algorithm using Python, we studied the original UCB1 algorithm's performance as it appears in Equation 2, and then we varied some of the algorithm's parameters in order to observe the impact different components of the algorithm had on its performance results. We observed that the natural log is a crucial component of the algorithm's good performance, that the factor in the added term substantially and proportionately influences the degree of exploration, and that weighting the mean term can help increase exploitation, as shown in the regret and reward figures below. Hence, we can identify that for different applications and settings in which the MAB model is applicable, one may use different versions or modified versions of the UCB1 algorithm according to the information and preferences one may have about the arms and their respective distributions. For example, if a particular application of the MAB problem does not need to place very much emphasis on exploration and is simply seeking to exploit good arms as early on as possible, one might choose to reduce the factor 2 to be a much smaller value such as 1/4.

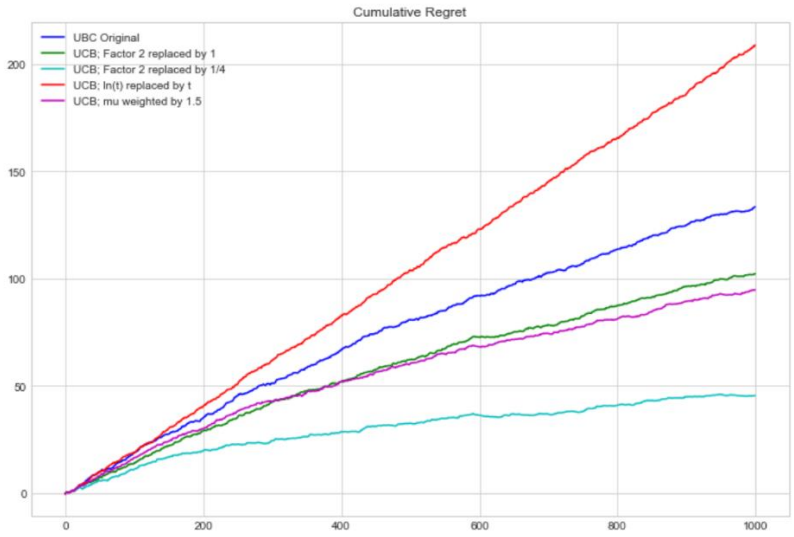

Figure 1: Cumulative Regret Comparisons for various modifications to the UCB1 algorithm

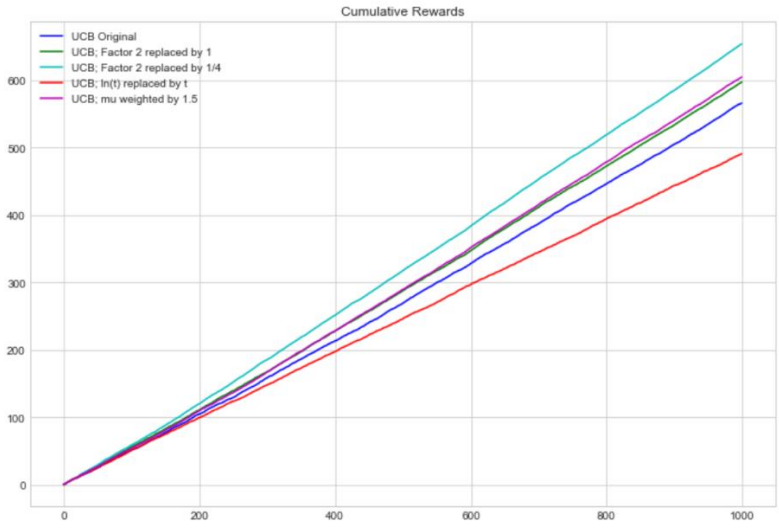

Figure 2: Cumulative Reward Comparisons for various modifications to the UCB1 algorithm

The framework described up until this point pertains to the simple MAB model, however there are many modifications and variations to this model, one of the most important in terms of real-world applicability being the Combinatorial Multi-Armed Bandit, also known as the Super Arm. This model stipulates that the user or player pulls not one arm at a time, but some number K arms, where K is less than the total arms k. Hence, in each round t, there are K arms played and the rewards yielded in each round is the sum of the rewards of each arm played. The K arms

constitute a “Super Arm” and it can be recognized that the simple MAB discussed above is just a special case of the super arm model, wherein each super arm consists of only one simple arm rather than *K* simple arms.

To solve the Combinatorial Multi-Armed Bandit (CMAB) problem, the standardly used algorithm is the “Combinatorial Upper Confidence Bound” algorithm (CUCB), which is based on the traditional UCB algorithm and is outlined in the work of Chen et al. [3]. Since the top *K* arms are selected in each round, we can use the term “window” to describe the chosen set of arms, where the window is of size *K*. The CUCB algorithm can be referred to as a “disjoint window” model, because once the user has chosen the top *K* arms, the window is shifted by *K* so that two consecutive sets of chosen arms from two windows cannot overlap or intersect, and therefore must be disjoint. This model is likely to have hindrances for gaining an understanding of the interrelationship or interdependency between different combinations of arms.

Inherent in the CMAB model is the concept of joint probability distributions between the arms. Since each arm can be interpreted as representing a choice, if the user is to make *K* choices in each round, there is the possibility of some choices impacting or depending on other choices. This implies that the arms might have some underlying relationship or interdependency, hence this can be analyzed as a joint probability distribution between the interdependent arms. In order to have a better opportunity to understand the potential interdependency between arms and in order to bypass the hindrance of needing to shift the window by *K* which prevents overlapping windows, a new concept was conceived of by Dr. Wu which we will refer to as the “sliding window” model.

Distinct from the disjoint window, the goal of this sliding window is to provide a greater opportunity for learning the potential joint distributions or interdependency between arms, because rather than shifting by *K*, this model allows you to shift by 1 arm each round (where arms are ranked by best performance up until that round), unless a currently chosen arm has already been chosen in the most recent *K*-1 rounds. An arm, once chosen, must therefore be disabled from being chosen for *K*-1 rounds in order to prevent the possibility of any arm being pulled twice within one super arm window (which is not permissible in the CMAB model). If an arm is disabled, the next best arm that is not disabled is selected within the super arm.

An important aspect to be accounted for in both the disjoint and sliding window models is that the total reward and the value of $n_{i,t}$ needed to be normalized. Normalization was performed fairly in both the disjoint and sliding window model to allow for fair comparison between the two models' performances. This was carried out by dividing the total reward per round by K (due to K arms being selected and contributing to the total reward in each round) and by increasing the $n_{i,t}$ counter by $1/K$ for each arm selected within every round. See Figures 3.a. and 3.b., and 4.a. and 4.b. in the Addendum for the full code for the disjoint and sliding window models respectively.

For simulating these two models, the data structures that needed to be employed were: arrays and lists for storing the history of the rewards obtained in each round, the history of regret obtained in each round, cumulative reward, cumulative regret, the UCB score for each arm to be updated in each round, the history of arms selected (in order to disable arms if they were selected in the past K-1 rounds, as well as to verify that the algorithm was accurately able to uncover the optimal arms); and tables and data-frames for creating various kinds of joint probability distributions amongst the 18 arms, with different reward values and varied pairwise, three-way and four-way interdependencies. Figure 5 in the Addendum provides part of the code used to create one of the data-frames used in simulation.

The visualizations used were line plots and bar charts. Using line plots, we portrayed the non-cumulative regret in order to observe how the regret got smaller and smaller gradually over each round as the algorithm learnt about the arms' performance. We also portrayed the non-cumulative regret in step-sizes of 50 and 100 out of the 10,000 rounds, meaning that we have just one point plotted for every 50 and 100 rounds, in order to see the trend of the diminishing regret more clearly. We plotted cumulative regret and cumulative reward as line plots as well. Then, as bar charts, we plotted the total sum of regret over all 10,000 rounds for the disjoint and sliding windows adjacently, as well as the total sum of rewards achieved over all 10,000 rounds for these two models. See Figures 6.a. and 6.b. for the code used for visualization of the results.

Our simulation results show that the disjoint and sliding window models' performance are very similar, and that under certain conditions, the sliding window provides a better performance, whereas under different conditions, the disjoint window outperforms it. The sliding window model consistently performs better than the disjoint window for three-way joint

distributions. This can be seen in Figure 7.a., b., and c, which each show the sliding window's regret outperforming that of the disjoint window where the lines intersect, for three different three-way joint distribution datasets. However, as can be seen in Figures 8.a. and 8.b., the sliding window performed worse than the disjoint window for datasets that had pairwise joint distributions, and four-way joint distributions. Please see the attached PDF materials for further graphs from different datasets with different joint distributions.

It is worth highlighting that although the sliding window model performed worse in the pairwise and four-way distributions according to having accrued a higher regret, it is clear from the graphs in the PDF that the range between which the sliding window model's regret fluctuates is consistently a smaller range than that within which the disjoint model's regret fluctuates. The disjoint model also benefits from having more frequent negative values for regret (which occurs when the achieved reward surpasses the expected optimal reward), which cancel out a portion of the higher positive regret values, making the overall total regret seem lower. If, however, one was to just sum the magnitudes of regret achieved by the disjoint model, the sliding window model would have outperformed it even in the pairwise and four-way distributions. The steeper curves in the sliding window model versus the disjoint model which are visible in the non-cumulative regret plots suggest that the sliding window model has a greater aptitude for learning and can do so more quickly than the disjoint window (which had a shallower curve regardless of the different datasets). These simulation results, therefore, seem to indicate that the sliding window might have specific strengths or applicability in certain types of datasets and distributions, or that with a larger number of rounds or some modifications, it could grow to perform better than the disjoint window consistently.

References:

[1] Volodymyr Kuleshov, V., Precup, D., “Algorithms for the multi-armed bandit problem”, 2000

[2] Silva, N., Werneck, H., Silva, T., Pereira, A. C. M., Rocha, L., “Multi-Armed Bandits in Recommendation Systems: A survey of the state-of-the-art and future directions”, 2022

[3] Chen, W., Wang, Y., Yuan, Y., “Combinatorial Multi-Armed Bandit: General Framework, Results and Applications”

Addendum:

```
#Disjoint
#implementing CUCB, 10,000 runs
#K=4 arms being selected
rounds = len(df2.index) # number of rounds;
k = 18 # number of arms
K = 4 # window size
mu = 0 # initialize to 0, the empirical mean
mu_arr = []
N = np.zeros(k) # number of times a given arm has been selected before current round; initialize to 0 as no arm selected ye
rewardSum = np.zeros(k)
hist_UCB_rewards_norm = [] #stores history of UCB CHOSEN rewards
alpha = 1
beta = 1
r_mu = 0
#calculate best superarm of window size K=4
opt_mu = (allmean[13]+allmean[14]+allmean[9]+allmean[15])/K #divide by K

for t in range(rounds):
    UCB_Values = np.zeros(k) #array holding ucb values. re-initialize to 0 at start of each round
    temp_UCB_Values = np.zeros(k)
    arm_selected = 0
    for a in range(k):
        if (N[a] > 0):
            #calculate UCB below
            mu = rewardSum[a]/N[a]
            ucb_value = mu + math.sqrt(3/2*math.log(t)/N[a])
            UCB_Values[a] = ucb_value
        elif (N[a] == 0): #i.e. if N is 0 for arm a, then allow exploration of that arm
            UCB_Values[a] = 1e500 #infinity
        temp_UCB_Values[a] = UCB_Values[a] #this is a copy of UCB_Values[a], to be modified below to find runner-up arms

    arm1_selected = np.argmax(UCB_Values) #NB: argmax gives index of max value in a list/array
    temp_UCB_Values[arm1_selected] = 0
    arm2_selected = np.argmax(temp_UCB_Values)
    temp_UCB_Values[arm2_selected] = 0
    arm3_selected = np.argmax(temp_UCB_Values)
    temp_UCB_Values[arm3_selected] = 0
    arm4_selected = np.argmax(temp_UCB_Values)

    #update Values as of round t
    reward = df2.values[t,arm1_selected+1]+df2.values[t,arm2_selected+1]+df2.values[t,arm3_selected+1]+df2.values[t,arm4_se
    r_mu = reward/K #reward divide by K, this is normalized reward
    #idea: increment each arm by 1/K
    N[arm1_selected] += 1/K
    N[arm2_selected] += 1/K
    N[arm3_selected] += 1/K
    N[arm4_selected] += 1/K

    #this below is the way to increment N according to the superarm paper, however
    #for the sake of fair comparison with sliding model, we will increment N as per above.
#     if r_mu < alpha*opt_mu: #superarm S is bad if: reward sum from S < alpha * mean Reward of Optimal Arm.
#         #this superarm is a bad superarm
#         #only increment the simple arm that has min N within this superarm
#         if (N[arm1_selected]<N[arm2_selected]):
#             N[arm1_selected] += 1
#         else:
#             N[arm2_selected] += 1
     #else if superarm S is not bad then don't increment N at all for any simple arms within that superarm

    rewardSum[arm1_selected] += df2.values[t, arm1_selected+1]  #do this K times
    rewardSum[arm2_selected] += df2.values[t, arm2_selected+1]
    rewardSum[arm3_selected] += df2.values[t, arm3_selected+1]
    rewardSum[arm4_selected] += df2.values[t, arm4_selected+1]
    hist_UCB_rewards_norm.append(r_mu)
    UCB_cumu_reward_d = np.cumsum(hist_UCB_rewards_norm)
    UCB_total_reward = sum(hist_UCB_rewards_norm)
```

Figure 3.a. The disjoint window model using the CUCB algorithm (modified in terms of normalization, for the purpose of fair comparison with the sliding model)

```
regret_total_d = opt_mu*rounds - UCB_total_reward
print("Total regret after ", rounds, " rounds: ", regret_total_d)

#regret, not cumulative
regret_arr_d = np.ones(rounds)*opt_mu*1 - hist_UCB_rewards_norm
regret_arr_d

plt.plot(regret_arr_d,color = 'r')
plt.title("Regret Per Round (Not Cumulative)")
plt.show()
```

Figure 3.b. Computing non-cumulative regret for the disjoint window

```
#Sliding
#implementing CUCB, 10,000 runs
#K=4 arms being selected
rounds = len(df2.index)
k = 18 # number of arms
K = 4 # window size
mu = 0 # initialize to 0, the empirical mean
N = np.zeros(k) # number of times a given arm has been selected before current round; initialize to 0 as no arm selected yet
rewardSum = np.zeros(k)
hist_UCB_rewards = [] #stores history of UCB CHOSEN rewards
hist_window_reward = []
hist_a = [-1, -1, -1] #stores history of arms selected
ctr = []
temp = [] #stores history of rewards

for t in range(rounds):
    UCB_Values = np.zeros(k) #array holding ucb values. re-initialize to 0 at start of each round, because each round, selec
    temp_copy = np.zeros(k)
    arm_selected = 0
    for a in range(k):
        if (N[a] > 0):
            mu = rewardSum[a]/N[a]
            ucb_value = mu + math.sqrt(3/2*math.log(t)/N[a])
            UCB_Values[a] = ucb_value
        elif (N[a] == 0): #i.e. if N is 0 for arm a, then allow exploration of that arm. ensures initialization of all arms
            UCB_Values[a] = 1e500
        temp_copy[a] = UCB_Values[a]
    arm_selected = np.argmax(UCB_Values) #NB: argmax gives index of max value in a list/array
    #hist_a.append(arm_selected)
    if t>=K: #this is for disabling a redundant arm (arm that was selected more than once in window size K)
        for h in np.arange(1,K):
            if (arm_selected==hist_a[t-3]) or (arm_selected==hist_a[t-2]) or (arm_selected==hist_a[t-1]):
                print("match")
                #hist_a.pop()
                temp_copy[arm_selected]=0
                arm_selected = np.argmax(temp_copy)
            else:
                print("Not match")
    hist_a.append(arm_selected) #append with the proper arm_selected
    N[arm_selected] += 1/K #should be incremented by 1/K
    reward = df2.values[t, arm_selected+1]
    temp = np.append(temp, reward)
    rewardSum[arm_selected] += reward
    hist_UCB_rewards.append(reward)
    UCB_cumu_reward = np.cumsum(hist_UCB_rewards)
    #UCB_total_reward = sum(hist_UCB_rewards)

for i in np.arange(K-1,rounds): #rounds divided by K. 10000 rounds for fair comparison with disjoint window
    window_reward_norm = (temp[i]+temp[i-1]+temp[i-2]+temp[i-3])/K #divide by K
    hist_window_reward.append(window_reward_norm)

window_cumu_reward = np.cumsum(hist_window_reward)
window_total_reward = sum(hist_window_reward)

hist_window_reward = [temp[0]] + [temp[1]] + [temp[2]] + hist_window_reward
```

Figure 4.a. The sliding window model

```
regret_total_s = opt_mu*rounds - window_total_reward
print("Total regret after ", rounds, " rounds: ", regret_total_s)

#regret, not cumulative
regret_arr_s = np.ones(rounds)*opt_mu*1 - hist_window_reward
regret_arr_s

plt.plot(regret_arr_s,color = 'r')
plt.title("Regret Per Round (Not Cumulative)")
plt.show()
```

Figure 4.b. Computing non-cumulative regret for the sliding window

```
data2 = {'Probability': np.ones(10)*0.1,
        'Arm1': [0.05, 0.1, 0.15, 0.3, 0.4, 0.5, 0.55, 0.6, 0.7, 0.77],
        'Arm2': [0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85],
        'Arm3': [0.1, 0.15, 0.2, 0.35, 0.45, 0.55, 0.6, 0.65, 0.75, 0.82],
        'Arm4': [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9],
        'Arm5': [0, 0.05, 0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85],
        'Arm6': [0.1, 0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95],
        'Arm7': [0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85],
        'Arm8': [0, 0.05, 0.1, 0.25, 0.35, 0.45, 0.5, 0.55, 0.65, 0.72],
        'Arm9': [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95],
        'Arm10': [0.1, 0.11, 0.15, 0.2, 0.3, 0.3, 0.45, 0.5, 0.5, 0.51],
        'Arm11': [0, 0.3, 0.31, 0.32, 0.33, 0.6, 0.61, 0.62, 0.63, 0.64],
        'Arm12': [0, 0.01, 0.05, 0.1, 0.2, 0.2, 0.35, 0.4, 0.4, 0.41],
        'Arm13': [0.5, 0.55, 0.6, 0.65, 0.7, 0.8, 0.95, 0.99, 1, 1],
        'Arm14': [0.5, 0.55, 0.6, 0.65, 0.7, 0.8, 0.95, 0.99, 1, 1],
        'Arm15': [0.45, 0.5, 0.55, 0.6, 0.65, 0.75, 0.9, 0.94, 0.95, 0.95],
        'Arm16': [0, 0.3, 0.31, 0.32, 0.33, 0.6, 0.61, 0.62, 0.63, 0.64],
        'Arm17': [0, 0.01, 0.05, 0.1, 0.2, 0.2, 0.35, 0.4, 0.4, 0.41],
        'Arm18': [0, 0.3, 0.31, 0.32, 0.33, 0.6, 0.61, 0.62, 0.63, 0.64]
       }

df2 = pd.DataFrame(data2)
df2
```

Figure 5. Part of the code for creating one of the data-frames to be used for simulation

```
#Non-Cumulative Regret Plots for Disjoint and Sliding Window on same set of axes
plt.plot(regret_arr_d, label = "Disjoint Window", color = "c")
plt.plot(regret_arr_s, label = "Sliding Window", color = 'g')
plt.title("Regret Per Round (Not Cumulative)")
plt.rcParams["figure.figsize"] = (14,9)

plt.legend()
plt.show()
```

Figure 6.a. Creating the final line plots that contain the disjoint and sliding window results overlayed for ease of analysis

```
#Bar Plot Total Regret Comparison
height = [regret_total_d, regret_total_s]
bars1 = "Disjoint", "Sliding"
x_pos = np.arange(len(bars1))
plt.bar(x_pos, height, color=(0.8, 0.1, 0.4, 0.8))
plt.xticks(x_pos, bars1)

print("Disjoint Window Model: Total Regret after ", rounds, " rounds: ", regret_total_d)
print("Sliding Window Model: Total Regret after ", rounds, " rounds: ", regret_total_s)

plt.title("Total Regret Comparison")
plt.show()
```

Figure 6.b. Creating the final bar plots that contain the disjoint and sliding window total regret results adjacently for comparison

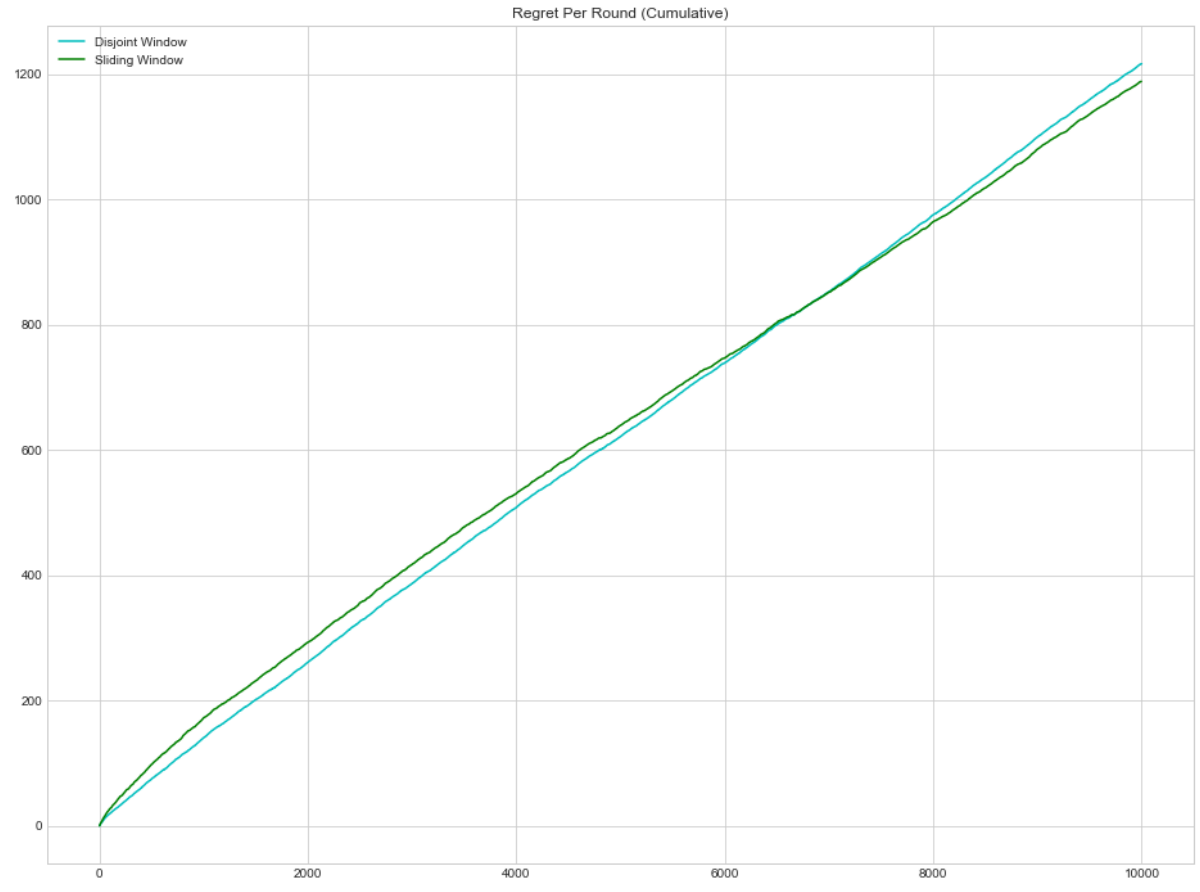

Figure 7.a. Disjoint and sliding window regret for Three-way interdependency dataset 1. Shows that the sliding window outperforms the disjoint window where the lines intersect

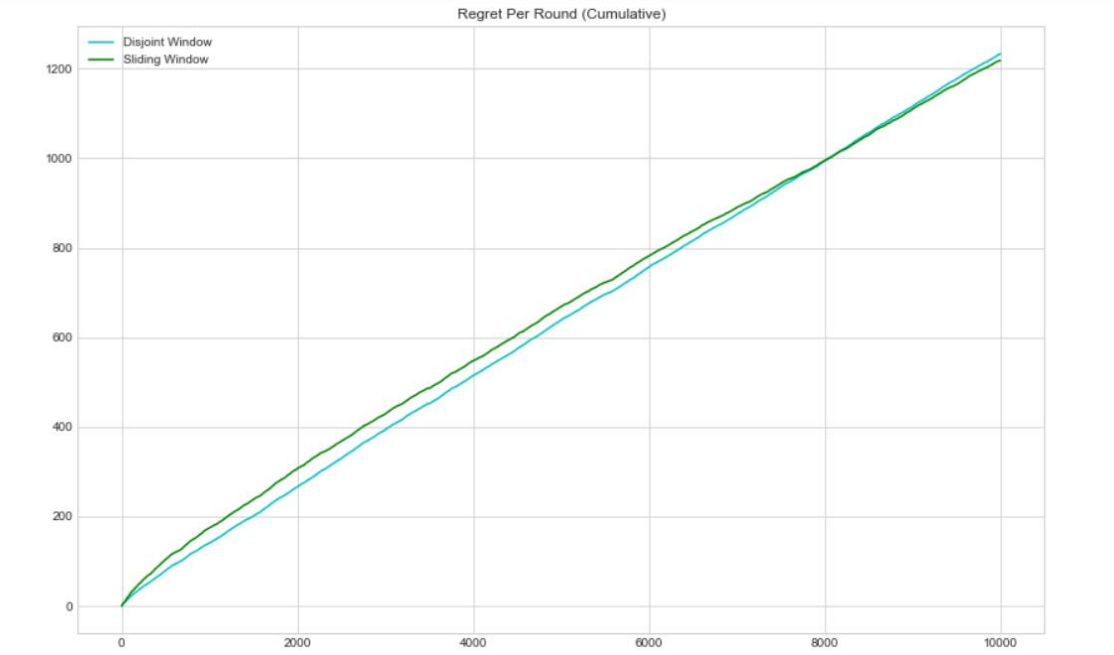

Figure 7.b. Disjoint and sliding window regret for Three-way interdependency dataset 2. Shows that the sliding window outperforms the disjoint window where the lines intersect

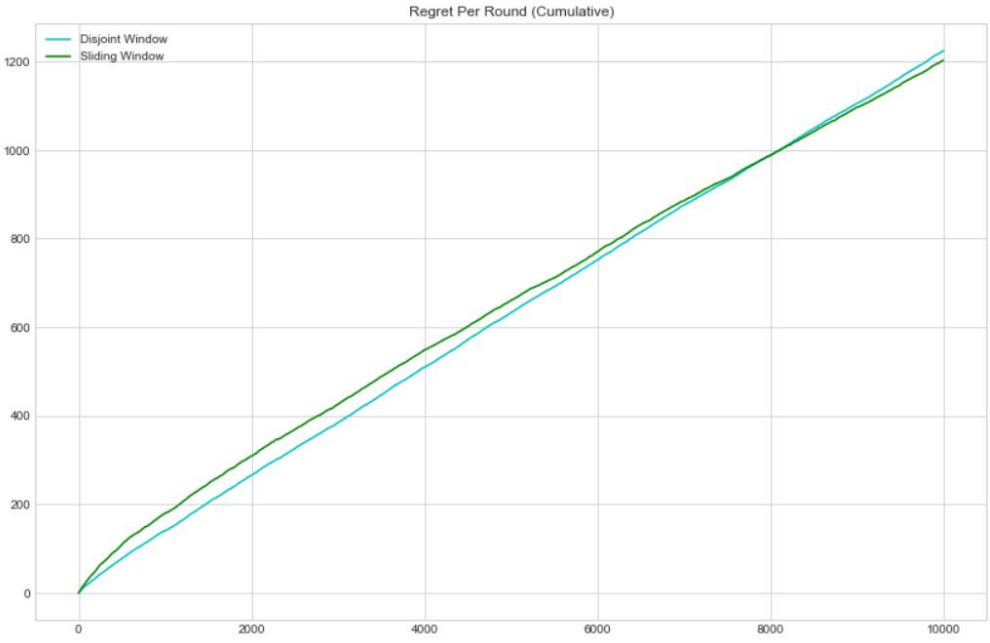

Figure 7.c. Disjoint and sliding window regret for Three-way interdependency dataset 3. Shows that the sliding window outperforms the disjoint window where the lines intersect

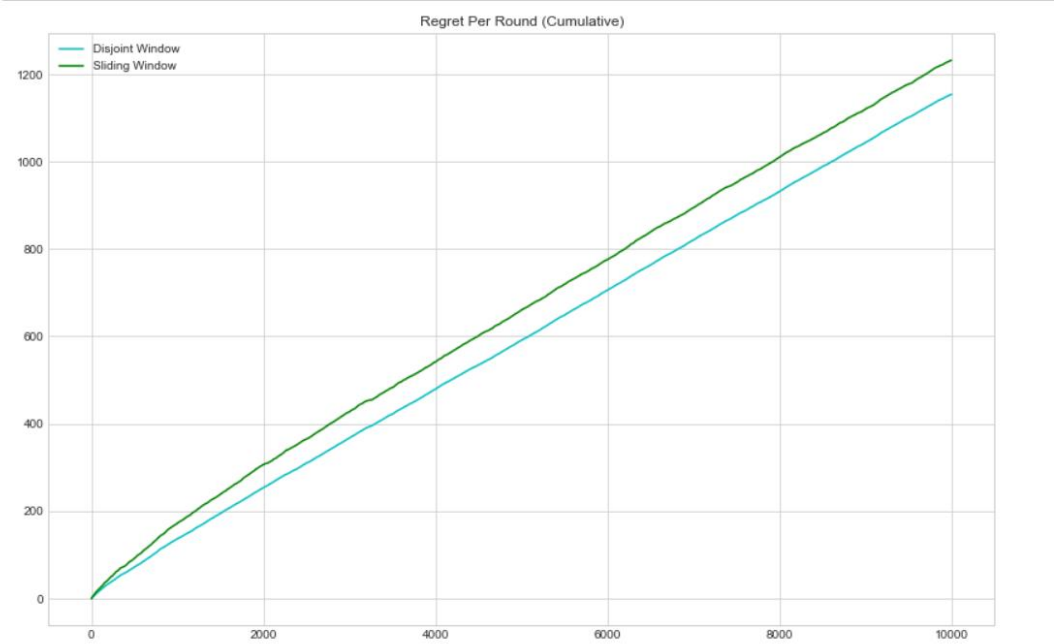

Figure 8.a. Disjoint and sliding window regret for Pairwise interdependency dataset 1. Shows that the sliding window performs worse than the disjoint window due to lines diverging, with sliding window having the higher regret values.

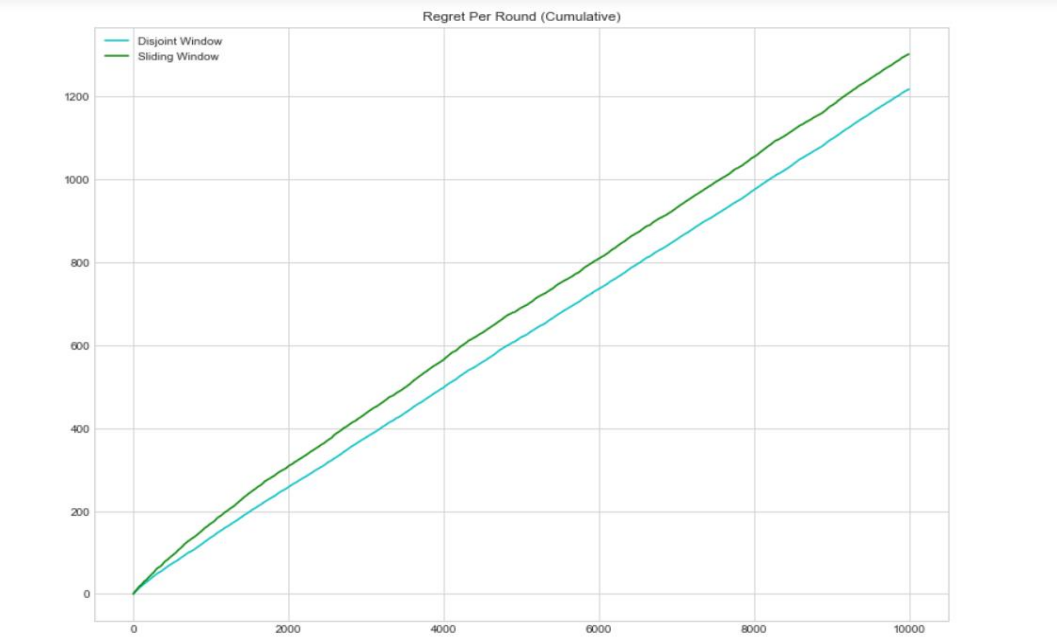

Figure 8.b. Disjoint and sliding window regret for Four-way interdependency dataset 1. Shows that the sliding window performs worse than the disjoint window due to lines diverging, with sliding window having the higher regret values.