CS 747: Programming Assignment 1

Vaibhav Singh (22M0827)

February 18, 2025

Task 1

1.1 Epsilon-Greedy

Figure 1: In epsilon-greedy, at each step, with probability ϵ we explore arms uniformly at random or we choose the best arm based on the empirical mean with probability $1 - \epsilon$. Here, we observe that the cumulative regret (R) grows linearly with the horizon (T). In other words, $R = \Omega(T)$.

For sub-sections 1.2 1.3 and 1.4, we provide code snippets with explanatory comments to modifications made on top of the Algorithm Class for UCB, KL-UCB and Thompson Sampling, respectively.

1.2 UCB

```
class UCB(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # START EDITING HERE
        self.counts = np.zeros(num_arms)
```

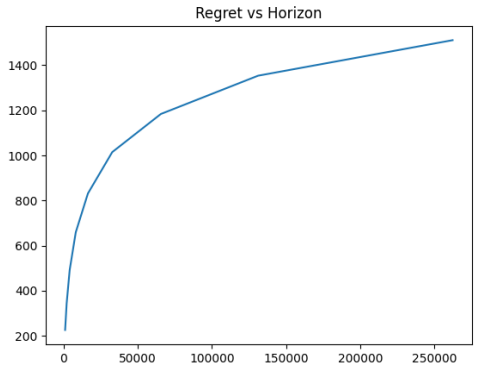

Figure 2: In UCB, the algorithm selects the best arm based on the empirical mean with an added factor ($\sqrt{\frac{2 \ln t}{u_a^t}}$). This factor decays when an arm is pulled more frequently. Here, we observe that UCB achieves a logarithmic growth in cumulative Regret (R) with the horizon (T). In other words, $R = O(\log T)$.

```
        self.values = np.zeros(num_arms)

        self.num_arms = num_arms
        self.num_plays = 0 # Keeps track of the total number
            of plays.
        self.pull_init_arm = -1 # Helper to track arm-number
            for the round-robin phase
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        if self.num_plays < self.num_arms:
            self.num_plays += 1
            self.pull_init_arm += 1
            return self.pull_init_arm # Initially we will
                pull each arm once, before starting the UCB
                loop (as per Auer et al. https://homes.di.
                unimi.it/~cesabian/Pubblicazioni/ml-02.pdf)
        else:
            best_arm = np.argmax(self.values + np.sqrt(2 *
                math.log(self.num_plays) / self.counts)) #
                UCB loop
            self.num_plays += 1
            return best_arm
        # END EDITING HERE
```

Listing 1: Python code for give_pull() method of Task 1: UCB

1.3 KL-UCB

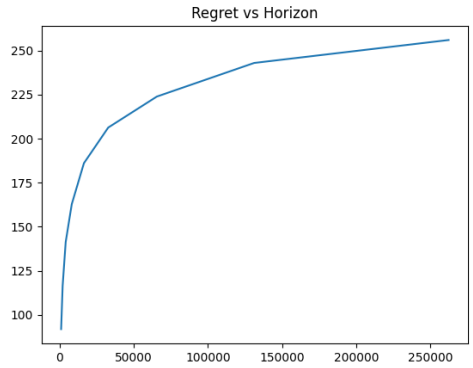

Figure 3: We observe that KL-UCB gives a tighter confidence bound than UCB. Similar to UCB, $R = O(\log T)$.

```
def bernoulli_kldiv(p, q):
    p = np.clip(p, 1e-8, 1 - 1e-8) # Clipping to avoid 0, 1
        in math.log
    q = np.clip(q, 1e-8, 1 - 1e-8)
    return p * math.log(p / q) + (1 - p) * math.log((1 - p)
        / (1 - q))

def binary_search(empirical_mean, num_arm_plays, num_plays,
    c=3): # Binary search for find the optimal q satisfying
    the KL-UCB conditon
    num_iters = 10 # Number of iterations to run search for.
    start = empirical_mean # q belongs to [empirical_mean,
        1]
    end = 1
    for i in range(num_iters):
        q = (start + end) / 2 # Start at midpoint of
            interval and iteratively shrink interval until we
             exhaust num_iters.
        if (num_arm_plays * bernoulli_kldiv(empirical_mean,
            q)) <= (math.log(num_plays) + c * math.log(math.
            log(num_plays))):
            start = q
        else:
            end = q

    return (start + end) / 2 # Return midpoint estimate

class KL_UCB(Algorithm):
    def __init__(self, num_arms, horizon):
```

```
        super().__init__(num_arms, horizon)
        # You can add any other variables you need here
        # START EDITING HERE
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)

        self.num_arms = num_arms
        self.num_plays = 0 # Keeps track of the total number
            of plays.
        self.pull_init_arm = -1 # Helper to track arm-number
            for the round-robin phase
        # END EDITING HERE

    def give_pull(self):
        # START EDITING HERE
        if self.num_plays < self.num_arms:
            self.num_plays += 1
            self.pull_init_arm += 1
            return self.pull_init_arm # Initially we will
                pull each arm once, before starting the KL-
                UCB loop (as per Garivier and Cappe, 2011,
                https://arxiv.org/abs/1102.2490)
        else:
            q_estimate = np.zeros(self.num_arms)
            for arm_idx in range(self.num_arms): # KL-UCB
                loop
                q_estimate[arm_idx] = binary_search(self.
                    values[arm_idx], self.counts[arm_idx],
                    self.num_plays)
            best_arm = np.argmax(q_estimate)
            self.num_plays += 1
            return best_arm
        # END EDITING HERE
```

Listing 2: Python code for give_pull() method for KL-UCB

1.4 Thompson Sampling

```
class Thompson_Sampling(Algorithm):
    def __init__(self, num_arms, horizon):
        super().__init__(num_arms, horizon)
        # You can add any other variables you need here
        # START EDITING HERE
        self.alpha = np.ones(num_arms) # Setting a uniform
            prior for each arm.
        self.beta = np.ones(num_arms)
        # END EDITING HERE

    def give_pull(self):
```

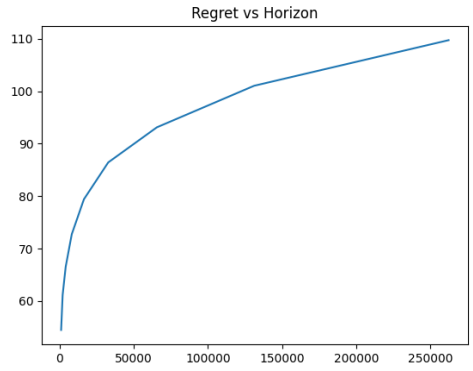

Figure 4: As compared to Epsilon-Greedy, UCB and KL-UCB, Thompson sampling gives the optimal regret.

```
        # START EDITING HERE
        best_arm = np.argmax(np.random.beta(self.alpha, self
            .beta)) # Sample from beta for each arm, and
            choose best arm.
        return best_arm
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.alpha[arm_index] += reward # Update params
            based on reward
        self.beta[arm_index] += (1 - reward)
        # END EDITING HERE
```

Listing 3: Python code for Task 1: Thompson Sampling

Task 2

For the Costly Bandit setting, we incorporate Thompson sampling based on the findings of Task 1. Here, the modifications are towards the query set generation where we would like to balance out arm exploration and arm exploitation. For this, we start with a Uniform prior for every arm, with Beta parameters $\alpha = 1$ and $\beta = 1$. At every time step (t), we draw a sample (x_a^t) from the Beta distribution of every arm (a). We add to the query set, the arms with $x_a^t >= k$, where k is a threshold hyperparameter. To also favor arm exploration, we append to the query set, the arm indices whose Beta distribution variance (V) is $\geq 1e-2$.

```
class CostlySetBanditsAlgo(Algorithm):
```

```
def __init__(self, num_arms, horizon):
    # You can add any other variables you need here
    self.num_arms = num_arms
    self.horizon = horizon
    # START EDITING HERE
    self.counts = np.zeros(self.num_arms)
    self.values = np.zeros(self.num_arms)
    self.alpha = np.ones(num_arms) # Setting a uniform
        prior for each arm.
    self.beta = np.ones(num_arms)
    self.k = 0.5 # Threshilding hparam
    self.query_set = None
    # END EDITING HERE

def give_query_set(self):
    # START EDITING HERE
    samples = np.random.beta(self.alpha, self.beta)
    var = (self.alpha * self.beta) / ((self.alpha + self
        .beta + 1) * (self.alpha + self.beta) ** 2)
    mean = (self.alpha) / (self.alpha + self.beta)
    self.query_set = [i for i, s in enumerate(samples)
        if s >= self.k] # Select arms where sample from
        beta is greater or equal to 0.5 for exploitation
    self.query_set.extend([i for i, s in enumerate(
        samples) if var[i] > 1e-2]) # We also select arms
         with high variance for exploration
    self.query_set = list(set(self.query_set))
    if len(self.query_set) == 0:
        self.query_set = np.argsort(-samples)[:self.
            num_arms // 2].tolist() # In case, query set
            is empty, we take top-k arms where k =
            num_arms / 2
    return self.query_set
    # END EDITING HERE

def get_reward(self, arm_index, reward):
    # START EDITING HERE
    self.alpha[arm_index] += reward # Update params
        based on reward
    self.beta[arm_index] += (1 - reward)
    #END EDITING HERE
```

Listing 4: Python code for Task 2

Task 3

We modify the Epsilon-Greedy algorithm provided in Task 1, to include the initial round-robin phase, where each arm is pulled once before the epsilon-greedy loop. The code for running a single simulation or multiple simulations

remains largely the same as mentioned in simulator.py except for the fact that Horizon (T) is fixed to 30000 and Epsilon is varied incrementally from 0 to 1 with a step size of 0.01. Figure 5 showcases the trade-off between Regret and Epsilon for the Epsilon-Greedy Algorithm.

```
class VaryingEpsGreedy(Algorithm):
    def __init__(self, num_arms, horizon, eps):
        super().__init__(num_arms, horizon)
        self.eps = eps
        self.counts = np.zeros(num_arms)
        self.values = np.zeros(num_arms)
        self.num_plays = 0 # Keeps track of the total number
            of plays.
        self.pull_init_arm = -1 # Helper to track arm-number
            for the round-robin phase

    def give_pull(self):
        if self.num_plays < self.num_arms: # Initially we
            will pull each arm once
            self.num_plays += 1
            self.pull_init_arm += 1
            return self.pull_init_arm
        else:
            if np.random.random() < self.eps: # Epsilon-
                greedy loop
                self.num_plays += 1
                return np.random.randint(self.num_arms)
            else:
                self.num_plays += 1
                return np.argmax(self.values)

    def get_reward(self, arm_index, reward):
        self.counts[arm_index] += 1
        n = self.counts[arm_index]
        value = self.values[arm_index]
        new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.values[arm_index] = new_value

def single_sim_task3(seed=0, ALGO=Algorithm, PROBS=[0.7,
    0.6, 0.5, 0.4, 0.3], HORIZON=30000, EPS=0.1):
    np.random.seed(seed)
    shuffled_probs = np.random.permutation(PROBS)
    bandit = BernoulliBandit(probs=shuffled_probs)
    algo_inst = ALGO(num_arms=len(shuffled_probs), horizon=
        HORIZON, eps=EPS)
    for t in range(HORIZON):
        arm_to_be_pulled = algo_inst.give_pull()
        reward = bandit.pull(arm_to_be_pulled)
        algo_inst.get_reward(arm_index=arm_to_be_pulled,
            reward=reward)
```

```
    return bandit.regret()

# DEFINE simulate_task3() HERE
def simulate_task3(algorithm, probs, horizon, eps, num_sims
    =50):
    def multiple_sims(num_sims=50):
        with Pool(10) as pool:
            sim_out = pool.starmap(single_sim_task3,
                [(i, algorithm, probs, horizon, eps) for i
                    in range(num_sims)])
        return sim_out

    sim_out = multiple_sims(num_sims)
    regrets = np.mean(sim_out)

    return regrets

# DEFINE task3() HERE
def task3(algorithm, probs, num_sims=50):
    eps_range = [i for i in np.arange(0, 1.01, 0.01)] #
        Epsilon step-size 0.01
    regrets = []
    for eps in eps_range:
        regrets.append(simulate_task3(algorithm=algorithm,
            probs=probs, horizon=30000, eps=eps, num_sims=
            num_sims)) # Fixed horizon (T) = 30000

    print(f"Epsilon value corresponding to least Regret: {
        eps_range[np.argmin(np.array(regrets))]}")
    plt.plot(eps_range, regrets)
    plt.title("Regret vs Epsilon for finite Horizon (T) =
        30000")
    plt.savefig("task3-{}-{}.png".format(algorithm.__name__,
         time.strftime("%Y%m%d-%H%M%S")))
    plt.clf()

# Call task3() to generate the plots
task3(VaryingEpsGreedy, probs=[0.7, 0.6, 0.5, 0.4, 0.3])
```

Listing 5: Python code for Task 3

Figure 5: We observe the lowest regret at $\epsilon = 0.04$. In Epsilon-Greedy, when $\epsilon = 0$, the algorithm runs only in the exploitation state, pulling the best arm based on the empirical mean. However, the empirical means are not calibrated in the absence of an exploration phase, resulting in a sub-optimal regret. Similarly, when $\epsilon = 1$, the algorithm always pulls arms uniformly at random, without taking into account the arm's running empirical mean, leading to a sub-optimal regret. The cumulative regret at $\epsilon = 0$ is lower than regret at $\epsilon = 1$ because of the initial round-robin phase, where each arm is pulled once to set the initial empirical means. Hereafter, the algorithm keeps pulling the best arm (even if sub-optimal), leading to lower regret than choosing any arm at random at each time step.