Statistical Techniques in Robotics 16-831 Sping 2021 - The Robotics Institute

Homework 3: Multi-Armed Bandits

Deadline: March 31th, 2021 11:59 PM

Instructor: Kris Kitani TA: Xingyu Lin

Total points: 100

1 Introduction

A multi-armed bandit problem [1] is a sequential allocation problem defined by a set of actions. At each time step, a unit resource is allocated to an action and some observable payoff is obtained. The goal is to maximize the total payoff obtained in a sequence of allocations. The name bandit refers to the colloquial term for a slot machine: a "one-armed bandit". In a casino, a sequential allocation problem is obtained when the player is facing many slot machines at once (a "multi-armed bandit") and must repeatedly choose where to insert the next coin.

We will begin by understanding why naively applying Generalized Weighted Majority does not work in the bandit setting. We will then work through two classes of bandit algorithms: one that makes assumptions about nature (stochastic bandits), and one that does not make such assumptions (adversarial bandits). The goal of this homework is to understand the behaviour of these algorithms and also understand which one to use in a real-world application.

1.1 Instructions

Submit this homework on Gradescope. Make sure your Gradescope student account is made using your **Andrew email ID** and your official name in the CMU system - this is essential to correctly record grades. Remember, you can only use a maximum of 2 late days (out of 5 for the semester) for any assignment. Beyond that, you will be penalized by a 1/3rd deduction in points per day. Detailed instructions on what to submit are given in Section 8.

2 Framework (7 points)

2.1 Notation

We will first define a framework to study bandit algorithms. The two main components of this framework are the **Game** and the **Policy**.

A **Game** is defined as follows. There is a set of actions $n = 1, 2, \dots N$. The game proceeds in rounds. Each round is indexed by a timestep $t \in \{1, 2, \dots, T\}$. At time step t, each action n is associated with a reward $g_n^t \in [0, 1]$ determined a priori by the game. This implies that the game creates a table of rewards.

$$G = \begin{bmatrix} g_1^1 & g_1^2 & \cdots & g_1^T \\ g_2^1 & g_2^2 & \cdots & g_2^T \\ \vdots & \vdots & \ddots & \vdots \\ g_N^1 & \cdots & \cdots & g_N^T \end{bmatrix}_{N \times T} \tag{1}$$

The game is played by a **Policy**. The policy is determined by a bandit algorithm. At every time step t, the policy selects an action a^t. The game provides a reward from its table of rewards $g_{a^t}^t$. The regret of a policy is defined as

$$R^T = \max_{i=1,\dots,N} \sum_{t=1}^{T} g_i^t - \sum_{t=1}^{T} \mathbb{E}_{a^t}\ g_{a^t}^t \tag{2}$$

2.1.1 Connection to PWEA terminology

- Trials —In PWEA we used the term *trial.* Each round of the Game is a trial.
- Prediction —In PWEA we used the term *prediction.* This is equivalent to an Action.
- Online Algorithm —In PWEA we used various online algorithms like Greedy, Halving etc. These are equivalent to policies in this setting.

2.2 Programming

You can use either `Matlab` or `Python` (with numpy, scipy, matplotlib, etc.) for this assignment. Skeleton code has been provided for both languages to help you get started. We strongly encourage you to follow this skeleton code. `Matlab` will be used as the running example in this handout, but the equivalent functions can also be found in the `Python` skeleton as well.

2.3 Files

For matlab, you have been provided with the code that contains the following files

- `Game.m` —abstract game class
- `gameConstant.m` —game where rewards are constant
- `gameGaussian.m` —game where rewards are drawn from Gaussian
- `gameAdversarial.m` —game where rewards are an adversarial sequence
- `gameLookupTable.m` —game where rewards are read from an external table
- `Policy.m` —abstract policy class
- `policyConstant.m` —policy chooses a constant action
- `policyRandom.m` —policy chooses actions randomly
- `policyGWM.m` —policy is GWM
- `policyEXP3.m` —policy is EXP3
- `policyUCB.m` —policy is UCB
- `simpleDemo.m` —script to apply policy on a game
- `data/` —folder containing datasets

2.4 Game

The file `Game.m` defines an abstract class Game. This class defines common functionality for a game. The member variables of the class are as follows.

```
nbActions   % number of actions
totalRounds % number of rounds of the game
tabR        % table of rewards (nbActions x totalRounds)
N           % counter for the current round of the game
```

A concrete class that inherits from this class will define values for each of these variables. The member functions are as follows. The first three functions have already been filled in for you.

```
function [reward, action, regret] = play(self, policy)
    %   The function simulates the entire game where at each time
    %   step it class the policy to get an action, evaluates reward
    %   for that action and also evaluates regret for the policy
    %   till the timestep.

function resetGame(self)
    %   Resets the current counter to the beginning

function r = reward(self, a)
    %   Returns the reward for an action a

function r = cumulativeRewardBestActionHindsight(self)
    %   Returns the cumulative reward of the best fixed action in
    %   hindsight
```

(3 points) Code: Fill in the function `cumulativeRewardBestActionHindsight`. This will be used to calculate the regret based on Equation 2. Make sure to read the code and understand how this function will be used.

In the coming sections, we will define a variety of games. In each case, we will create a new class that inherits from this class. As an example for a concrete class, examine the file `gameConstant.m`. This is a game with only 2 actions which have constant rewards for all time steps. We will use this class to sanity check our implementations.

2.5 Policy

The file `Policy.m` defines an abstract class Policy. This class defines common functionality for a policy. This class has no member variables. The member functions are all abstract and are as follows

```
function init(self, nbActions);
% Initialize the policy with number of actions. This function is
% called before a game is played

function a = decision(self);
% Choose an action at the current round

function getReward(self, reward);
% Receive a reward for the chosen action and update internal model.
```

In the coming sections, you will define a variety of policies. In each case, we will create a new class that inherits from this class. As an example for a concrete class, examine the file `policyConstant.m`. In this case the policy decides an action from the beginning and sticks to it. Also examine `policyRandom.m` which randomly selects an action at every time step.

2.6 Application

We will now apply a policy on a game. Use the file `simpleDemo.m`.

(2 points) Code: Generate two plots: (1) regret over time for both policies (in different colors), and (2) actions taken over time, for both policies. Be sure to include a legend.

(2 points) Explain:

- Does regret rise over time for policyRandom? What if you try multiple random restarts? Do you always get the same results? Why?
- Does regret rise over time for policyConstant? What if you try multiple random restarts? Do you always get the same results? Why?

We have now gone through the whole framework. In subsequent sections we will make new policies and games and make similar plots to have a better understanding.

3 EXP3 (Exponential Weights for Exploration and Exploitation) (39 points)

We learned in class that the EXP3 algorithm was a modification to the Generalized Weighted Majority (GWM) algorithm to work in the Bandit setting. The GWM algorithm is an extension of the Randomized Weighted Majority (RWM) algorithm in that we consider a continuous loss as opposed to the zero-one loss.

An action n at time t receives a loss $l_n^t \in [0,1]$. The weights of the experts are updated as $w_n^{t+1} = w_n^t \exp(-\eta l_n^t)$ every iteration. The rest of the algorithm remains the same. It is straightforward to go from reward g_n^t to loss l_n^t by applying the following transformation:

$$l_n^t = 1 - g_n^t \tag{3}$$

We will also use this conversion from rewards given by the game to loss while we develop the EXP3 policy in the loss setting.

3.1 Generalized Weighted Majority for bandits

Here, we have the pseudo code for the GWM algorithm (Alg. 1). We will apply it in the bandit setting.

Algorithm 1: Generalized Weighted Majority (GWM)

```
1 $w_n^1 \leftarrow 1$;
2 for $t = 1, 2, \ldots, T$ do
3     $p_n^t \leftarrow \frac{w_n^t}{\sum_{i=1}^N w_n^t}$ ;
4     $a^t \leftarrow$ Multinomial($p_n^t$);
5     $l^t \leftarrow$ GetLoss();
6     $w_n^{t+1} = w_n^t e^{-\eta^t l_n^t}$;
```

The regret R^T for GWM is:

$$\begin{aligned} R^T &= \sum_{t=1}^T \mathbb{E}_{a^t \sim p_n^t}\, l_{a^t}^t - \sum_{t=1}^T l_{n^*}^t \\ &\leq \sum_{t=1}^T \epsilon l_{n^*}^t + \frac{\log N}{\epsilon} \end{aligned} \tag{4}$$

where $\epsilon \in (0, 0.5)$.

GWM is a no-regret algorithm in the online setting where the game reveals the entire vector l^t, i.e, all the losses incurred by all the actions at timestep t. However in the bandit setting only $l_{a^t}^t$ is revealed. This is equivalent to receiving a loss vector $\hat{l}_n^t$ where

$$\hat{l}_n^t = l_n^t \cdot \mathbb{1}_{a^t=n} \tag{5}$$

We will implement GWM with such a loss vector. The file `policyGWM.m` and `policy.py` contains the concrete class derived from Policy that has to be filled up. The comments contain a general guide as to what each function should implement. Since we are doing an *anytime* version of the algorithm where the time horizon is not known to GWM, use $\eta^t = \sqrt{\frac{\log N}{t}}$

3.1.1 Implement GWM

(5 points) Code: Complete `policyGWM.m` or `class policyGWM(Policy)` in `policy.py`. Test it on the constant game. Plot the regret and the actions chosen.

(3 points) Explain:

- Did the GWM algorithm work well here?
- Does it score better than the random policy?
- Is the GWM algorithm no regret? If so, under what assumptions?

You will notice that it is a loose regret bound. This is clearly due to GWM using the loss vector $\hat{l}_n^t$. Now we define a *scaled loss.* If the probability distribution over actions maintained by GWM at time t is p_n^t then the scaled loss is defined as:

$$\tilde{l}_n^t = \frac{l_n^t}{p_n^t} \mathbb{1}_{a^t=n} \tag{6}$$

3.1.2 Unbiased estimator of regret

(4 points) Derive: Show that $\sum_{t=1}^{T} \tilde{l}_n^t$ is an unbiased estimator (in expectation over the actions selected) of $\sum_{t=1}^{T} l_n^t$ for any action n. An estimator $\hat{Y}$ of some quantity Y is said to be unbiased if $E[\hat{Y}] = Y$.

Hints: Try to develop some intuition about what is asked to be proved. We want to estimate the sum of the losses for all time steps of any one action. Lets pick action 1 to be concrete. Action 1 only has a probability of being selected p_1^t at time step t. When it is selected we observe its true loss, and when its not selected we pretend its loss is 0.

Algorithm 2: EXP3

```
1 $w_n^1 \leftarrow 1$, for $n = 1, \ldots, N$;
2 for $t = 1, 2, \ldots, T$ do
3     $p_n^t \leftarrow \frac{w_n^t}{\sum_{i=1}^N w_n^t}$ ;
4     $a^t \leftarrow$ Multinomial($p_n^t$);
5     $l_{a^t}^t \leftarrow$ GetLoss();
6     $\tilde{l}_{a^t}^t \leftarrow \frac{l_{a^t}^t}{p_{a^t}^t}$;
7     $w_{a^t}^{t+1} = w_{a^t}^t e^{-\eta^t \tilde{l}_{a^t}^t}$;
```

3.2 Derive the Regret Bound for EXP3

We show the regret bound for EXP3 using the reward. In this section we will derive the regret bound for EXP3 using the slightly different notation with the loss $\tilde{l}_{a_t}^t$ at each time step. The pseudo code is shown in Algorithm 2. To simplify the derivation, we assume that η is fixed during learning for deriving the bound but you will use a time dependent η_t for the code implementation later.

Recall the pseudo-regret is defined as follows:

$$\bar{R} = \max_n \mathbb{E}\Big[\sum_{t=1}^T \big(l_{a_t}^t - l_n^t\big)\Big].$$

We will prove that

$$\bar{R} \le \sqrt{2TN \log N}.$$

We will follow the general strategy of first defining the potential function then get the upper bound and lower bound of the potential function, and finally chain the upper bound and the lower bound to get the regret bound.

3.2.1 Upper bound the potential function

Denote the sum of the weights at step t as $z^t = \sum_n w_n^t$. We define the potential function as $\Phi = \ln \frac{z^{t+1}}{z^1} = \sum_{t=1}^T \ln \frac{z^{t+1}}{z^t}$

(2 points) Derive: Show that the potential function can be upper bounded by

$$\Phi \le \sum_{t=1}^T \Big(-\eta \sum_n p_n^t \tilde{l}_n^t + \frac{\eta^2}{2} \sum_n p_n^t (\tilde{l}_n^t)^2 \Big)$$

Hints: The following inequalities can be useful: $\forall x \le 0, e^x \le 1 + x + \frac{x^2}{2}$ and $\ln(1+x) \le x$.

3.2.2 Lower bound the potential function

(2 points) Derive: Show that the potential function can be lower bounded by

$$\Phi \geq -\ln N - \eta \sum_{t=1}^{T} \tilde{l}_j^t, \forall j$$

3.2.3 Chain upper and lower bounds

(3 points) Derive: Combined the upper bound and lower bound above, and show that

$$\max_j \mathbb{E}\Big[\sum_t l_{a^t}^t\Big] - \sum_t l_j^t \leq \frac{\ln N}{\eta} + \frac{\eta NT}{2}$$

Apply expectation on both sides:

$$\mathbb{E}\Big[\sum_t l_{a^t}^t\Big] - \sum_t l_j^t \leq \frac{\ln N}{\eta} + \frac{\eta}{2}\sum_t \sum_n \mathbb{E}\Big[(l_n^t)^2\Big], \quad \forall j$$

$$\mathbb{E}\Big[\sum_t l_{a^t}^t\Big] - \sum_t l_j^t \leq \frac{\ln N}{\eta} + \frac{\eta}{2} NT, \quad \forall j$$

We then obtain

$$\max_j \mathbb{E}\Big[\sum_t l_{a^t}^t\Big] - \sum_t l_j^t \leq \frac{\ln N}{\eta} + \frac{\eta NT}{2}.$$

3.2.4 Find optimal η

(3 points) Derive: Finally, derive the optimal η for the bound above and show that the final bound for EXP3 is

$$\bar{R} \leq \sqrt{2TN \ln N}$$

3.3 Implementation

(5 points) Code: Complete your implementation of `policyEXP3.m` or the *class policyEXP3(Policy)* in `policy.py`. Test EXP3 on the constant game. Plot the regret. Plot the actions chosen. Interpret how EXP3 behaves from the plot. If you see this is not no-regret, check your answer in 2.4 with TAs or other students. Use the included comments as a guide for what each function should do. Choose $\eta^t = \sqrt{\frac{\log N}{tN}}$.

(2 points) Explain:

- How does EXP3 behave near the beginning of training?
- How does EXP3 behave near the end of training?

3.4 Gaussian Game

We will now test EXP3 on a more realistic game. In this game, the rewards for each action is drawn from a Gaussian distribution.

$$g_n^t \sim \mathcal{N}(\mu_n, \sigma_n^2) \tag{7}$$

The file `gameGaussian.m` and `game.py` contains the concrete class derived from Game that has to be filled up. Choose μ_n to be uniform random $[0, 1]$. Choose the standard deviation σ_n to be uniform random $[0, 1]$. Clip the sampled g_n^t to the range $[0, 1]$. Note that you should only sample n mean and variance (where n is the number of actions) instead of sampling a new mean for each time step.

3.4.1 Implement the Gaussian Game

(5 points) Code: Complete `gameGaussian.m` or *class gameGaussian(Game)* in `game.py`.

3.4.2 Test EXP3

(5 points) Code: Test EXP3 on the Gaussian game. Plot the regret. Create a Gaussian game with 10 actions that go on for 10^4 timesteps.

4 Upper Confidence Bound (UCB) Algorithm (22 points)

The EXP3 algorithm made no assumptions about how l_n^t was distributed. This property was also shared by the online algorithms that we looked at in the full information setting such as RWMA, GWM, etc.

However there is a large class of algorithms that do make such an assumption, i.e., treat the losses for an action to be i.i.d. One such approach is Upper Confidence Bound (UCB).

For the implementation of UCB we shall revert back to using rewards g_n^t to be consistent with the literature.

4.1 Optimism Under Uncertainty

UCB employs the principle of optimism in the face of uncertainty. This principle is a very useful heuristic to deal with the exploration exploitation trade-off. At each time step, despite the uncertainty in what actions are best, we will construct an optimistic estimate as to how good the expected reward of each action is, and pick the action with the highest estimate.

If the action incurs low reward, then the optimistic estimate will quickly decrease and we will be compelled to switch to a different action. If the action incurs a good reward, it results in exploitation of that action and we incur little regret. In this way, we balance exploration and exploitation.

4.2 Upper Confidence Bound

We will now use the principle of optimism under uncertainty to select the action at each time step. We assume that the loss of each action is i.i.d. At any time step, we have some confidence interval for the mean loss of an action based on the finite number of times we have selected it. We compare the upper confidence bound for all the actions and pick the best one.

4.3 Implementation

The UCB algorithm is as follows. Note that Lines 3-6 contain the initialization procedure: we pull each arm once to make sure that $C_i \neq 0$ when we enter into the main procedure in Line 8 to 11.

Algorithm 3: UCB

```
1  $S_i \leftarrow 0 \quad \forall i \in \{1, \ldots, N\}$;
2  $C_i \leftarrow 0 \quad \forall i \in \{1, \ldots, N\}$;
3  for $i = 1, 2, \ldots, N$ do
4      Pull arm $i$ and receive reward $g_i^0$ for arm $i$;
5      $S_i = g_i^0$;
6      $C_i = 1$;
7  for $t = 1, 2, \ldots, T$ do
8      $a^t \leftarrow \arg\max_{i=1,\ldots,N} \frac{S_i}{C_i} + \sqrt{\alpha \frac{\log t}{2C_i}}$;
9      $g_{a^t}^t \leftarrow$ GetReward();
10     $S_{a^t} \leftarrow S_{a^t} + g_{a^t}^t$;
11     $C_{a^t} \leftarrow C_{a^t} + 1$;
```

The parameter α is exploration-exploitation trade-off parameter. For this assignment, a good starting place is $\alpha = 1$.

The files `policyUCB.m` and `policy.py` contains the concrete class derived from Policy that has to be filled up. The comments contain a general guide as to what each function should implement.

(5 points) Code: Complete `policyUCB.m` or *class policyUCB(Policy)* in `policy.py`. Test UCB on the constant game. Plot the regret. Plot the actions chosen. Plot the confidence of each action.

(3 points) Explain:

- How does UCB behave near the beginning of training?
- How does UCB behave near the end of training?
- Which action is the policy most confident in?

4.4 Gaussian Game

(5 points) Code: Test UCB on the Gaussian Game. Plot the regret of UCB vs that of EXP3. (Be sure to include a legend.)

(2 points) Explain:

- Which policy performs better on average?
- Which policy performs better after 10,000 rounds?

4.5 Adversarial Game

We will now design an adversarial game.

(5 points) Code: Implement Adversarial Game by completing `gameAdversarial.m` or *class game Adverserial(Game)* in `game.py`. To keep things simple, choose only 2 actions. Let the game run for 1000 rounds. Generate plots with UCB and EXP3 together (in different colors), showing (1) actions chosen, and (2) regret over time. (Be sure to include a legend.)

(2 points) Explain:

- Briefly describe how your adversarial game works.
- Which policy performs better on average?
- Considering the algorithm that did better: why did it do better? What is the critical difference from the other algorithm?

5 Real Datasets (20 points)

We will now test these bandit algorithms on two real-world datasets. This will help us understand how these algorithms behave in practice.

5.1 Lookup Table Game

Before we proceed, we will have to define a game class that helps us easily test on databases. We will call this a "lookup table game." This will help us learn policies given a special matrix/table of loss values. In the table, each row corresponds to an action, and each column corresponds to a round, so that $table(i, j)$ indicates the loss of taking action i on round j.

(5 points) Code: Implement the lookup table game in `gameLookupTable.m` or the *class gameLookupTable(Game)* in `game.py`.

5.2 University Website Latency Dataset

This dataset corresponds to a real-world data retrieval problem where redundant sources are available. This problem is also commonly known as the Content Distribution Network problem (CDN). An agent must retrieve data through a network with several redundant sources available. For each retrieval, the agent selects one source and waits until the data is retrieved. The objective of the agent is to minimize the sum of the delays for the successive retrievals.

We will use a university website latency dataset from [2]. The dataset corresponds to the retrieval latencies of more than 700 university homepages. The pages have been probed every 10 mins for more than one week in May 2004 from an internet connection located in New York, NY, USA.

To fit into our bandit setting, these latencies been normalized to the interval $[0, 1]$. The normalized table is provided in the file `univLatencies.mat` in the folder `data/`. The rows of the matrix correspond to different universities. The columns correspond to different times. Note that the table is a collection of *losses*, so when this is passed to the constructor of `gameLookupTable`, the loss flag should be set to true.

(5 points) Code: Test EXP3, UCB, and the random policy on this dataset. Generate two plots: one with each algorithm's regret over time, and another with each algorithm's actions over time. (Be sure to include a legend.)

(3 points) Explain:

- How do UCB and EXP3 compare with each other?
- How do UCB and EXP3 each compare with the random policy?
- In the actions-over-time plot, there should be a visible difference between the action exploration strategy of UCB vs that of EXP3. What is this difference? Where does this come from (e.g., what line(s) in the algorithm)?

5.3 Planner Dataset

This dataset corresponds to a simulated 2D robot navigating at high speeds through an environment. The robot must use a planning algorithm from its database of planning algorithms to plan a path within a strict time budget. At each planning instance, the robot selects one planning algorithm and gets the cost of the path that is planned (which corresponds to loss).

The planning database (corresponding to actions) is created by selecting planners from OMPL with different parameters. The loss of a selected planner on an environment is the normalized cost of the planned path. The environments come from a non-stationary distribution. The robot first moves in a environment with sparse set of obstacles (favouring planning algorithms that aggressively collision check long edges) to a cluttered environment (favouring planning algorithms that do ordered search over smaller edges).

The normalized table is provided in the file `plannerPerformance.mat` in the folder `data/`. The rows of the matrix correspond to different planners. The columns correspond to different environments. Note that the table is a collection of *losses*, so when this is passed to the constructor of `gameLookupTable`, the loss flag should be set to true.

(5 points) Code: Test EXP3, UCB, and the random policy on this dataset. Plot actions over time, and regret over time.

(2 points) Explain:

- How do UCB and EXP3 compare with each other?
- How do UCB and EXP3 each compare with the random policy?

6 Incorporating context (12 pts)

Consider a robot with two jobs: (1) retrieving data from websites, and (2) navigating at high speeds through its environment.

6.1 Setup

(5 points) Code: Make a big "lookup table game" with *both* the website task and the planner task, so that on each round, the robot may be tasked with either navigation or website-retrieval. Set the loss to 1.0 for executing an irrelevant action. That is, if the robot executes a website-retrieval action during a navigation round, or executes a navigation action during a website-retrieval round, it should receive a loss of 1.0.

More specifically, merge the two tables, to make a new table with N1+N2 rows (corresponding to actions) and T1+T2 columns (corresponding to rounds). Fill the missing data with ones (since both source tables are collections of losses).

Shuffle the rounds, so that it is random whether the task is a website problem or a navigation problem.

Evaluate a UCB policy on this big game. (You will plot its performance in the next part.)

6.2 Context-aware bot

(5 points) Code: Make a context-aware version of this robot, so that it "knows" which task is being asked on each round, and executes a particular policy depending on the context. In other words, create a boolean variable that indicates the task (e.g., 0 = website lookup, 1 = planner), and use this to switch between two policies. Use UCB as before.

Evaluate your context-aware robot, and plot its results against the naive robot. As usual, include both regret over time and actions over time.

(2 points) Explain:

- Did the robot perform better when it was context-free, or context-aware? Why?

7 Bonus: Proof of Hoeffding's inequality (10 pts)

This inequality is key for analysing bandit problems. It is a good exercise to go through its proof. In the class we saw a simplified form of the inequality, here is the generalized form.

If $X_1, X_2, ..X_n$ are n independent random variables with $a_i \leq X_i \leq b_i$ for all i and $\bar{X} = (X_1 + .. + X_n)/n$, then for any $\epsilon > 0$,

$$Pr[|\bar{X} - E[\bar{X}]| \geq \epsilon] \leq 2e^{-2n^2\epsilon^2/\sum_{i=1}^{n}(b_i - a_i)^2}$$

As the proof is involved, you may assume other results like Markov inequality, Chernoff bounds and Jensen's inquality in your proof.

8 What to Submit

Your submission should consist of:

1. a zip file named `<AndrewId>.zip` consisting of a folder `code` containing all the code and data (if any) files you were asked to write and generate. Submit this to `Homework 3 - Programming` on Gradescope.

2. a `pdf` named `<AndrewId>.pdf` with the answers to the theory questions and the results, explanations and images asked for in the coding questions. **It is compulsory to type-set your answers; images of hand-written documents will not be accepted**. Submit this to `Homework 3 - Theory` on Gradescope. For easier grading, please use the Gradescope feature for tagging the pages and position for each question. 5 style points will be deducted from this homework if this is not followed.

9 Academic Integrity

You are encouraged to work together BUT you must do your own work (code and write up). If you work with someone, you must include their name in your write up and inside any code that has been discussed. If we find highly identical write-ups or code without proper accreditation of collaborators, we will take action according to university policies. If you use someone's material, you must give proper credit by including a citation where necessary

References

[1] S. Bubeck, and N. Cesa-Bianchi, “Regret analysis of stochastic and non-stochastic multi-armed bandit problems.” arXiv preprint arXiv:1204.5721 (2012).

[2] J. Vermorel, and M. Mohri,“Multi-armed bandit algorithms and empirical evaluation,” Machine Learning: ECML 2005. Springer Berlin Heidelberg, 2005. 437-448.