Statistical Techniques in Robotics 16-831 - The Robotics Institute

Project 2: MultiArmed Bandits

Deadline: October 19th, 2015 11:59 PM (submit via Blackboard)

1 Introduction

A multi-armed bandit problem (or, simply, a 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 ("one-armed bandit" in American slang). 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.

In project 1, we looked at the Prediction with Expert Advice (PWEA) problem. 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 (adversarial bandits). The goal of this project is to understand the behaviour of these algorithms and also understand which one to use in a real world application.

2 Framework (5 points)

2.1 Notation

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

A **Game** is defined as follows. There is a set of actions $n = 1, 2, \dots N$. The game proceeds in rounds indexed by time $t = 1, 2, \dots, T$. 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.

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

The game is played by a **Policy**. The policy is 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

$$R^T = \min_{i=1,\ldots,N} \sum_{t=1}^{T} g_i^t - \sum_{t=1}^{T} E_{a^t}\ g_{a^t}^t \tag{2}$$

2.1.1 Connection to PWEA terminology

- Trials —In PWEA we used the term trial. This is each round of the Game.
- Prediction —In PWEA we used the term prediction. This is equivalent to Action in this setting
- Online Algorithm —In PWEA we used various online algorithms like Greedy, Halving etc. This is equivalent to Policy in this setting.

2.2 Files

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

2.3 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

```
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
```

The first three functions have already been filled in for you.

2.3.1 Fill in the function cumulativeRewardBestActionHindsight (5 points)

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.

2.4 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.

2.5 Application

We will now apply a policy on a game. Examine the file `simpleDemo.m`. Plot the regret for both policies. Also plot the actions taken by both policies.

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) (50 points)

We learnt in class that the EXP3 algorithm was a modification to the Generalized Weighted Majority (GWM) algorithm to work in the Bandit setting. We will revisit this discussion here.

Before we proceed, note that we studied Generalized Weighted Majority in the loss setting. An action n at time t receives a loss l_n^t. It is straightforward to go from gain to loss by applying the following transformation

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

We will also develop EXP3 policy in the loss setting by converting the rewards given by the game to a loss.

3.1 Generalized Weighted Majority (GWM) in bandit setting

We will start by revisiting GWM (Alg. 1) and then proceed to apply it in the bandit setting. Recall that the difference between GWM and RWMA is that nature hands a loss vector where loss for each action (expert) is continuous in $[0, 1]$.

The regret R^T for GWM was derived in class to be

$$\begin{aligned} R^T &= \sum_{t=1}^{T} E_{a^t \sim p_n^t}\ l_{a^t}^t - \sum_{t=1}^{T} l_{n^*}^t \\ &\leq \sum_{t=1}^{T} \eta l_{n^*}^t + \frac{2 \log N}{\eta} \end{aligned} \tag{4}$$

Algorithm 1: Generalized Weighted Majority (GWM)

```
$w_n^1 \leftarrow 1$;
for $t = 1, 2, \ldots, T$ do
    $a^t \leftarrow$ Multinomial($p_n^t$);
    $l^t \leftarrow$ GetLoss();
    $w_n^{t+1} = w_n^t e^{-\eta^t l_n^t}$;
```

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

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

We will implement GWM with such a loss vector. The file `policyGWM.m` 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 by completing `policyGWM.m`. Test it on the constant game. Plot the regret and the actions chosen. Explain the behaviour of GWM (10 points)

So we see that GWM is not at all working! But we know that in the full information setting it is no-regret. Revisit the regret bound for GWM and confirm that it is no longer valid under the bandit setting.

3.1.2 Derive the regret bound for GWM in the bandit setting (10 points)

It is pretty clear that the loose regret bound is 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

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

3.1.3 Show that $\sum_{t=1}^{T} \tilde{l}^t_n$ is an unbiased estimator (in expectation over the actions selected) of $\sum_{t=1}^{T} l^t_n$ for any action n (5 points)

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^t_1 at time step t. When it is selected we observe its true loss, and when its not selected we pretend its loss is 0.

3.2 Implementation

The EXP3 algorithm (Alg. 2) is GWM applied to the scaled loss. The hope is that since we showed that the scaled loss is an unbiased estimator of the true loss, EXP3 should work.

Algorithm 2: EXP3

```
$w^1_n \leftarrow 1$;
for $t = 1, 2, \dots, T$ do
    $a^t \leftarrow$ Multinomial($p^t_n$);
    $l^t_{a^t} \leftarrow$ GetLoss();
    $\tilde{l}^t_{a^t} \leftarrow \frac{l^t_n}{p^t_n}$;
    $w^{t+1}_{a^t} = w^t_{a^t} e^{-\eta^t \tilde{l}^t_{a^t}}$;
```

The file `policyEXP3.m` 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. Choose $\eta^t = \sqrt{\frac{\log N}{tN}}$.

3.2.1 Implement EXP3 by completing `policyEXP3.m`. Test EXP3 on the constant game. Plot the regret. Plot the actions chosen. Interpret from the plot how EXP3 behaves. (10 points)

3.3 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.

$$g^t_n \sim \mathcal{N}(\mu_n, \sigma^2_n) \tag{7}$$

The file `gameGaussian.m` 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 $[0, 1]$. Remember to ensure that the sampled $g^t_n \in [0, 1]$.

3.3.1 Implement Gaussian Game by completing `gameGaussian.m` (5 points)

We will now test EXP3 on this game. Create a Gaussian game with 10 actions that go on for 10^4 timesteps.

3.3.2 Test EXP3 on the Gaussian Game. Plot the regret. (5 points)

3.4 Variance Issues

Even though we showed $\sum_{t=1}^{T} \tilde{l}_n^t$ is an unbiased estimator of the true losses, we did not examine the variance of the estimate.

3.4.1 Derive the variance of the estimator $\sum_{t=1}^{T} \tilde{l}_n^t$. Is the variance bounded? (5 points)

Hint: $\text{Var}(\text{X}) = \text{E}[\text{X}^2] - (E[X])^2$

Even though we will proceed with the vanilla version of EXP3, there are variants of EXP3 that fix this issue [1].

4 Upper Confidence Bound (UCB) Algorithm (30 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 tradeoff. 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'll be compelled to switch to a different action. But if the action had a good reward, it resulted 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.

We will use Hoeffding's inequality to come up with this upper confidence bound. Let $X_1, X_2, \ldots, X_m$ be i.i.d random variables in $[0, 1]$. Let μ be the true mean. Let $\hat{\mu} = \sum\limits_{i=1}^{m} X_i$ be the sample mean. Then Hoeffding's inequality states

$$P(\mu - \hat{\mu} \geq \varepsilon) \leq e^{-2m\varepsilon^2} \tag{8}$$

4.2.1 Show that the upper bound of the mean is the following with probability atleast $1 - \delta$ (5 points)

$$\mu \leq \sum_{i=1}^{m} X_i + \sqrt{\frac{\log(\delta^{-1})}{2m}} \tag{9}$$

Now we return to the bandit setting. Let the number of times an action n has been selected uptil a time step t be maintained by the counter C_n^t. Let μ_n^t be the true mean of rewards for action n. Let $\hat{\mu}_n^t$ be the sample mean of rewards for action n.

Assume we want to drive the probability that the confidence bound is correct to 1 as $t \rightarrow \infty$. We can do so by choosing $\delta = \frac{1}{t}$.

4.2.2 Show that the upper bound of the mean reward for an action n is the following (5 points)

$$\mu_n^t \leq \hat{\mu}_n^t + \sqrt{\frac{\log t}{2C_n^t}} \tag{10}$$

4.3 Implementation

The UCB algorithm is as follows

The file `policyUCB.m` 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.

Algorithm 3: UCB

```
$S_n \leftarrow 0$;
$C_n \leftarrow 0$;
for $t = 1, 2, \ldots, T$ do
    $a^t \leftarrow \arg\max_{i=1,\ldots,N} \frac{S_n}{C_n} + \sqrt{\frac{\log t}{2C_n^t}}$;
    $g_{a^t}^t \leftarrow \texttt{GetReward}()$;
    $S_{a^t} \leftarrow S_{a^t} + g_{a^t}^t$;
    $C_{a^t} \leftarrow C_{a^t} + 1$;
```

4.3.1 Implement UCB by completing `policyUCB.m`. Test UCB on the constant game. Plot the regret. Plot the actions chosen. Plot the upper confidence of both actions. Interpret from these plots how UCB behaves. (10 points)

4.4 Gaussian Game

We will now test UCB on the Gaussian game that we created in the previous section to test EXP3.

4.4.1 Test UCB on the Gaussian Game. Plot the regret of UCB vs that of EXP3. Explain the performance difference (5 points)

4.5 Adversarial Game

UCB is a deterministic algorithm. We studied in class that such algorithms are vulnerable to deterministic sequences. We will now design such an adversarial game.

To keep things simple choose only 2 actions. The game must run for atleast 1000 rounds.

4.5.1 Implement Adversarial Game by completing `gameAdversarial.m`. Plot the actions chosen by UCB and EXP3. Compare the regret. Interpret the plots. (5 points)

5 Real Datasets (15 points)

We will now test these bandit algorithms on two 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 where the class is given a table of rewards as input.

The file `gameLookupTable.m` contains the concrete class derived from Game that has to be filled up.

5.1.1 Implement the lookup table game in `gameLookupTable.m` (5 points)

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 have to be first normalized in 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*. When this is passed to the constructor of `gameLookupTable`, the loss flag must be set to true.

5.2.1 Test EXP3 and UCB on this dataset. Plot the corresponding regrets. (5 points)

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*. When this is passed to the constructor of `gameLookupTable`, the loss flag must be set to true.

5.3.1 Test EXP3 and UCB on this dataset. Plot the corresponding regrets. (5 points)

6 What to Submit

Your submission should consist of only 1 file, a zip file named `<AndrewId>.zip`. This zip file should contain

- a folder `code` containing all the code and data (if any) files you were asked to write and generate.
- a `pdf` named `writeup.pdf` containing the results, explanations and images asked for in the coding section and the answers to the theory questions.

References

[1] S. Bubeck, and N. Cesa-Bianchi, “Regret analysis of stochastic and nonstochastic 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.