lab09_student_version

November 8, 2021

Lab 9: Bayesian and Frequentist Takes on Multi-Armed Bandits

Welcome to the ninth DS102 lab!

The goals of this lab is to implement and gain a better understanding of the pros and cons of the Upper Confidence Bounds (UCB) and Thompson Sampling algorithms for the multi-armed bandits problem.

The code you need to write is commented out with a message "TODO: fill in". There is additional documentation for each part as you go along.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the labs, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** in the cell below.

Gradescope Submission

To submit this assignment, rerun the notebook from scratch (by selecting Kernel > Restart & Run all), and then print as a pdf (File > download as > pdf) and submit it to Gradescope.

This assignment should be completed and submitted before Wednesday, Nov 10th, 2021 at 11:59 PM. PST

Collaborators

Write the names of your collaborators in this cell.

`<Collaborator Name> <Collaborator e-mail>`

[1]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import seaborn as sns
from matplotlib.widgets import Button, CheckButtons
from matplotlib import gridspec
import functools
from Bandit_env import BanditEnv,␣
↪Interactive_UCB_Algorithm,Interactive_TS_Algorithm
```

```
%load_ext autoreload
%autoreload 2
%matplotlib notebook
```

Multi-Armed Bandits (MAB)

In this lab we will be implementing two of the most common approaches to solving stochastic Multi-Armed Bandit (MAB) problems. We first define the problem and then you will have a chance to implement the Upper Confidence Bound (UCB) algorithm and the Thompson Sampling (TS) algorithm from lecture and analyze their performance.

Setup:

A MAB problem is a simple setting in which it is easy to analyze the **exploration/exploitation** tradeoff that is extremely common in machine learning. The setup is as follows:

A MAB instance is a set $\mathcal{A}$ of K arms. Each arm $a \in \mathcal{A}$ is associated with a reward distribution $X_a \sim \mathbb{P}_a$ which is unique to that arm. The mean of an arm $a \in \mathcal{A}$ is denoted as $\mu_a = \mathbb{E}[X_a]$.

At each time $t = 1, 2, ...,$ an algorithm that is interacting with a MAB must choose an arm, $A_t \in \mathcal{A}$. The algorithm then receives a reward $X_{A_t}^{(t)}$ sampled independently from $\mathbb{P}_{A_t}$.

The **goal** of an algorithm interacting with a MAB instance is to find the arm A^* with the highest mean reward μ^* as fast as possible while maintaining performance. That is, it would like to find the single optimal arm A^* such that:

$$A^* = arg\max_{a \in \mathcal{A}} \mu_a$$

where $\mu^* = \mu_{A^*}$.

This is often encoded as wanting to find an algorithm that minimizes the **regret** over a time horizon T. Intuitively, the regret is the best the algorithm could have done in hindsight if it had known which was the optimal arm (A^*). The regret of an algorithm is defined as:

$$Regret(T) = \sum_{t=1}^{T} X_{A^*}^{(t)} - X_{A_t}^{(t)}$$

Most of the time, it is simpler to analyze the **pseudo-regret**, which is the mean of the regret.

$$R_T = T\mu^* - \mathbb{E}\left[\sum_{t=1}^{T} X_{A_t}^{(t)}\right]$$

Lab setup:

In this lab, the MAB instances will have a set of arms numbered $0, 1, ..., K-1$. Each arm $a = 0, 1, ..., K-1$ is associated with a Gaussian reward distribution with mean μ_a and standard deviation of $\sigma_a = 1.5$. To be able to analyze the various algorithms, the optimal arm A^* will always be arm 0, and its mean will always be $\mu^* = 10$.

By running the following cells, you can interact with a MAB instance of the type we will be using in this lab. You can see the reward distributions as well as the expected cumulative regret you incur when pulling each arm.

Verify for yourself that explore-then-commit strategies can get stuck pulling the wrong arm.

[2]:
```python
# Run this cell to initialize the parameters for the arms that we will be pulling from.

# Mean reward for each arm. Arm 0 has the highest mean, but the algorithm doesn't know that yet.
means=[10,9,8,7,6,4]

# Variance of the reward for each arm.
variance=1.5
standard_deviations=[np.sqrt(variance) for arm in range(len(means))]

# Initialize the interactive environment for pulling arms.
bandit_env=BanditEnv(means,standard_deviations)
```

[3]:
```python
"""
Creates an interactive bandit instance.

    - Pull an arm by clickling on the colored button
    - The true means of the distributions are shown with the dashed horizontal lines
    - Large solid circle is the sample mean of the arm
    - Small empty circle is a sample from the arm
    - The reward distribution of each arm is shown on the right and can be toggled on/off by checking the box
    - Running Pseudo-regret is shown on the bottom and can be toggled on/off by checking the box
"""

# You may need to rerun this cell to restart the gui

%matplotlib notebook
plt.rcParams['figure.figsize']=[7,6]
bandit_env.run_Interactive()

# You can click on the arms to see how it selecting sub-optimal arms accumulates regret
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/home/jovyan/fa21/lab/lab09/Bandit_env.py:70: MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be
removed two minor releases later. Use manager.set_window_title or GUI-specific
methods instead.
  self.figure.canvas.set_window_title('Bandits Demo')
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
```

```
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
```

```
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
```

Question 1. The Frequentist Approach: Upper Confidence Bounds (UCB)

The first algorithm we will analyze is the frequentist take on multi-armed bandits, known as the Upper Confidence Bounds (UCB) algorithm.

For each arm $a \in \{0, 1, ..., K-1\}$, you keep track of:

1. $T_a(t)$: the number of times arm a has been pulled up to and including iteration t.
2. $X_a^{(1)}, ..., X_a^{(T_a(t))}$: the samples you have received from arm a. Let $\hat{\mu}_{a,T_a(t)}$ be the mean of those samples: $\hat{\mu}_{a,T_a(t)} = \frac{1}{T_a(t)} \sum_{i=1}^{T_a(t)} X_a^{(i)}$

Using this information, you compute an upper confidence bound, $C_a(T_a(t), \delta)$ that encompasses the true mean μ_a with probability at least $1-\delta$, for some $\delta \in [0,1]$. $C_a(T_a(t), \delta)$, must therefore satisfy:

$$\mathbb{P}\Bigg(\mu_a < \hat{\mu}_{a,T_a(t)} + C_a(T_a(t), \delta)\Bigg) > 1 - \delta.$$

As an edge case, after 0 samples, we simply set the upper bound on μ_a to ∞, since it's always true that $\mathbb{P}(\mu_a < \infty) > 1 - \delta$.

The algorithm then pulls, at each round t, the arm with the highest upper confidence bound based on the results we saw up to time $t-1$:

$$A_t = \underset{a \in \{0,1,\ldots,K-1\}}{\operatorname{argmax}} \hat{\mu}_{a,T_a(t-1)} + C_a(T_a(t-1), \delta).$$

1a. The UCB algorithm

We will now implement the classic version of the UCB algorithm using the Hoeffding bound that we derived in lecture.

In lecture, we used the Hoeffding bound for the sample mean of bounded random variables. Here instead we will use a related bound for the sample mean of normally distributed random variables. The sample mean of Gaussian random variables $X_a^{(1)}, ..., X_a^{(T_a(t))}$, $X_a^{(i)} \sim \mathcal{N}(\mu_a, \sigma_a)$, satisfies the following form of the Hoeffding Inequality:

$$P(\hat{\mu}_{a,T_a(t)} - \mu_a \leq -\epsilon) \leq e^{\frac{T_a(t)\epsilon^2}{2\sigma_a^2}}.$$

Take a minute to compare this bound with the bounds discussed in lecture.

This bound results in the upper confidence bound on μ_a:

$$P\left(\mu_a < \hat{\mu}_{a,T_a(t)} + \sigma_a\sqrt{\frac{2\log 1/\delta}{T_a(t)}}\right) > 1-\delta.$$

where $\hat{\mu}_{a,T_a(t)}$ is the current sample mean for arm a:

$$\hat{\mu}_{a,T_a(t)} = \frac{1}{T_a(t)}\sum_{i=1}^{T_a(t)} X_a^{(i)}$$

and the confidence bound term added to $\hat{\mu}_{a,T_a(t)}$ is:

$$C_a(T_a(t),\delta) = \sigma_a\sqrt{\frac{2\log 1/\delta}{T_a(t)}}.$$

To handle the edge case where we've seen 0 samples from arm a so far (i.e. $T_a(t-1)=0$), we set the upper bound on μ_a to ∞. Specifically, we set

$$C_a(T_a(t),\delta) = \begin{cases} \infty & \text{if } T_a(t)=0 \\ \sigma_a\sqrt{\frac{2\log 1/\delta}{T_a(t)}} & \text{if } T_a(t)>0 \end{cases}$$

and

$$\hat{\mu}_{a,T_a(t)} = \begin{cases} \infty & \text{if } T_a(t)=0 \\ \frac{1}{T_a(t)}\sum_{i=1}^{T_a(t)} X_a^{(i)} & \text{if } T_a(t)>0 \end{cases}$$

Finally, as mentioned earlier, we choose the arm A_t at time t as follows:

$$A_t = \underset{a\in\{0,1,\dots,K-1\}}{\operatorname{argmax}}\ \hat{\mu}_{a,T_a(t-1)} + C_a(T_a(t-1),\delta).$$

We will choose a δ that decreases with time to ensure that we will explore the arms at first:

$$\delta = \frac{1}{t^3}$$

`TODO`: Now, use this formula for A_t to fill out the following function which returns the choice of arm as well as the upper confidence bounds of each arm. In the code below, we use the variable "confidence_bound" to refer to the entire term $\hat{\mu}_{a,T_a(t)}+C_a(T_a(t),\delta)$.

[5]:
```
# TODO: complete the function
def UCB_pull_arm(t, standard_deviations, times_pulled, rewards):
    """ Implement the choice of arm for the UCB algorithm

    Inputs:
        t : int, current iteration
        standard_deviations : a list of length K (where K is the number of␣
↪arms) of the
```

```
            standard deviations associates with each arm
        times_pulled: a list of length K (where K is the number of arms) of the␣
↪number
            of times each arm has been pulled.
        rewards: a list of K lists. Each of the K lists holds the samples␣
↪received from
            pulling each arm up to iteration t.

    Outputs:
        arm: an integer representing the arm that the UCB algorithm would␣
↪choose.
        confidence_bounds: a list of the confidence bounds for each arm
    """

    K = len(times_pulled)
    delta = 1/(t**3)

    confidence_bounds=[]
    for arm in range(K):
        if times_pulled[arm]==0:
            c_bound = np.inf
            confidence_bounds.append(c_bound)
        else:
            delta = 1/(t**3)
            c_bound = np.mean(rewards[arm]) + standard_deviations[arm]*np.
↪sqrt(2*np.log(1/delta)/times_pulled[arm])
            # TODO: fill in
            # Hint: the \hat\mu_{a, T_a(t-1)} value is the mean of␣
↪rewards[arm],
            #       and the T_a(t-1) value is equal to times_pulled[arm].
            confidence_bounds.append(c_bound)

    arm = np.argmax(confidence_bounds)

    return arm, confidence_bounds
```

[6]:
```
# Validation tests, do not modify
K_test = 5
times_pulled_test = [3, 5, 7, 4, 0]
t_test = np.sum(times_pulled_test) + 1
standard_deviations_test = [0.4, 0.2, 0.1, 0.2, 0.5]
rewards_test = [[10.4, 10.6, 11],
                [8, 13, 12, 11, 9],
                [9, 10, 10, 8, 9.5, 10.5, 11],
                [8.3, 9.6, 7.9, 8.1],
                []]
```

```
test_arm, test_confidence_bounds = UCB_pull_arm(t_test,␣
↪standard_deviations_test, times_pulled_test, rewards_test)
assert test_arm == 4
assert np.isinf(test_confidence_bounds[-1])
opt_vals = [11.64, 10.98, 9.87, 8.90]
for a in range(K_test-1):
    assert (np.abs(opt_vals[a] - test_confidence_bounds[a]) <= 0.1)
print("Test Passed!")
```

```
Test Passed!
```

Given the function you have filled out, let us investigate the pseudo-regret of the UCB algorithm. Since the pseudo-regret is an expectation of the regret, the following cell runs the algorithm 20 times and computes the average pseudo-regret across all runs.

[7]:
```
#Initialize Figure
plt.rcParams['figure.figsize']=[9,4]
plt.figure()

# Define the time horizon of each run, and the number of runs of each the␣
↪algorithm.
T=1000
num_runs=20

#Initialize pseudo-regret
UCB_pseudo_regret=0
for runs in range(num_runs):
    #Initialize Bandit_environment
    bandit_env.initialize(make_plot=0)
    for t in range(1,T+1):
        #Choose arm using UCB algorithm
        arm,confidence_bounds=UCB_pull_arm(t, standard_deviations, bandit_env.
↪times_pulled,bandit_env.rewards)

        #Pull Arm
        bandit_env.pull_arm(arm)

    #Keep track of pseudo-regret
    UCB_pseudo_regret+=np.array(bandit_env.regret)

#Make plot
plt.plot(UCB_pseudo_regret/num_runs)
plt.xlabel('Time')
plt.ylabel('Pseudo-Regret')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

<IPython.core.display.HTML object>

Visualize Your Algorithm

If you want to visualize your algorithm, you can use the following interactive demo (If it is lagging, do not worry this part is not graded and is meant to build your intuition for the algorithm):

[8]:
```
plt.rcParams['figure.figsize']=[9,8]

"""
Creates an interactive bandit instance with an option to test your algorithm.
    - Pull an arm by clickling on the colored button
    - Allow your algorithm to choose the arm by clicking on the ``UCB'' button␣
 ↪in the lower right.
    - The true means of the distributions are shown with the dashed horizontal␣
 ↪lines
    - Large solid circle is the sample mean of the arm
    - Solid vertical line is the upper confidence bound you have calculated
    - The reward distribution of each arm is shown on the right and can be␣
 ↪toggled on/off by checking the box
    - Running Pseudo-regret is shown on the bottom and can be toggled on/off by␣
 ↪checking the box
"""


# You may need to rerun this cell to restart the gui
alg=Interactive_UCB_Algorithm(bandit_env,UCB_pull_arm,'UCB')
alg.run_Interactive_Alg()
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

/home/jovyan/fa21/lab/lab09/Bandit_env.py:70: MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be
removed two minor releases later. Use manager.set_window_title or GUI-specific
methods instead.
  self.figure.canvas.set_window_title('Bandits Demo')
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
```

```
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
```

```
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
```

Question 2. The Bayesian Approach: Thompson Sampling

Noq we will analyze is the Bayesian take on multi-armed bandits, known as Thompson Sampling. In this setting, you begin with a prior over the mean of each arm $\mu_a \sim \pi_a$.

At each round $t = 1, 2...$, the algorithm computes the posterior probability $q_{a,t}$ that arm $a \in \mathcal{A}$ has the highest mean reward.

$$q_{a,t} = \mathbb{P}\left(\mu_a = \max_{a'} \mu_{a'} \;\middle|\; X_{1,A_1}, ..., X_{t-1,A_{t-1}}\right).$$

The choice of arm is then randomly sampled from the distribution q_t over $\mathcal{A}$, where each arm $a \in \mathcal{A}$ has probability $q_{a,t}$:

$$A_t \sim q_t$$

2.a Implementing Thompson Sampling

Since the posterior distribution over each arm having the maximum mean is often intractable to compute, in practice we often implement a simpler algorithm that nevertheless accomplishes the same task.

At each rount $t = 1, 2....,$ you keep track of the posterior distribution over μ_a, for each arm $a \in \{0, 1, ..., K-1\}$, given all the samples you have observed from that arm $X_{a,1}, ..., X_{a,T_a(t-1)}$. Denote by $P_{a,t}$ the posterior distribution of the mean reward associated with arm a, after observing all the samples up to time t.

$$P_{a,t} = \mathbb{P}(\mu_a | X_{a,1}, ..., X_{a,T_a(t-1)}).$$

You then take one posterior sample from $P_{a,t}$ and choose the arm with the highest sample:

1. $\tilde{\mu}_{a,t} \sim P_{a,t}$ for $a \in \{0, 1, ..., K-1\}$.
2. Choose arm:

$$A_t = \underset{a \in \{0,1,...,K-1\}}{\text{argmax}} \quad \tilde{\mu}_{a,t}$$

Since the reward distributions in this lab are Gaussians with known variance σ_a^2, we know from our investigation of conjugate priors that if we have Gaussian priors: $\mu_a \sim \mathcal{N}(\mu_{a,0}, \sigma_{a,0}^2)$, and Gaussian likelihoods $X_a^{(i)} | \mu_a \sim \mathcal{N}(\mu_a, \sigma_a)$ the posterior distribution for each arm will also be a Gaussian.

Therefore, to implement Thompson Sampling in this lab, the posterior distributions for each arm in this lab at each time $t = 1, 2, ...$ are given by:

$$P_{a,t} = \mathcal{N}(\hat{\mu}_{a,t}, \hat{\sigma}_{a,t}^2)$$

where,

$$\hat{\sigma}_{a,t}^2 = \left(\frac{1}{\sigma_{a,0}^2} + \frac{T_a(t-1)}{\sigma_a^2} \right)^{-1}$$

$$\hat{\mu}_{a,t} = \hat{\sigma}_{a,t}^2 \left(\frac{\mu_{a,0}}{\sigma_{a,0}^2} + \frac{\sum_{i=1}^{T_a(t-1)} X_{a,i}}{\sigma_a^2} \right)$$

`TODO`: Fill out the following function that implements the choice of arm for the Thompson Sampling algorithm with Gaussian arms and prior.

[9]:
```
# TODO: complete the function
def TS_pull_arm(t,variances,times_pulled,rewards,prior_means,prior_variances):
    """
    Implement the choice of arm for the Thompson Sampling Algorithm when the␣
 ↪arms and priors are Gaussians.
```

```
    Inputs:
        t: int, number of iteration of the bandit algorithm.
        variances: a list of length K (where K is the number of arms) of the␣
↪variances
            corresponding to each arm (\sigma_a^2 in the likelihood expression␣
↪above)
        times_pulled: a list of length K of the number of times each arm has␣
↪been pulled.
        rewards: a list of K lists. Each of the K lists holds the samples␣
↪received from pulling each arm up
            to iteration t.
        prior_means: a list of length K with the mean of the priorsfor each arm.
        prior_mea: a list of length K with the variance of the prior for each␣
↪arm.

    Outputs:
        arm: integer representing the arm that the TS algorithm would choose.
        posterior_samples: list of samples from the posterior used to choose␣
↪the arm.
        posterior_means: list of means of the posterior for each arm
        posterior_vars: list of variances of the posteriors of each arm
    """

    K=len(times_pulled)

    posterior_samples=[]
    posterior_means=[]
    posterior_vars=[]
    for arm in range(K):
        arm_var_hat= (1/prior_variances[arm] + (times_pulled[arm]/
↪variances[arm]))**(-1)
        # TODO: fill in arm_var_hat, which is \hat\sigma^2_{a,t}.
                    # Hint: \hat\sigma^2_{a,0} is prior_variances[arm],␣
↪\sigma^2_a is variance,
                    #       and T_a(t-1) is times_pulled[arm] (as before).

        mean_hat= arm_var_hat * (prior_means[arm]/prior_variances[arm] + np.
↪sum(rewards[arm])/variances[arm])
        # TODO: fill in mean_hat, which is \hat\mu_{a,t}.
                # Hint: \mu_{a,0} is prior_means[arm], and X_a^{(i)} is␣
↪rewards[arm] (as before).


        posterior_samples.append(np.random.normal(mean_hat,arm_var_hat,1))
        posterior_means.append(mean_hat)
```

```
        posterior_vars.append(arm_var_hat)

    arm= np.argmax(posterior_samples)

    return arm, posterior_samples, posterior_means, posterior_vars
```

[10]:
```
# Validation tests: Do not modify
K_test = 5
times_pulled_test = [3, 5, 7, 4, 1]
prior_means_test=[8,5,7,9,6]
prior_variances_test=[2.5, 0.1, 1.6, 2.3, 1.7]
t_test = np.sum(times_pulled_test) + 1
variances_test = [0.4, 0.2, 0.1, 0.2, 0.5]
rewards_test = [[10.4, 12.6, 11],
                [8, 13, 12, 11, 9],
                [9, 10, 10, 8, 9.5, 10.5, 11],
                [8.3, 9.6, 7.9, 8.1],
                [8]]
test_arm, posterior_samples_test, posterior_means_test, posterior_vars_test = TS_pull_arm(t_test,
            variances_test,
            times_pulled_test,
            rewards_test,
            prior_means_test,
            prior_variances_test)
assert test_arm == 0
opt_vals_means = [11.16, 9.0, 9.69, 8.49, 7.55]
opt_vals_vars = [0.123, 0.0286, 0.014,0.049,0.386]
for a in range(K_test):
    assert (np.abs(opt_vals_means[a] - posterior_means_test[a]) <= 0.1)
    assert (np.abs(opt_vals_vars[a] - posterior_vars_test[a]) <= 0.01)
print("Test Passed!")
```

```
Test Passed!
```

Thompson Sampling with Good Priors

As we saw in class, the performance of Thompson Sampling can vary drastically with the quality of the prior.

First, let us analyze the performance of Thompson Sampling when the priors reflect the correct rankings of the arms (meaning that the prior mean for arm 0 is the highest). We will compare it to the performance of the UCB algorithm.

[11]:
```
# No TODOs here, just run the code and inspect the plot
#Initialize Figure
plt.rcParams['figure.figsize']=[9,4]
plt.figure()

# Variance of the reward for each arm.
variance=1.5
true_variances=[variance for arm in range(len(means))]

#Define Prior Means and Variances
prior_means=[12,9,8,7,4,3]
prior_vars=[3.2,3.2,3.2,3.2,3.2,3.2]


#Initialize pseudo-regret
TS_pseudo_regret=0

for runs in range(num_runs):
    #Initialize bandit environment
    bandit_env.initialize(make_plot=0)
    for t in range(1,T+1):
        #Choose arm with Thompson Sampling
        arm,samples,means,variances=TS_pull_arm(t,true_variances,bandit_env.
 ↪times_pulled,bandit_env.rewards,prior_means,prior_vars)

        #Pull Arm
        bandit_env.pull_arm(arm)

    #Keep track of regret Regret
    TS_pseudo_regret+=np.array(bandit_env.regret)

#Plot Thompson Sampling vs. UCB regret
plt.plot(TS_pseudo_regret/num_runs ,label='TS Regret')
plt.plot(UCB_pseudo_regret/num_runs ,label='UCB Regret')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Pseudo-Regret')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Thompson Sampling with Bad Priors

Now let us analyze the performance of Thompson Sampling when the priors have completely incorrect correct rankings of the arms, meaning that the prior mean for arm 0 is the lowest.

[12]:

```
# No TODOs here, just run the code and inspect the plot
#Initialize Figure
plt.rcParams['figure.figsize']=[9,4]
plt.figure()

#Define prior means and standard deviations
prior_means=[2,3,4,5,6,7]
prior_vars=[3.2,3.2,3.2,3.2,3.2,3.2]

#Initialize pseudo-regret
TS_pseudo_regret=0
for runs in range(num_runs):
    #Initialize bandit environment
    bandit_env.initialize(make_plot=0)
    for t in range(1,T+1):
        #Chosoe arm with Thompson Sampling
        arm,samples,means,variances=TS_pull_arm(t,true_variances,bandit_env.
 ↪times_pulled,bandit_env.rewards,prior_means,prior_vars)

        #Pull Arm
        bandit_env.pull_arm(arm)

    #Keep track of regret Regret
    TS_pseudo_regret+=np.array(bandit_env.regret)

#Plot Thompson Sampling vs. UCB regret
plt.plot(TS_pseudo_regret/num_runs ,label='TS Regret')
plt.plot(UCB_pseudo_regret/num_runs ,label='UCB Regret')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Pseudo-Regret')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Thompson Sampling with the same prior for each arm

Now let us analyze the performance of Thompson Sampling when the priors are the same for all arms.

[13]:

```
# No TODOs here, just run the code and inspect the plot
plt.rcParams['figure.figsize']=[9,4]
plt.figure()

#Define prior means and variances
prior_means=[8,8,8,8,8,8]
```

```
prior_vars=[2.5,2.5,2.5,2.5,2.5,2.5]

#Initialize pseudo-regret
TS_pseudo_regret=0
for runs in range(num_runs):
    #Initialize bandit environment
    bandit_env.initialize(make_plot=0)
    for t in range(1,T+1):
        #Chose arm with Thompson Sampling
        arm,samples,means,variances=TS_pull_arm(t,true_variances,bandit_env.
 ↪times_pulled,bandit_env.rewards,prior_means,prior_vars)

        #Pull Arm
        bandit_env.pull_arm(arm)

    #Keep track of regret Regret
    TS_pseudo_regret+=np.array(bandit_env.regret)

#Plot Thompson Sampling vs. UCB regret
plt.plot(TS_pseudo_regret/num_runs ,label='TS Regret')
plt.plot(UCB_pseudo_regret/num_runs ,label='UCB Regret')
plt.legend()
plt.xlabel('Time')
plt.ylabel('Pseudo-Regret')
plt.show()
# You may need to rerun this cell to restart the gui
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Visualize Your Algorithm

If you want to visualize your algorithm, you can use the following interactive demo (If it is lagging, do not worry this part is not graded and is meant to build your intuition for the algorithm):

[14]:
```
plt.rcParams['figure.figsize']=[9,8]

"""
Creates an interactive bandit instance with an option to test your algorithm.
    - Pull an arm by clickling on the colored button.
    - Allow your algorithm to choose the arm by clicking on the ``TS'' button␣
 ↪in the lower right.
    - The true means of the distributions are shown with the dashed horizontal␣
 ↪lines.
    - Large solid circle is the sample mean of the rewards for the arm.
    - Solid vertical line shows the 95% credible interval for the arm.
```

```
    - The reward distribution of each arm is shown on the right and can be␣
 ↪toggled on/off by checking the box.
    - Running Pseudo-regret is shown on the bottom and can be toggled on/off by␣
 ↪checking the box.
"""

#Define prior means and variances
prior_means=[8,8,8,8,8,8]
prior_vars=[2.5,2.5,2.5,2.5,2.5,2.5]

# You may need to rerun this cell to restart the gui
alg=Interactive_TS_Algorithm(bandit_env,TS_pull_arm,'TS',prior_means,prior_vars)
alg.run_Interactive_Alg()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/home/jovyan/fa21/lab/lab09/Bandit_env.py:70: MatplotlibDeprecationWarning:
The set_window_title function was deprecated in Matplotlib 3.4 and will be
removed two minor releases later. Use manager.set_window_title or GUI-specific
methods instead.
  self.figure.canvas.set_window_title('Bandits Demo')
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
```

```
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
```

```
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1689:
FutureWarning: The `vertical` parameter is deprecated and will be removed in a
future version. Assign the data to the `y` variable instead.
  warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.9/site-packages/seaborn/distributions.py:1708:
UserWarning: Support for alternate kernels has been removed. Using Gaussian
kernel.
  warnings.warn(msg, UserWarning)
```

Question 3. Pros and Cons of UCB and Thompson Sampling

In the following cell, write a few sentences comparing and contrasting UCB and Thompson Sampling. What are some pros and cons of UCB and of Thompson Sampling?

TODO: Thompson Sampling performs better when we use a good prior. This means that Thompson Sampling accumulates less regret pseudo-regret over time than UCB if we use a good prior, but it ends up accumulating more pseudo-regret over time than UCB if we use a bad prior. As a result, in deciding whether to use Thompson Sampling or UCB, it is important to consider whether you have enough prior domain knowledge or not: if you have domain knowledge then it is probably a good idea to use Thompson Sampling.

UCB is generally more consistent (shape and y values of psuedo-regret). This is because there is no random sampling step and it does not depend on whether the prior is good or bad. On the other hand, Thompson Sampling varies more because there is the `np.random.normal(mean_hat,arm_var_hat,1)` step, which introduces randomness. I would use Thompson Sampling if I had a lot of domain knowledge about my problem, and use UCB if I knew nothing.

[15]:
```
%matplotlib inline
import matplotlib.image as mpimg
img = mpimg.imread('husky.jpg')
imgplot = plt.imshow(img)
imgplot.axes.get_xaxis().set_visible(False)
imgplot.axes.get_yaxis().set_visible(False)
```

```python
print("Yay, you've made it to the end of Lab 8!")
plt.show()
```

```
Yay, you've made it to the end of Lab 8!
```

[ ]:

```python

```

[ ]:

```python

```