Economics 691-06: Assignment 6

Nihar Shah

Solutions are marked in blue.

Problem 1: Multi-Armed Bandits

AV Club, an online publisher known for its review of television shows and movies, is debating between two possible headlines for a short article. Those two headlines are:

- *Why the last episode of The Simpsons left us with a cliffhanger*
- *What the cliffhanger means for the next episode of The Simpsons*

In both cases, the articles are the same (reviewing the last episode and forecasting the next episode); but the headline can either advertise its commentary on the last episode or on the next one. For the sake of notation, suppose the test is the first headline (the last episode) and the control is the second headline (the next episode). AV Club wants to deploy a bandit to maximize time spent reading the article.

The response functions for article readers is below, in both Python and R. These represent the amount of time, in seconds, that readers spend reading the article. Notice that the response function differ from the one discussed in class in one crucial dimension: the response to treatment changes over time. Early readers have a relatively positive reaction to the first headline compared to the second one, while later readers have a relatively negative reaction. This should not be surprising, given that the first headline focuses on the last episode and so does best right after the episode has been aired; and the second headline focuses on the next episode and so does best right before that episode is to be aired. In this simulation, we will always assume there are five hundred readers who come to the article, so i ranges from 1 to 500.[1]

```
# Define a function giving an individual's response, given a treatment status
# and the individual's place in the simulation

# Python
import numpy as np
def individual_response(treatment, i):
  return(max(0, 12 + (5 - i/70) * treatment + np.random.normal(0, 5, 1)[0]))

# R
individual_response = function(treatment, i) {
  return(max(0, 12 + (5 - i/70) * treatment + rnorm(1, 0, 5)))
  }
```

[1]Since Python is zero-indexed, i will range from 0 to 499. You do not need to make any adjustments for this.

1. **(5 points)** In class, we covered three approaches: the standard experiment (in which each user is independently assigned to treatment with 50% probability), the constant-ϵ bandit, and the declining-ϵ bandit. Include them in your code, and update them in two ways. First, update both their default n and their calls to the individual_response function. Second, have the functions *only* return the average time-spent, in seconds, on the article. This is because the coefficient and standard error are no longer useful attributes of the data, because of the variation in the coefficient across time and across the order of units.

   - Note that you may wish to partially or fully consolidate these functions to make them more readable. In the solutions, for instance, I will consolidate all three approaches and the following approach (Thompson sampling) into a single function. This is encouraged but not required for full credit.

   The answer for this question is bundled into the next question.

2. **(25 points)** AV Club also wants to implement a bandit that utilizes Thompson sampling, and asks for your assistance. Implement one, using the hints and guidance below; and demonstrate that it works by running the function once.

   - While Thompson sampling does not technically need a phase with uniform random sampling (as the other bandit algorithms do), most Thompson sampling algorithms use a "burn-in" period with uniform random sampling for the first few observations. As such, allow the algorithm to explore randomly between test and control for the first **thirty** observations.
   - Once this burn-in period has elapsed, sample each arm with the probability that it is the better arm. As a hint to calculate this probability, recall the following formula for the two-sample t-test; and consider how it may apply in this case.

$$t = \frac{\bar{x}_1 - \bar{x}_0}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_0^2}{n_0}}}$$

   The solution for Thompson sampling is coded in the function below, along with the other three approaches – the experiment, the constant-ϵ algorithm, and the declining-ϵ algorithm. This unification is because all four approaches share strong similarities. For instance, all four have a component that involves uniform random sampling: the experiment does it 100% of the time, the constant-ϵ and declining-ϵ algorithm do it some fraction of the time, and the Thompson algorithm does it for the first thirty draws.

   To compute the probability for each arm, consider the hint. We want the probability that the test arm outperforms the control arm; and we have a sample of data for each arm. Mathematically, this is the same as using a two-sample t-test to identify whether the test arm outperforms the control arm. The only difference is that, in a formal hypothesis test, we map the t-statistic to a probability and check whether it is below some threshold α. In this case, we want to see where on the normal distribution the t-statistic lies; and how much probability mass lies below it. In other words, we use the following expression:

$$\mathbb{P}(\mu_1 > \mu_0) = \Phi_t\left(\frac{\bar{x}_1 - \bar{x}_0}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_0^2}{n_0}}}\right)$$

```
# Load libraries
import numpy as np
import pandas as pd
import statsmodels.formula.api as sm
import scipy.stats as stats

# Define a function giving an individual's response, given a treatment status
# and the individual's place in the simulation
def individual_response(treatment, i):
  return(max(0, 12 + (5 - i/70) * treatment + np.random.normal(0, 5, 1)[0]))

# Define the bandit function, that takes in three arguments: the number of
# observations; the sampling scheme (one of experiment, constant, declining,
# or Thompson); and an additional parameter
def bandit(n = 500, sampling = 'experiment', parameter = None):

  # Define the response and indicator function, as np.arrays as these can
  # be more easily searched during the 'exploit' phase
  response = np.array([])
  indicator = np.array([])

  # Iterate through each person
  for i in range(n):

    # Set epsilon for uniform sampling, which always happens for experiments,
    # sometimes happens for constant / declining-epsilon, and only happens in
    # the first thirty iterations for Thompson sampling
    if sampling == 'experiment':
      epsilon = 1
    elif sampling == 'constant':
      epsilon = parameter
    elif sampling == 'declining':
      epsilon = np.exp(-parameter * i)
    elif sampling == 'thompson':
      epsilon = int(i < 30)
    else:
      raise ValueError('You must pick a valid sampling scheme.')

    # Implement 'explore' phase with uniform sampling
    if np.random.rand(1)[0] < epsilon:
        if np.random.rand(1)[0] < 0.5:
          treatment = 1
        else:
          treatment = 0

    # Else implement 'exploit' phase
    else:
      # For constant/declining-epilson, we exploit greedily, taking the best
      # arm
      if (sampling == 'constant') or (sampling == 'declining'):

        # Note that we add 0.001 to avoid division-by-zero issues
        y1 = sum(response[indicator == 1])/(sum(indicator) + 0.001)
        y2 = sum(response[indicator == 0])/(i - sum(indicator) + 0.001)
        if y1 > y2:
          treatment = 1
        else:
          treatment = 0

      # For Thompson sampling, we exploit probabilistically, sampling each arm
      # with the likelihood that it is better
      elif sampling == 'thompson':

        # Compute the t-statistic of the difference between the two arms
        x = response[indicator == 1]
        y = response[indicator == 0]
        t = (np.mean(x) - np.mean(y))/(
            np.sqrt(np.var(x)/len(x) + np.var(y)/len(y)))

        # Draw a random uniform variable and see if it is below the CDF of
        # that t-statistic

        # If the t-statistic is large (e.g. 2), then stats.t.cdf(t, df) will
        # return a value close to 1, and we will almost always sample from the
        # treatment arm over the control arm. And if the t-statistic is small
        # (e.g. -2), then stats.t.cdf(t, df) will return a value close to 0,
        # and we will almost always sample from the control arm
        # Note that we can make an approximation on the degrees of freedom, as
        # it will not matter quantitatively
        if np.random.rand(1)[0] < stats.t.cdf(t, len(response) - 2):
          treatment = 1
        else:
          treatment = 0

    # Get and save the individual's response
    response = np.append(response, individual_response(treatment, i))
    indicator = np.append(indicator, treatment)

  # Return the results of the simulation
  return(sum(response)/n)

# Test the algorithm works on Thompson sampling
bandit(sampling = 'thompson')
13.005011951256892
```

3. **(10 points)** Evaluate the constant-ϵ algorithm under four parameterizations: $\epsilon = 0.5$, $\epsilon = 0.3$, $\epsilon = 0.1$, and $\epsilon = 0.01$. To mitigate noise, run each parameterization on one hundred different streams of data, and take the average of the hundred runs per parameterization.[2] Explain the results: which ϵ performs best, which ϵ performs worst, and why?

   The code below gets results for the four different parameterizations, each averaged over one hundred streams of data.

```
# Define a helper function to run each algorithm on one hundred streams of data
def bandit_analysis(sampling = 'experiment', parameter = None):
  results = []
  for i in range(100):
    np.random.seed(i)
    results.append(bandit(sampling = sampling, parameter = parameter))
  return(np.mean(results))

print("With epsilon = 0.50, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'constant', parameter = 0.50), 4)))
With epsilon = 0.50, the average time-spent is: 13.0578

print("With epsilon = 0.30, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'constant', parameter = 0.30), 4)))
With epsilon = 0.30, the average time-spent is: 13.1815

print("With epsilon = 0.10, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'constant', parameter = 0.10), 4)))
With epsilon = 0.10, the average time-spent is: 13.1822

print("With epsilon = 0.01, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'constant', parameter = 0.01), 4)))
With epsilon = 0.01, the average time-spent is: 12.3311
```

   The best-performing ϵ is 0.10, in which we exploit 90% of the time but explore 10% of the time. The worst-performing ϵ is 0.01, in which we explore only 1% of the time. Notice that the performance is not monotonic in ϵ. Exploiting more is generally good as it avoids waste by sending observations to suboptimal arms; but as we start to hit very low levels of ϵ, we explore too little. This is especially important because the environment changes over time; and so the optimal arm early in the simulation is no longer the optimal arm later in the simulation, making continuous exploration particularly valuable. Note that $\epsilon = 0.30$ also performs about as well, and that may well be the leading candidate in your answer.

4. **(10 points)** Evaluate the declining-ϵ algorithm under three parameterizations: $\alpha = 0.01$, $\alpha = 0.005$, and $\alpha = 0.001$. To mitigate noise, run each parameterization on one hundred different streams of data, and take the average of the hundred runs per parameterization. Explain the results: which α performs best, which α performs worst, and why?

   The code below gets results for the three different parameterizations, each averaged over one hundred streams of data.

[2] In reality, note that this sort of approach is difficult given a single realization of the data.

```
print("With alpha = 0.01, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'declining', parameter = 0.01), 4)))
With alpha = 0.01, the average time-spent is: 13.1024

print("With alpha = 0.005, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'declining', parameter = 0.005), 4)))
With alpha = 0.005, the average time-spent is: 12.9461

print("With alpha = 0.001, the average time-spent is: " +
      str(round(bandit_analysis(sampling = 'declining', parameter = 0.001), 4)))
With alpha = 0.001, the average time-spent is: 12.7491
```

The best-performing α is 0.01, in which we exploit relatively more; and the worst-performing α is 0.001, in which we exploit relatively little. This may be counterintuitive, especially since the environment changes. However, this likely because the low-α algorithms *explore too much* early in the approach, whereas the high-α algorithms exploit then. More generally, this underscores a broader issue with the various ϵ-based algorithms, which is that they require a careful tuning of the parameters. Finally, although this is not required for credit, note that all variants of the declining-ϵ algorithm generally perform worse than the constant-ϵ algorithms, because they cannot easily adapt to the changing environment.

5. **(5 points)** Evaluate the four approaches: the experiment, the best constant-ϵ algorithm, the best declining-ϵ algorithm, and Thompson sampling. To mitigate noise, run each algorithm on one hundred different streams of data, and take the average of the hundred runs per algorithm.

   The code below gets results for the four different algorithms, each averaged over one hundred streams of data.

```
print("The experiment yields a time-spent of: " +
      str(round(bandit_analysis(sampling = 'experiment'), 4)))
The experiment yields a time-spent of: 12.7409

print("The best constant-epsilon algorithm yields a time-spent of: " +
      str(round(bandit_analysis(sampling = 'constant', parameter = 0.10), 4)))
The best constant-epsilon algorithm yields a time-spent of: 13.1822

print("The best declining-epsilon algorithm yields a time-spent of: " +
      str(round(bandit_analysis(sampling = 'declining', parameter = 0.01), 4)))
The best declining-epsilon algorithm yields a time-spent of: 13.1024

print("The Thompson sampling algorithm yields a time-spent of: " +
      str(round(bandit_analysis(sampling = 'thompson'), 4)))
The Thompson sampling algorithm yields a time-spent of: 13.3162
```

6. Using these results, answer the following questions comparing Thompson sampling to each of the algorithms in turn.

   (a) **(5 points)** Compare Thompson sampling to the experiment. Which performs better and why?

   Thompson sampling does very well against the experiment, as the experiment wastes many observations on the suboptimal arms. Thompson sampling (and all bandit strategies, for that matter) do successfully exploit knowledge accumulated during the experiment, and sends relatively more observations to optimal arms.

   (b) **(5 points)** Compare Thompson sampling to the best constant-ϵ algorithm. Which performs better and why?

   Thompson sampling performs comparably to the best constant-ϵ algorithm, but still does better. While both explore the suboptimal arm constantly, Thompson sampling does so more efficiently: it explores the suboptimal arm less, until the evidence starts to arrive that the suboptimal arm may no longer be suboptimal (as the environment changes) – at which point it starts to explore that more and in turn reallocate its exploitation strategy.

   (c) **(5 points)** Compare Thompson sampling to the best declining-ϵ algorithm. Which performs better and why?

   Thompson sampling does well against the declining-ϵ algorithm. The declining-ϵ algorithm explores little near the end, even as the environment shifts and the control arm goes from suboptimal to optimal. However, Thompson sampling continues to explore at that point, and so can identify this change and exploit it successfully. In a changing environment, this difference in algorithmic ability to explore turns out to be massively valuable.

Problem 2: Assignment Survey

(20 points) Congrats on completing all homework assignments! Please fill out a short survey of the assignments, linked here: https://forms.gle/XjYEp3F7HA2mM9zW8.