Machine Learning: algorithms, Code Lecture 7

Multi-armed bandits part b, Thompson sampling

Nicky van Foreest

June 4, 2021

Contents

1 Overview

- Previous lecture:
  - Boosting
  - simple algos for multi-armed bandits
- This lecture: Advanced algos for multi-armed bandits.
  - eps greedy strategies
  - UPC1
  - Thompson sampling.

  We finish the course with a Bayesian flavored algorithm!

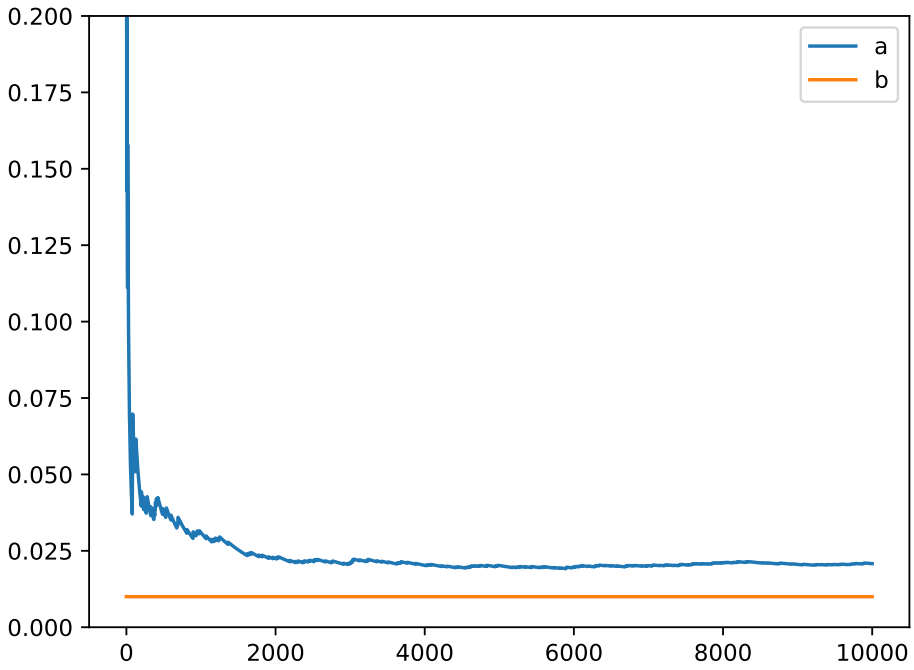
2 Optimizing a web site, part 2

2.1 Previous lecture

Recall:

- We have two flavors of a web page.
- The problem is to find out the flavor that converts best.
- We developed simple strategies to estimate the conversion ratios of each flavor.
- We could get stuck in a local minimum.
- We should improve our exploration-exploitation strategy.

The next figure recalls our earlier example of getting stuck on the 'wrong page'.



2.2 A policy that ensures to keep exploring

Suppose we assign always at least 3% of the visitors to each the two pages. Then I guess that the loss cannot be really bad. So, let's see whether we can support this guess.

Recall that we assumed that conversion loss of flavors a and b are $p = 0.02$ and $q = 0.05$, respectively, and we were in the process of developing algorithms to see whether we can filter out page b as the better of the two versions.

If we would give page b to 97% of the visitors and to rest of the visitors page `a`, then we make a expected profit of

$$0.03p + 0.97q = 0.03 \cdot 0.02 + 0.97 \cdot 0.05 = 0.491 \tag{1}$$

instead of 0.05 always. The benefit of giving page a to 3% of the visitors is that we will learn pretty quickly that page a is not the best page.

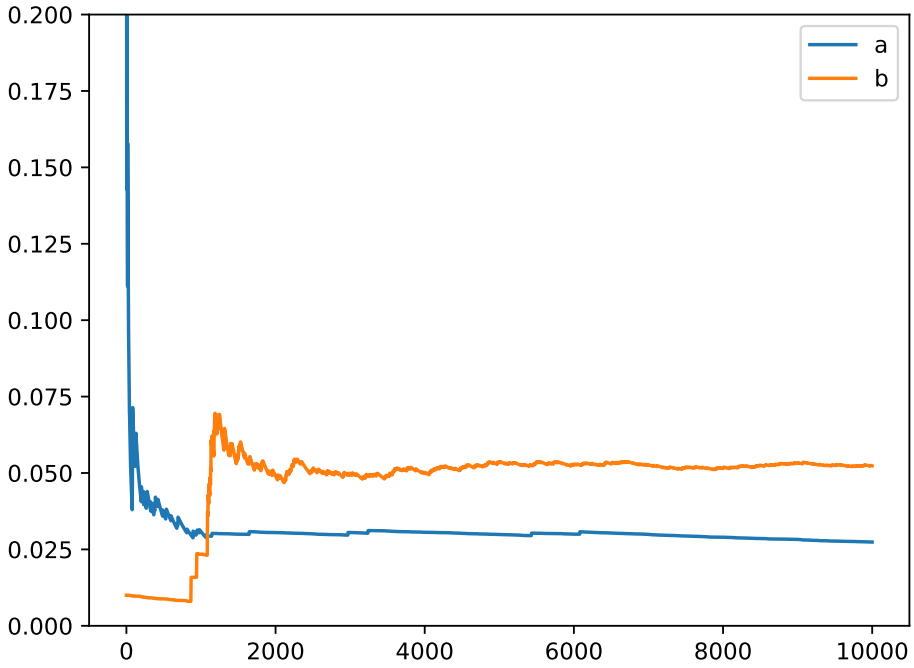
Here is some code to plot the estimators for p and q.

```python
import numpy as np
import matplotlib.pyplot as plt

p, q, eps = 0.02, 0.05, 0.03

def make_plot(estimator, fname):
    plt.clf()
    plt.ylim(0, 0.2)
    xx = range(len(estimator))
    plt.plot(xx, estimator[:, 0], label="a")
    plt.plot(xx, estimator[:, 1], label="b")
    plt.legend()
    plt.savefig(fname)
```

```python
def run(a, b, n_a, n_b, n=1000):
    np.random.seed(30)
    a_convert = np.random.binomial(1, p, size=n)
    b_convert = np.random.binomial(1, q, size=n)
    flip = np.random.uniform(size=n)

    estimator = np.zeros((n, 2))

    for i in range(n):
        if a / n_a >= b / n_b:
            if flip[i] > eps:
                a  += a_convert[i]
                n_a += 1
            else:
                b += b_convert[i],
                n_b += 1
        else:
            if flip[i] <= eps:
                a  += a_convert[i]
                n_a += 1
            else:
                b += b_convert[i],
                n_b += 1
        estimator[i, :] = [a / n_a, b / n_b]
    return estimator
```

And now we plot. Note that we give page b a very bad start: our estimate of page b being successful is $1/100$.

```python
estimator = run(a=1, n_a=1, b=1, n_b=100, n=10000)
make_plot(estimator, "figures/policy_3.pdf")
```



We see that both conversion ratios are well estimated. And, even though page b starts badly, it recovers quickly. But, can we do better? (Note again, as with nearly any problem, the algorithms we invent ourselves are often pretty bad. It's smart to be suspicious about one own's intelligence, rather than the other way round.)

2.3 A simple revision of the code

Before coding some algorithms of others, I need to revise my code a bit, so as to make it easier to implement the other algorithms in a similar way. In particular, I need to track the number of successful conversions for each page. Let $a_s(t)$ be the number of conversions up to time t for page a, and $a_f(t)$ the number of failures. Consequently, the number of visitors for page `a` has been $a_s(t) + a_f(t)$. The notation for the other page is likewise.

For each round t I determine a winner, which is the page that gets the visitor for that round. The `trace` keeps track of the a_f, etc.

Besides the estimated success ratio, I want to track the average reward, which is given by

$$r_t = \frac{a_s(t) + b_s(t)}{t} \tag{2}$$

up to round t. We know that the best we could have done is qt, in expectation for course. So, by comparing r_t to q we have an idea of the performance of our algorithm.

Here is my revised version. The most important part is the part in which we determine the *winner*. In fact, the goal of the algorithms is to find the winner, the rest of the code is not essential.

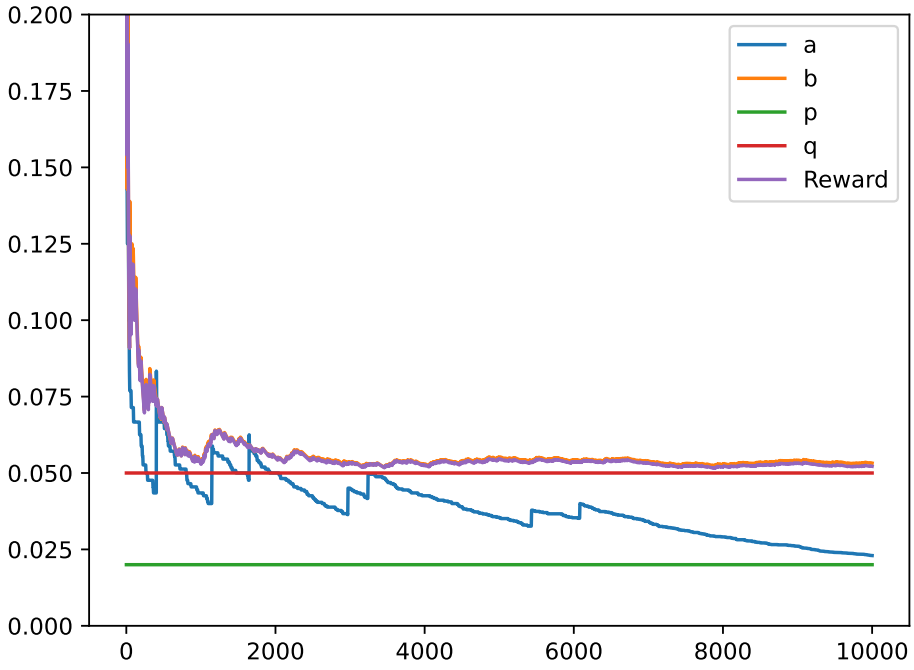
```python
def eps_greedy(a_s, a_f, b_s, b_f, n=1000):
    np.random.seed(30)
    convert = np.zeros((n, 2))
    convert[:, 0] = np.random.binomial(1, p, size=n)
    convert[:, 1] = np.random.binomial(1, q, size=n)
    flip = np.random.uniform(size=n)

    trace = np.zeros((n, 4))
    trace[0, :] = [a_s, a_f, b_s, b_f]

    for t in range(1, n):
        p_hat = a_s / (a_s + a_f)
        q_hat = b_s / (b_s + b_f)
        winner = 0 if p_hat >= q_hat else 1
        winner = 1 - winner if flip[t] <= eps else winner

        if winner == 0:
            a_s += convert[t, winner]
            a_f += 1 - convert[t, winner]
        else:
            b_s += convert[t, winner]
            b_f += 1 - convert[t, winner]
        trace[t, :] = [a_s, a_f, b_s, b_f]

    return trace
```

I have to update the plotting function accordingly.

```python
def make_plot(trace, fname):
    plt.clf()
    plt.ylim(0, 0.2)
    xx = np.arange(len(trace)) + 1  # prevent division by 0
    plt.plot(xx, trace[:, 0] / (trace[:, 0] + trace[:, 1]), label="a")
    plt.plot(xx, trace[:, 2] / (trace[:, 2] + trace[:, 3]), label="b")
    plt.plot(xx, (trace[:, 0] + trace[:, 2]) / xx, label="Reward")
    plt.plot(xx, p*np.ones(len(xx)),  label="p")
    plt.plot(xx, q*np.ones(len(xx)),  label="q")
    plt.legend()
    plt.savefig(fname)
```

Let's run it, and see whether we get something similar as before.

```python
trace = eps_greedy(a_s=1, a_f=1, b_s=1, b_f=1, n=10000)
make_plot(trace, "figures/policy_greedy.pdf")
```



This is interesting. The reward lies slightly below the estimate for q, and both p and q seem to be reasonably well estimated by a and b. However, the quality of the estimator of p is lagging, but this is because we give page `a` 'much less attention'.

There are many interesting variations on this type of algorithm. For instance, ϵ can be made time dependent. So, start with a large ϵ to learn rapidly, and then make ϵ smaller and smaller over time, with the intuition that we learn less and less from new measurements.

2.4 Making a common testing ground.

In the code `multi_armed_bandits.py` in the `code` directory I wrote a class `Strategy` to run a simulation and make a plot in one go. The only thing that has to be provided when subclassing `Strategy` is a `run` method that implements the actual algorithm that decides the winner, i.e., the page to give to a visitor that arrives at time t.

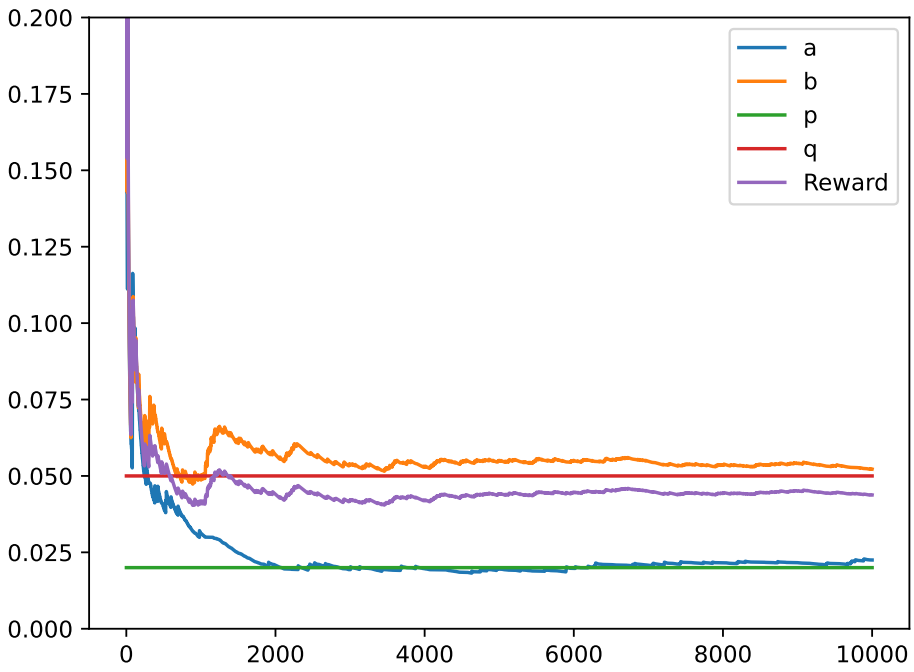
Here is an implementation of our earlier, so-called, epsilon-greedy strategy.

```
class Eps_greedy(Strategy):
    def run(self):
        flip = np.random.uniform(size=self.n)
        for t in range(1, self.n):
            p_hat = self.a_s / (self.a_s + self.a_f)
            q_hat = self.b_s / (self.b_s + self.b_f)
            winner = 0 if p_hat >= q_hat else 1
            winner = winner if flip[t] >= eps else 1 - winner
            self.update(winner, t)
```

2.5 UCB1 sampling

The UCB1 policy works similar to the eps greedy policy. Here is a nice page that explains a bit about the background of this algorithm, and the intuition behind x_a and x_b.

```
class UCB1(Strategy):
    def run(self):
        for t in range(1, self.n):
            n_a = self.a_s + self.a_f
            n_b = self.b_s + self.b_f
            x_a = self.a_s / n_a + np.sqrt(2 * np.log(t) / n_a)
            x_b = self.b_s / n_b + np.sqrt(2 * np.log(t) / n_b)
            winner = 0 if x_a >= x_b else 1
            self.update(winner, t)
```



2.6 Thompson sampling

Thompson sampling is a really nice idea to decide which page to sample. Search the web for the details. I particularly liked:

1. Russo D.J et al: A tutorial on Thompson sampling
2. O. Chapelle and Li L.: An empirical evaluation of Thompson sampling

I encourage you to at least browse through both documents.

The main idea is like this. Assume we have a guess $f(p)$ for the PDF of the success of page a. We can start with the uniform PDF (probability density function) $f(p) = 1_{p\in[0,1]}$, but it

can also another PDF. If we assign page a to a visitor, let $X \in \{0, 1\}$ be the rv that corresponds to the success, or not, of the conversion of page a for the visitor. Given the outcome of X, we like to update $f(p)$, because if $X = 1$, we believe that the conversion probability of page a should increase, and if $X = 0$, we believe it should decrease.

To update $f(p)$ after a measurement we use Bayes' formula:

$$f(p|X = k) = \frac{\mathsf{P}[X = k, p]}{\mathsf{P}[X = k]} = \frac{\mathsf{P}[X = k|p]\, f(p)}{\mathsf{P}[X = k]}. \tag{3}$$

Let us take $p \sim \text{Beta}(a, b)$, i.e., $f(p) = p^{a-1}(1-p)^{b-1}/\beta(a, b)$, where $\beta(a, b)$ is a normalization constant. Then,

$$f(p|X = k) = \frac{\mathsf{P}[X = k|p]\, f(p)}{\mathsf{P}[X = k]} \tag{4}$$

$$\propto p^k(1-p)^{1-k}p^{a-1}(1-p)^{b-1} = p^{a+k-1}(1-p)^{b-k}. \tag{5}$$

Comparing this to the PDF of $f(p)$ we see that $f(p|X = k) = \text{Beta}(a + k, b + 1 - k)$. Thus, the posterior PDF $f(p|X = k)$ is still a Beta distribution!

In general, we can start with an arbitrary number of successes a and failures b. If a new measurement results in a success, we add 1 to a, if it's a failure, we add 1 to b. Typically, we start with $a = b = 1$ as this is the uniform prior distribution.

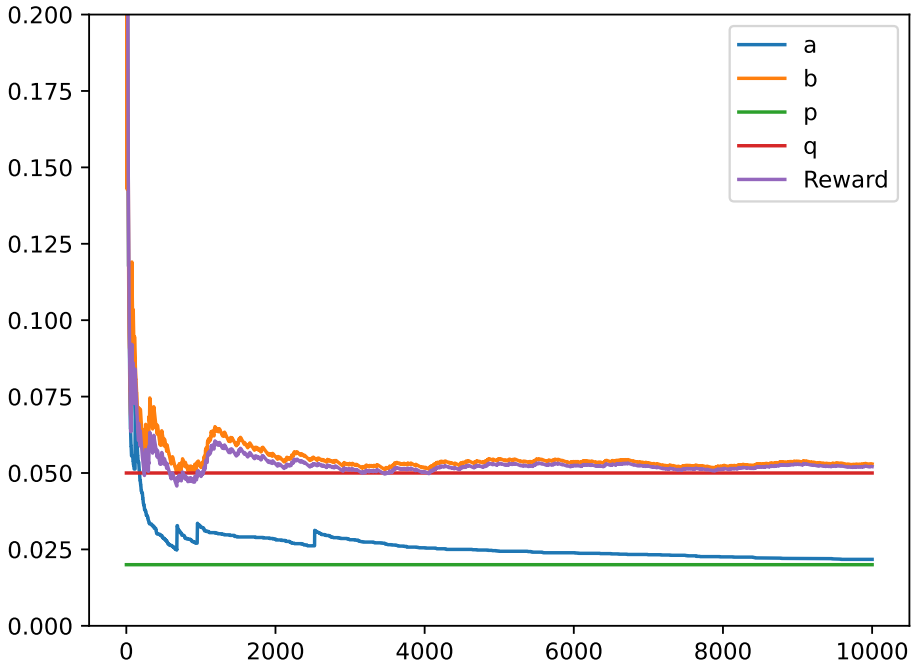
Applying this idea to selecting web pages is now straightforward. Assuming we have seen a_s and a_f successes and failures for page a, and b_s and b_f for page b, then sample

$$X_a \sim \text{Beta}(a_s, a_f), \qquad X_b \sim \text{Beta}(b_s, b_f). \tag{6}$$

If $X_a < X_b$ then page b is the winner, otherwise page a. Supposing that page a is the winner, update a_s and a_f according to whether a converversion occurred or not.

```
class Thompson(Strategy):
    def run(self):
        for t in range(1, self.n):
            x_a = np.random.beta(self.a_s, self.a_f)
            x_b = np.random.beta(self.b_s, self.b_f)
            winner = 0 if x_a >= x_b else 1
            self.update(winner, t)
```

Here is the result of a simulation.



Comparing this to the graphs of the other policies, we see that Thompson sampling works very well. Interestingly, the eps greedy policy works also very well, but we know that its performance is less than q.

3 Closing remarks

For the course:

- Finish the report
- Prepare a bit for the oral exam. All group members should be able to explain each and every part of the report and the code.
- We'll plan the oral exam together with you. If you have exams for other courses, we can plan the oral exam after that.
- Next steps in your professional life:
  - Finish the MSc
  - Read about learning strategies. It's a really interesting field, with a mix of probability, statistics, algorithms, optimization, and applications.
  - Find a job in which you are happy (On the long run, financial reward does not compensate for loss of time.)