Machine Learning: algorithms, Code Lecture 7

Multi-armed bandits

Nicky van Foreest

May 27, 2021

Contents

https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-pytho
https://jamesrledoux.com/algorithms/bandit-algorithms-epsilon-ucb-exp-python/

1 Overview

- Previous lecture:
  - Boosting for regression
  - Boosting for classification: AdaBoost
- This lecture:
  - Decision making based on data

2 Optimizing a web site

2.1 The problem

We have a web site on which we offer something to sell, e.g., hotel rooms. We have two aims:

1. Attract people to visit our site
2. Seduce visitors into buying something, or making a reservation for a hotel room, for instance.

The second step is called *conversion*: visitors cost resources, paying visitors bring in money. Hence, in web design, increasing the *conversion rate*, the fraction of visitors that buy something, is very important. The basis question is thus: How to do increase the conversion rate?

As the conversion rate might depend on the design of the web pages, companies try many different designs. Let's look at a simple example. Suppose we have a web page with the button

BUY in green, and we have another page that looks the same except that the button to buy is in red and has the text *Buy now*. Does the appearance of the button affect the conversion rate?

How would you try to find this out?

2.2 Some simple ideas

1. Initially we don't know which of the two designs is better, so we offer both web pages randomly to visitors with probability 1/2.

2. After some time, we get some updates.

3. If one design converts always, and the other never converts, we learn very rapidly.

4. But, typically, conversion rates are somewhere between 2 and 5%. Figuring out which design is better is much harder now.

So, let's build a simulator and test some ideas on how to assign a page to a new visitor. In other words, we are going to analyse a *policy* to make automatic decisions on which page to choose.

2.3 A simple policy

Suppose we have a conversions for web page type `a`, and b for type `b`. It seems reasonable to assign a new visitor to `a`, if $a >= b$. (If $a = b$, I don't care.) Suppose the pages have probabilities p and q, respectively, to convert a visitor. Suppose that $p < q$.

The `success` vectors have 0, 1 elements. If element i is 1, the visitor does convert, and if 0, the visitor does not.

```
import numpy as np

np.random.seed(3)

p, q = 0.02, 0.05
a, b = 0, 0

n = 10000  # number of visitors
succes_a = np.random.binomial(1, p, size=n)
succes_b = np.random.binomial(1, q, size=n)

for i in range(n):
    if a >= b:
        a += succes_a[i]  # if 1, conversion occurred
    else:
        b += succes_b[i]

print(a, b)
```

```
188 0
```

Here is the result:

188 0

What do we see? Page a gets all the visitors. But, $q > p$, hence page b must be converting better. Obviously we are using a bad policy. How to repair? Can you make a simple plan to improve?

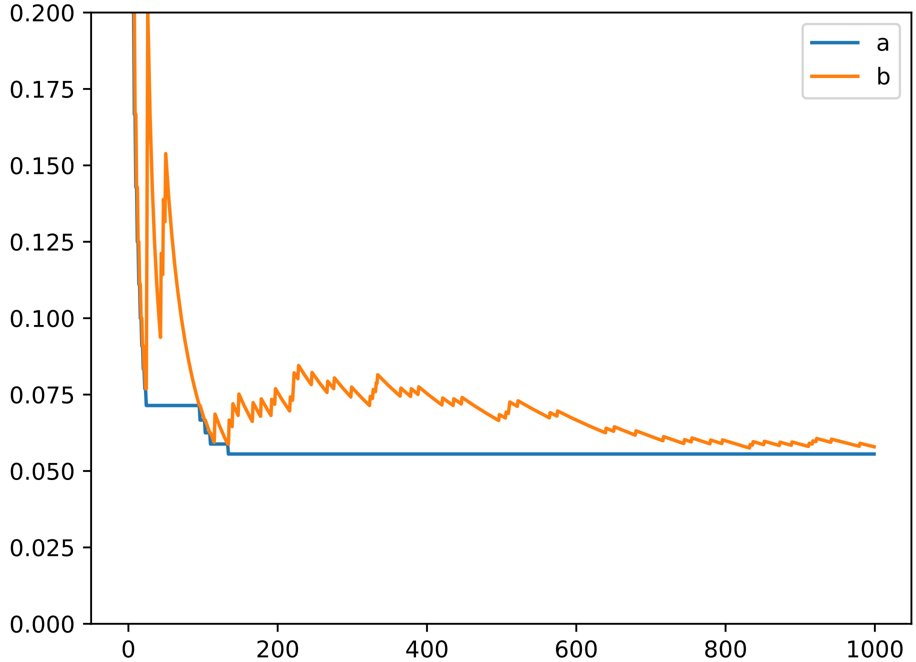
2.4 A better policy

Clearly, a and b do not correspond to the actual conversion probabilities. It must be better to use a/n_a, where n_a is the number of visitors given to type a, as an estimator for p.

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


p, q = 0.02, 0.05
a, b = 1, 1
n_a, n_b = 1, 1


def run(a, b, n_a, n_b, n=1000):
    np.random.seed(3) # ensure the same starting conditions
    succes_a = np.random.binomial(1, p, size=n)
    succes_b = np.random.binomial(1, q, size=n)

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

    for i in range(n):
        if a / n_a >= b / n_b:
            a += succes_a[i]
            n_a += 1
        else:
            b += succes_b[i]
            n_b += 1
        estimator[i, :] = [a / n_a, b / n_b]
    return estimator
```

Make a plot.

```python
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)


estimator = run(a=1, n_a=1, b=1, n_b=1)
make_plot(estimator, "figures/policy_2_1.pdf")
```

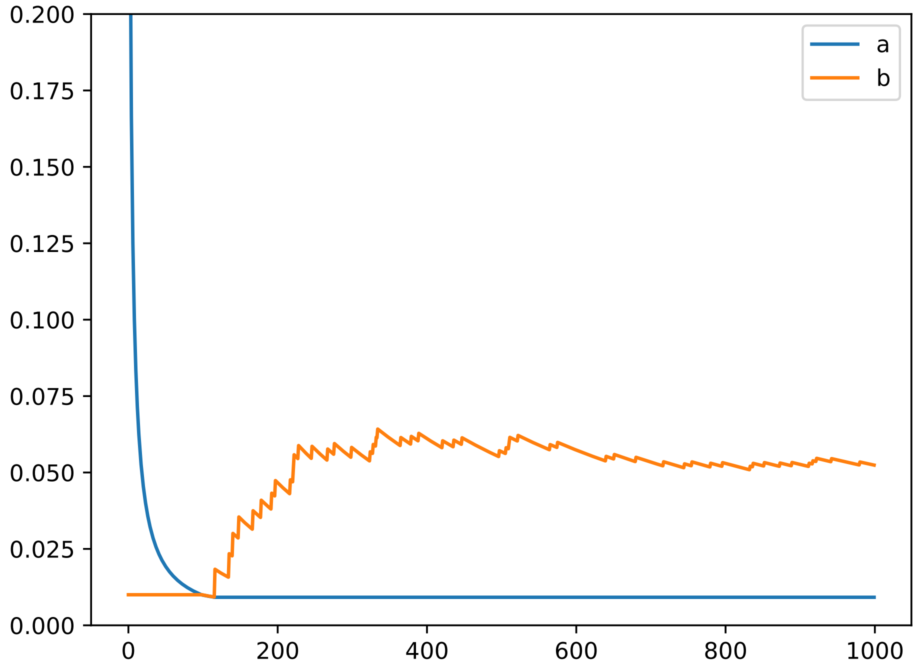


This seems to be ok. We are giving most of the visitors to page `b`. But is it robust?

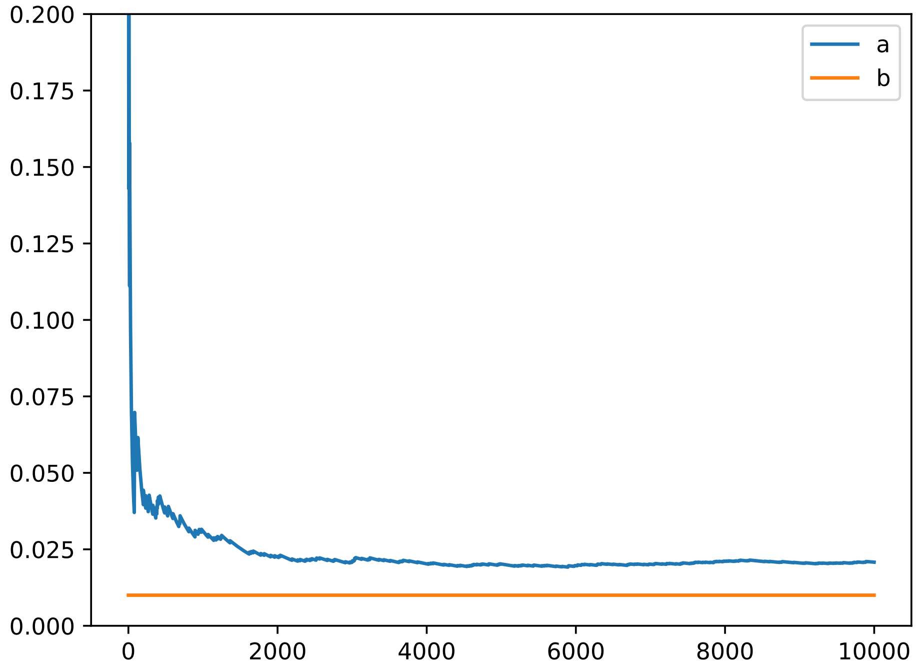
Suppose we would have started with a dumb guess, or by change we would have given lots of visitors to page `a`. Will we ever recover from bad luck?

To test, let's set $b/n_b = 1/100$. I take a value of 1% because I know that $p = 0.02$; I expect we will never find out that `b` is better, because we will always send page `a` to visitors.

```
estimator = run(a=1, n_a=1, b=1, n_b=100)
make_plot(estimator, "figures/policy_2_2.pdf")
```



Apparently we find again that b is better. However, I tried a run with the seed 30, and then I get this graph.



I think it is better not use this policy, because it can happen that we don't send enough visitors to page `b` to get a good estimate for q. This problem is known as the *exploration-exploitation* problem. We turn too rapidly to believing that page `a` is the better of the two alternative. Think a bit about how you could repair this.

2.5 A policy that ensures to keep exploring

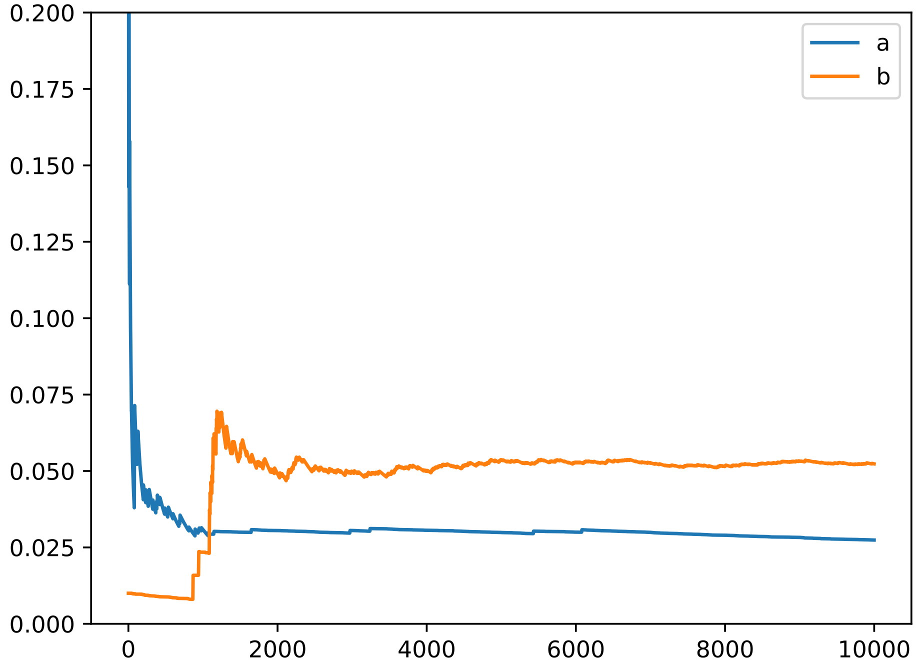
Suppose we assign always at least 3% of the visitors to each the two pages. Then the loss cannot be really bad. As an estimate, in 95% of the cases we give page `b` to a visitor, in 3% we give page `a`. We then make a profit of

$$0.97 * 0.05 + 0.03 * 0.02 \tag{1}$$

instead of 0.05 always. The relative difference is $0.97 + 0.03 * p/q$. This is still nearly 1.

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


p, q = 0.02, 0.05
eps = 0.03


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, n_a = a + a_convert[i], n_a + 1
            else:
                b, n_b = b + b_convert[i], n_b + 1
        else:
            if flip[i] > eps:
                b, n_b = b + b_convert[i], n_b + 1
            else:
                a, n_a = a + a_convert[i], n_a + 1
        estimator[i, :] = [a / n_a, b / n_b]
    return estimator
```

```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.

Can we do better? As with nearly any algorithm, the ones we invent ourselves are often pretty bad. Before we try that, I need to revise the code a bit, so as to make it easier to implement other algoritms in a similar way. In particular, I need to track the number of successful conversions for each page. Let a_s be the number of conversions and a_f the number of failures for page `a`, hence the number of visitors for page `a` has been $a_s + a_f$. 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

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

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.

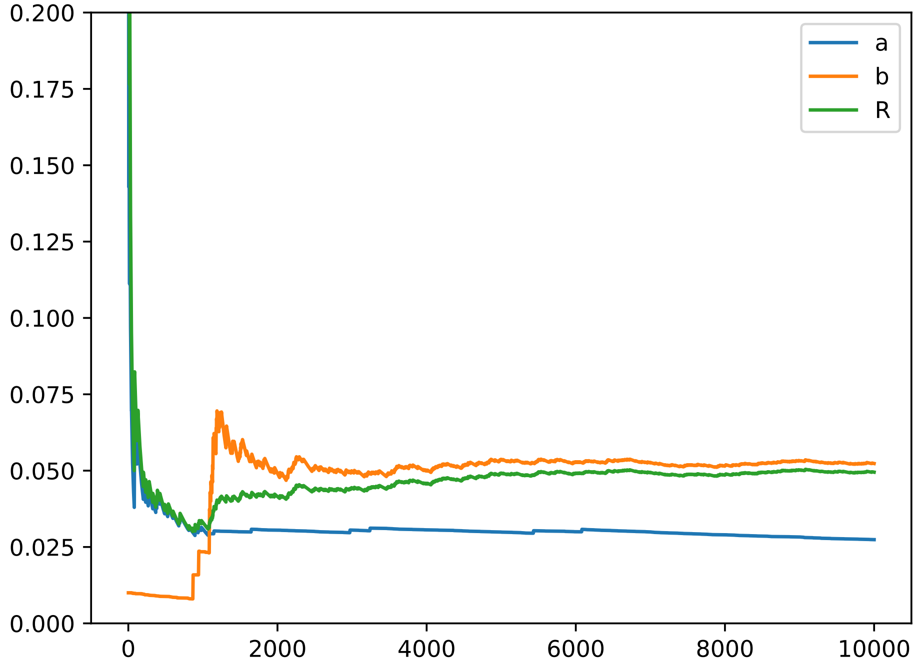
```python
def eps_greedy(a_s, a_f, b_s, b_f, n=1000):
    np.random.seed(30)
    convert = np.zeros((n, 2))
    convert[:, 0] = np.random.binomial(1, p, size=n)
    convert[:, 1] = np.random.binomial(1, q, size=n)
    flip = np.random.uniform(size=n)

    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="R")
    plt.legend()
    plt.savefig(fname)
```

Let's call it, and see whether we get the same graph as earlier.

```python
trace = eps_greedy(a_s=1, a_f=1, b_s=1, b_f=99, n=10000)
make_plot(trace, "figures/policy_greedy.pdf")
```



3 Exercises

Dealing with changing demand rates?