Machine Learning: algorithms, Code Lecture 6

Boosting, multi-armed bandits part A

Nicky van Foreest

June 14, 2021

Contents

1 Overview

- Last lecture
  - Trees
- This lecture:
  - Boosting for regression
  - Boosting for classification: AdaBoost
  - Multi-armed bandits, part a
- Next lecture:
  - Initial feedback on your reports.
  - Multi-armed bandits, part b, Thompson sampling

2 Boosting for regression

2.1 Implementation

I basically follow the ideas of DSML, but I turn their code into decent python code.

First some imports. I use the `DecisionTreeRegressor` from `sklearn` as a weak learner. It's hard to beat such implementations.

```python
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
```

Next comes my implementation for a `Boost` class for regression. Notice how I use *list comprehensions* in the `predict` methods, and *broadcasting*, which means for instance that numbers are turned into arrays of proper size when there is `+` between a number and an array.

To see how the implementation follows from the algorithm in pseudo code, observe that (in vector notation)

$$e^{b+1} = y - g_b = y - (g_{b-1} + \gamma h_b) = e^b - \gamma h_b. \tag{1}$$

Hence, we can update the residuals e^b by subtracting (γ times) the prediction of a weak learner. This is what we do in the code.

```python
class Boost:
    def __init__(self, gamma, rounds):
        self.rounds = rounds
        self.gamma = gamma

    def fit(self, X, Y):
        self.g0 = Y.mean()
        residuals = Y - self.g0
        self.learners = []

        for b in range(self.rounds):
            clf = DecisionTreeRegressor(max_depth=1)
            clf.fit(X, residuals)
            residuals -= self.gamma * clf.predict(X)
            self.learners.append(clf)

    def predict(self, x):
        y = self.g0
        y += self.gamma*sum(l.predict(x) for l in self.learners)
        return y
```

Note that in the `predict` method I take the γ out of the loop. If you look at the code of DSML it's evident that γ can be put in front, thereby replacing many multiplications by just one. Of course, it's not a big deal, but I tend to implement quick wins always, just out of habit. Over the years it all adds up.

2.2 A test

To see whether I get any output at all, I use a simple test.

```python
def test():
    X, Y = make_regression(n_samples=5, n_features=1, n_informative=1, noise=10)

    boost = Boost(gamma=0.05, rounds=10)
    boost.fit(X, Y)
    y = boost.predict(np.array([[3]]))
    print(y)
```

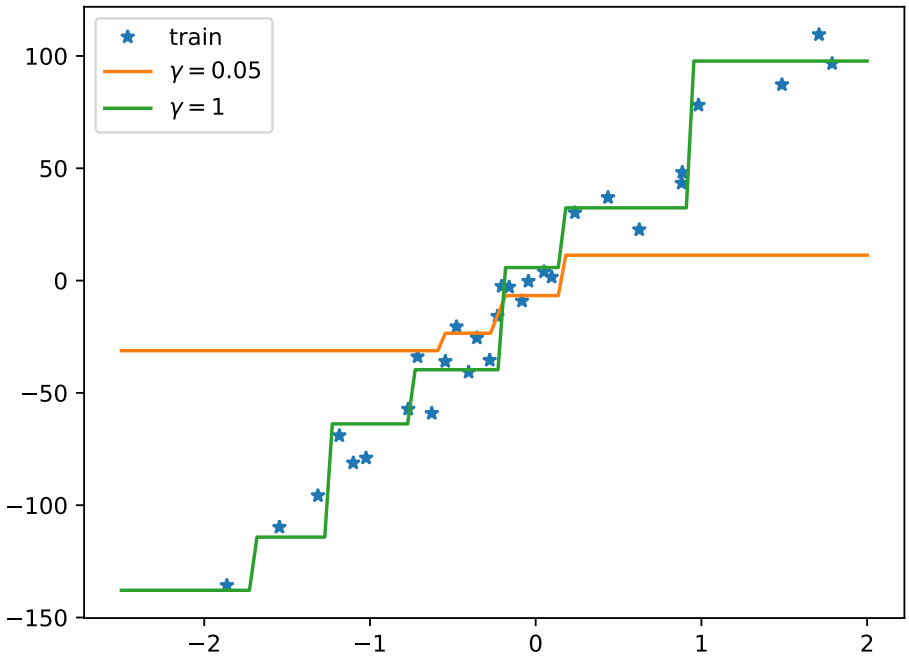
2.3 Making plots

With the following code we can make some interesting plots of the performance of boosting.

```python
def make_plots():
    np.random.seed(3)
    X, Y = make_regression(n_samples=30, n_features=1, n_informative=1, noise=10)
    plt.plot(X, Y, "*", label="train")

    n = 100
    x = np.linspace(-2.5, 2, n).reshape(n, 1)

    rounds = 100
    for gamma in [0.05, 1]:
        boost = Boost(gamma=gamma, rounds=rounds)
        boost.fit(X, Y)
        y = boost.predict(x)
        plt.plot(x, y, label=f"$\gamma = {gamma}$")

    plt.legend()
    plt.show()
    # plt.savefig("boost_test.pdf")
```

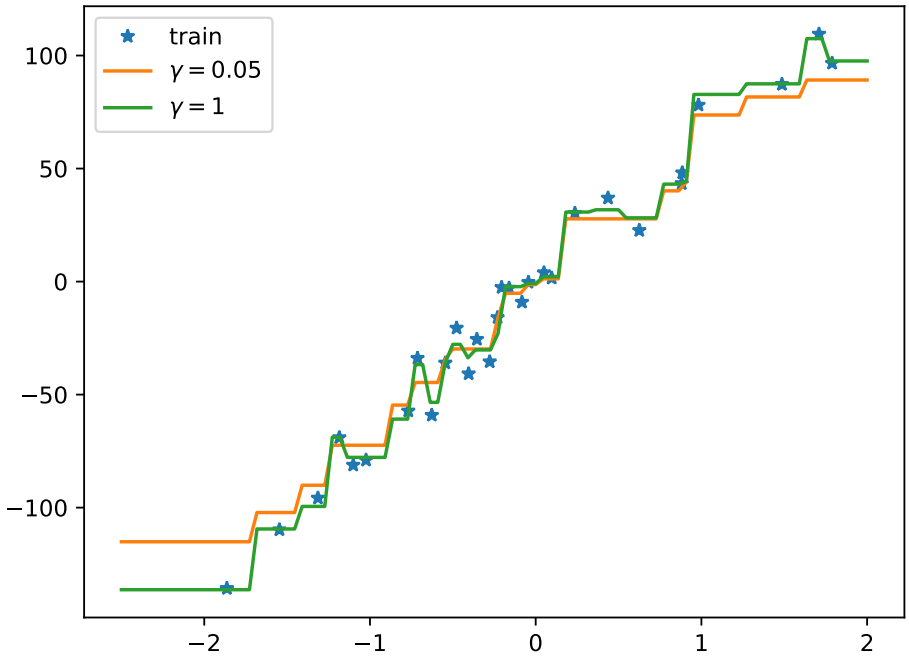
2.4 Effect of 10 rounds

Let us first investigate the effect of $\gamma \in \{0.05, 1\}$ for a small number of rounds, like 10.



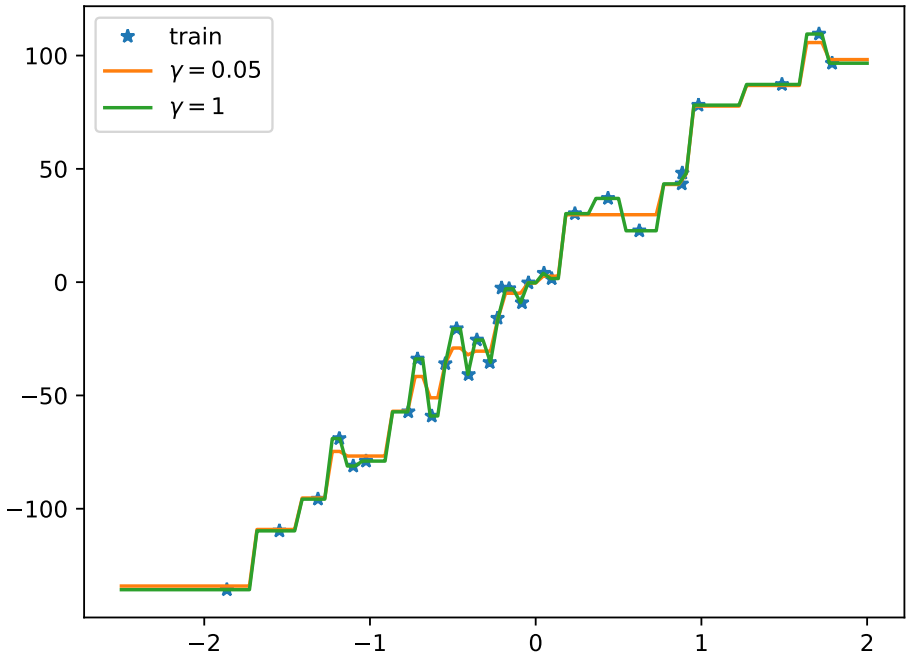
We see that for just 10 round and $\gamma = 1$, we get already quite a good predictor. For $\gamma = 0.05$, the predictor is still near the horizontal line `Y.mean()`.

2.5 Effect of 100 rounds



Using $\gamma = 1$ is over-fitting: each and every point of the train data is followed. For $\gamma = 0.05$ it appears that the predictor is still trying to catch up at the boundaries.

2.6 Effect of 1000 rounds



Clearly, fitting with $\gamma = 0.05$ for 1000 rounds seems to be over-fitting too (at least, when I eye ball the graph). Perhaps using 500 rounds with $\gamma = 0.05$, is better. However, if this much better than using $\gamma = 1$ and just running 10 rounds is debatable.

3 AdaBoost

3.1 Implementation

You can find the code in `adaboost.py` on github.

The implementation for AdaBoost is very similar to boosting with regression. Rather than a `DecisionTreeRegressor` we now need a `DecisionTreeClassifier`.

```
1 import numpy as np
2 from sklearn.datasets import make_blobs
3 from sklearn.tree import DecisionTreeClassifier
```

For my implementation of the `AdaBoost` class, you should know that `x * y` for two `numpy` arrays (with the same dimensions) results in a point wise multiplication, i.e.,

$$x * y = (x_1 y_1, \ldots, x_n y_n). \tag{2}$$

I use this idea to update `w`, and also in the computation of the exponential loss, see below.

In `predict` I use `zip` to iterate over two lists at the same time; very convenient.

```python
class AdaBoost:
    def __init__(self, rounds):
        self.rounds = rounds

    def fit(self, X, Y):
        n, p = X.shape
        w = np.ones(n) / n
        self.alphas = []
        self.learners = []

        for b in range(self.rounds):
            clf = DecisionTreeClassifier(max_depth=1)
            clf.fit(X, Y, sample_weight=w)
            self.learners.append(clf)
            y = clf.predict(X)

            error = w[y != Y].sum() / w.sum()
            alpha = 0.5 * np.log(1 / error - 1)
            self.alphas.append(alpha)
            w *= np.exp(-alpha * y * Y)

    def predict(self, x):
        res = sum(a * l.predict(x) for a, l in zip(self.alphas, self.learners))
        return np.sign(res)
```

3.2 A test

To see whether I get any output at all, I use a simple test.

```python
def exp_loss(y, yhat):
    return np.exp(-y * yhat).sum() / len(y)


def test():
    np.random.seed(4)
    X, Y = make_blobs(n_samples=13, n_features=2, centers=2, cluster_std=20)
    Y = 2 * Y - 1  # don't forget this!

    ada = AdaBoost(8)
    ada.fit(X, Y)
    y = ada.predict(X)
    print(exp_loss(y, Y))
```

4 Optimizing a web site, multi-armed bandits

4.1 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. In other words, we want to turn a visitor into a *customer*.

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 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?

This problem is known in the literature as A/B testing. Here I illustrate the problem for web sites, but the problem occurs in many more different settings, such as medicine testing.

4.2 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.

With regard to point 4: even a 1% difference is interesting when a company receives 10 000 visitors per day. Booking.com (used to) get(s) 1 000 000 (or more ?) visitors per day.

So, let's build a simulator and test some policies on how to assign a page to a new visitor. In other words, we are going to analyze various *policy* that make automatic decisions on which page to give to a visitor.

4.3 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.

```python
import numpy as np

np.random.seed(3)

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

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

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

print(a, b)
```

```
188 0
```

What do we see? Page `a` gets all the visitors. So, obviously the first conversion occurred for page `a`, and therefore the other page never got a new visitor. But we took as conversion probabilities $q > p$, 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?

4.4 A better policy

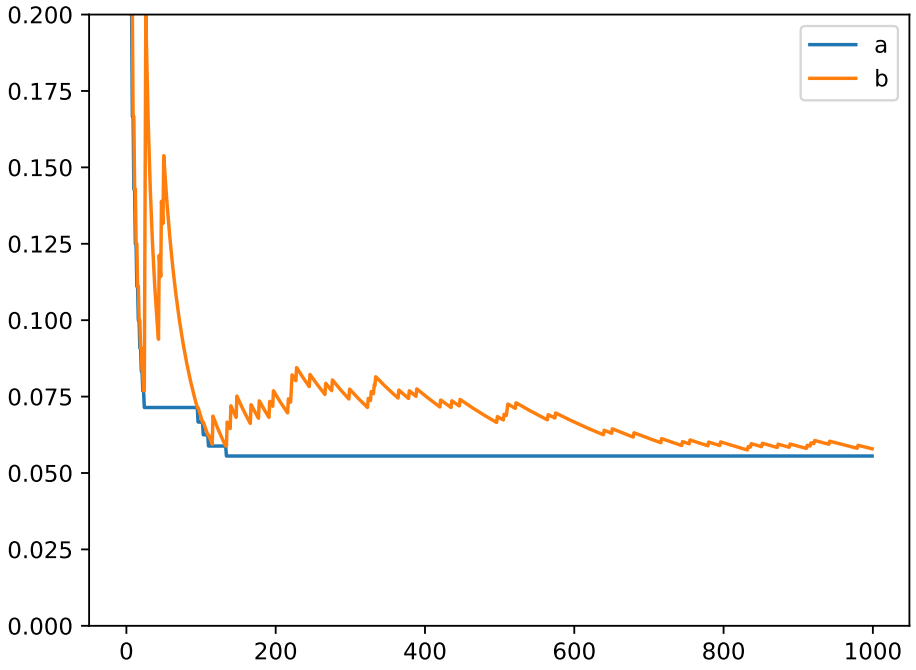
Clearly, a and b do not correspond to the actual conversion probabilities. It must be better to use a/n_a, where n_a is the number of visitors given to type `a`, as an estimator for p. And of course b/n_b for the other page.

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


p, q = 0.02, 0.05


def run(a, b, n_a, n_b, n=1000):
    np.random.seed(3) # ensure the same starting conditions
    success_a = np.random.binomial(1, p, size=n)
    success_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 += success_a[i]
            n_a += 1
        else:
            b += success_b[i]
            n_b += 1
        estimator[i, :] = [a / n_a, b / n_b]
    return estimator
```

Let's make a few plots.

I set $a = b = n_a = n_b = 1$ initially. This prevents divisions by zero, and it ensures that my *prior probabilities* are $1/2$ for both pages.

```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")
```

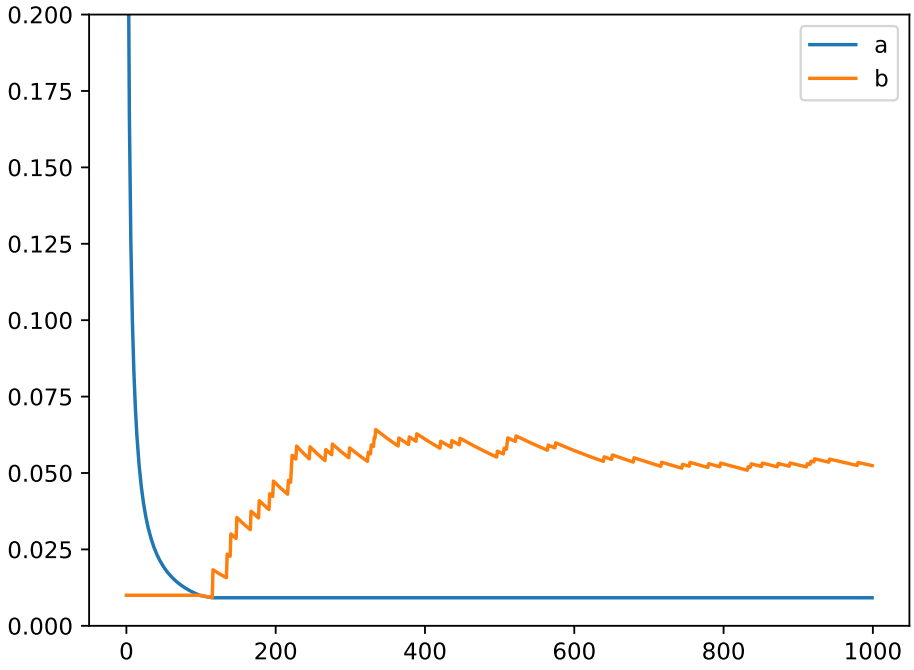


This seems to be OK: we are giving most of the visitors to page `b`. How do I see this in the figure? Well, the estimator for page `a` remains constant for long stretches, which means that it does not get a visitor. And since it does not a visitor, its rate is not estimated well, but I don't care to know what its conversion rate is; for the moment all I want to know is which of the two pages converts better.

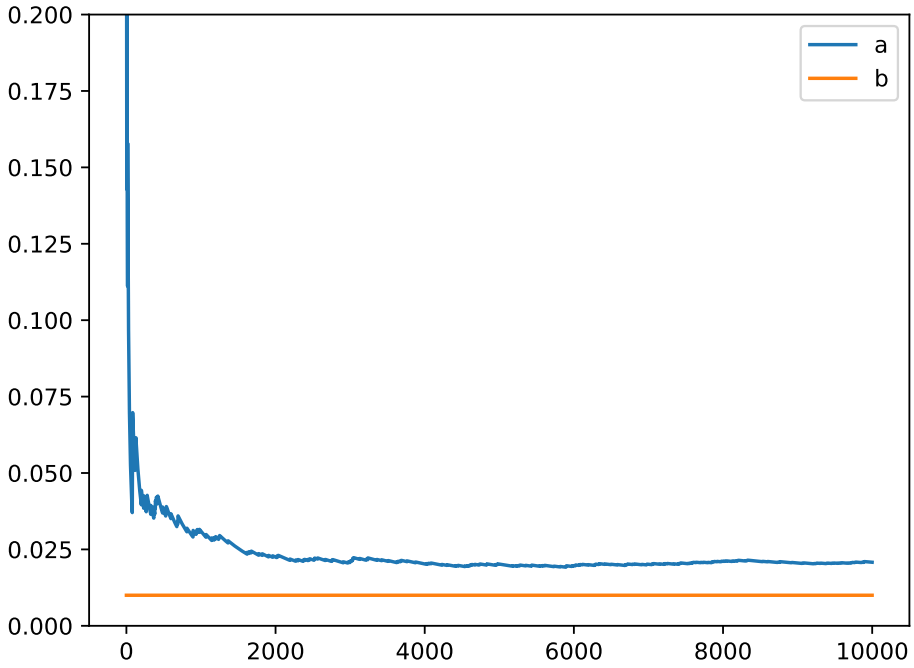
However, is this policy to assign pages to visitors robust? Suppose, by chance, initially we would have given lots of visitors to page `a`. Will we ever recover from such cases of bad luck?

To test, let's set $b/n_b = 1/100$. I take a value of 1% on purpose because I know that $p = 0.02$. My guess is that 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 do find that b is better. However, this is just a single example; let's try a run with the another seed. Then I get this graph.



To conclude, the policy is not robust, because it can happen that we don't send enough visitors to page `b` to get a good estimate for q.

Here is slight digression. I believe that eventually the `b` page will 'win'. But if that takes years to find out, then by the time I found out, the knowledge is not relevant anymore, because in the mean time I have changed lots of other things. Hence, just the fact that a policy is *asymptotically* optimal is not all that relevant. I need a policy that gives good results in a reasonable amount of learning time. (So, now you ask what is 'reasonable', and then I say that there is no guideline for this, but that it depends on the context. Let's not enter that discussion here again. The best guideline is this: Use your common sense! And what common sense is, I don't know.)

This problem is in general known as the *exploration-exploitation* problem. With the above policies it can happen that we believe too rapidly that page `a` is the better of the two alternatives. If we would explore better, or longer, we can come to different conclusions about which alternative is better.

5 Exercises

5.1 Exercise 1.

Compare my implementation of the boosting classes to those in DSML.

5.2 Exercise 1

Next lecture we'll discuss a very smart method to improve the policies we tried above. To get in the mood, you can try to invent a few modifications of the policy and test these. Even if your ideas don't turn out to work well, you'll learn quite a number of useful skills.

5.3 Exercise

Read the math behind AdaBoost in DSML. It's a really clever algorithm, and a joy to read.

5.4 Exercise

Suppose you would use `max_depth=2` in the weak learners. Would that improve the situation in that we need less rounds to get the same error rate as when using `max_depth=1`?

5.5 Exercise

Read on the web on the *exploration-exploitation* problem. (Do your future self a favor, and really do this exercise. The trade-off between exploration and exploitation is very important to be aware of. Even if you don't apply it mathematically, it is important to recognize when learning new things.) If you come across Thompson sampling, then you can skip that, because we will deal with this in code lecture 7. Or. read it, so that you are better prepared.

5.6 Exercise

Think a bit about how you could repair this.

5.7 Exercise

Can you modify the above policies for the multi-armed bandit such they can deal with non-stationary demand? Here is dumb example: visitors tend to prefer the green button in the morning (i.e., the green button converts better), but favor the red one more in the evening. Can you design a measurement policy that helps to find this out? How to adapt the policy to exploit this? (Of course, in real life, you don't know that visitors prefer green over red buttons, you can only use the data to make hunches.)