Machine Learning

Lecture 5.7

Loss functions and Gradient Descent

Boosting is a technique which has been very successful as of late. It’s performance has rivalled and in many cases surpassed the methods using *Deep Learning (Neural Nets).*

Adaptive Boosting (AdaBoost) is based on the idea of building up a classifier from “weak” classifiers, like a tree with only 2 nodes where the decision boundary is just a hyperplane parallel to a coordinate plane. In 2 dimensions it is just a line parallel to the x-axis or the y-axis. So the classifier just divides the data space into two half-spaces. We can, however also use more complicated classifiers like RandomForests

In the classifiers we have looked at so far, the classifier outputs a label, in fact a data point goes through a decision tree and ends up in a leaf. The output is then the majority label of the data points in the leaf.

We could also have computed the frequencies of the labels and the probability distribution over the labels in the leaf and then output the label with the highest probability.

In order to understand boosting we need to understand the notion of loss function and algorithms to minimize loss functions.

In our previous examples we have used accuracy of predictions on the training set as our measure of performance, and training amounts to optimizing this accuracy.

A boosting classifier outputs real numbers and so we first need to use these real numbers to output a label

Here is an example: consider a data set $\{x_i, y_i\}$ where the y_i are the labels. Let's say there are k labels.

Instead of a classifier explicitly outputting a label, assume it outputs a k-dimensional vector, $f(x) = (f_1(x), f_2(x), \ldots, f_k(x))$ i.e. k real numbers. We then have to turn this vector into a prediction of a label.

We consider the softmax function:

$$p(y_i) = \frac{\exp(f_i(x))}{\sum_{j=1,2,\ldots k} \exp(f_j(x))}$$

This defines a probability distribution over the set of labels, indeed each $p(y_j) \geq 0$ and they sum to 1.

What is the true distribution over the labels? The true label attached to x_i is y_i with certainty so the true distribution is

$$q(y_i) = 1, \qquad q(y_j) = 0, j \neq i$$

To use our classifier to predict a label, y we would choose the label with the highest softmax probability $p(y)$. So in order for this to be the true label we would want the softmax distribution to be “close” to the true distribution q

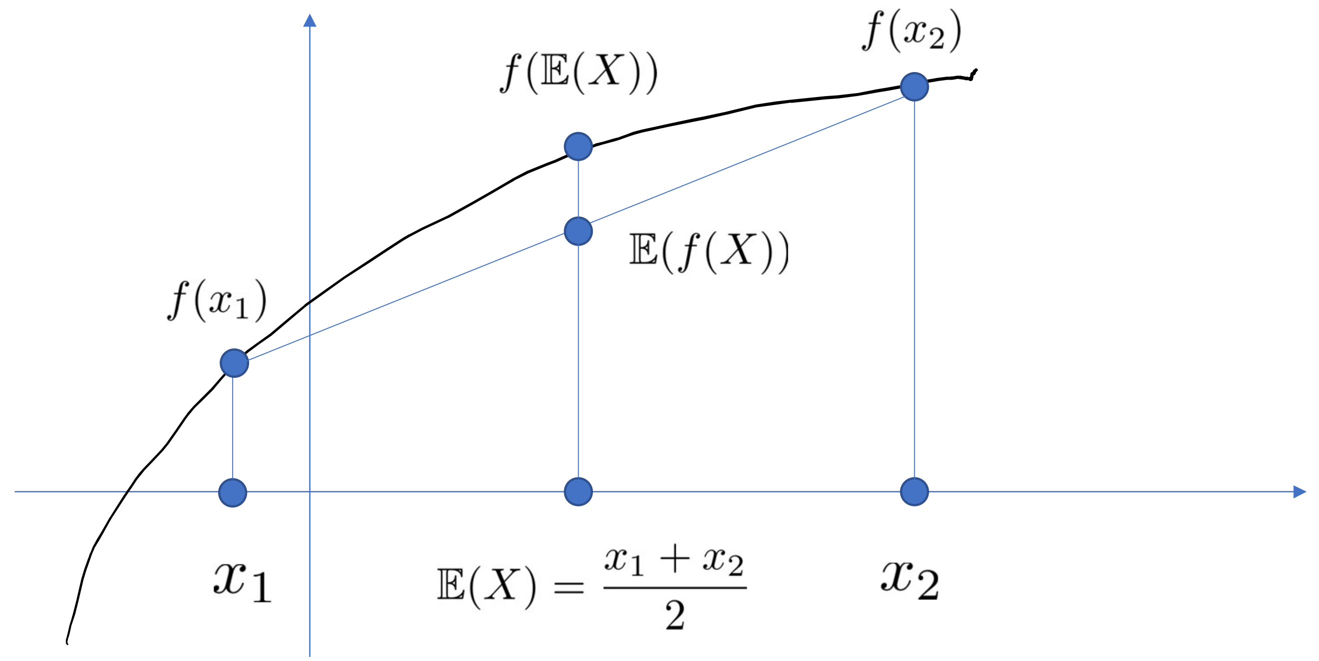
One way to measure how close two probability distributions are, is the Kullback-Leibler Divergence. It is defined by

$$D_{KL}(q||p) = -\sum_j q(y_i) \log \frac{p(y_i)}{q(y_i)}$$

If $p = q$ so $p(y_i) = q(y_i), \forall i$ then clearly $D_{KL}(q,p) = 0$, also $D_{KL}(q,p) \geq 0$ with equality only if $p = q$. This follows from Jensen's inequality:

If f is a concave function and X is a random variable then

$$\mathbb{E}_Q(f(X)) \leq f(\mathbb{E}_Q(X))$$


$f(x_2)$
$f(\mathbb{E}(X))$
$\mathbb{E}(f(X))$
$f(x_1)$
x_1
$\mathbb{E}(X) = \frac{x_1 + x_2}{2}$
x_2

We notice that

$$D_{KL}(q||p) = -\mathbb{E}_q(\log \frac{p}{q})$$

and since $\log$ is a concave function

$$\mathbb{E}_q(\log \frac{p}{q}) \leq \log \mathbb{E}_q(\frac{p}{q})$$

But

$$\mathbb{E}_q(\frac{p}{q}) = \sum_j q(y_j) \frac{p(y_j)}{q(y_j)} = \sum_j p(y_j) = 1$$

The KL-divergence between the true distribution and the softmax distribution is

$$-\log p(y_i) = -\log \left(\frac{\exp(f_i(x)}{\sum_j \exp(f_j(x))} \right)$$

so this is the expression we want to minimize for every data point. We can then define the softmax loss function

$$\mathcal{L}(\{x_t\}_t, f) = -\frac{1}{N} \sum_t \log \left(\frac{\exp(f_i(x_t))}{\sum_j \exp(f_j(x_t))} \right)$$

Usually the output function f will depend on a set of parameters Θ and so training the model means finding the set of parameters that minimizes the loss function. The softmax loss function is a convex function so we can find the minimum by solving the equation

$$\nabla_\Theta \mathcal{L}(\{x_t\}_t, f_\Theta) = 0$$

This is usually not possible to directly solve this equation so we resort to iterative methods to find approximate solutions.

The idea is to start with some initial value Θ_0 and then take small steps in the direction of the minimum and that way approach the Θ that minimizes the loss function

To figure out which direction to go consider the Taylor series expansion

$$\mathcal{L}(\Theta_0 - \Delta\Theta) \simeq \mathcal{L}(\Theta_0) - \nabla_\Theta \mathcal{L}|_{\Theta_0} \cdot \Delta\Theta$$

Thus we get the largest decrease in the value of the loss function when $|\nabla_\Theta \mathcal{L}|_{\Theta_0} \cdot \Delta\Theta|$ is maximal. From the Cauchy-Schwartz inequality we have

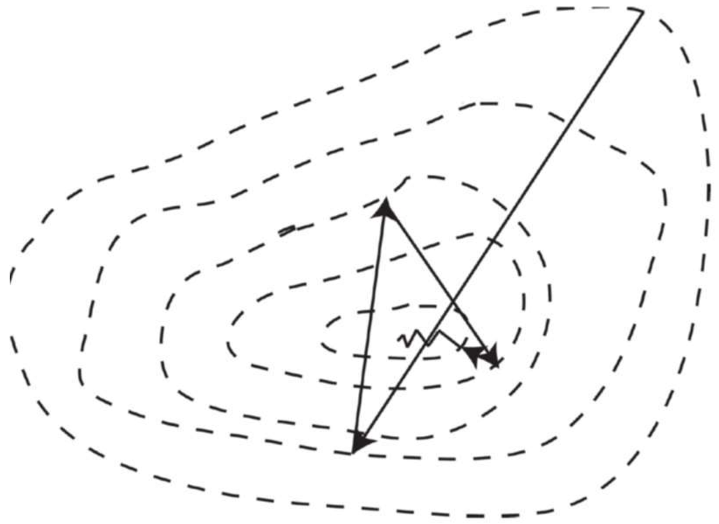
$$|\nabla_\Theta \mathcal{L}|_{\Theta_0} \cdot \Delta\Theta| \leq ||\nabla_\Theta \mathcal{L}|_{\Theta_0}||||\Delta\Theta||$$

with equality precisely when

$$\Delta\Theta = \mu \nabla_\Theta \mathcal{L}|_{\Theta_0}$$

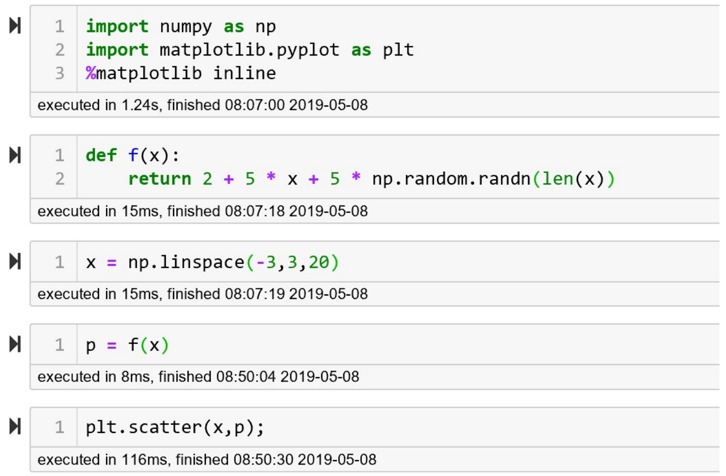
i.e. the increment is in the direction of the gradient

So the iterative algorithm, which is known a Gradient Descent is to take steps in the opposite direction of the gradient. The factor μ is called *the learning rate* and a good value for the learning rate has to be determined. If it is too small we may get very slow convergence and if it is too large we may not get convergence at all.



There are many kinds of loss functions for instance for regression problems we can use the squared error as the loss function.

Let's code up a simple example:



```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

executed in 1.24s, finished 08:07:00 2019-05-08

```
def f(x):
    return 2 + 5 * x + 5 * np.random.randn(len(x))
```

executed in 15ms, finished 08:07:18 2019-05-08

```
x = np.linspace(-3,3,20)
```

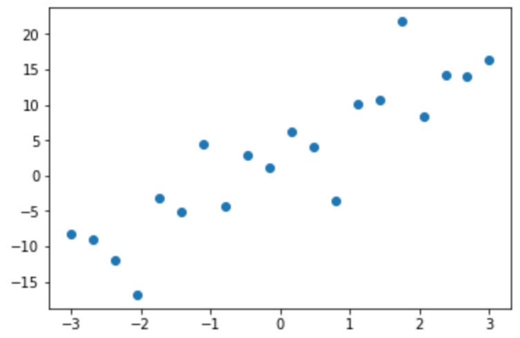
executed in 15ms, finished 08:07:19 2019-05-08

```
p = f(x)
```

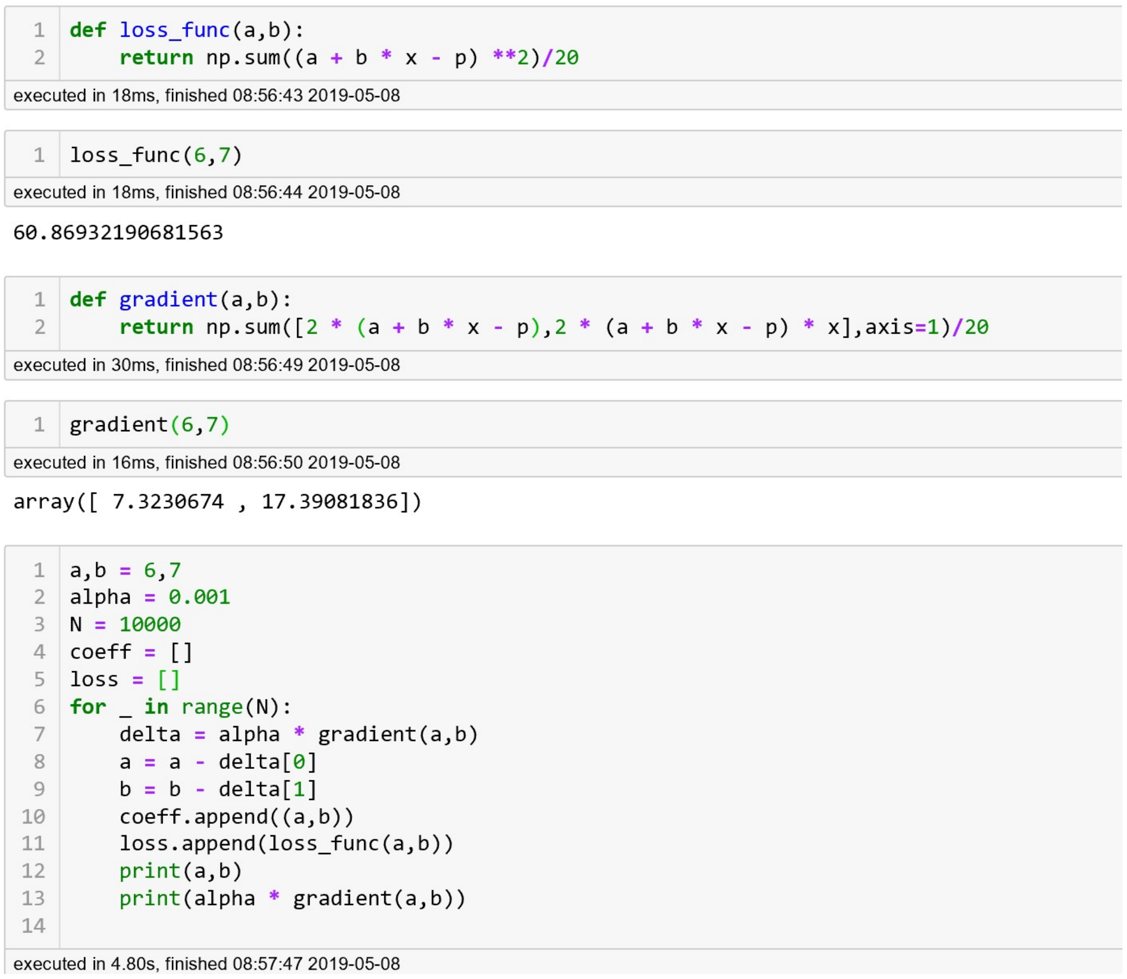
executed in 8ms, finished 08:50:04 2019-05-08

```
plt.scatter(x,p);
```

executed in 116ms, finished 08:50:30 2019-05-08



Gradient descent with learning rate = 0.001 and 10000 steps



```
def loss_func(a,b):
    return np.sum((a + b * x - p) **2)/20
```

executed in 18ms, finished 08:56:43 2019-05-08

```
loss_func(6,7)
```

executed in 18ms, finished 08:56:44 2019-05-08

```
60.86932190681563
```

```
def gradient(a,b):
    return np.sum([2 * (a + b * x - p),2 * (a + b * x - p) * x],axis=1)/20
```

executed in 30ms, finished 08:56:49 2019-05-08

```
gradient(6,7)
```

executed in 16ms, finished 08:56:50 2019-05-08

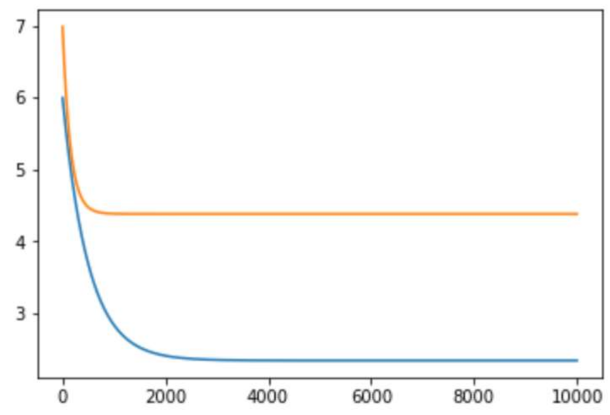
```
array([ 7.3230674 , 17.39081836])
```

```
a,b = 6,7
alpha = 0.001
N = 10000
coeff = []
loss = []
for _ in range(N):
    delta = alpha * gradient(a,b)
    a = a - delta[0]
    b = b - delta[1]
    coeff.append((a,b))
    loss.append(loss_func(a,b))
    print(a,b)
    print(alpha * gradient(a,b))

```

executed in 4.80s, finished 08:57:47 2019-05-08

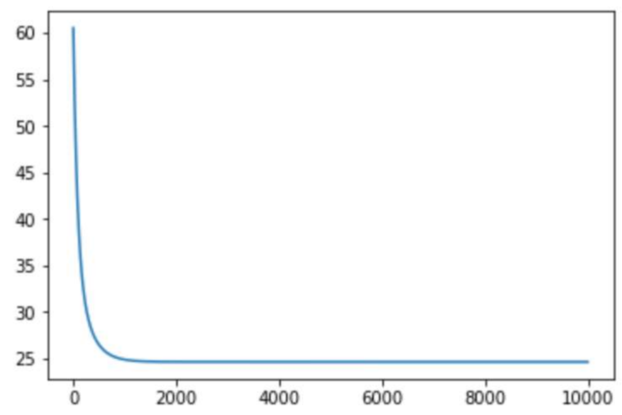
```
plt.plot(range(N),coeff);
```

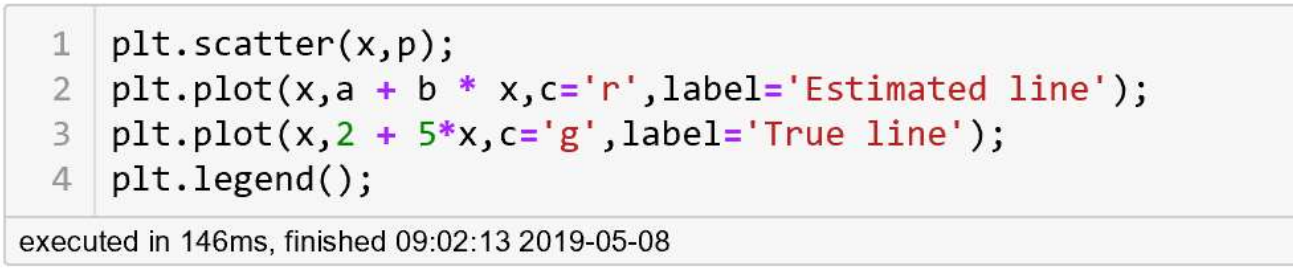
```
executed in 111ms, finished 08:59:17 2019-05-08
```



```
plt.plot(range(N),loss);
```

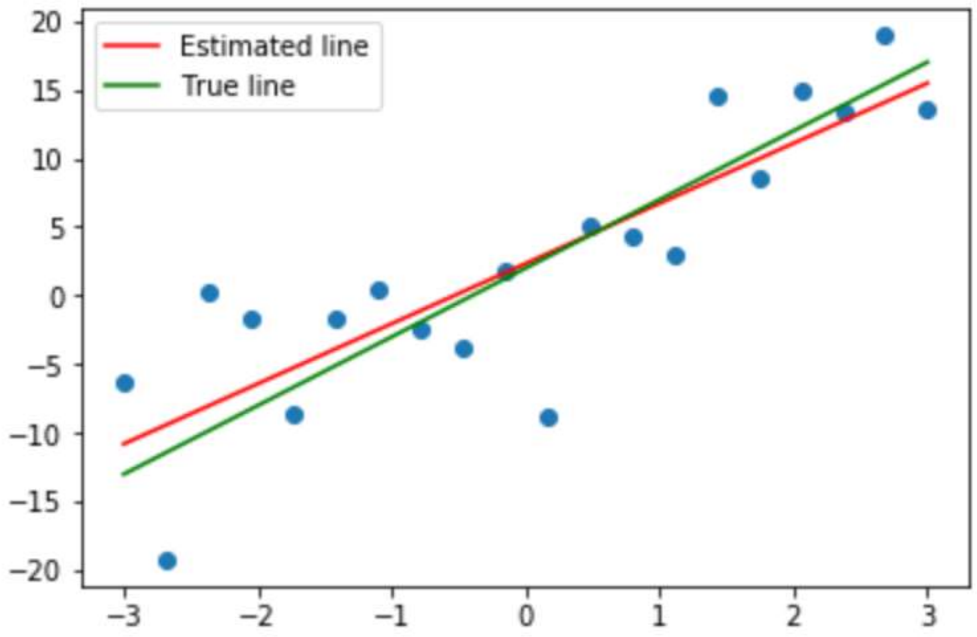
```
executed in 129ms, finished 08:59:21 2019-05-08
```



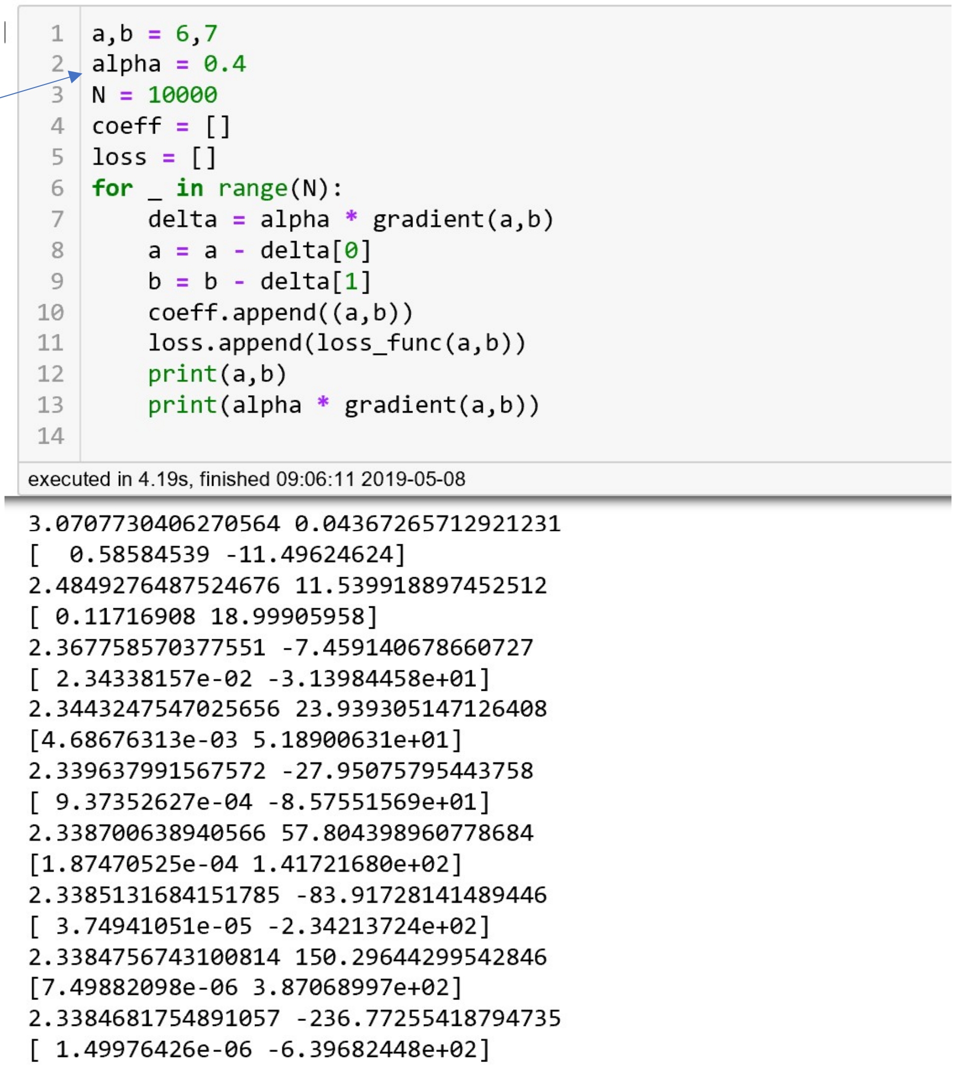


```
plt.scatter(x,p);
plt.plot(x,a + b * x,c='r',label='Estimated line');
plt.plot(x,2 + 5*x,c='g',label='True line');
plt.legend();
```

```
executed in 146ms, finished 09:02:13 2019-05-08
```



If we take the learning rate too large we do not get convergence

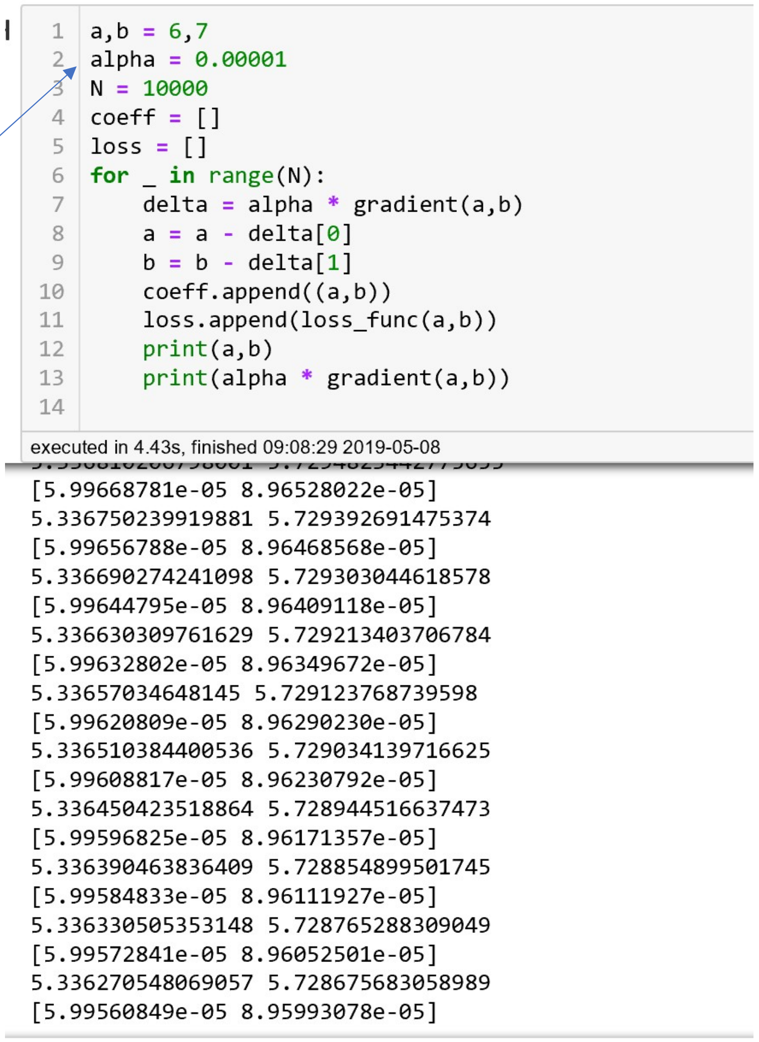


```
a,b = 6,7
alpha = 0.4
N = 10000
coeff = []
loss = []
for _ in range(N):
    delta = alpha * gradient(a,b)
    a = a - delta[0]
    b = b - delta[1]
    coeff.append((a,b))
    loss.append(loss_func(a,b))
    print(a,b)
    print(alpha * gradient(a,b))

```

executed in 4.19s, finished 09:06:11 2019-05-08

```
3.0707730406270564 0.04367265712921231
[  0.58584539 -11.49624624]
2.4849276487524676 11.539918897452512
[ 0.11716908 18.99905958]
2.367758570377551 -7.459140678660727
[ 2.34338157e-02 -3.13984458e+01]
2.3443247547025656 23.939305147126408
[4.68676313e-03 5.18900631e+01]
2.339637991567572 -27.95075795443758
[ 9.37352627e-04 -8.57551569e+01]
2.338700638940566 57.804398960778684
[1.87470525e-04 1.41721680e+02]
2.3385131684151785 -83.91728141489446
[ 3.74941051e-05 -2.34213724e+02]
2.3384756743100814 150.29644299542846
[7.49882098e-06 3.87068997e+02]
2.3384681754891057 -236.77255418794735
[ 1.49976426e-06 -6.39682448e+02]
```

If we take the learning rate too small we get very slow convergence



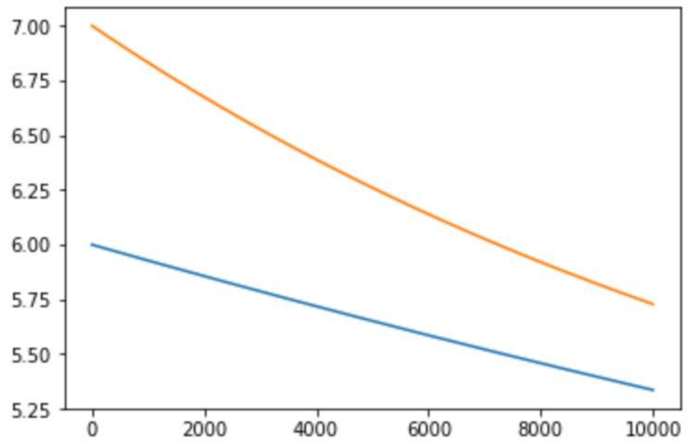
```python
a,b = 6,7
alpha = 0.00001
N = 10000
coeff = []
loss = []
for _ in range(N):
    delta = alpha * gradient(a,b)
    a = a - delta[0]
    b = b - delta[1]
    coeff.append((a,b))
    loss.append(loss_func(a,b))
    print(a,b)
    print(alpha * gradient(a,b))

```

executed in 4.43s, finished 09:08:29 2019-05-08

```
[5.99668781e-05 8.96528022e-05]
5.336750239919881 5.729392691475374
[5.99656788e-05 8.96468568e-05]
5.336690274241098 5.729303044618578
[5.99644795e-05 8.96409118e-05]
5.336630309761629 5.729213403706784
[5.99632802e-05 8.96349672e-05]
5.33657034648145 5.729123768739598
[5.99620809e-05 8.96290230e-05]
5.336510384400536 5.729034139716625
[5.99608817e-05 8.96230792e-05]
5.336450423518864 5.728944516637473
[5.99596825e-05 8.96171357e-05]
5.336390463836409 5.728854899501745
[5.99584833e-05 8.96111927e-05]
5.336330505353148 5.728765288309049
[5.99572841e-05 8.96052501e-05]
5.336270548069057 5.728675683058989
[5.99560849e-05 8.95993078e-05]
```

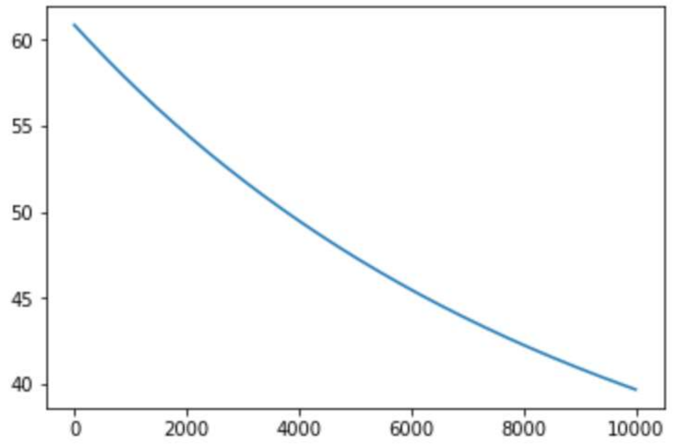
```
plt.plot(range(N),coeff);
```

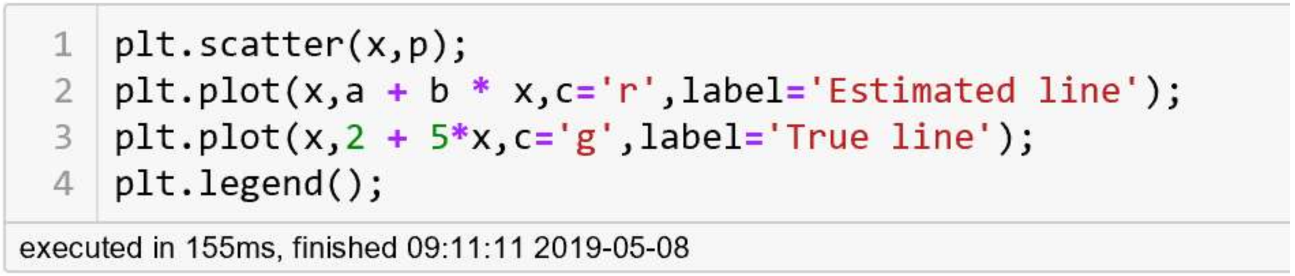
executed in 153ms, finished 09:10:32 2019-05-08



```
plt.plot(range(N),loss);
```

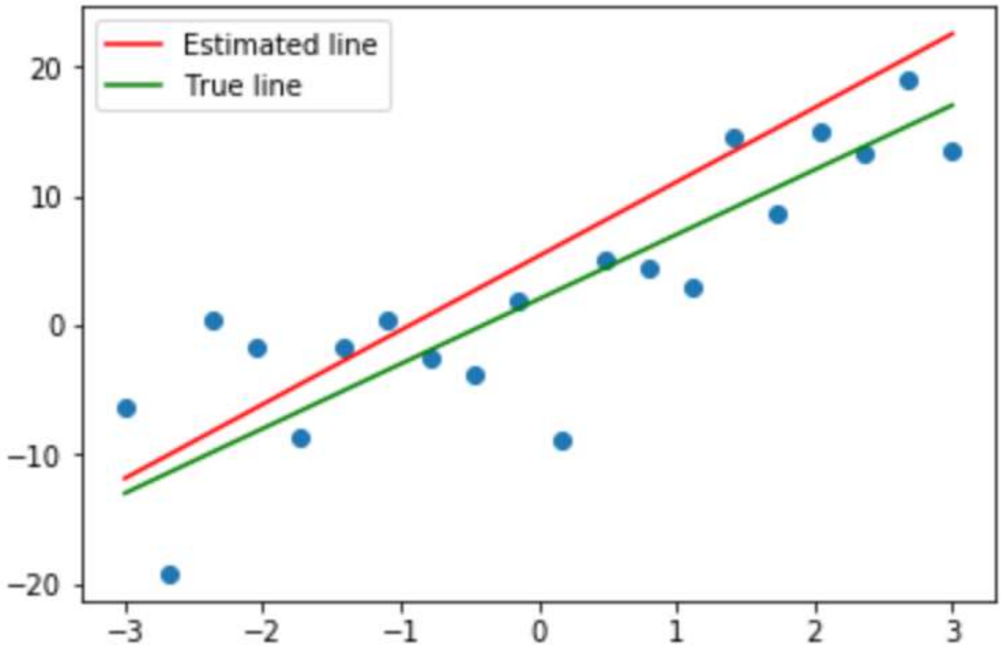
executed in 151ms, finished 09:10:33 2019-05-08



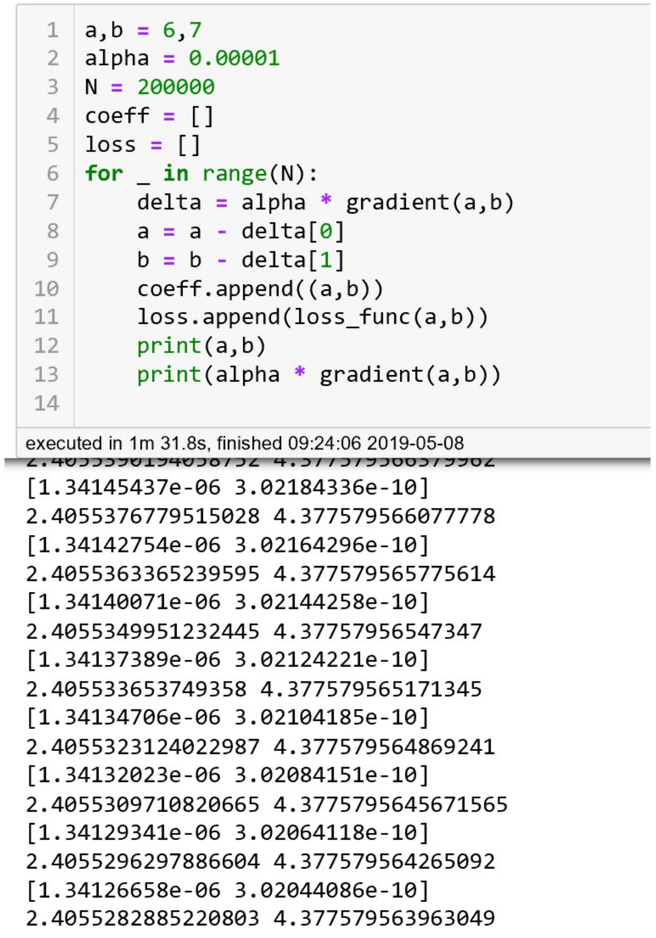


```
plt.scatter(x,p);
plt.plot(x,a + b * x,c='r',label='Estimated line');
plt.plot(x,2 + 5*x,c='g',label='True line');
plt.legend();
```

executed in 155ms, finished 09:11:11 2019-05-08



After 200,000 steps it gets there



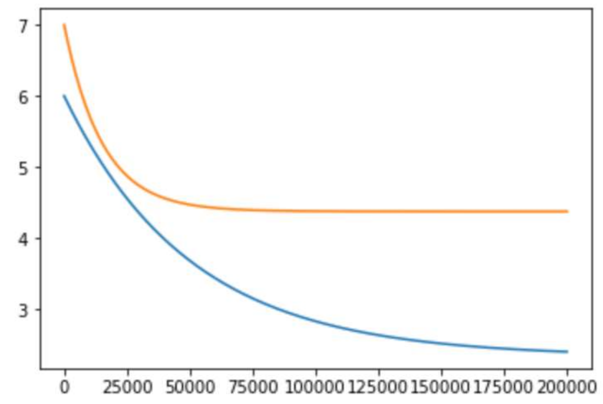
```
a,b = 6,7
alpha = 0.00001
N = 200000
coeff = []
loss = []
for _ in range(N):
    delta = alpha * gradient(a,b)
    a = a - delta[0]
    b = b - delta[1]
    coeff.append((a,b))
    loss.append(loss_func(a,b))
    print(a,b)
    print(alpha * gradient(a,b))

```

executed in 1m 31.8s, finished 09:24:06 2019-05-08

```
[1.34145437e-06 3.02184336e-10]
2.4055376779515028 4.377579566077778
[1.34142754e-06 3.02164296e-10]
2.4055363365239595 4.377579565775614
[1.34140071e-06 3.02144258e-10]
2.4055349951232445 4.37757956547347
[1.34137389e-06 3.02124221e-10]
2.405533653749358 4.377579565171345
[1.34134706e-06 3.02104185e-10]
2.4055323124022987 4.377579564869241
[1.34132023e-06 3.02084151e-10]
2.4055309710820665 4.3775795645671565
[1.34129341e-06 3.02064118e-10]
2.4055296297886604 4.377579564265092
[1.34126658e-06 3.02044086e-10]
2.4055282885220803 4.377579563963049
```

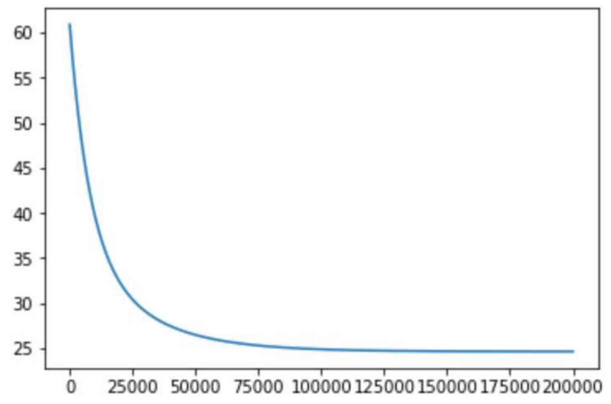
```
plt.plot(range(N),coeff);
```

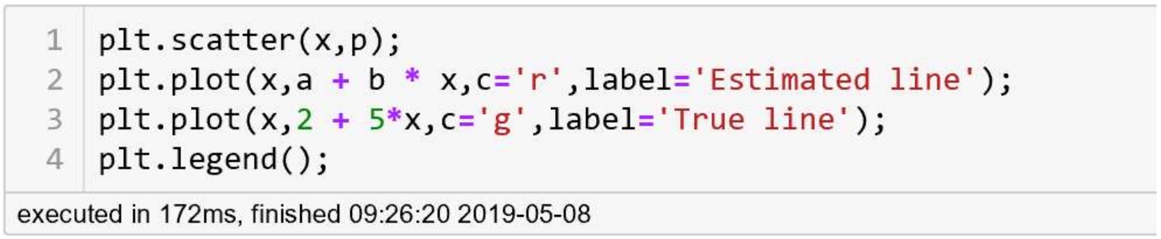
executed in 287ms, finished 09:24:16 2019-05-08



```
plt.plot(range(N),loss);
```

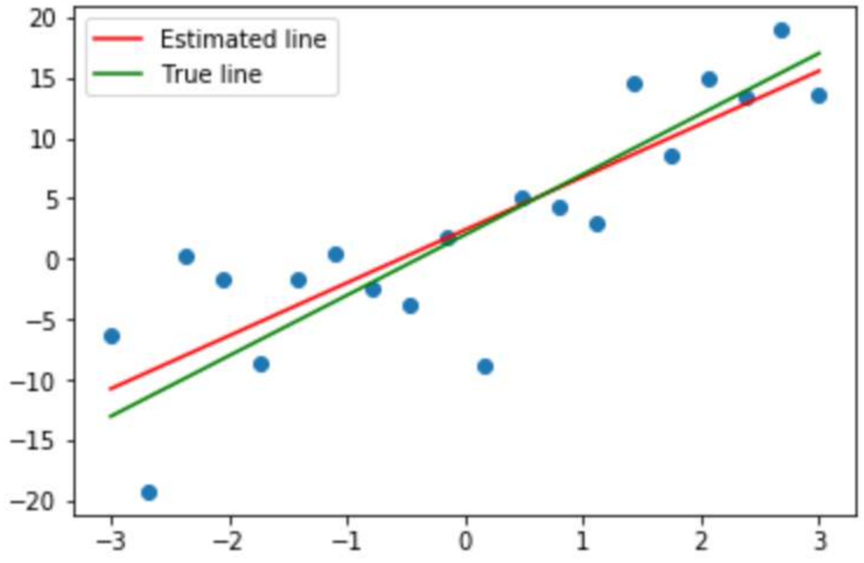
executed in 215ms, finished 09:24:20 2019-05-08





```python
plt.scatter(x,p);
plt.plot(x,a + b * x,c='r',label='Estimated line');
plt.plot(x,2 + 5*x,c='g',label='True line');
plt.legend();
```

```
executed in 172ms, finished 09:26:20 2019-05-08
```


Estimated line
True line