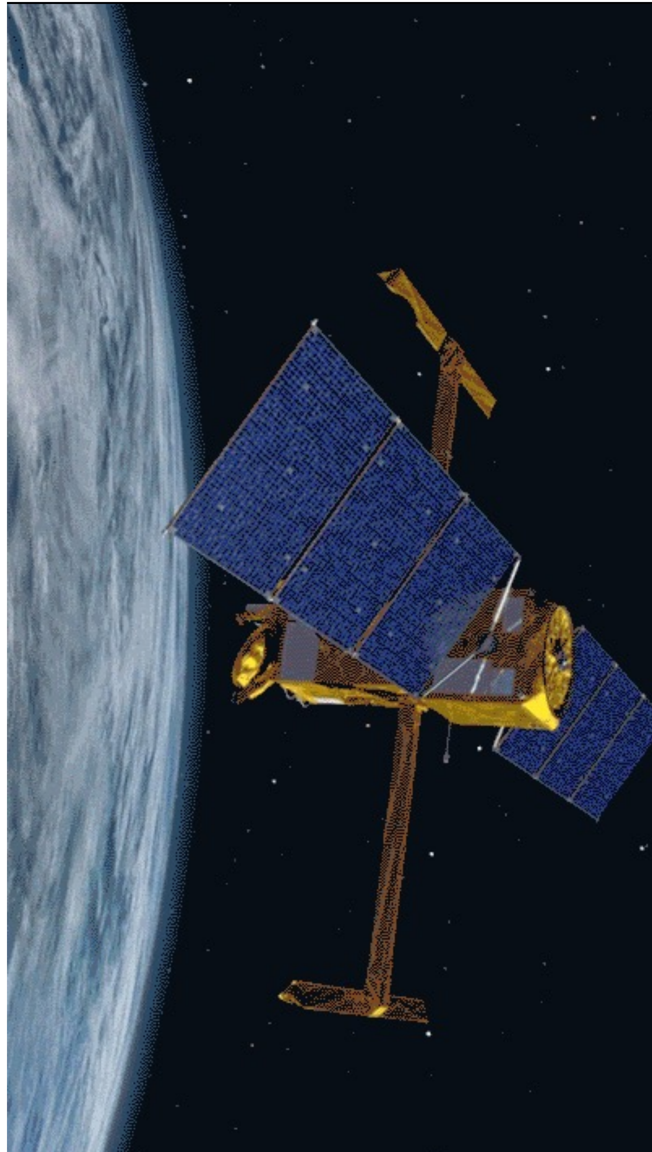
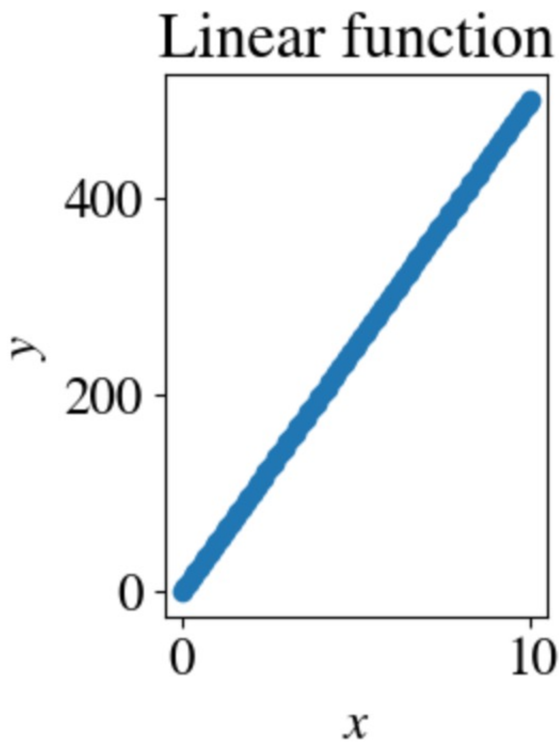
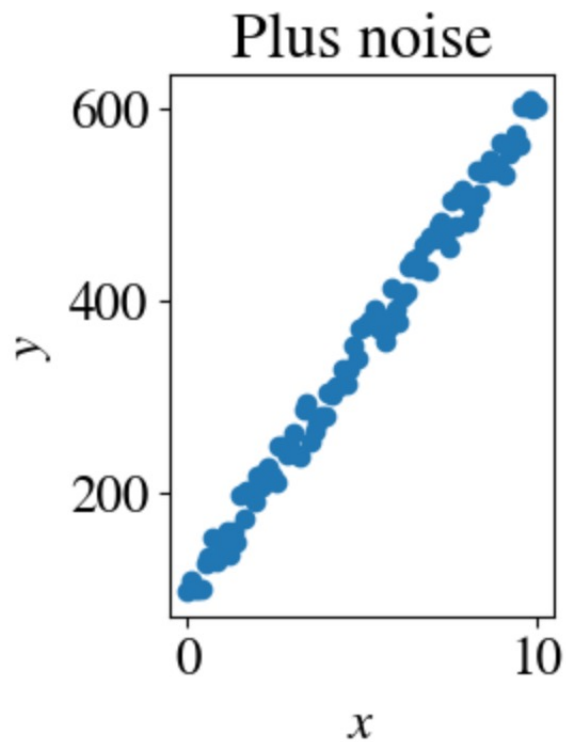


Lecture 2
Linear Numerical Regression

Problem Statement





- **1 Feature:** x (m values/**samples**)
- 1 **Target**: y
- Supervised learning: m known labels y_{true}
- **Hypothesis**: $y_{\text{pred}} = \theta_0 + \theta_1 x$

- Example: $\theta_0 = 100$, $\theta_1 = 50$
- Mean-squared-error (MSE):

$$\frac{1}{2\,m}\sum_{i=1}^{m}(y_{i,\text{pred}} - y_{i,\text{true}})^2 = 106$$

Objective of Linear Regression:

Fit linear function to (generally noisy) data, i.e.:

Find slope (θ_1) & intersect (θ_0) with smallest MSE between y_{true} and $y_{\text{pred}} = \theta_0 + \theta_1 x$ for feature x.

How?

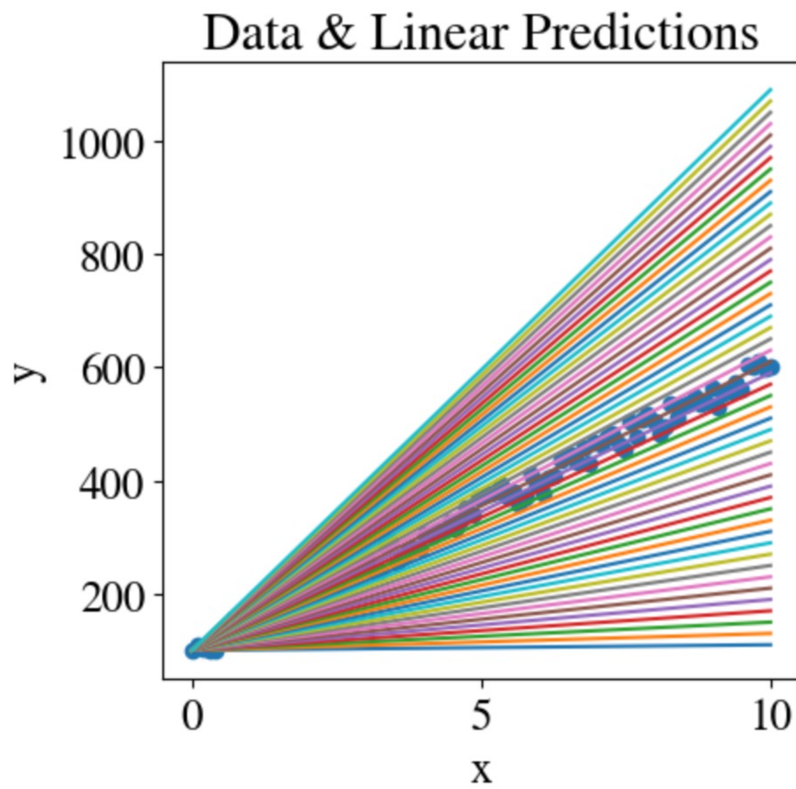
Brute force fit of slope

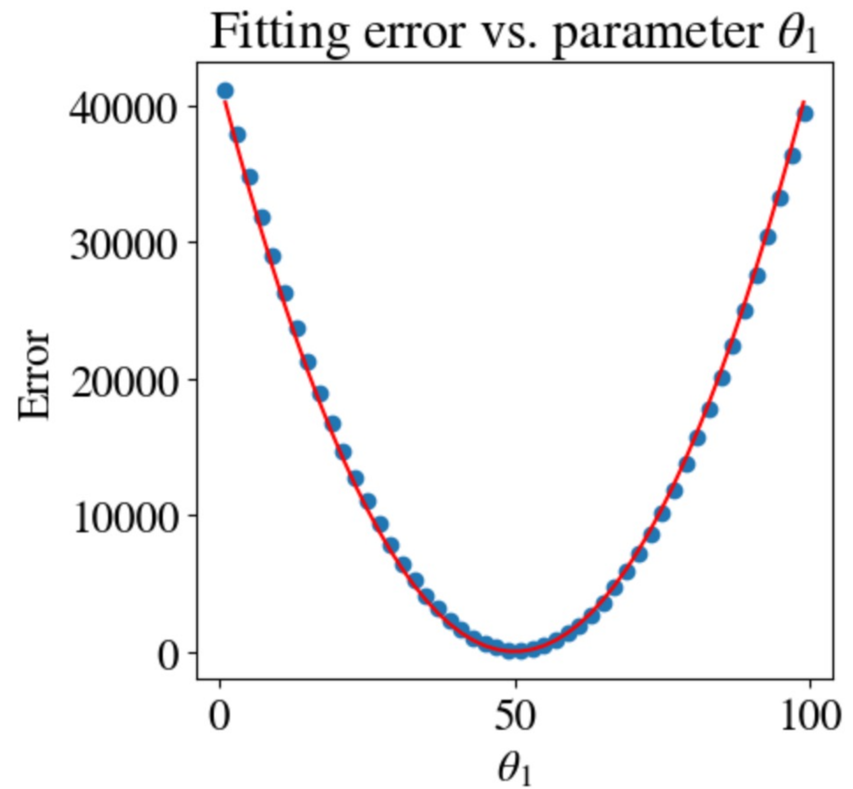
- Simplest example, consider known intercept θ_0 = 100 and only fit θ_1:

```
1  theta0 = 100
2  errors = np.zeros(50)
3  theta  = np.zeros(50)
4
5  for i in range(50):
6      theta[i] = 1 + i*2
7      y_pred = theta0 + theta[i]*x
8      errors[i] = mean_squared_error(y_pred , y)/2
```

Result

- Best fit: θ_1 = 51.0
- **CPU time: 650 ms**
- **MSE = 105.8**





Interpretation of error plot

- If 'true' values were exactly linear (slope of 50) without noise:

$$MSE = \frac{1}{2\,m}\sum_{i=1}^{m}(y_{i,\text{pred}} - y_{i,\text{true}})^2 =$$

$= \frac{1}{2\,m}\sum_{i=1}^{m}((100 + \theta_1 x_i) - (100 + 50\,x_i))^2 =$

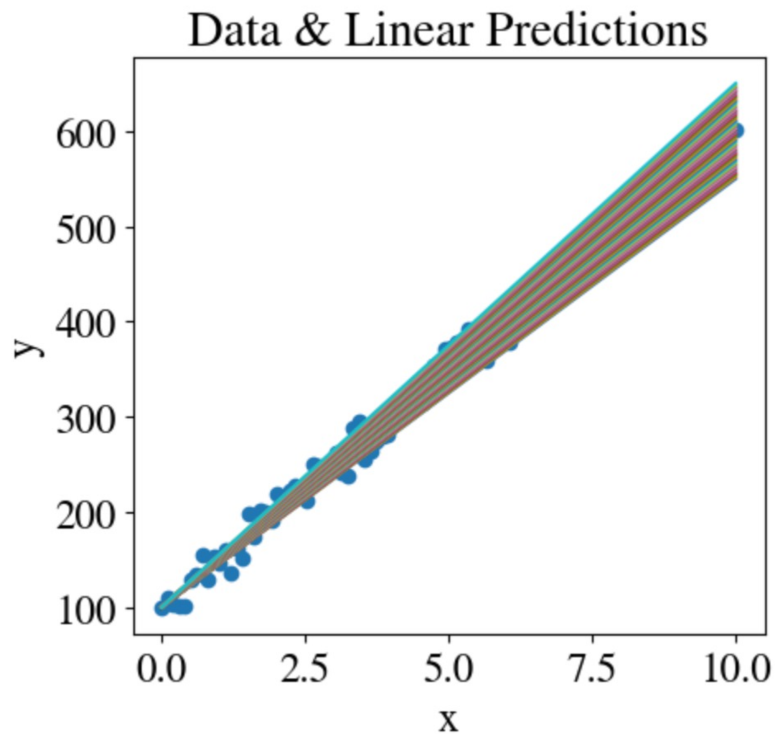
$= (\theta_1 - 50)^2 \times \frac{1}{2\,m}\sum_{i=1}^{m}(x_i)^2 \;\approx 17(\theta_1 - 50)^2$

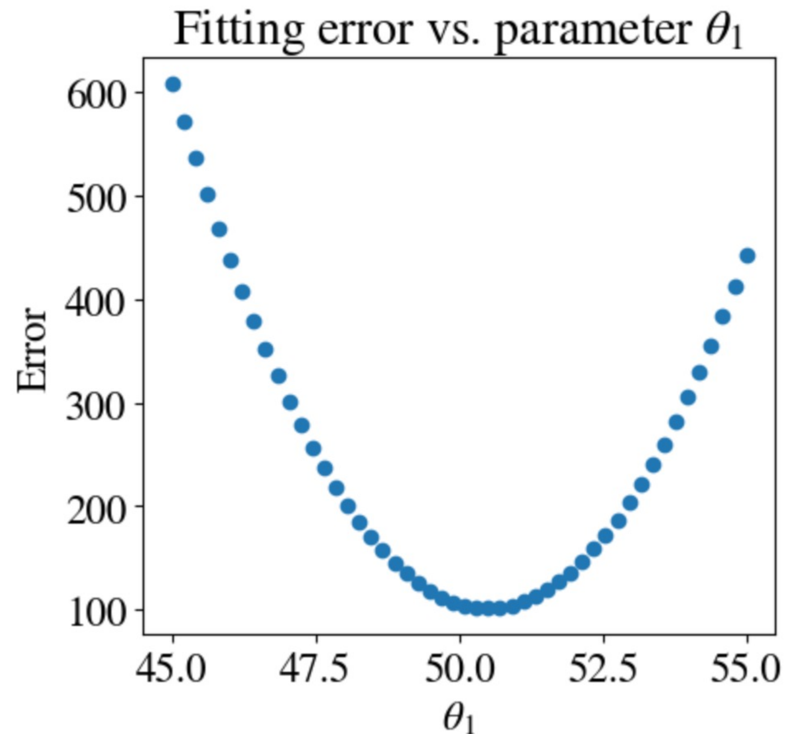
- How do we improve this?

Try narrower range

- Best fit: θ_1 = 50.5
- **CPU time: 650 ms**
- **MSE = 101.6**

- 45 < θ_1 < 55, again in 50 steps





Brute force fit of slope **and** intercept

```
nrfits = 100

errors = np.zeros([nrfits,nrfits])
theta  = np.zeros(nrfits)
theta0 = np.zeros(nrfits)

for j in range(nrfits):
    theta0[j] = j * 200/nrfits
    for i in range(nrfits):
        theta[i] = i * 100/nrfits
        y_pred = theta0[j] + theta[i] * x
        errors[i,j] = mean_squared_error(y_pred , y)/2
```

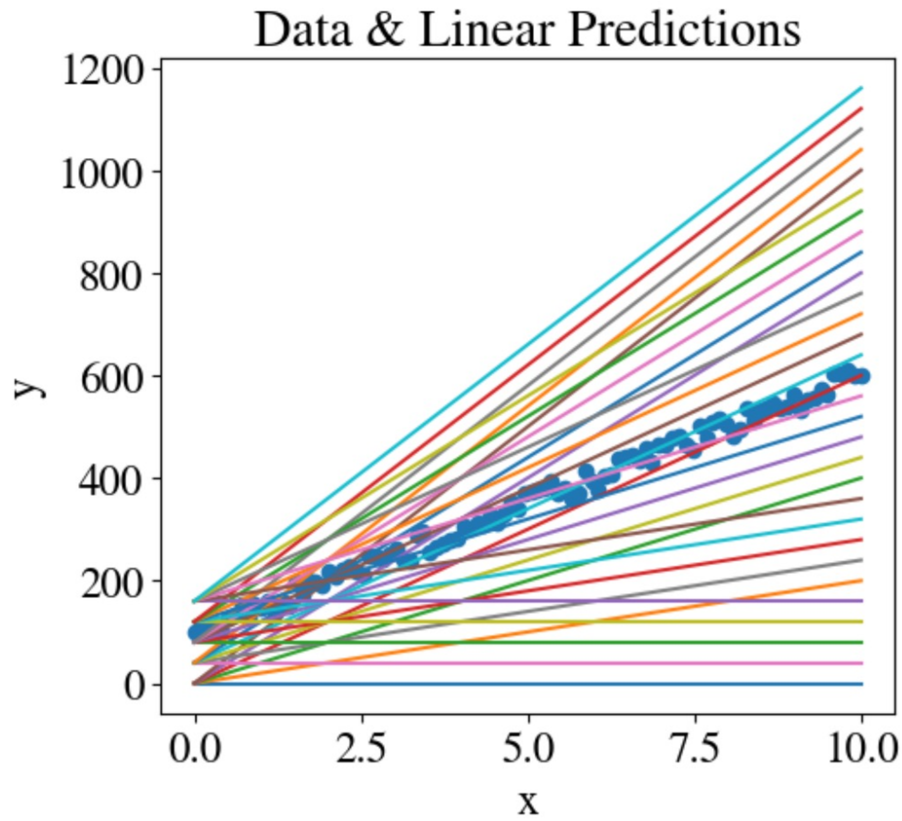
Better:

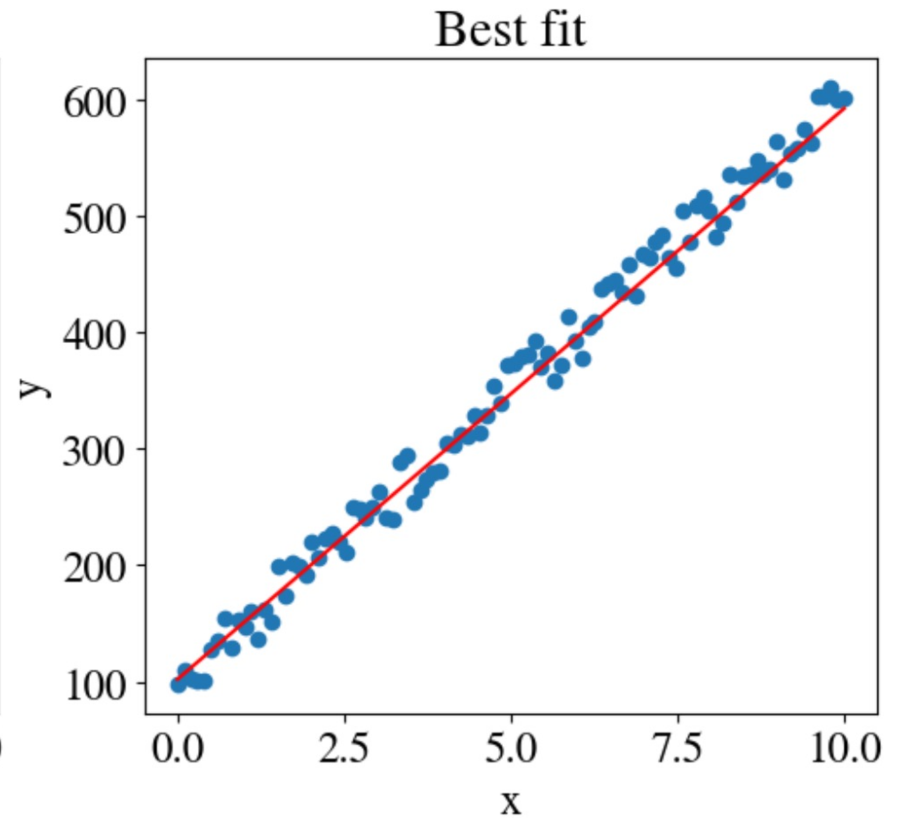
theta0 = np.linspace(0,200,nrfits)

theta = np.linspace(0,100,nrfits)

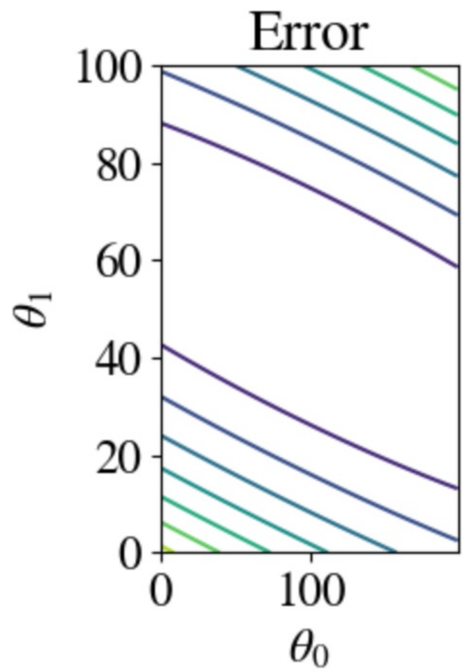
Result

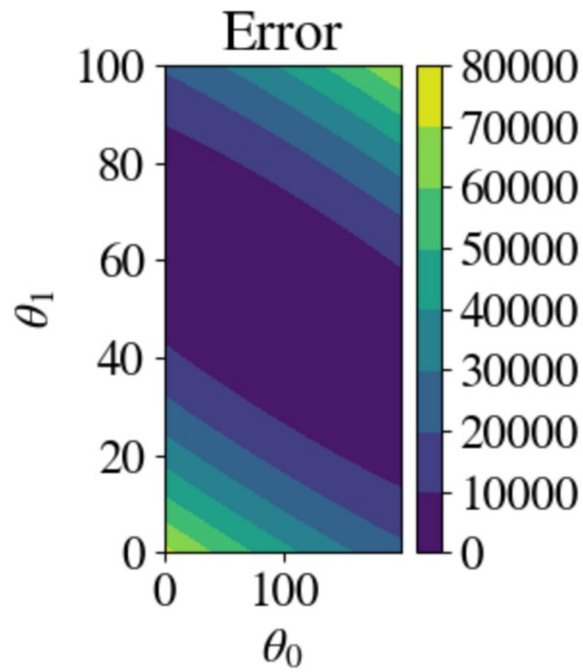
- Best fit: θ_0 = 102, and θ_1 = 49
- 10,000 tries
- **CPU time: 5.1 s**
- **MSE = 102.2**

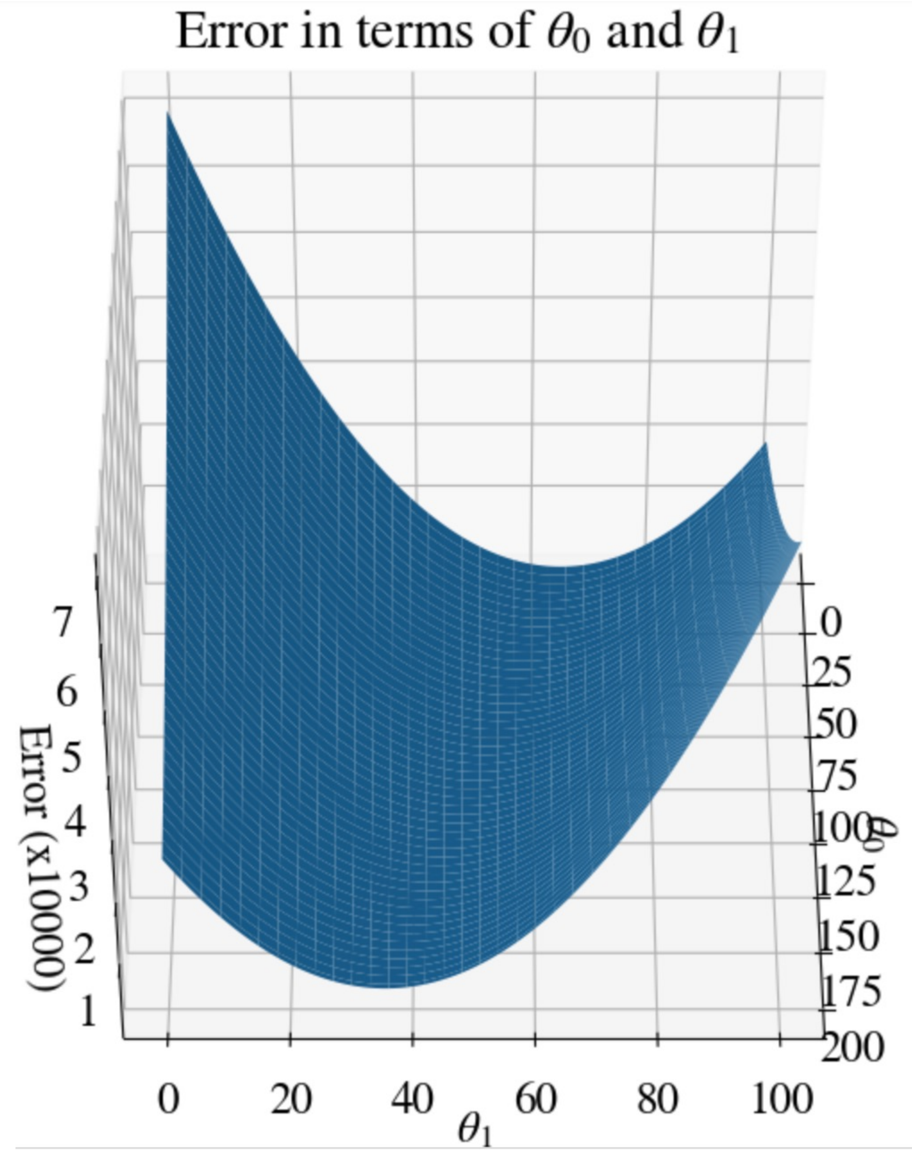




MSE in 2D

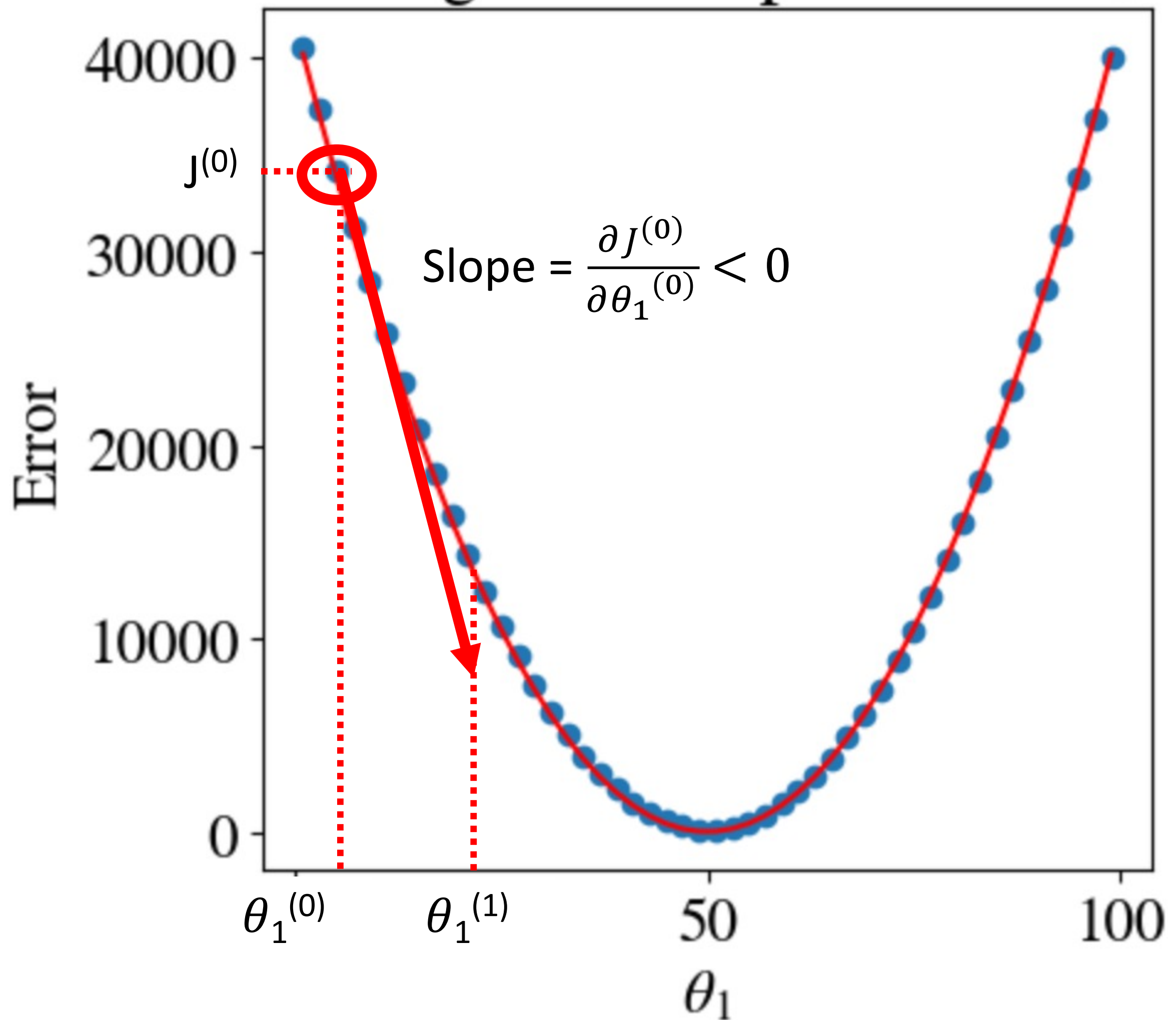






Gradient Descent Method

Navigating out of mountainous terrain



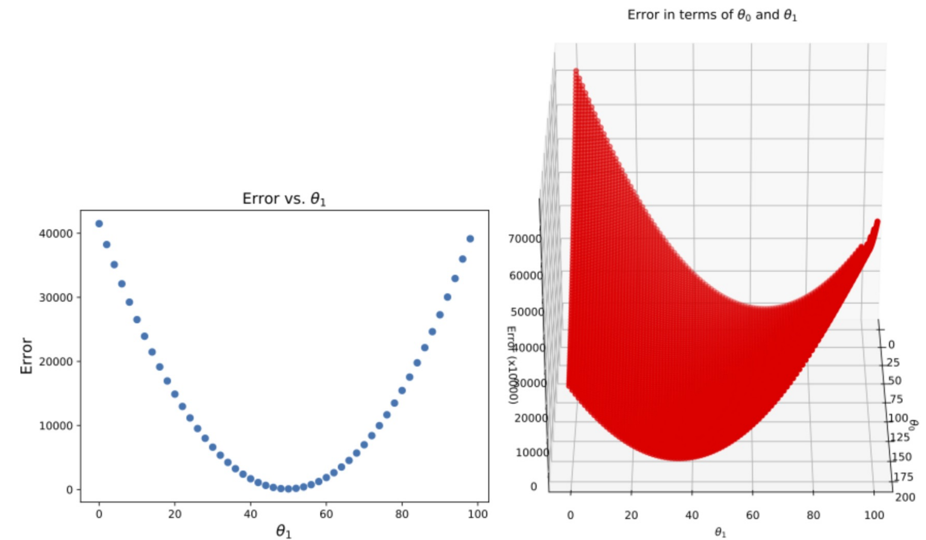
- Pick initial guess $\theta_1^{(0)}$,
- Calculate MSE, $J^{(0)}$,
- Calculate slope = ***gradient*** $\frac{\partial J^{(0)}}{\partial \theta_1^{(0)}}$,
- Pick new guess as:

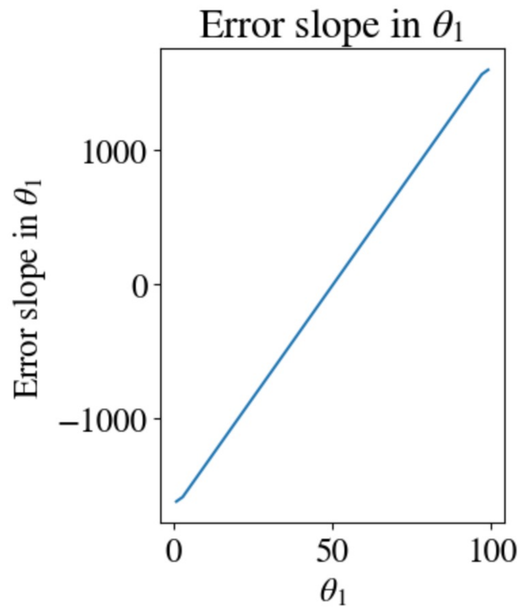
$$\theta_1^{(1)} = \theta_1^{(0)} - \alpha \frac{\partial J^{(0)}}{\partial \theta_1^{(0)}}$$

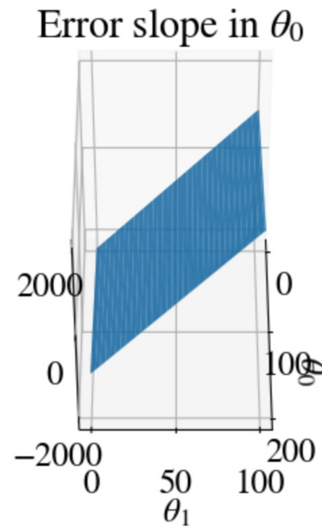
- $J^{(1)} < J^{(0)}$ if $\boldsymbol{\alpha}$ small enough
- $\boldsymbol{\alpha}$ is the **learning rate**

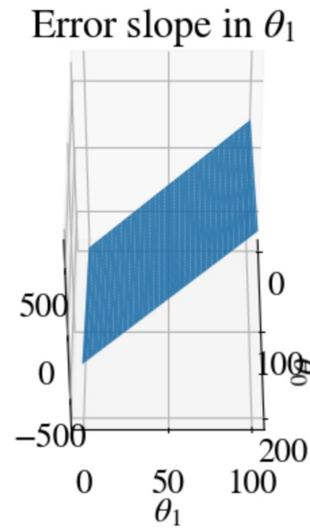
Use derivative

- MSE is quadratic in θ_0 and θ_1, so slope/gradient *of errors* is linear in θ_0 and θ_1









Explicit equations

$$J = \frac{1}{2m}\sum_{i=1}^{m}(h(x_i,\theta) - y_i)^2 = \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_i - y_i)^2$$

- $\frac{\partial J}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_i - y_i).$
- $\frac{\partial J}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_i - y_i)x_i.$

- $\theta_0^{\text{new}} = \theta_0 - \alpha\frac{\partial J}{\partial \theta_0} = \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_i - y_i),$
- $\theta_1^{\text{new}} = \theta_1 - \alpha\frac{\partial J}{\partial \theta_1} = \theta_1 - \frac{\alpha}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_i - y_i)x_i.$

Python code

```
%%time
nrsteps = 10000
thetas = np.zeros([nrsteps,2])
errorsGD = np.zeros(nrsteps)

# Initial guess:
thetas[0,:] = [25,10]

# Step-size:
alphas = 0.01

for i in range(nrsteps-1):
    hypothesis = thetas[i,0] + thetas[i,1]*x
    errorsGD[i] = mean_squared_error(hypothesis,y) /2

    thetas[i+1,0] = thetas[i,0] - (10*alphas/len(y))  * sum(hypothesis - y)
    thetas[i+1,1] = thetas[i,1] - (alphas/len(y))  * sum((hypothesis - y)*x)

```

Still using many steps

Note: examples all start with 'bad' initial guess

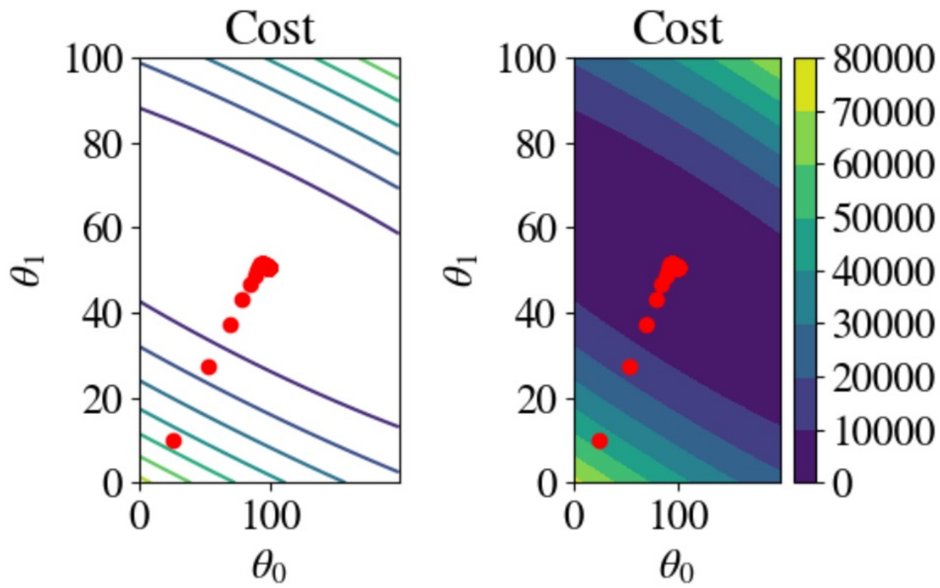
Learning rate (hyper-parameter)

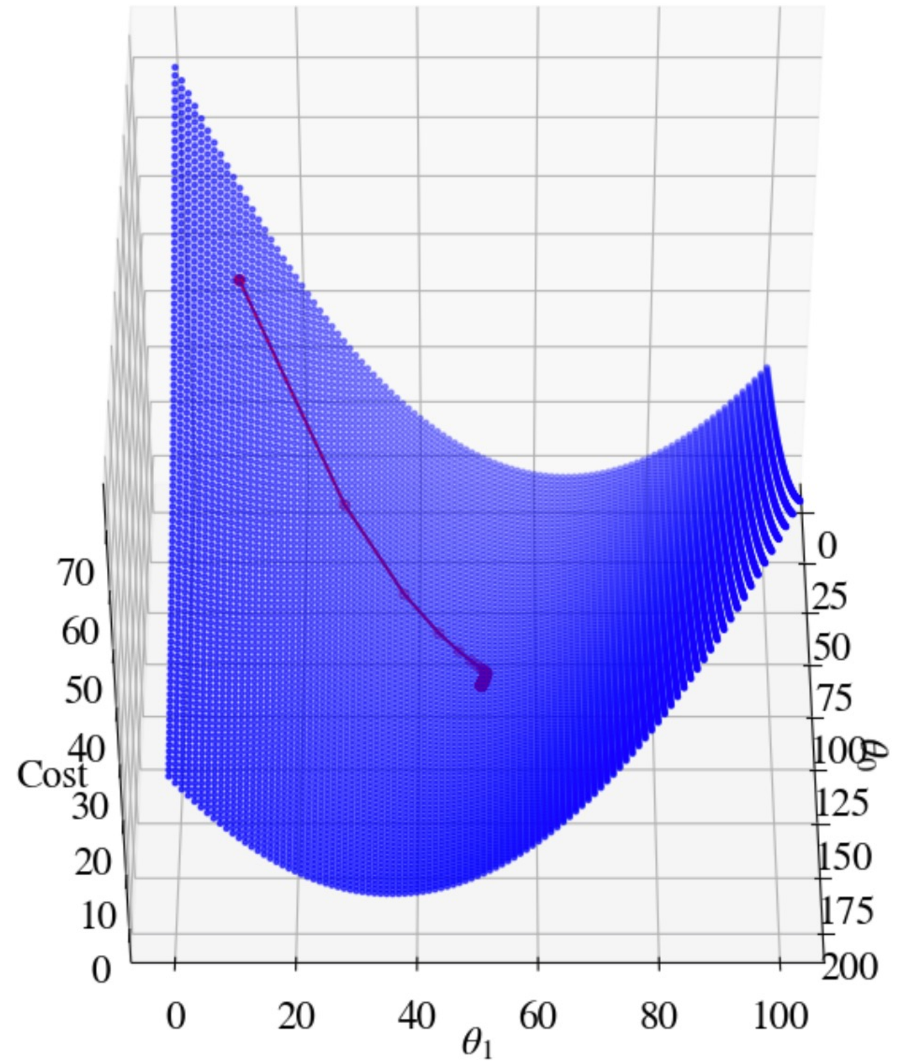
Only 1 loop!

Cheating with 10x learning rate

How did we do?

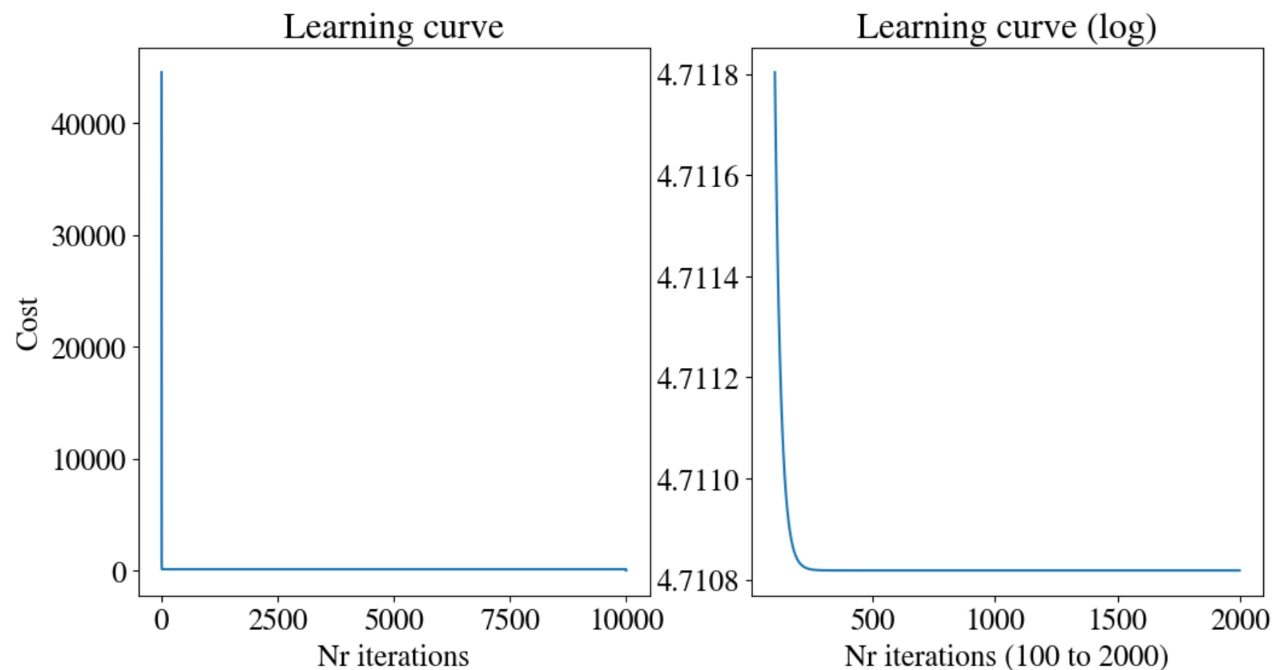
- Best fit: θ_0 = 89.9, and θ_1 = 50.7
- 10,000 tries
- **CPU time: 3.6 s**
- **MSE = 101.4**





Learning curve

- Plot error as function of nr of iterations
- Clearly, best fit was already found after few iterations



Vectorization

- Add a column of all ones to the feature array ***x*** (‘**bias**’)
- Define 2-element vector $\boldsymbol{\theta}$ = [θ_0 , θ_1]
- Then hypothesis (prediction) becomes simply:
- Vector ***y*** = h(***x***, $\boldsymbol{\theta}$) **= x ·** $\boldsymbol{\theta}$
- And 2D gradient is
- $\left(\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}\right) = \nabla J = (h(\boldsymbol{x}, \boldsymbol{\theta}) - \boldsymbol{y}) \cdot \boldsymbol{x} = (\boldsymbol{x} \cdot \boldsymbol{\theta} - \boldsymbol{y}) \cdot \boldsymbol{x}$

- $\frac{\partial J}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x_i - y_i).$
- $\frac{\partial J}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x_i - y_i) x_i.$

```
1  x_aug = np.zeros([m,2])
2  x_aug[:,0] = 1
3  x_aug[:,1] = x
4  x_aug
```

```
[ 1.        ,  3.93939394],
[ 1.        ,  4.04040404],
[ 1.        ,  4.14141414],
[ 1.        ,  4.24242424],
[ 1.        ,  4.34343434],
[ 1.        ,  4.44444444],
[ 1.        ,  4.54545455],
[ 1.        ,  4.64646465],
[ 1.        ,  4.74747475],
[ 1.        ,  4.84848485],
[ 1.        ,  4.94949495],
[ 1.        ,  5.05050505],
[ 1.        ,  5.15151515],
[ 1.        ,  5.25252525],
[ 1.        ,  5.35353535],
[ 1.        ,  5.45454545],
[ 1.        ,  5.55555556],
[ 1.        ,  5.65656566],
[ 1.        ,  5.75757576],
[ 1.        ,  5.85858586],
[ 1.        ,  5.95959596],
[ 1.        ,  6.06060606],
[ 1.        ,  6.16161616],
[ 1.        ,  6.26262626],
[ 1.        ,  6.36363636],
[ 1.        ,  6.46464646],
[ 1.        ,  6.56565657],
[ 1.        ,  6.66666667],
[ 1.        ,  6.76767677],
  1.        ,  6.86868687],
  1.        ,  6.96969697],
  1.        ,  7.07070707],
  1.        ,  7.17171717],
  1.        ,  7.27272727],
  1.        ,  7.37373737],
  1.        ,  7.47474747],
  1.        ,  7.57575758],
  1.        ,  7.67676768],
```

Python code for **non**-vectorized GD

```
%%time
nrsteps = 10000
thetas = np.zeros([nrsteps,2])
errorsGD = np.zeros(nrsteps)

# Initial guess:
thetas[0,:] = [25,10]

# Step-size:
alphas = 0.01

for i in range(nrsteps-1):
    hypothesis = thetas[i,0] + thetas[i,1]*x
    errorsGD[i] = mean_squared_error(hypothesis,y) /2

    thetas[i+1,0] = thetas[i,0] - (10*alphas/len(y))  * sum(hypothesis - y)
    thetas[i+1,1] = thetas[i,1] - (alphas/len(y))  * sum((hypothesis - y)*x)

```

Python code for vectorized gradient descent

```python
%%time
nrsteps = 10000
m = len(y)
thetas = np.zeros([nrsteps,2])
errorsGD = np.zeros(nrsteps)

alpha_m = alphas/m

x_aug = np.zeros([m,2])
x_aug[:,0] = 1
x_aug[:,1] = x

# Initial guess:
thetas[0,:] = [25,10]

for i in range(nrsteps-1):
    hypothesis = x_aug.dot(thetas[i,:])
    errorsGD[i] = mean_squared_error(hypothesis,y) /2

    thetas[i+1,:] = thetas[i,:] -  alpha_m * (hypothesis - y).dot(x_aug)
```

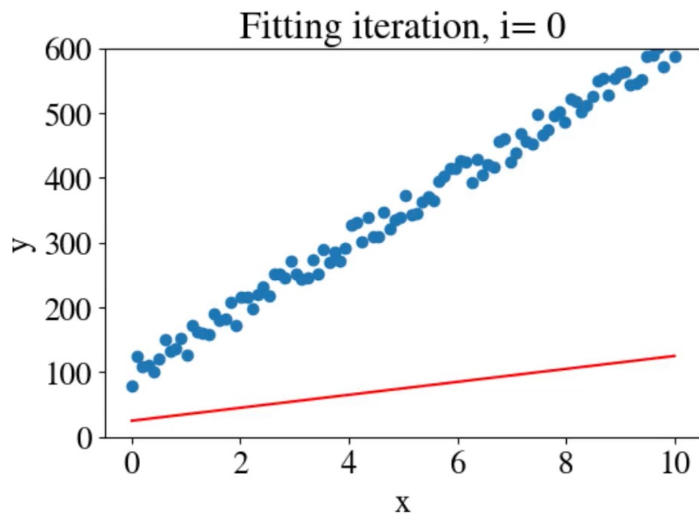
One more optimization

- Track changes, i.e., incremental reduction in MSE error, and stop when error is below certain tolerance (‘while-loop’):

```
i=0
converged = False
tolerance = 1e-4
while not converged:
    hypothesis = x_aug.dot(thetas[i,:])
    errorsGD[i] = mean_squared_error(hypothesis,y) /2

    thetas[i+1,:] = thetas[i,:] -  alpha_m * (hypothesis - y).dot(x_aug)
    if np.abs(errorsGD[i]-errorsGD[i-1]) < tolerance:
        converged = True
        print("Converged in", i, "iterations with thetas: ", thetas[i+1,:], "and error ", errorsGD[i])
    i+=1
```

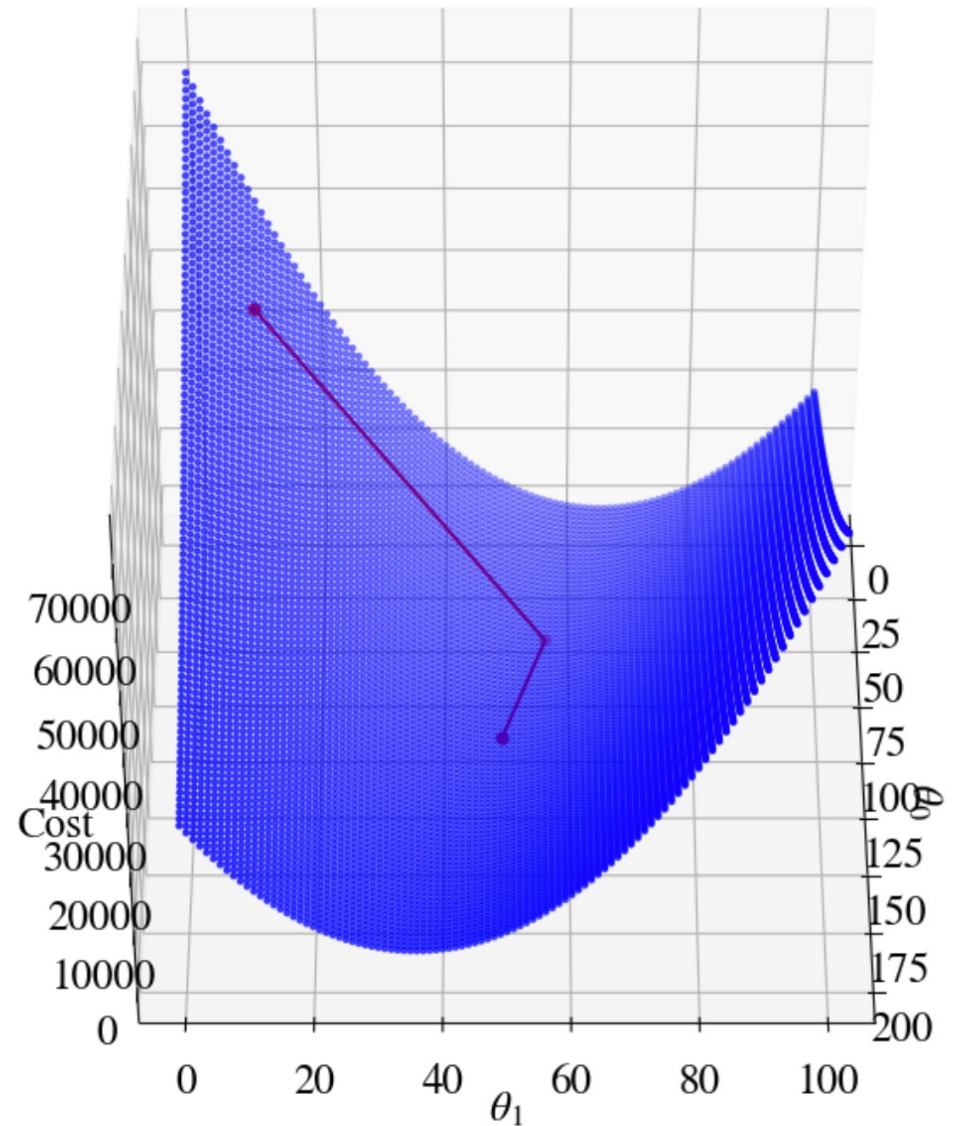
How did we do?

- Best fit: θ_0 = 98.7, and θ_1 = 50.7
- 10,000 tries => 1,099 tries
- **CPU time: 5.1 s => 248 ms**
- **MSE = 102.2 => 101.4**



Even faster!

- Use ***second* derivative** to determine *adaptive* learning rate $\boldsymbol{\alpha}$
- (Truncated) Newton method



- Best fit: θ_0 = 98.9, and θ_1 = 50.7
- 10,000 tries => **2 steps** (!!!)
- **CPU time: 5.1 s => 5 ms** (1/1000×)
- **MSE = 102.2 => 101.4**

Stochastic Gradient Descent

Stochastic Gradient Descent

- Many ML problems involve (many) thousands of features (columns of input data) and millions of data points (rows of input data)
- Gradient Descent can require too much RAM
- Key idea behind *stochastic* GD: Only fit to 1 point at a time!
  1. Initial guess of slope and intercept $\boldsymbol{\theta} = [\theta_0^{(0)}, \theta_1^{(0)}]$
  2. Randomly pick a point $(x_i, y_{i,true})$ and predict target $y_{i,pred} = \theta_0^{(0)} + \theta_1^{(0)} x_i$
  3. Compute MSE error $J^{(0)}$ and its derivatives, only w.r.t. that single point
  4. Update fitting parameters to $\boldsymbol{\theta} = [\theta_0^{(1)}, \theta_1^{(1)}]$
  5. Return to Step 2 and iterate/loop through all points x_i
  6. If accuracy is not satisfactory after looping through all points, set $[\theta_0^{(0)}, \theta_1^{(0)}]$ to last values and start again from Step 1

Python code I/II

- Randomly shuffle (x_i, $y_{i,true}$) pairs and set aside memory for $\boldsymbol{\theta}$ and errors

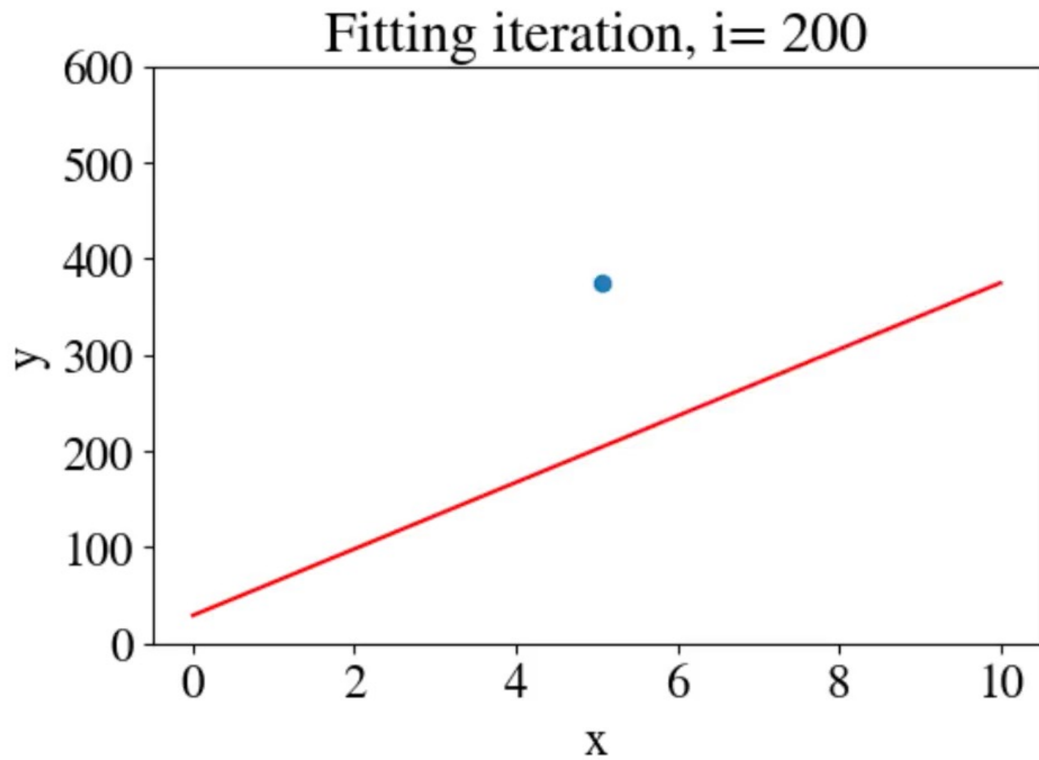
```
 7   p = np.random.permutation(len(xtmp))
 8   xshuf = xtmp[p]
 9   yshuf = ytmp[p]
10
11   m = len(yshuf)
12   nrouter = 10
13   nrsteps = m
14   # Allocate memory for variables:
15   savethetas = np.zeros([(nrsteps-1)*nrouter,2])
16   saveerrors = np.zeros((nrsteps-1)*nrouter)
17   thetas = np.zeros([nrsteps,2])
```

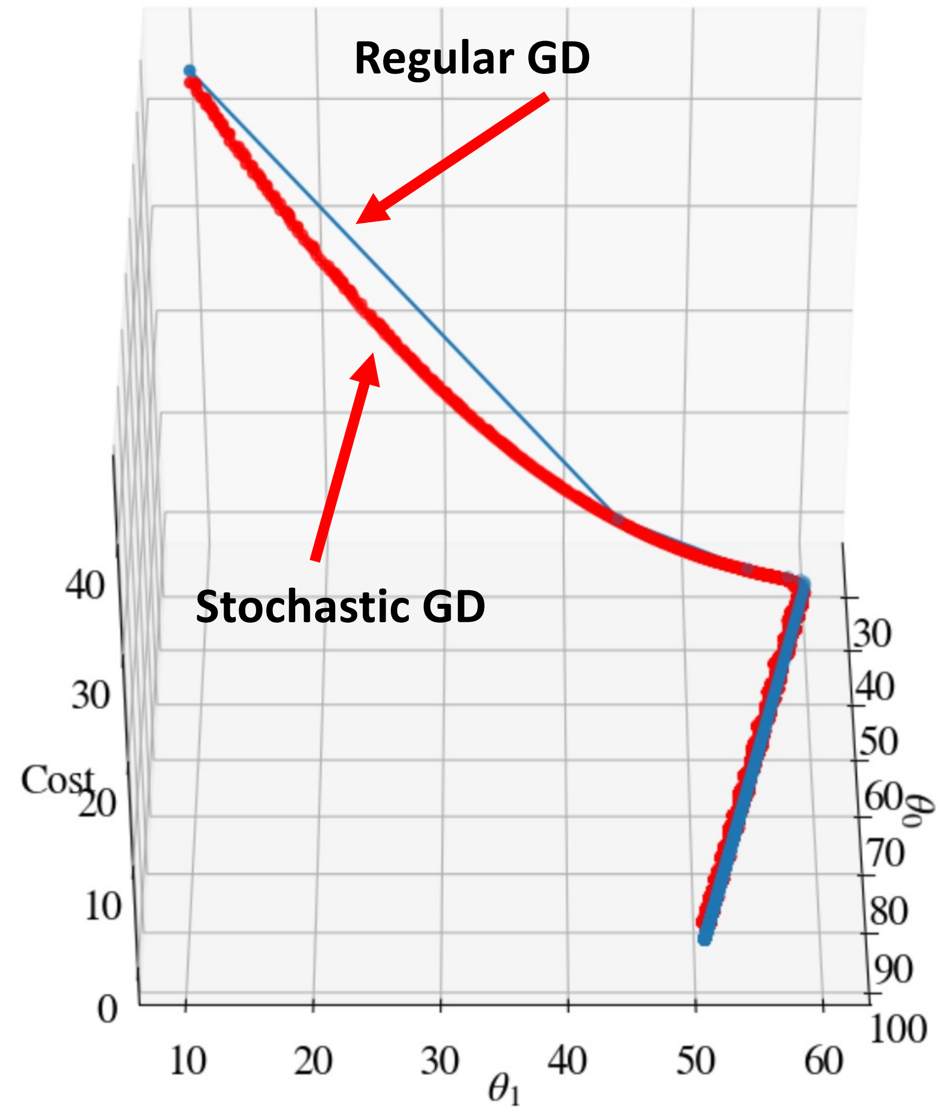
Python code II/II

- Non-vectorized implementation:

```
# Initial guess:
thetas[0,:] = [25,10]
alpha = 1

cnt = 0
for j in range(nrouter):
    for i in range(m-1):
        thetas[i+1,0] = thetas[i,0] - (alpha/m)  * (thetas[i,0] + thetas[i,1]*xshuf[i] - yshuf[i])
        thetas[i+1,1] = thetas[i,1] - (alpha/m)  * ((thetas[i,0] + thetas[i,1]*xshuf[i] - yshuf[i])*xshuf[i])

        # Compute error:
        saveerrors[cnt] = mean_squared_error(thetas[i+1,0]+thetas[i+1,1]*xshuf,yshuf) /2
        cnt = cnt+1
    # Set initial guess for the next run through all input values:
    thetas[0,:] = thetas[m-1,:] #
```

- Best fit: θ_0 = 94.4, and θ_1 = 50.9
- 100,000 tries ☹
- **CPU time: 23 s** ☹
- **MSE = 110.4** ☹





Mini-Batch Gradient Descent

Mini-Batch Gradient Descent

- Best of both worlds:
- Instead of fitting to all points at once (regular gradient descent), or to a single point at a time (stochastic gradient descent), fit to small mini-batches in each iteration
- Example: for 1,000 data-points, loop through 100 iterations that each fit to 10 points
- Repeat if error still too large (using $\boldsymbol{\theta}$ from previous pass as new initial guess)

Python code

- Vectorized

Batches of 10 points

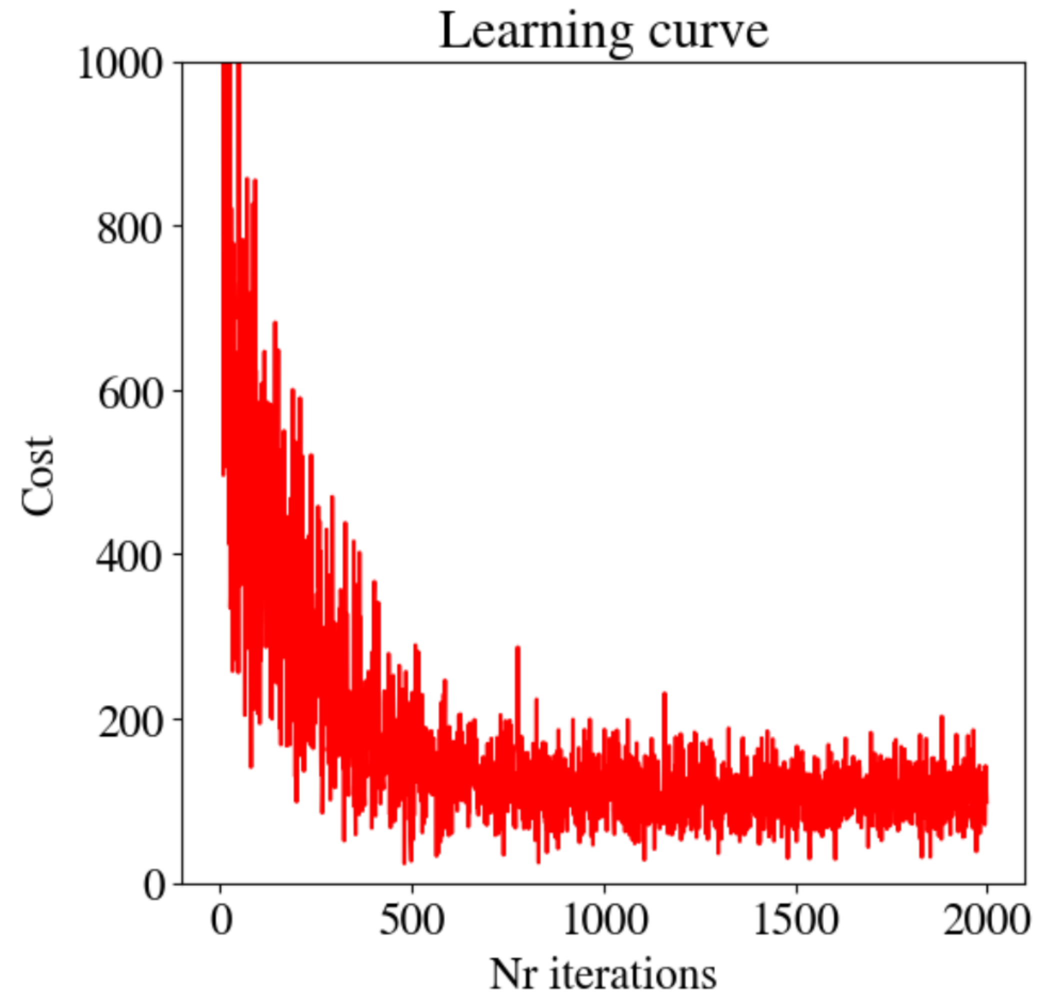
```
batch = 10
m = len(xshuf)
alpha_m = 0.01/batch

nrsteps = int(m/batch)
x_i = np.zeros([batch,2])
x_i[:,0] = 1
thetas = np.zeros([nrsteps+1,2])
thetas[0,:] = [25,10]

mb_iter=0
for j in range(1):
    for i in range(nrsteps):
        x_i[:,1] = xshuf[i*batch:(i+1)*batch]
        y_i = yshuf[i*batch:(i+1)*batch]
        hypothesis = x_i.dot(thetas[i,:])
        thetas[i+1,:] = thetas[i,:] - alpha_m * (hypothesis - y_i).dot(x_i)
        mb_iter += 1

    thetas[0,:] = thetas[i+1,:]
```

Single pass

Vectorized

Learning curve

- Noisier than before
- Error for each batch of 10 points can be higher/lower, but still converge to global minimum



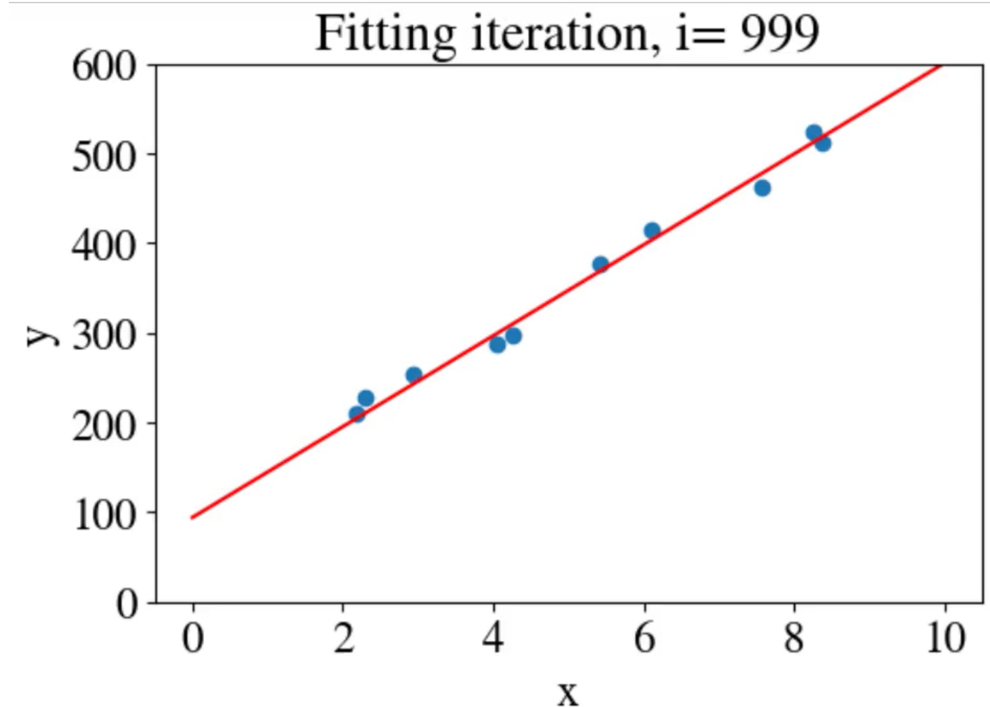
How did we do?

- Two passes:
- Best fit: θ_0 = 99.4, and θ_1 = 49.8
- 2,000 tries ☺
- **CPU time: 15.2 ms** ☺
- **MSE = 107.5** ☺

✓Memory efficient

✓CPU efficient (10× faster than vectorized regular GD)

✓Accurate



Summary

Summary

- Linear regression fits linear **hypothesis** to labeled **features** + **targets**
- Double nested loop of trials for slope and intercept is inefficient
- **Gradient Descent methods** (GD) takes advantage of error derivatives: slopes of the error landscape
- Standard GD uses a user-defined **learning rate** that can be too small (slow convergence) or too large (overshooting)
- **Accelerated GD** uses second derivative as *adaptive* learning rate
- **Stochastic GD** fits 1 point at a time for less memory
- **(Mini-)Batch GD** fits, say, 10 points per batch: more efficient when memory allows