Training Models

Lecturer: Ngo Huy Bien
Software Engineering Department
Faculty of Information Technology
University of Science
Ho Chi Minh City, Vietnam
nhbien@fit.hcmus.edu.vn

Objectives

- To find linear regression parameters using *analytical method*
- To find linear regression parameters using *gradient descent*
- To explain machine learning *terminologies* via examples

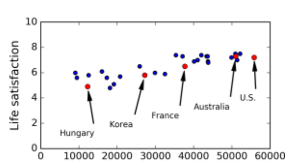
Contents

References

1. Aurelien Geron (2017). Hands on Machine Learning with Scikit Learn and TensorFlow. O'Reilly Media.
2. David C. Lay et al. (2016). Linear Algebra and Its Applications. 5th Edition. Pearson Education.

Does Money Make People Happier? [1]

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2



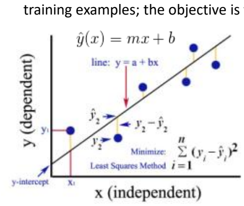
- `GDP per capita = 22587`
- `Life satisfaction = ?`

life_satisfaction = θ_0 + θ_1 × GDP_per_capita

How to find θ_0 and θ_1?

Cost Function

- You can either define a *utility function* (or *fitness function*) that measures how *good* your model is, or you can define a *cost function* that measures how *bad* it is.
- For linear regression problems, people typically use a cost function that measures the *distance* between the linear model's predictions and the training examples; the objective is to minimize this distance.



Mean Square Error (MSE):

$$\mathrm{MSE}(m, b) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

$$\mathrm{MSE}(m, b) = \frac{1}{n} \sum_{i=1}^{n} (mx_i + b - y_i)^2$$

Scalar Derivative Rules Review

- https://explained.ai/matrix-calculus/index.html

Rule	$f(x)$	Scalar derivative notation with respect to x	Example
Constant	c	0	$\frac{d}{dx}99 = 0$
Multiplication by constant	cf	$c\frac{df}{dx}$	$\frac{d}{dx}3x = 3$
Power Rule	x^n	nx^{n-1}	$\frac{d}{dx}x^3 = 3x^2$
Sum Rule	$f+g$	$\frac{df}{dx} + \frac{dg}{dx}$	$\frac{d}{dx}(x^2+3x) = 2x+3$
Difference Rule	$f-g$	$\frac{df}{dx} - \frac{dg}{dx}$	$\frac{d}{dx}(x^2-3x) = 2x-3$
Product Rule	fg	$f\frac{dg}{dx} + \frac{df}{dx}g$	$\frac{d}{dx}x^2x = x^2 + x2x = 3x^2$
Chain Rule	$f(g(x))$	$\frac{df(u)}{du}\frac{du}{dx}$, let $u = g(x)$	$\frac{d}{dx}ln(x^2) = \frac{1}{x^2}2x = \frac{2}{x}$

- Example:

$$\frac{d}{dx}9(x+x^2) = 9\frac{d}{dx}(x+x^2) = 9(\frac{d}{dx}x + \frac{d}{dx}x^2) = 9(1+2x) = 9+18x$$

Analytical Solution for Simple Linear Model

Compare the derivatives to zero

$$\frac{\partial \text{MSE}}{\partial m} = \frac{2}{n}\sum_{i=i}^{n}(mx_i + b - y_i)x_i = 0$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n}\sum_{i=i}^{n}(mx_i + b - y_i) = 0$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n}\sum_{i=i}^{n}(mx_i + b - y_i)$$

$\bar{x}$ **Mean value of *x* across all samples.**

$$= \frac{2}{n}(mn\bar{x} + nb - n\bar{y})$$

$$= 2m\bar{x} + 2b - 2\bar{y} = 0$$

$$\frac{\partial \text{MSE}}{\partial m} = 2m\overline{x^2} + 2b\bar{x} - 2\overline{xy} = 0$$

Solution: $m = \dfrac{\bar{x}\bar{y} - \overline{xy}}{\bar{x}^2 - \overline{x^2}}$ $\quad b = \bar{y} - \bar{x}\dfrac{\bar{x}\bar{y} - \overline{xy}}{\bar{x}^2 - \overline{x^2}}$

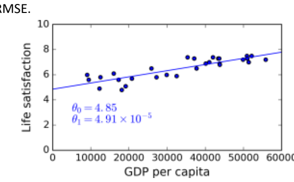
Finding Analytic Solution

```
def estimate_coefficients(X, y):

    mean_X, mean_y = np.mean(X), np.mean(y)
    mean_Xy = np.mean(X*y)
    s_mean_X = mean_X**2
    mean_X_s = np.mean(X**2)

    m = (mean_X * mean_y - mean_Xy) / (s_mean_X - mean_X_s)
    b = mean_y - mean_X*m

    return(b, m)

b, m = estimate_coefficients(X, y)
print ('coef:', m)
print ('intercept:', b)
```

```
coef: 4.9115445891584675e-05
intercept: 4.8530528002664415
```

Training Models Summary

- Training a model means *setting its parameters* so that the model best fits the training set.
- For this purpose, we first need a *measure of how well* (or poorly) the model fits the training data.
- The most common *performance measure* of a regression model is the Root Mean Square Error (RMSE).
- Therefore, to train a Linear Regression model, you need to find the value of θ that *minimizes* the RMSE.



Multiple Linear Regression [1]

- The *linear function*

$$\hat{y}(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

- The *mean square error* (over k samples)

$$\text{MSE} = \frac{1}{k}\sum_{i=1}^{k}(\hat{y}(x^{(i)}) - y^{(i)})^2$$

- The *partial derivative* of this cost by each θ

$$\frac{\partial \text{MSE}}{\partial \theta_j} = \frac{2}{k}\sum_{i=1}^{k}(\hat{y}(x^{(i)}) - y^{(i)})x_j^{(i)}$$ **0**

Matrices Review [2]

$$AB = \begin{bmatrix} 2 & 3 \\ 1 & -5 \end{bmatrix}\begin{bmatrix} 4 & 3 & 6 \\ 1 & -2 & 3 \end{bmatrix} = \begin{bmatrix} \square & \square & 2(6)+3(3) \\ \square & \square & \square \end{bmatrix} = \begin{bmatrix} \square & \square & 21 \\ \square & \square & \square \end{bmatrix}$$

If $A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \end{bmatrix}$ then $A^T = \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 3 & 4 \end{bmatrix}$

If $A = \begin{bmatrix} 2 & 5 \\ -3 & -7 \end{bmatrix}$ and $C = \begin{bmatrix} -7 & -5 \\ 3 & 2 \end{bmatrix}$, then

$$AC = \begin{bmatrix} 2 & 5 \\ -3 & -7 \end{bmatrix}\begin{bmatrix} -7 & -5 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ and}$$

$$CA = \begin{bmatrix} -7 & -5 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} 2 & 5 \\ -3 & -7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Thus $C = A^{-1}$.

$\mathbf{u} = \begin{bmatrix} 2 \\ -5 \\ -1 \end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix} 3 \\ 2 \\ -3 \end{bmatrix}$

$$\mathbf{u}\cdot\mathbf{v} = \mathbf{u}^T\mathbf{v} = [2 \;\; -5 \;\; -1]\begin{bmatrix} 3 \\ 2 \\ -3 \end{bmatrix} = (2)(3) + (-5)(2) + (-1)(-3) = -1$$

$$\mathbf{v}\cdot\mathbf{u} = \mathbf{v}^T\mathbf{u} = [3 \;\; 2 \;\; -3]\begin{bmatrix} 2 \\ -5 \\ -1 \end{bmatrix} = (3)(2) + (2)(-5) + (-3)(-1) = -1$$

Vectorized Representation (I)

- Expressing the regression *coefficients* as a vector:

$$\begin{pmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1}$$

- The *linear function* $\hat{y}(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$ becomes (for single *data elements* and *n features)*:

$$\Rightarrow \quad [\theta_0 \;\; \theta_1 \;\; \dots \;\; \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x \quad \Rightarrow \quad \hat{y}(x) = \theta^T x$$

Vectorized Representation (II)

- For *m data elements* and *n features*:

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & x_3^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \qquad \begin{pmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1}$$

$$\hat{y} = X\hat{\theta}$$

The Normal Equation (Solution): $\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$

- The Normal Equation computes the *inverse* of $X^T \cdot X$, which is an n × n matrix (where n is the number of features).

The Normal Equation Proof (I)

- https://eli.thegreenplace.net/2014/derivation-of-the-normal-equation-for-linear-regression

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

$$\begin{pmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1} \qquad h_\theta(x) = \theta^T x$$

$$J(\theta_{0\dots n}) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y) \Rightarrow \cancel{\frac{1}{2m}}$$

$$J(\theta) = ((X\theta)^T - y^T)(X\theta - y)$$

$$J(\theta) = (X\theta)^T X\theta - (X\theta)^T y - y^T(X\theta) + y^T y$$

$$J(\theta) = \theta^T X^T X\theta - 2(X\theta)^T y + y^T y$$

$$\frac{\partial J}{\partial \theta} = 2X^T X\theta - 2X^T y = 0 \Rightarrow \theta = (X^T X)^{-1} X^T y$$

The Normal Equation Proof (II)

- https://eli.thegreenplace.net/2015/the-normal-equation-and-matrix-calculus/

$$J(\theta) = \theta^T X^T X\theta - 2(X\theta)^T y + y^T y \Rightarrow \frac{\partial J}{\partial \theta} = 2X^T X\theta - 2X^T y = 0$$

$$P(\theta) = 2(X\theta)^T y$$

$$P(\theta) = 2\left[\begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & \dots & \dots & x_{2n} \\ \dots \\ x_{m1} & \dots & \dots & x_{mn} \end{pmatrix}\begin{pmatrix} \theta_1 \\ \theta_2 \\ \dots \\ \theta_n \end{pmatrix}\right]^T \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{pmatrix} \qquad P(x) = 2\left[\begin{pmatrix} x_{11}\theta_1 + \dots + x_{1n}\theta_n \\ x_{21}\theta_1 + \dots + x_{2n}\theta_n \\ \dots \\ x_{m1}\theta_1 + \dots + x_{mn}\theta_n \end{pmatrix}\right]^T \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{pmatrix}$$

$$P(x) = 2(x_{11}\theta_1 + \dots + x_{1n}\theta_n)y_1 + 2(x_{21}\theta_1 + \dots + x_{2n}\theta_n)y_2 + \dots + 2(x_{m1}\theta_1 + \dots + x_{mn}\theta_n)y_m$$

$$P(x) = 2\sum_{r=1}^{m} y_r(x_{r1}\theta_1 + \dots + x_{rn}\theta_n) = 2\sum_{r=1}^{m} y_r \sum_{c=1}^{n} x_{rc}\theta_c$$

$$\frac{\partial P}{\partial \theta_1} = 2(x_{11}y_1 + \dots + x_{m1}y_m)$$

$$\frac{\partial P}{\partial \theta_2} = 2(x_{12}y_1 + \dots + x_{m2}y_m) \Rightarrow \frac{\partial P}{\partial \theta} = 2X^T y$$

$$\dots$$

$$\frac{\partial P}{\partial \theta_n} = 2(x_{1n}y_1 + \dots + x_{mn}y_m)$$

Training Linear Regression Model Using The Normal Equation

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

```
X_b = np.c_[np.ones((X.shape[0], 1)), X] # add x0 =
1 to each instance
theta_best =
np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
print (theta_best)
```

```
[[4.85305280e+00]
 [4.91154459e-05]]
```

Making Predictions

For single data element:

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

For m data elements:

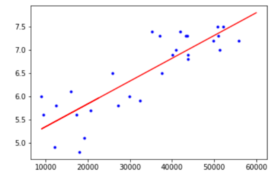
$$\hat{y} = X\hat{\theta}$$

```
X_new = np.array([[22587], [9054], [60000]]) # Cyprus' and Russia GDP per capita
X_new_b = np.c_[np.ones((X_new.shape[0], 1)), X_new] # add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
print(y_predict)
```

```
[[5.96242338]
 [5.29774405]
 [7.79997955]]
```

Plotting Model's Predictions

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.show()
```

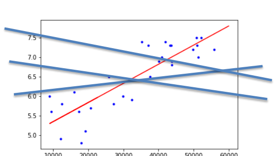


Computational Complexity

- The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation).
- In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.
- The Normal Equation gets *very slow* when the number of features grows large (e.g., 100,000).
- This equation is linear with regards to the number of instances in the training set (it is O(m)), so it handles large training sets efficiently, provided they can fit in *memory*.

Gradient Descent

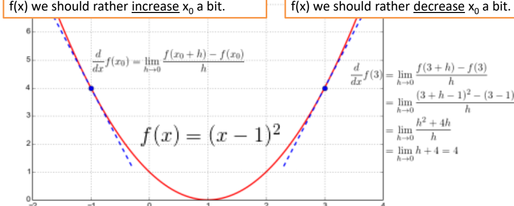
- Gradient Descent is a very *generic optimization algorithm* capable of finding optimal solutions to a wide range of problems.
- The general idea of Gradient Descent is to *tweak parameters iteratively* in order to minimize a cost function.



Building Intuition

The slope of df/dx at $x_0 = -1$ is –4; for a very small positive change h to x at that point, the value of f(x) will *decrease* by 4h. Therefore, to get closer to the minimum of f(x) we should rather increase x_0 a bit.

The slope of df/dx at $x_0 = 3$ is 4; for a very small positive change h to x at that point, the value of f(x) will *increase* by 4h. Therefore, to get closer to the minimum of f(x) we should rather decrease x_0 a bit.

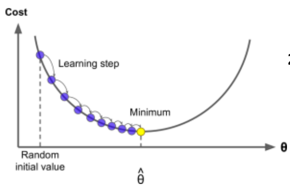


https://eli.thegreenplace.net/2016/understanding-gradient-descent/

How Gradient Descent Works (I)

1. Select a model.

$$h_\theta(x) = \theta_0 + \theta_1 x$$



2. Select a cost function.

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

How Gradient Descent Works (II)

3. Calculate partial derivatives. $\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

$$\frac{d}{d\theta_0}J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

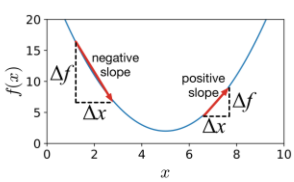
$$\frac{d}{d\theta_1}J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)\cdot x^{(i)}$$

4. Fill θ_0 and θ_1 with *random* values (this is called *random initialization*).

5. Compute cost using x^i, y^i θ_0 and θ_1 $\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

How Gradient Descent Works (III)

6. Increase or decrease θ_0 and θ_1 based on derivative value.



$$\theta_0 := \theta_0 - \alpha\frac{d}{d\theta_0}J(\theta_0,\theta_1)$$

$$\theta_1 := \theta_1 - \alpha\frac{d}{d\theta_1}J(\theta_0,\theta_1)$$

- An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter.

Computing Cost

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \qquad J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y)$$

```
def compute_cost(X, y, theta_1, theta_0):
    yhat = theta_1 * X + theta_0
    diff = yhat - y
    # Vectorized computation using a dot product to compute sum of
squares.
    cost = np.dot(diff.T, diff) / float(X.shape[0])
    # Cost is a 1x1 matrix, we need a scalar.
    return cost.flat[0]

print ('theta_1 = 2, theta_0 = 3, cost =', compute_cost(X, y, 2, 3))
print ('theta_1 = 1, theta_0 = 0, cost =', compute_cost(X, y, 1, 0))
```

```
theta_1 = 2, theta_0 = 3, cost = 5288423882.877398
theta_1 = 1, theta_0 = 0, cost = 1321778841.117839
```

Implementing Gradient Descent

```
def gradient_descent(X, y, learning_rate = 0.001):
    m = X.shape[0]
    theta_1, theta_0 = 0, 0
    cost = compute_cost(X, y, theta_1, theta_0)
    print ('initial theta_1 =', theta_1)
    print ('initial theta_0 = ', theta_0)
    print ('initial cost =', cost)

    for step in range(1, 3):
        yhat = theta_1 * X + theta_0
        diff = yhat - y
        dtheta_1 = (diff * X).sum() * 2 / m
        dtheta_0 = diff.sum() * 2 / m
        theta_1 -= learning_rate*dtheta_1
        theta_0 -= learning_rate*dtheta_0
        cost = compute_cost(X, y, theta_1, theta_0)
        print ('step', step)
        print ('theta_1 =', theta_1)
        print ('theta_0 = ', theta_0)
        print ('cost =', cost)

    return theta_1, theta_0, cost

theta_1, theta_0, cost = gradient_descent(X, y)
```

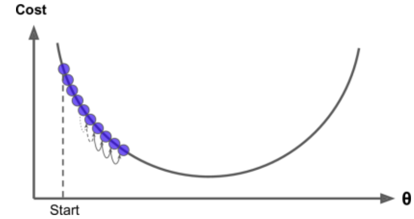
```
initial theta_1 = 0
initial theta_0 =  0
initial cost = 42.841034482758616
step 1
theta_1 = 453.98794905517235
theta_0 =  0.012986206896551725
cost = 272518659234307.88
step 2
theta_1 = -1200554594.4757342
theta_0 =  -30318.877466546877
cost = 1.9057755484797597e+27
```

$$\theta_0 := \theta_0 - \alpha\frac{d}{d\theta_0}J(\theta_0,\theta_1)$$

$$\theta_1 := \theta_1 - \alpha\frac{d}{d\theta_1}J(\theta_0,\theta_1)$$

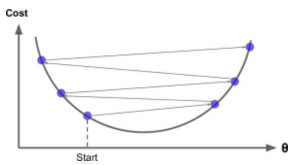
Too Small Learning Rate

- If the learning rate is *too small*, then the algorithm will have to go through many iterations to converge, which will take a long time.



Too High Learning Rate

- If the learning rate is *too high*, you might jump across the valley and end up on the other side, possibly even higher up than you were before.
- This might make the algorithm diverge, with larger and larger values, *failing* to find a good solution.



Summary

1. Selecting a model:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

2. Selecting a cost function:

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

3. Calculating cost partial derivatives:

$$\frac{d}{d\theta_0}J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$$

$$\frac{d}{d\theta_1}J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

4. Updating parameters:

$$\theta_0 := \theta_0 - \alpha\frac{d}{d\theta_0}J(\theta_0,\theta_1)$$
$$\theta_1 := \theta_1 - \alpha\frac{d}{d\theta_1}J(\theta_0,\theta_1)$$

5. Computing cost:

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Multiple Parameters

1. Selecting a model:

$$\hat{y}(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

2. Selecting a cost function:

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

3. Calculating cost partial derivatives:

$$\begin{bmatrix} \frac{\partial J}{\partial\theta_0} \\ \frac{\partial J}{\partial\theta_1} \\ \frac{\partial J}{\partial\theta_2} \\ \vdots \\ \frac{\partial J}{\partial\theta_n} \end{bmatrix} = \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}\right) \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}\right) \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}\right) \\ \vdots \\ \sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)}\right) \end{bmatrix}$$

4. Updating parameters:

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

5. Computing cost:

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Gradient Vector Review (I)

https://explained.ai/matrix-calculus/index.html

- Let f be a *scalar-valued function*, f: $\mathbf{R}^2 \rightarrow \mathbf{R}$

$$f(x, y) = 3x^2y$$

- The *gradient* of f is simply a vector of its partials (denoted as ∇f)

$$\nabla f(x,y) = [\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}] = [6yx, 3x^2]$$

Gradient Vector Review (II)

- Let f be a *scalar-valued function*, f: $\mathbf{R}^n \rightarrow \mathbf{R}$
- Combine multiple parameters into a single vector argument $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$
- The *gradient* of f is simply a vector of its partial derivatives (denoted as ∇f)

$$Df(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}(\mathbf{x}) \quad \frac{\partial f}{\partial x_2}(\mathbf{x}) \quad \cdots \quad \frac{\partial f}{\partial x_n}(\mathbf{x})\right]$$

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \cdots, \frac{\partial f}{\partial x_n}(\mathbf{x})\right)$$

Vectorized Representation (I)

- The *mean square error* (over m samples)

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

becomes

$$\mathrm{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

- Partial derivatives of the *cost function* with regards to parameter θ_j:

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Vectorized Representation (II)

- Partial derivatives of the *cost function* with regards to parameter θ_j: $\frac{\partial}{\partial\theta_j}\mathrm{MSE}(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$

can be represented in *vectorized form* as:

$$\nabla_\theta \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathrm{MSE}(\theta) \\ \frac{\partial}{\partial\theta_1}\mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X}\cdot\theta - \mathbf{y})$$

Batch Gradient Descent [1]

- Once you have the *gradient vector*, which points uphill, just go in the opposite direction to go downhill.
- This means subtracting $\nabla_\theta\mathrm{MSE}(\theta)$ from θ.

$$\theta^{(\text{next step})} = \theta - \eta\nabla_\theta\mathrm{MSE}(\theta)$$

where

$$\nabla_\theta \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathrm{MSE}(\theta) \\ \frac{\partial}{\partial\theta_1}\mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X}\cdot\theta - \mathbf{y})$$

Batch

Why The Updating Rule Works? [2]

$$\theta^{(\text{next step})} = \theta - \eta\nabla_\theta\mathrm{MSE}(\theta)$$

2 Definition The **directional derivative** of f at (x_0, y_0) in the direction of a unit vector $\mathbf{u} = \langle a, b\rangle$ is

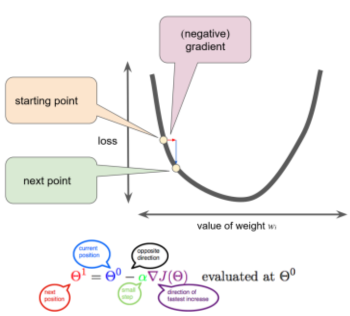
$$D_\mathbf{u} f(x_0, y_0) = \lim_{h\to 0}\frac{f(x_0 + ha, y_0 + hb) - f(x_0, y_0)}{h}$$

if this limit exists.

3 Theorem If f is a differentiable function of x and y, then f has a directional derivative in the direction of any unit vector $\mathbf{u} = \langle a, b\rangle$ and

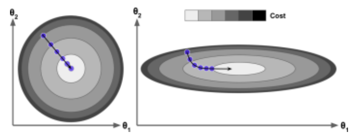
$$D_\mathbf{u} f(x, y) = f_x(x, y)\,a + f_y(x, y)\,b$$

Summary



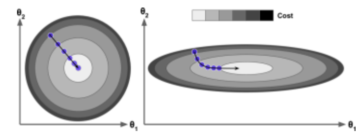
Feature Scaling

- Gradient Descent
  - on a training set where features 1 and 2 have the *same scale* (on the left), and
  - on a training set where feature 1 has much *smaller values* than feature 2.



Revisiting Training Model Meaning

- This diagram also illustrates the fact that *training a model* means searching for a combination of model parameters that minimizes a cost function (over the training set).
- It is a search in the model's *parameter space*: the more parameters a model has, the more dimensions this space has, and the harder the search is.



Implementing Feature Scaling

```
from copy import deepcopy

def NormalizeX(X):
    X_standalized = deepcopy(X)
    mu_X = np.mean(X_standalized[:,1])
    sigma_X = np.std(X_standalized[:,1])

    # Standardization
    X_standalized[:,1]=(X_standalized[:,1]-mu_X)/sigma_X
    print('mu_X:' , mu_X)
    print('sigma_X:', sigma_X)

    return X_standalized

X_b = np.c_[np.ones((X.shape[0], 1)), X] # add x0 = 1 to each instance
X_b_standalized = NormalizeX(X_b)
print(X_b_standalized[0:5])
```

```
mu_X: 33391.74913793103
sigma_X: 14395.272687078264
[[ 1.         -1.690613  ]
 [ 1.         -1.66404469]
 [ 1.         -1.4693612 ]
 [ 1.         -1.45161648]
 [ 1.         -1.20873105]]
```

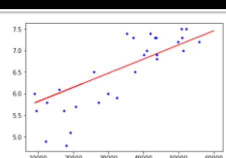
Implementing BGD

```
m, n = X_b_standalized.shape
print ('Examples:', m)
print ('Features:', n)
print ('Labels:', y.shape)
learning_rate = 0.00001
n_epochs = 800000

theta_bgd = np.random.randn(n, 1) # random initialization
print ('Randomly initialized theta:', theta_bgd)

for epoch in range(n_epochs):
    gradients = (2/m) *
X_b_standalized.T.dot(X_b_standalized.dot(theta_bgd) - y)
    theta_bgd = theta_bgd - learning_rate * gradients
    if 0 == epoch % (int(n_epochs/5)):
        mse = np.sum(np.square(X_b_standalized.dot(theta_bgd)-y))/m
        print ('Cost ' + str(epoch) + ': ' + str(mse))

print ('Final BGD theta:', theta_bgd)
```

```
Examples: 29
Features: 2
Labels: (29, 1)
Randomly initialized
theta: [[ 0.09771652]
 [-1.48551093]]
Cost 0:
45.887132732116264
Cost 160000:
0.25668924884667826
Cost 320000:
0.18087650583438167
Cost 480000:
0.1807505466815824
Cost 640000:
0.18075033740663013
Final BGD theta:
[[6.49310273]
 [0.70702999]]
```

Making Predictions

```
X_new = np.array([[22587], [9054], [60000]]) # Cyprus' and Russia GDP
per capita
X_new_b = np.c_[np.ones((X_new.shape[0], 1)), X_new] # add x0 = 1 to
each instance
X_new_b_standalized = NormalizeX(X_new_b)
y_predict_bgd = X_new_b_standalized.dot(theta_bgd)
print(y_predict_bgd)
plt.plot(X_new, y_predict_bgd, "r-")
plt.plot(X, y, "b.")
plt.show()
```



BGD Without Feature Scaling

```
m, n = X_b.shape
print ('Examples:', m)
print ('Features:', n)
print ('Labels:', y.shape)
learning_rate = 0.00001
n_epochs = 800000

theta_bgd_without_scaling = np.random.randn(n, 1) # random initialization
print ('Randomly initialized theta:', theta_bgd_without_scaling)

for epoch in range(n_epochs):
    gradients = (2/m) * X_b.T.dot(X_b.dot(theta_bgd_without_scaling) - y)
    theta_bgd_without_scaling = theta_bgd_without_scaling - learning_rate *
gradients
    if 0 == epoch % (int(n_epochs/5)):
        mse = np.sum(np.square(X_b.dot(theta_bgd_without_scaling)-y))/m
        print ('Cost ' + str(epoch) + ': ' + str(mse))

print ('Final BGD theta:', theta_bgd_without_scaling)
```

BGD Without Feature Scaling Issue

```
Examples: 29
Features: 2
Labels: (29, 1)
Randomly initialized theta: [[-0.32663899]
 [ 1.05828644]]
Cost 0: 782931448.607165

c:\users\admin\ml\env\lib\site-packages\ipykernel_launcher.py:14:
RuntimeWarning: overflow encountered in square

c:\users\admin\ml\env\lib\site-packages\ipykernel_launcher.py:13:
RuntimeWarning: invalid value encountered in subtract
  del sys.path[0]

Cost 160000: nan
Cost 320000: nan
Cost 480000: nan
Cost 640000: nan
Final BGD theta: [[nan]
 [nan]]
```

Evaluating The Model

```
print ('R-squared score (training):',
lin_reg_model.score(X, y))
```

```
R-squared score (training): 0.734441435543703
```

The R-Squared Test

- In statistics, there are many ways to evaluate *how good* a "fit" some model is on the given data.
- One of the most popular ones is the *r-squared* test ("coefficient of determination").
- It measures the *proportion* of the total variance in the output (y) that can be explained by the variation in *x*:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - (mx_i + b))^2}{n \cdot var(y)}$$

Computing Simple R-Squared

```
def compute_simple_rsquared(X, y, theta_1, theta_0):
    yhat = theta_1 * X + theta_0
    diff = yhat - y
    SE_line = np.dot(diff.T, diff)
    SE_y = len(y) * y.var()
    return 1 - SE_line / SE_y

print ('R-squared score (training):',
compute_simple_rsquared(X, y, theta_1, theta_0))
```

```
R-squared score (training): [[-2.79996722e+27]]
```

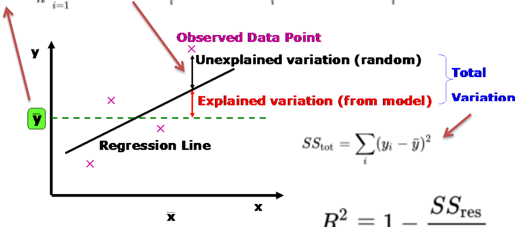
Computing R-Squared

```
def compute_rsquared(X_b_standalized, y, theta):
    yhat = X_b_standalized.dot(theta)
    #
https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.dot.html
    diff = yhat - y
    SE_line = np.dot(diff.T, diff)
    SE_y = len(y) * y.var()
    return 1 - SE_line / SE_y

print ('X_b_standalized.shape:', X_b_standalized.shape)
print ('theta_bgd.shape:', theta_bgd.shape)
print ('R-squared score (training/b/standalized):',
        compute_rsquared(X_b_standalized, y, theta_bgd))
```

```
X_b_standalized.shape: (29, 2)
theta_bgd.shape: (2, 1)
R-squared score (training/b/standalized): [[0.73444144]]
```

Coefficient Of Determination (R^2)

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \qquad SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2 \qquad SS_{\text{reg}} = \sum_i (f_i - \bar{y})^2$$



$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

$$R^2 \equiv 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

GD vs. Normal Equation

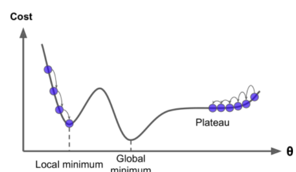
$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

$$\nabla_\theta \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\text{MSE}(\theta) \\ \frac{\partial}{\partial\theta_1}\text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\text{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

- Gradient Descent scales well with *the number of features*;
- Training a Linear Regression model when there are *hundreds of thousands of features* is much faster using Gradient Descent than using the Normal Equation.

Gradient Descent Pitfalls

- If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the global minimum.
- If it starts on the right, then it will take a very *long time* to cross the plateau, and if you stop too early you will never reach the global minimum.



Batch Gradient Descent Drawback

$$\nabla_\theta \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\text{MSE}(\theta) \\ \frac{\partial}{\partial\theta_1}\text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\text{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

- This formula involves calculations over the *full training set X*, at each Gradient Descent step!
- This is why the algorithm is called Batch Gradient Descent: it uses the *whole batch of training data* at every step.
- As a result it is *terribly slow* on very large training sets.

Stochastic Gradient Descent

- Stochastic Gradient Descent just picks a *random instance* in the training set at every step and computes the gradients based only on that single instance.
- Obviously this makes the algorithm *much faster* since it has very little data to manipulate at every iteration.
- It also makes it possible to train on *huge training sets*, since only one instance needs to be in memory at each iteration.

Repeat until convergence
{
$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$
}

Batch

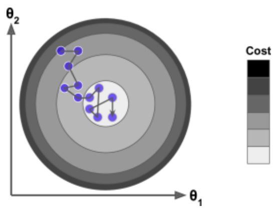
Randomly shuffle (reorder) training examples

Stochastic

Repeat {
for $i := 1, \ldots, m$ {
$\theta_j := \theta_j - \alpha (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$
(for every $j = 0, \ldots, n$)
}
}

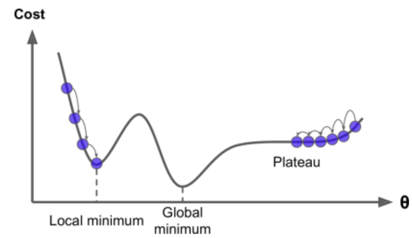
Change of Cost

- Due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the *cost function* will bounce up and down, decreasing only on average.



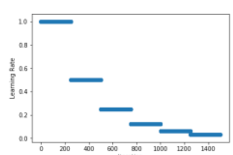
Pros and Cons

- Randomness is *good* to escape from local optima, but *bad* because it means that the algorithm can never settle at the minimum.



Learning Rate Schedule

- One solution to this dilemma is to *gradually reduce* the learning rate.
- The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.
- This process is called *simulated annealing*.
- The function that determines the learning rate at each iteration is called the *learning schedule*.



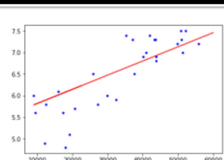
Implementing SGD

```
m, n = X_b_standalized.shape
print ('Examples:', m)
print ('Features:', n)
print ('Labels:', y.shape)

n_epochs = 80000
learning_rate = 0.01

t0, t1 = 5, 50 # learning schedule hyperparameters
def learning_schedule(t):
    return t0 / (t + t1)

theta_sgd = np.random.randn(n, 1) # random initialization
print ('Randomly initialized theta:', theta_sgd)

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b_standalized[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta_sgd) - yi)
        # theta_sgd = theta_sgd - learning_rate * gradients
        learning_rate = learning_schedule(epoch * m + i)
        theta_sgd = theta_sgd - learning_rate * gradients

    if 0 == epoch % (int(n_epochs/5)):
        mse = np.sum(np.square(X_b_standalized.dot(theta_sgd)-y))/m
        print ('Cost ' + str(epoch) + ': ' + str(mse))

print ('Final SGD theta:', theta_sgd)
```

```
Examples: 29
Features: 2
Labels: (29, 1)
Randomly initialized
theta: [[-0.14872021]
 [ 0.50054239]]
Cost 0:
0.3277784112790561
Cost 16000:
0.1807517856126915
Cost 32000:
0.18075052933479704
Cost 48000:
0.1807513083368629
Cost 64000:
0.18075043925114864
Final SGD theta:
[[6.49240039]
 [0.70769211]]
```

Making Predictions

```
X_new = np.array([[22587], [9054], [60000]]) # Cyprus' and Russia GDP
per capita
X_new_b = np.c_[np.ones((X_new.shape[0], 1)), X_new] # add x0 = 1 to
each instance
X_new_b_standalized = NormalizeX(X_new_b)
y_predict_sgd = X_new_b_standalized.dot(theta_sgd)
print(y_predict_sgd)
plt.plot(X_new, y_predict_sgd, "r-")
plt.plot(X, y, "b.")
plt.show()
```



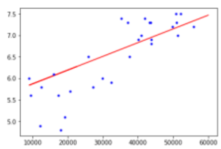
Using **SGDRegressor**

```
from sklearn.linear_model import SGDRegressor
# https://scikit-
learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.htm
l

sgd_reg_model = SGDRegressor(max_iter=50, tol = 1e-3, penalty=None,
eta0=0.1)
sgd_reg_model.fit(X_b_standalized, y.ravel())
# y.ravel(): Return a contiguous flattened array.
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html

print ('coef_:', sgd_reg_model.coef_)
print ('intercept_:', sgd_reg_model.intercept_)
print ('n_iter_:', sgd_reg_model.n_iter_)
```

```
coef_: [3.24026375 0.70052644]
intercept_: [3.24026375]
n_iter_: 8
```

Making Predictions

```
X_new = np.array([[22587], [9054], [60000]]) # Cyprus' and Russia GDP
per capita
X_new_b = np.c_[np.ones((X_new.shape[0], 1)), X_new] # add x0 = 1 to
each instance
X_new_b_standalized = NormalizeX(X_new_b)
y_predict_sgd = sgd_reg_model.predict(X_new_b_standalized)
print(y_predict_bgd)
plt.plot(X_new, y_predict_sgd, "r-")
plt.plot(X, y, "b.")
plt.show()
```



Mini-Batch Gradient Descent

- Mini-batch GD computes the gradients on small random sets of instances called *minibatches*.
- The main advantage of Mini-batch GD over Stochastic GD is that you can get a *performance boost* from hardware optimization of matrix operations, especially when using GPUs.
- Gradient Descent paths in parameter space:

Say $b = 10, m = 1000$.

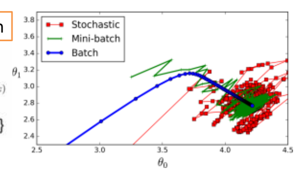
Mini-Batch

Repeat {

for $i = 1, 11, 21, 31, \ldots, 991$ {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every $j = 0, \ldots, n$) } }



Preparing Mini-Batches

```
import math
def random_mini_batches(X, y, mini_batch_size = 64, seed = 0):
    np.random.seed(seed)            # To make your "random" minibatches the same as ours
    m = X.shape[0]                  # number of training examples
    mini_batches = []

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[permutation, :]
    shuffled_y = y[permutation, :].reshape((m,1))

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini batches of size mini_batch_size in
your partitionning
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[k*mini_batch_size : (k+1)*mini_batch_size, :]
        mini_batch_y = shuffled_y[k*mini_batch_size : (k+1)*mini_batch_size, :]
        mini_batch = (mini_batch_X, mini_batch_y)
        mini_batches.append(mini_batch)

    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[num_complete_minibatches*mini_batch_size : m, :]
        mini_batch_y = shuffled_y[num_complete_minibatches*mini_batch_size : m, :]
        mini_batch = (mini_batch_X, mini_batch_y)
        mini_batches.append(mini_batch)

    return mini_batches
mini_batches = random_mini_batches(X_b_standalized, y, 5)
minibatch_X, minibatch_y = mini_batches[0]
print (minibatch_X)
print (minibatch_y)
```

```
[[ 1.         -1.4693612 ]
 [ 1.          0.7093555 ]
 [ 1.          1.14443937]
 [ 1.          0.13557137]
 [ 1.         -0.24488373]]
[[4.9]
 [7.3]
 [7.2]
 [7.4]
 [6. ]]
```

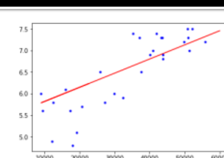
Implementing MBGD

```
m, n = X_b_standalized.shape
print ('Examples:', m)
print ('Features:', n)
print ('Labels:', y.shape)
n_epochs = 80000
learning_rate = 0.01

t0, t1 = 5, 50 # learning schedule hyperparameters
def learning_schedule(t):
    return t0 / (t + t1)

mini_batch_size = 5
seed = 0

theta_mbgd = np.random.randn(n, 1) # random initialization
print ('Randomly initialized theta:', theta_mbgd)

for epoch in range(n_epochs):
    # Define the random minibatches.
    # We increment the seed to reshuffle differently the dataset after each epoch
    seed = seed + 1
    minibatches = random_mini_batches(X_b_standalized, y, mini_batch_size, seed)
    for minibatch in minibatches:
        # Select a minibatch
        minibatch_X, minibatch_y = minibatch
        gradients = (2/len(minibatch_y)) * minibatch_X.T.dot(minibatch_X.dot(theta_mbgd) - minibatch_y)
        learning_rate = learning_schedule(epoch * m + mini_batch_size)
        theta_mbgd = theta_mbgd - learning_rate * gradients

    if 0 == epoch % (int(n_epochs/5)):
        mse = np.sum(np.square(X_b_standalized.dot(theta_mbgd)-y))/m
        print ('Cost ' + str(epoch) + ': ' + str(mse))

print ('Final MBGD theta:', theta_mbgd)
```

```
Examples: 29
Features: 2
Labels: (29, 1)
Randomly initialized theta:
[[1.25733501]
 [1.29229389]]
Cost 0: 2.640584779376952
Cost 16000:
0.18075033811252936
Cost 32000:
0.18075033707247637
Cost 48000:
0.18075033798285348
Cost 64000:
0.1807503392816367
Final MBGD theta:
[[6.49306315]
 [0.70705733]]
```

Making Predictions

```
X_new = np.array([[22587], [9054], [60000]]) # Cyprus' and Russia GDP
per capita
X_new_b = np.c_[np.ones((X_new.shape[0], 1)), X_new] # add x0 = 1 to
each instance
X_new_b_standalized = NormalizeX(X_new_b)
y_predict_mbgd = X_new_b_standalized.dot(theta_mbgd)
print(y_predict_mbgd)
plt.plot(X_new, y_predict_mbgd, "r-")
plt.plot(X, y, "b.")
plt.show()
```



Polynomial Regression

- What if your data is actually *more complex* than a simple straight line?
- Surprisingly, you can actually *use a linear model* to fit nonlinear data.
- A simple way to do this is to *add powers of each feature* as new features, then train a linear model on this extended set of features.
- This technique is called *Polynomial Regression*.

$$y = b_0 + b_1 x_1 + b_2 x_1^2 + \ldots + b_n x_1^n$$

Adding Powers Of Each Feature

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=3,
include_bias=False)
X_poly = poly_features.fit_transform(X_b_standalized)
print ('X:', X_b_standalized[0])
print ('X + powers:', X_poly[0])
```

```
X: [ 1.        -1.690613]
X + powers: [ 1.          -1.690613    1.          -1.690613
2.85817231  1.
 -1.690613    2.85817231 -4.83206327]
```

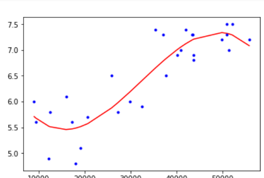
Training The Model

```
lin_poly_reg = LinearRegression()
lin_poly_reg.fit(X_poly, y)
print (lin_poly_reg.coef_)
print (lin_poly_reg.intercept_)
```

```
[[ 0.00000000e+00  4.03973993e-01 -1.11022302e-16  4.03973993e-
01
  -3.09222238e-02 -2.18952885e-47  4.03973993e-01 -3.09222238e-
02
  -3.00617661e-01]]
[6.48258625]
```

Making Predictions (On Training Set)

```
y_predict_poly_reg = lin_poly_reg.predict(X_poly)
plt.plot(X, y_predict_poly_reg, "r-")
plt.plot(X, y, "b.")
plt.show()
```



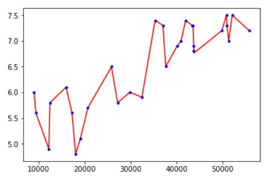
High-Degree Polynomial Regression

```
from sklearn.preprocessing import PolynomialFeatures
poly_features_30 = PolynomialFeatures(degree=30,
include_bias=False)
X_poly_30 = poly_features_30.fit_transform(X_b_standalized)

lin_poly_30_reg = LinearRegression()
lin_poly_30_reg.fit(X_poly_30, y)
print (len(lin_poly_30_reg.coef_[0]))
```

```
495
```

Is It Good Or Bad?

```
y_predict_poly_30_reg = lin_poly_30_reg.predict(X_poly_30)
plt.plot(X, y_predict_poly_30_reg, "r-")
plt.plot(X, y, "b.")
plt.show()
```



A Model's Generalization Error

- An important theoretical result of statistics and Machine Learning is the fact that *a model's generalization error* can be expressed as the sum of three very different errors:
  - *Bias*: This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to *underfit* the training data.
  - *Variance*: This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance, and thus to *overfit* the training data.
  - *Irreducible error*: This part is due to the *noisiness* of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

The Bias/Variance Tradeoff

- Increasing a *model's complexity* will typically increase its variance and reduce its bias.
- Conversely, reducing a model's complexity increases its bias and reduces its variance.
- This is why it is called a *tradeoff*.

Regularized Linear Models

- A good way *to reduce overfitting* is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data.
- For example, a simple way to *regularize a polynomial model* is to reduce the number of polynomial degrees.

Ridge Regression

- *Ridge Regression* (also called Tikhonov regularization) is a regularized version of Linear Regression: a regularization term equal to $\alpha\Sigma_{i=1}^{n} \theta_i^2$ is added to the cost function.
- Ridge Regression *cost function*:

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha\frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

- Ridge Regression *closed-form solution*:

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X} + \alpha\mathbf{A}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Ridge Regression Closed-Form Solution

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

```
from sklearn.linear_model import Ridge
ridge_reg_model = Ridge(alpha=1, solver="cholesky")
# Train the model
ridge_reg_model.fit(X, y)
print ('coef:', ridge_reg_model.coef_)
print ('intercept:', ridge_reg_model.intercept_)
```

```
coef: [[4.91154459e-05]]
intercept: [4.8530528]
```

Ridge Regression Using SGD

```
from sklearn.linear_model import SGDRegressor
# https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html

sgd_reg_model = SGDRegressor(max_iter=50, tol = 1e-3, penalty="l2", eta0=0.01)
sgd_reg_model.fit(X_b_standalized, y.ravel())
# y.ravel(): Return a contiguous flattened array.
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html

print ('coef:', sgd_reg_model.coef_)
print ('intercept:', sgd_reg_model.intercept_)
print ('Epochs:', sgd_reg_model.n_iter_)
```

```
coef: [3.22409025 0.64268936]
intercept: [3.22473792]
Epochs: 37
```

Lasso Regression

- Least Absolute Shrinkage and Selection Operator Regression (simply called *Lasso Regression*) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ1 norm of the weight vector instead of half the square of the ℓ2 norm.
- Lasso Regression *cost function*:

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \sum_{i=1}^{n} \left|\theta_i\right|$$

Implementing Lasso Regression

```
from sklearn.linear_model import Lasso
lasso_reg_model = Lasso(alpha=0.1)
# Train the model
lasso_reg_model.fit(X, y)
print ('coef:', lasso_reg_model.coef_)
print ('intercept:', lasso_reg_model.intercept_)
```

```
coef: [4.91149633e-05]
intercept: [4.85306891]
```

Elastic Net

- Elastic Net *cost function*

$$J(\theta) = \mathrm{MSE}(\theta) + r\alpha \sum_{i=1}^{n} \left|\theta_i\right| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

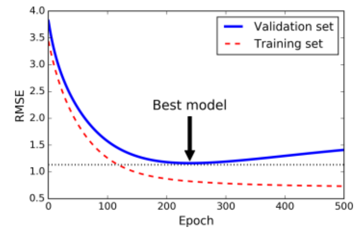
- When r = 0, Elastic Net is equivalent to Ridge Regression, and when r = 1, it is equivalent to Lasso Regression

Linear Regression, Ridge, Lasso, or Elastic Net?

- It is almost always preferable to have *at least a little bit of regularization*, so generally you should avoid plain Linear Regression.
- Ridge is a good default, but if you suspect that only *a few features* are actually useful, you should prefer Lasso or Elastic Net since they tend to reduce the useless features' weights down to zero.
- In general, *Elastic Net is preferred over Lasso* since Lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Early Stopping

- A very different way to regularize iterative learning algorithms such as Gradient Descent is to *stop training* as soon as the validation error reaches a minimum.
- This is called *early stopping*.



Implementing Early Stopping

```
es_sgd_reg_model = SGDRegressor(early_stopping=True,
tol=0.0001, validation_fraction=0.2,
                              max_iter=3000,
n_iter_no_change=3, warm_start=True, penalty=None,
learning_rate="constant", eta0=0.00005)
es_sgd_reg_model.fit(X_b_standalized, y.ravel())
print ('coef:', es_sgd_reg_model.coef_)
print ('intercept:', es_sgd_reg_model.intercept_)
print ('epochs:', es_sgd_reg_model.n_iter_)
```

```
coef: [3.25437434 0.63154547]
intercept: [3.25437434]
epochs: 2663
```

Thank You for Your Time