Multivariate Regression

Lecture 4

About the Course Textbook

Second edition will be published sometime in September. There will be hard copies of the book for you to checkout (either through UVA Library or myself).

In the meantime, you can access the first version of the book. If you are on Grounds, you will gain access via below link:

https://proquest.safaribooksonline.com/9781491962282

This link will also eventually be updated to the second edition of the book.

Many content is already in the lecture slides and code repo. So you won't miss out much!

So far

We've treated most of ML algorithms like "black boxes", and we are able to:

- Find the best fit line with a few regression models
- Build prediction systems for **world happiness** and **housing prices**
- Try out Google Colab coding (**wasn't it fun?**)

Understanding how things work

- Help **quickly** identify appropriate models to use
- Help **debug** issues and perform error analysis more effectively
- Build upon **knowledge** to more complex models (ie. deep neural networks)

We are about to learn know how a learning algorithm actually works under the hood!

Today: Learning Objectives

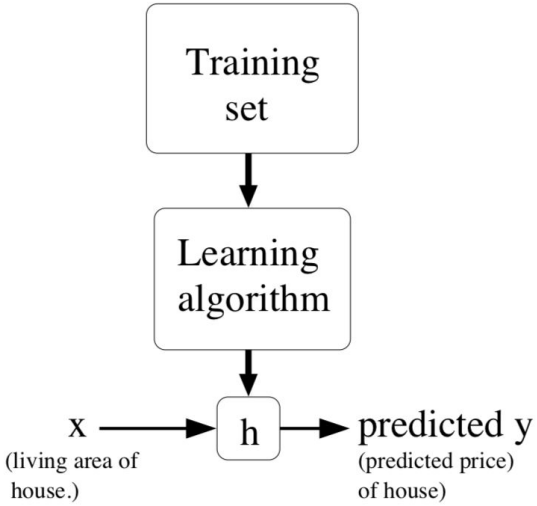
- ❑ Revisit **Linear Regression** at a closer perspective
- ❑ Formulate a direct **close-form equation** to compute model parameter
- ❑ Use an iterative optimization approach, called **Gradient Descent** (GD)
- ❑ Explore at a few variants of GD: **Batch**, **Mini-batch**, and **Stochastic**
- ❑ Learn about **Polynomial Regression** to fit non-linear data.

Data Representation: From Table to Matrix

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_n$	$\mathbf{y}$	
	total_bedrooms	population	households	median_income	median_house_value	
$\mathbf{x}^{(1)}$	129.0	322.0	126.0	. . . 8.3252	452600.0	$y^{(1)}$
$\mathbf{x}^{(2)}$	1106.0	2401.0	1138.0	. . . 8.3014	358500.0	$y^{(2)}$
$\mathbf{x}^{(3)}$	190.0	496.0	177.0	. . . 7.2574	352100.0	$y^{(3)}$
	⋮	⋮	⋮	⋮	⋮	
$\mathbf{x}^{(m)}$	280.0	565.0	259.0	. . . 3.8462	342200.0	$y^{(m)}$

$\mathbf{X}$

Linear Regression



A generalized linear model:

$$\hat{y} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

OR a more concisely vectorized (short) form of the model/hypothesis:

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

Hypothesis function

***1 x (n+1)* Parameter Vector**

***(n+1) x 1* Feature Vector**

Linear Algebra Review: Dot Product

Definition:

$$x \cdot y = x^T y \in \mathbb{R} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ x_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i.$$

$$x \cdot y = x^T y = y^T x$$

Hypothesis $h_\theta : \mathbf{X} \rightarrow \mathbf{Y}$

Predicted Outputs:

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} h_\theta(\mathbf{x}^{(1)}) \\ h_\theta(\mathbf{x}^{(2)}) \\ \vdots \\ h_\theta(\mathbf{x}^{(m)}) \end{bmatrix} = \begin{bmatrix} \theta^T \mathbf{x}^{(1)} \\ \theta^T \mathbf{x}^{(2)} \\ \vdots \\ \theta^T \mathbf{x}^{(m)} \end{bmatrix} = \mathbf{X}\theta$$

mx1

1x(n+1) *(n+1)x1*

mx(n+1) *(n+1)x1*

Loss Function → MSE (Mean Square Error)

$$\text{MSE}\ (\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2$$

Feature Matrix

Hypothesis

Feature vector of instance i^{th}

Label of instance i^{th}

We will derive this into a loss function and try to minimize it

Loss Function $J(\theta)$

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

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\underbrace{\boxed{\theta^T\mathbf{x}^{(i)} - y^{(i)}}}_{z^{(i)}})^2$$

Recall that $\sum_i {z^{(i)}}^2 = \mathbf{z}^T\mathbf{z}$, so:

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

Continue in Matrix form

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

Removing the constant and deriving the transpose:

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

Multiply them out: $(\mathbf{a}^T - \mathbf{b}^T)(\mathbf{a} - \mathbf{b}) = \mathbf{a}^T\mathbf{a} - \mathbf{a}^T\mathbf{b} - \mathbf{b}^T\mathbf{a} + \mathbf{b}^T\mathbf{b}$

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

Simplify further: $\mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a}$

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

To minimize the loss function w.r.t θ

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

To minimize J, we take **partial derivative** and then **set it to zero:**

$$\frac{\partial J}{\partial \theta} = 2\mathbf{X}^T\mathbf{X}\theta - 2\mathbf{X}^T\mathbf{y} = 0$$

Simplify 2 and move thing to the other side:

$$\mathbf{X}^T\mathbf{X}\theta = \mathbf{X}^T\mathbf{y}$$

Assuming that $(\mathbf{X}^T\mathbf{X})^{-1}$ is invertible, we can multiply both side by it:

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

← **It's a beautiful thing!**

The Normal Equation

We found the value of parameter $\hat{\theta}$ that **directly minimizes the loss** in a **closed-form solution** called the **normal equation**:

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

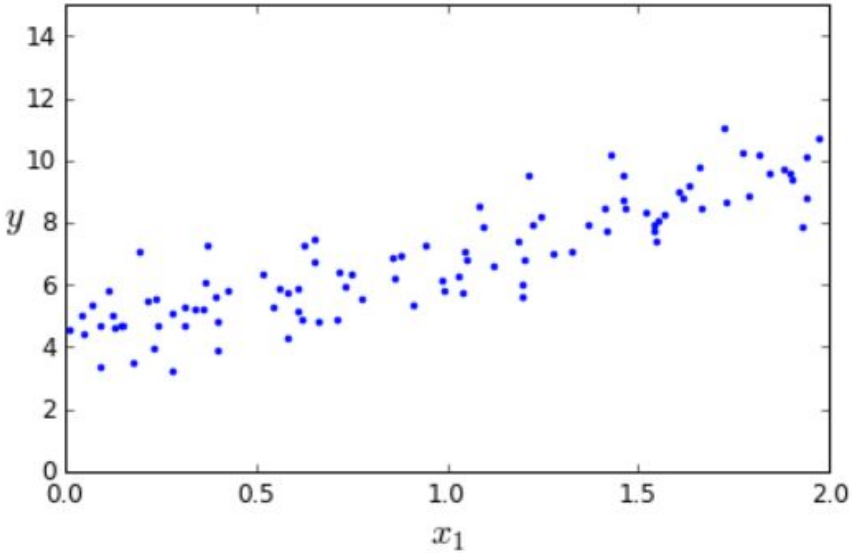
Optimal parameters

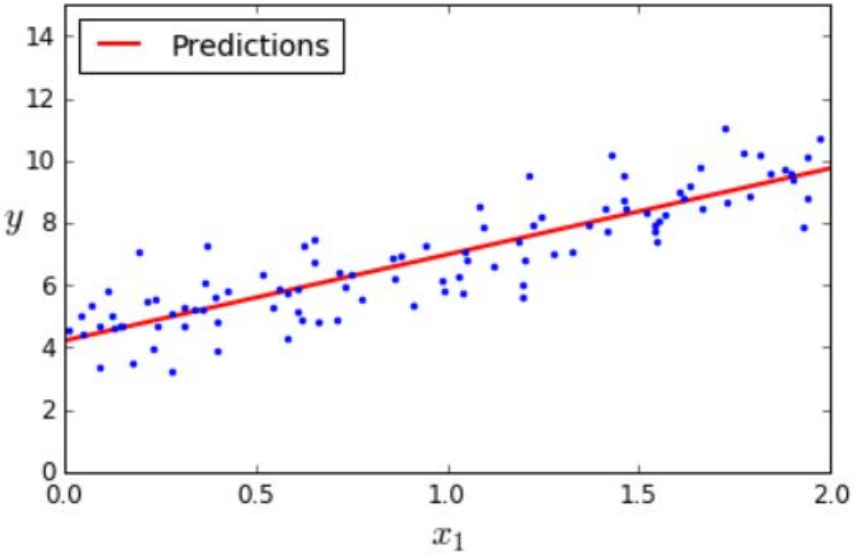
Training Set Feature Matrix

Training Label Vector

```
X_b = np.c_[np.ones((100, 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)
```

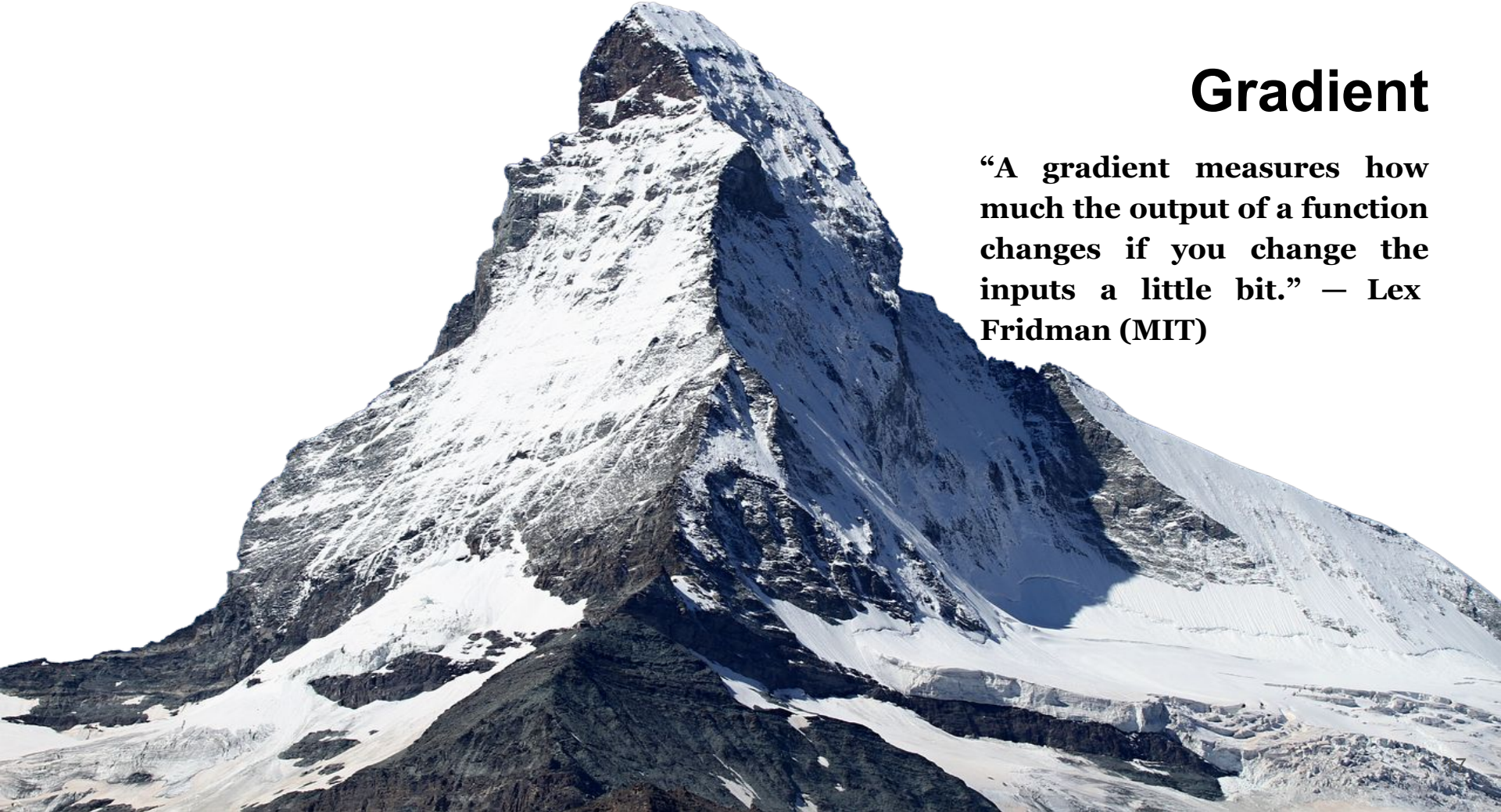
Demo on Colab notebook





Computational Complexity

- Inverting such a matrix is typically about **O(n^3)** where n is the number of features
- Grow linearly with the number of samples in the training set **O(m)** as long as you can fit in memory
- Once trained, the model work fast (linearly with number of samples)

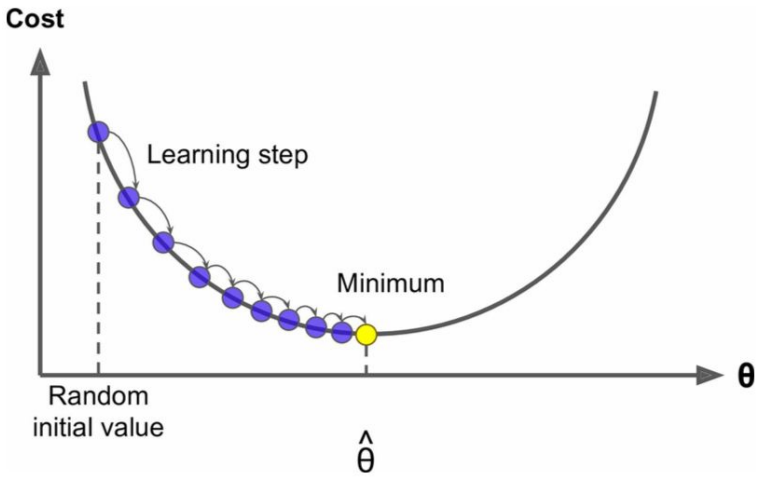


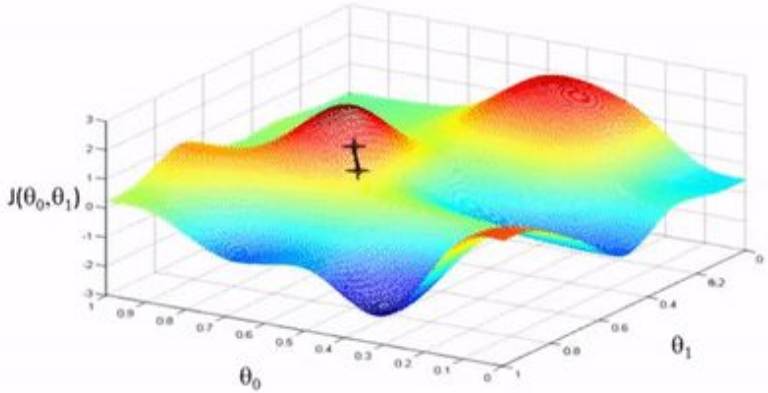
Gradient

“A gradient measures how much the output of a function changes if you change the inputs a little bit.” — Lex Fridman (MIT)

Gradient Descent (GD)

- Generic optimization algorithm to find optimal solution to wide range of problems.
- Tweak parameters **iteratively** in order to **minimize** a loss function.
- Determined by a ***learning rate*** hyperparameter





Andrew Ng

Learning Rate too small

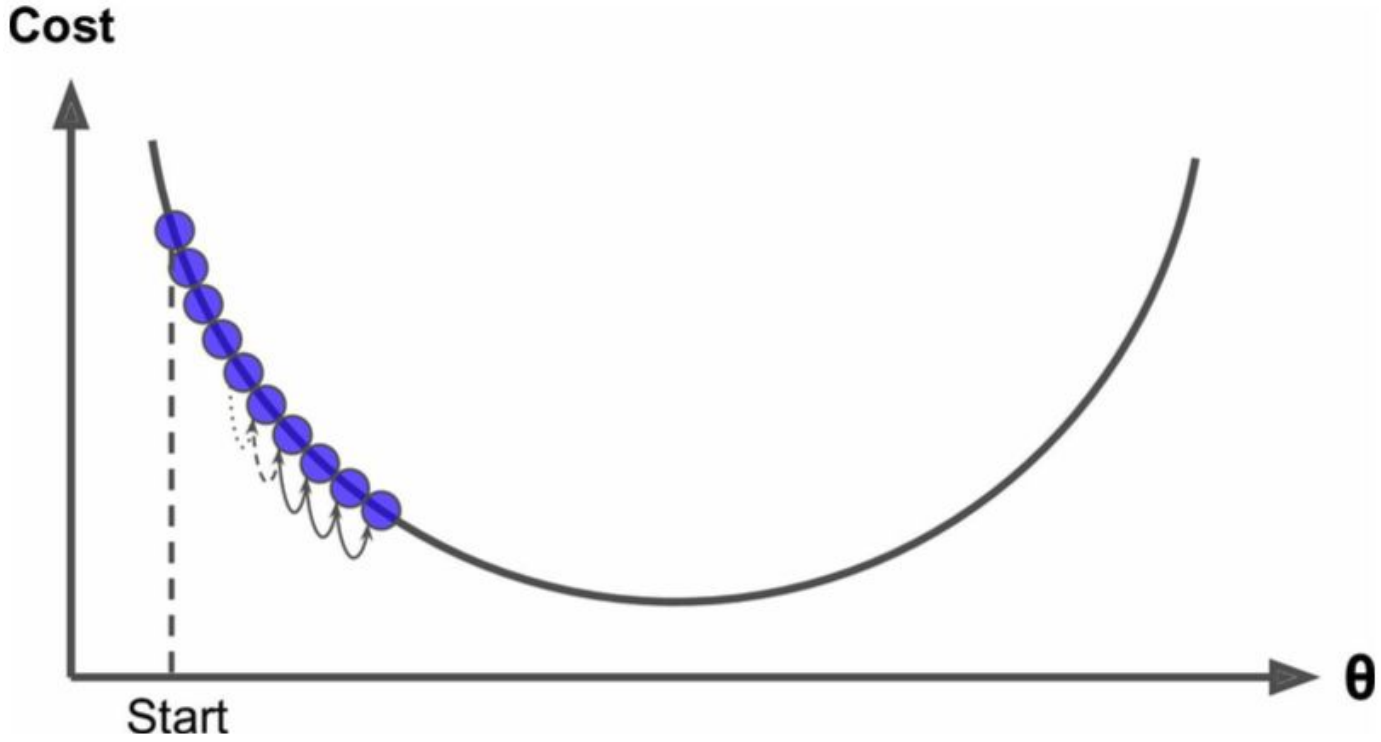


Figure 4-4. Learning rate too small

Learning rate too big

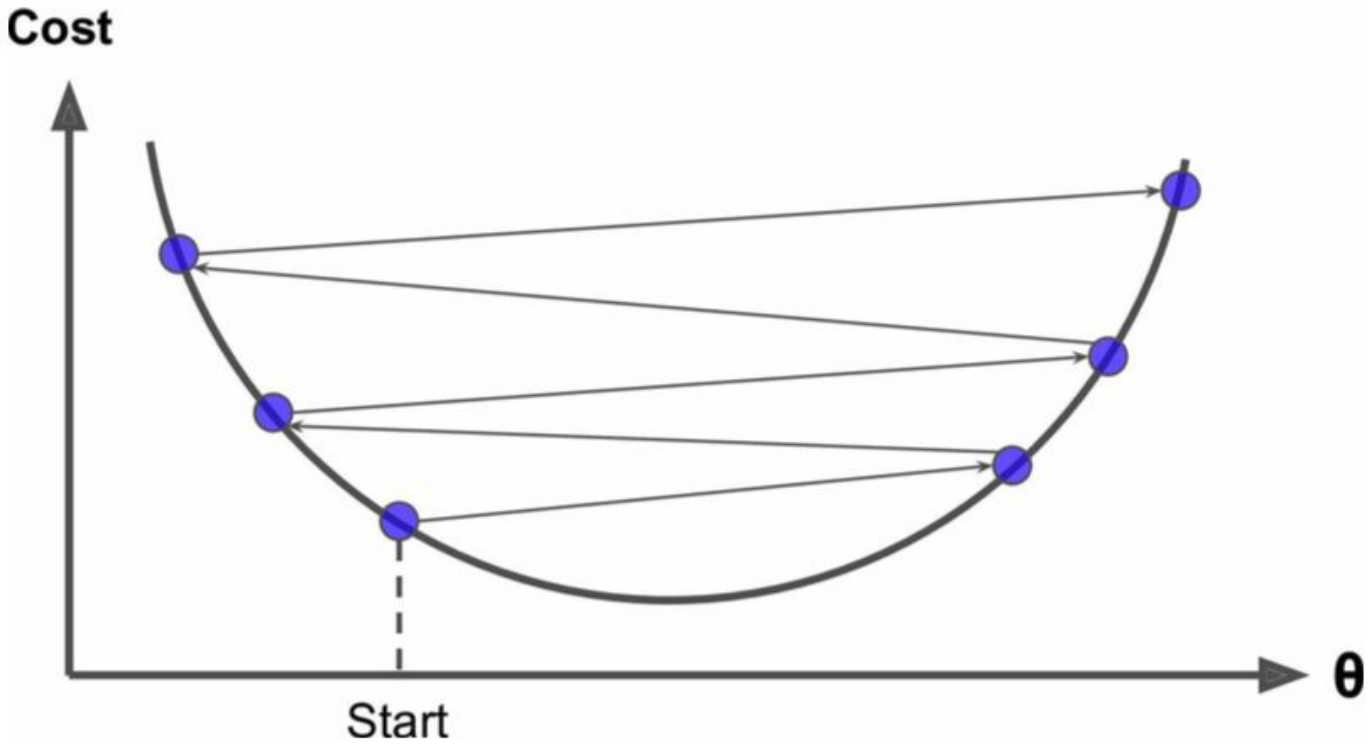
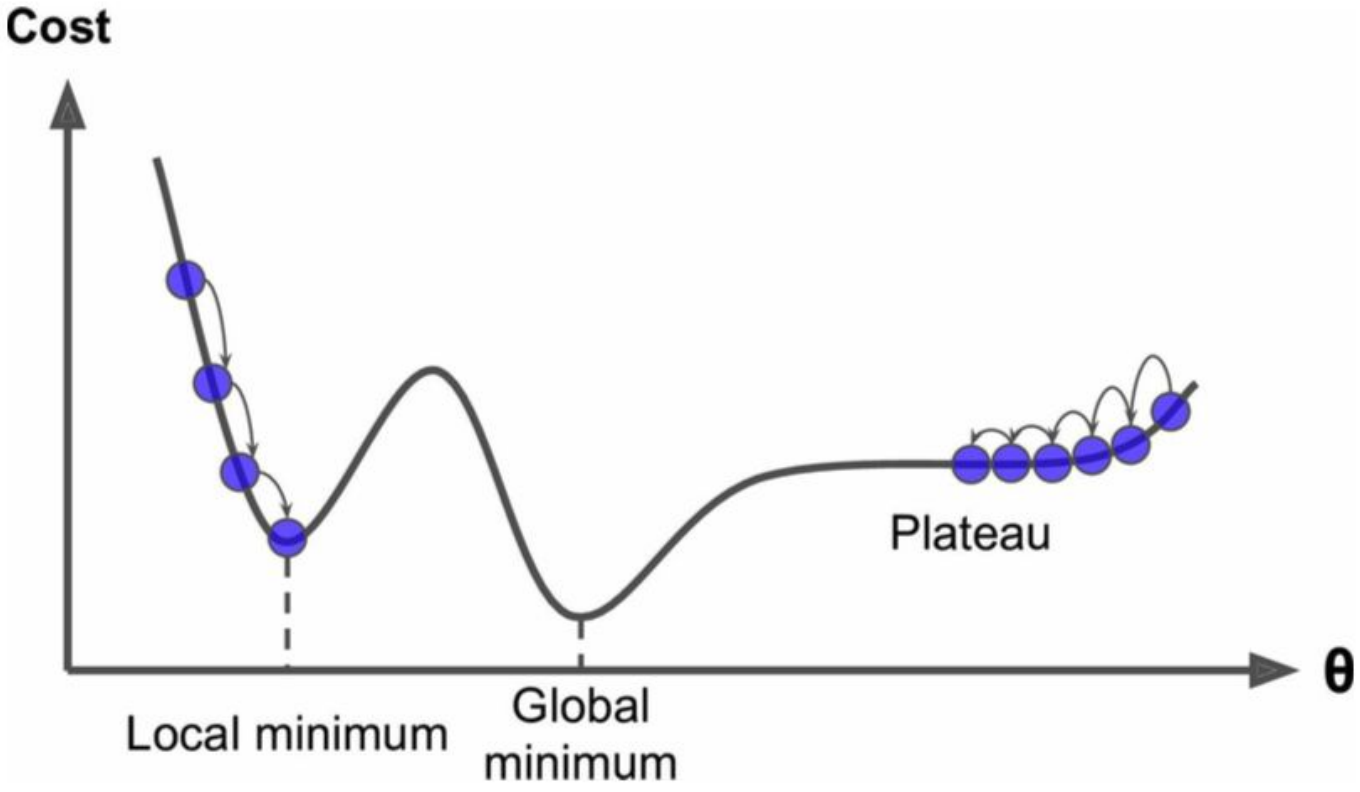


Figure 4-5. Learning rate too large

Local minimum vs. global minimum



Feature Scaling

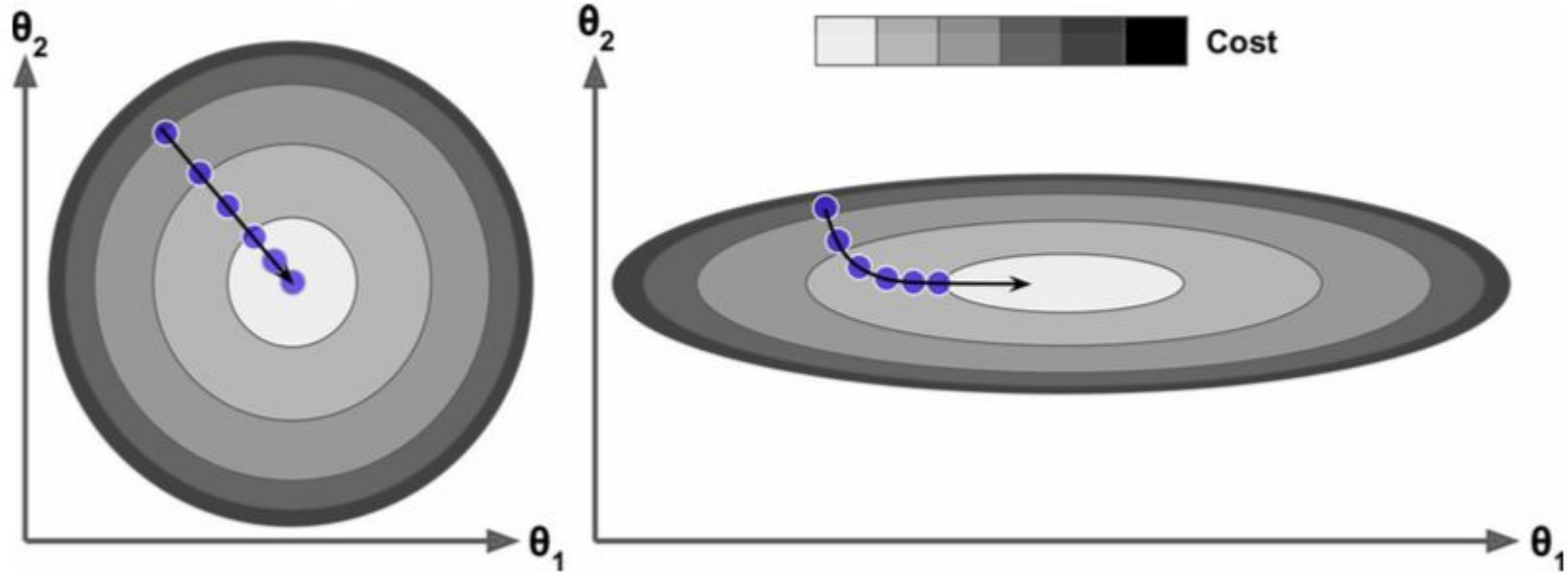


Figure 4-7. Gradient Descent with and without feature scaling

Batch Gradient Descent (BGD)

- Calculate how much the loss function will change if we change the parameter just a bit (ie. partial derivatives)
- Same as: “what is the slope of the mountain if I take a step to the east?” then as the same question for other directions
- Use all of training data → batch

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), (j = 1 \ldots n)$$

Learning Rate **Cost Function**

BGD Formulation

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta), (j = 1 \ldots n)$$

Expand loss function:

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

Take partial derivative:

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

← **Why is this term here?**

Calculus Review: Partial Derivative

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} \left(h_\theta(x) - y\right)^2 \\
&= 2 \cdot \frac{1}{2} \left(h_\theta(x) - y\right) \cdot \frac{\partial}{\partial \theta_j} \left(h_\theta(x) - y\right) \\
&= \left(h_\theta(x) - y\right) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{n} \theta_i x_i - y \right) \\
&= \left(h_\theta(x) - y\right) x_j
\end{aligned}
$$

Gradient Vector $\nabla J(\theta)$

- Need to calculate over the full training set **X**
- Use the whole batch at every step → can be slow on large data set

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\theta) \end{bmatrix} = \begin{bmatrix} \frac{2}{m} \sum_i \left(\theta^T \mathbf{x}^{(i)} - y^{(i)}\right) x_0^{(i)} \\ \frac{2}{m} \sum_i \left(\theta^T \mathbf{x}^{(i)} - y^{(i)}\right) x_1^{(i)} \\ \vdots \\ \frac{2}{m} \sum_i \left(\theta^T \mathbf{x}^{(i)} - y^{(i)}\right) x_n^{(i)} \end{bmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y})$$

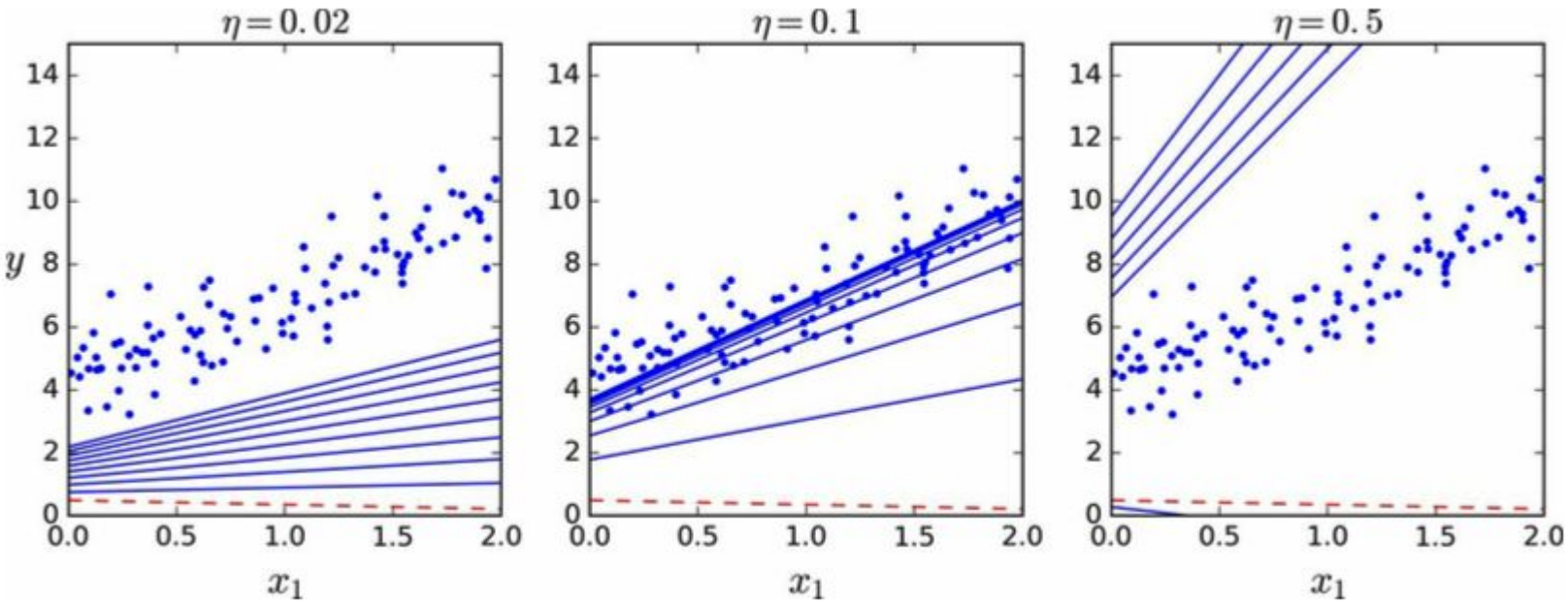
Gradient Descent (GD) Step

$$\theta := \theta - \alpha \nabla J(\theta)$$

$$\theta = \theta - \alpha \frac{2}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y})$$

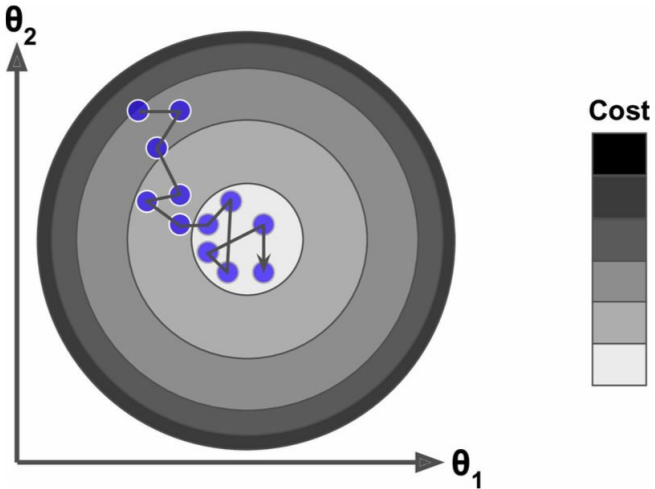
```
eta = 0.1
n_iterations = 1000
m = 100
theta = np.random.randn(2,1)

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

GD with various learning rate



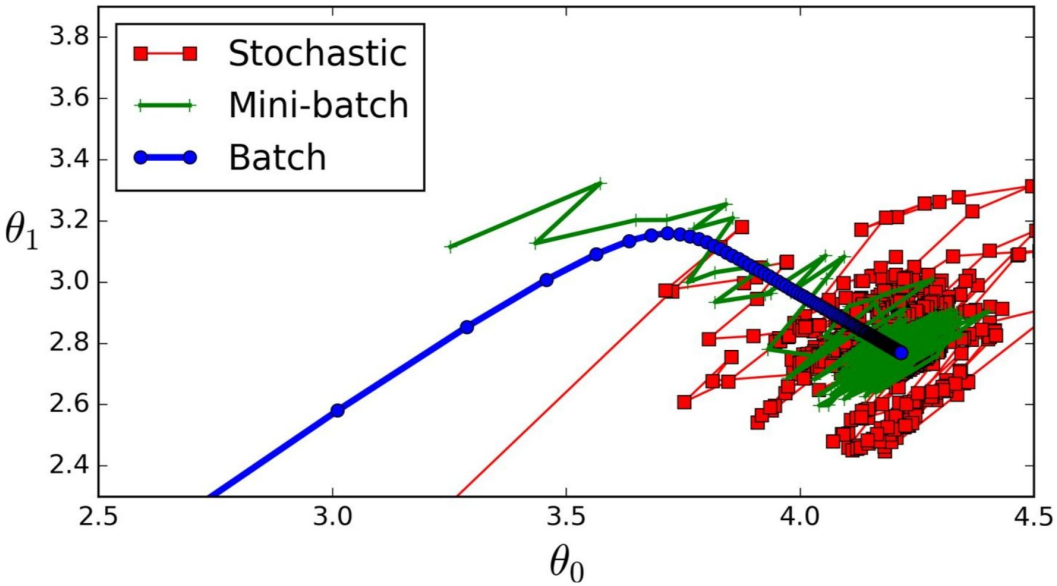
Stochastic Gradient Descent (SGD)

- Instead of using the whole training set, SGD picks **a random sample** in the training set at every step and compute the gradients based on **that** sample.
- It's extremely fast, but is "**stochastic**" (random) in nature, its final parameter values are bounce around the minimum, which are good, but not optimal.



Mini-batch Gradient Descent

Instead of training on the full set (Batch GD) or based on just one sample (Scholastic GD), Mini-batch GD computes gradients on **small random sets of samples** (10-1000 in size) called mini-batches → best of both world



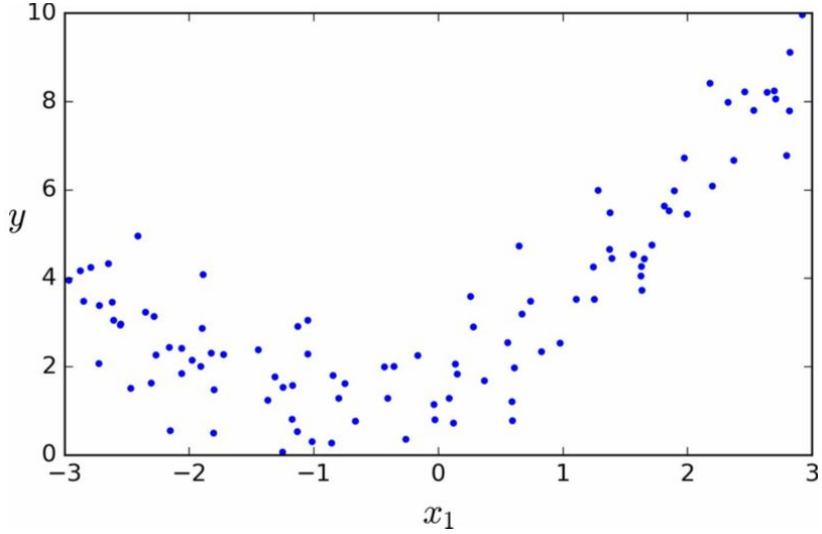
Regression Algorithms Comparison

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

What if we have non-linear data?

More complex than a straight line? → can we use linear model to fit nonlinear data?

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```



$$y = 2 + x + 0.5x^2 + \epsilon$$

Transform data to fit properly

We can add the square (2nd degree polynomial) to each feature of the training set

→ training data now contain the original feature and its square value.

$$\hat{y} = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3^2$$

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

```
array([-0.75275929])
```

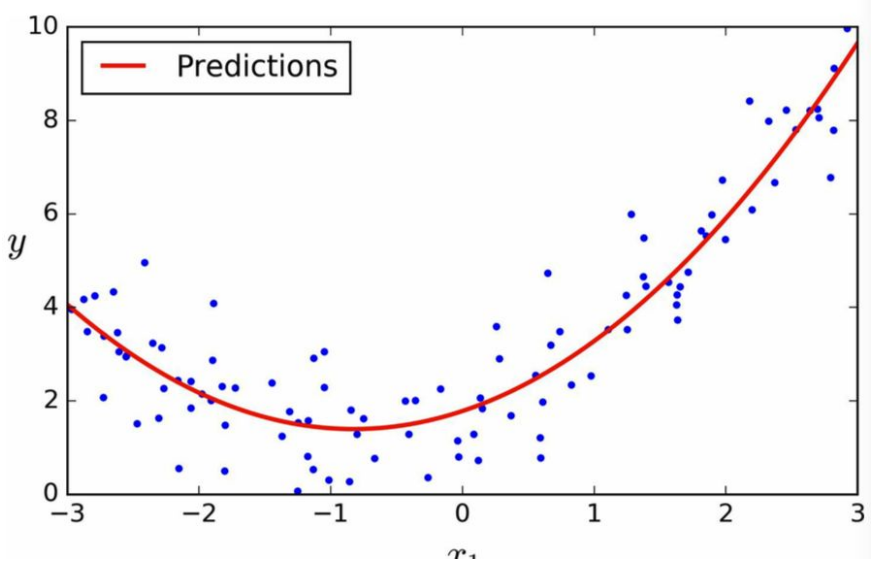
```
X_poly[0]
```

```
array([-0.75275929,  0.56664654])
```

Fit a linear regression to this data

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

```
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```



Does it fit well?

Original function:

```
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

$$y = 2 + x + 0.5x^2 + \epsilon$$

Prediction function:

```
lin_reg.intercept_, lin_reg.coef_
```

```
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```

$$\hat{y} = 1.78 + 0.93x + 0.56x^2$$

← **not too bad, right?**

Summary: Learning Objectives

- ✓ Revisit **Linear Regression** at a closer perspective
- ✓ Formulate a direct **close-form equation** to compute model parameter
- ✓ Use an iterative optimization approach, called **Gradient Descent** (GD)
- ✓ Explore at a few variants of GD: **Batch**, **Mini-batch**, and **Stochastic**
- ✓ Learn about **Polynomial Regression** to fit non-linear data.

Next: we will learn how to fix overfitting model by using **regularization**

Review: Matrix-Vector Multiplication

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^n$, their product is a vector $y = Ax \in \mathbb{R}^m$.

If we write A by rows, then we can express Ax as,

$$y = Ax = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} x = \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_m^T x \end{bmatrix}.$$

Review: Partial Derivative

- Scalar multiplication: $\partial_x[af(x)] = a[\partial_x f(x)]$
- Polynomials: $\partial_x[x^k] = kx^{k-1}$
- Function addition: $\partial_x[f(x) + g(x)] = [\partial_x f(x)] + [\partial_x g(x)]$
- Function multiplication: $\partial_x[f(x)g(x)] = f(x)[\partial_x g(x)] + [\partial_x f(x)]g(x)$
- Function division: $\partial_x \left[\frac{f(x)}{g(x)}\right] = \frac{[\partial_x f(x)]g(x) - f(x)[\partial_x g(x)]}{[g(x)]^2}$
- Function composition: $\partial_x[f(g(x))] = [\partial_x g(x)][\partial_x f](g(x))$
- Exponentiation: $\partial_x[e^x] = e^x$ and $\partial_x[a^x] = \log(a)e^x$
- Logarithms: $\partial_x[\log x] = \frac{1}{x}$