Training Models

Recap

- So far we have learned how to get things done
  - Without knowing anything about
  - What's under the hood
- However understanding how things work can help us in
  - Choosing the appropriate model
  - Right training algorithm to use
  - Good set of hyperparameters
  - Debugging issues and
  - Perform error analysis

CLOUD x LAB

Training Models

- In this session
  - We will learn about internals of how models are trained
- We'll cover various algorithms of Gradient Descent like
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-batch Gradient Descent
- Later we will learn different ways of regularizing linear models
  - Ridge Regression
  - Lasso Regression
  - Elastic Net

CLOUD x LAB

Training Models

- Then we will learn about logistic regression
- Most of the topics discussed in this chapter will also be essential in
  - Understanding
  - Building and
  - Training neural networks

CLOUD x LAB

Notations

- Let's understand the notations used in this course
- These notations will help in understanding the mathematical equations
- Let's take the example of housing dataset in end-to-end project for understanding notations

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Notations

- m is the number of instances in the training dataset
- For example
  - If we have 2000 districts in our training dataset
  - Then $m = 2000$

Notations

- $x^{(i)}$ is a vector of all the feature values (excluding the label)
  - Of the ith instance in the dataset and
  - $y^{(i)}$ is its label (the desired output value for that instance)

Notations

- For example, if the first district in the dataset has below values
  - longitude –118.29°
  - latitude - 33.91°
  - inhabitants - 1,416
  - median income - $38,372
  - median house value - $156,400
  - Then x(1) and y(1) will be

Notations

- For example, if the first district in the dataset has below values
  - longitude –118.29°
  - latitude - 33.91°
  - inhabitants - 1,416
  - median income - $38,372
  - median house value - $156,400

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,\ 416 \\ 38,\ 372 \end{pmatrix}$$

$$y^{(1)} = 156,\ 400$$

CLOUD x LAB

Notations

- **X** is a matrix
  - Contains all the feature values (excluding labels) of all instances in the dataset
  - There is one row per instance and
  - the ith row is equal to the transpose of $x^{(i)}$, noted $(x^{(i)})^T$

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

CLOUD x LAB

Notations

- *h* is system's prediction function
  - Also called as hypothesis
  - When we train a model using instance feature vector $x^{(i)}$
  - We get predicted value $\hat{y}^{(i)} = h(x^{(i)})$ for that instance
  - $\hat{y}$ is pronounced "y-hat"

CLOUD x LAB

Notations

- For example if model predicts that the
  - Median housing price in the first district is $158,400
  - Then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158{,}400$
- The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2{,}000$
- RMSE(X, *h*) is the cost function
  - Measured on the set of examples using hypothesis h

CLOUD x LAB

Notations

In general we will use

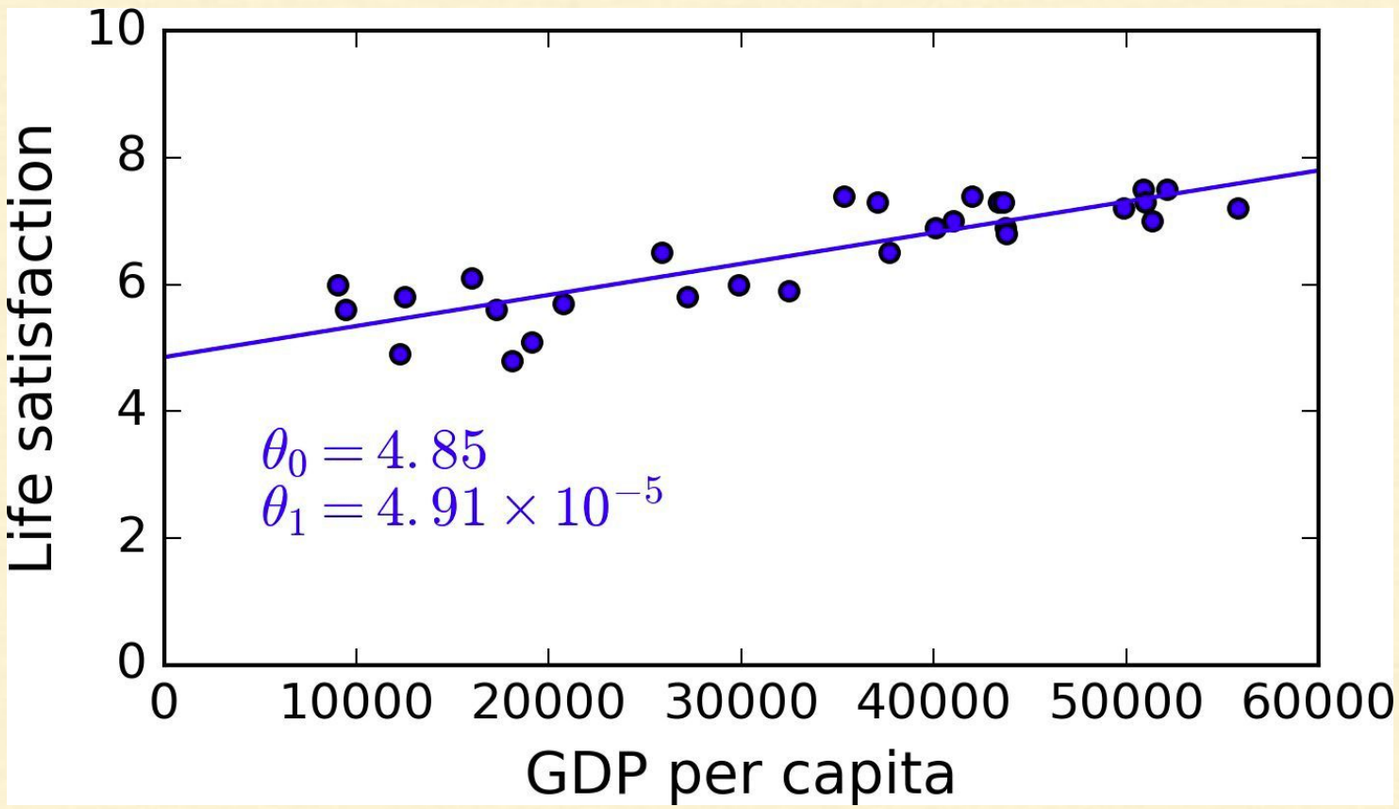
- Lowercase italic font for
  - Scalar values such as m or $y^{(i)}$ and
  - Function names such as h
- Lowercase bold font for vectors
  - Such as $\mathbf{x}^{(i)}$
- Uppercase bold font for matrices
  - Such as $\mathbf{X}$

CLOUD x LAB

Linear Regression

Linear Regression

- Let's look at the simple linear regression model of
  - Life satisfaction vs GDP per capita



$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$

CLOUD x LAB

Linear Regression

$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$

- θ_0 and θ_1 are the model's parameters
- Let's say our model has learned the parameters θ_0 and θ_1 as
  - $\theta_0 = 4.85$
  - $\theta_1 = 4.91 * 10^{-5}$
- Then the final equation becomes

$$\text{life_statisfaction} = \underbrace{4.85}_{\theta_0} + \underbrace{4.91 * 10^{-5}}_{\theta_1} * \text{GDP_per_capita}$$

CLOUD x LAB

Question

What does this equation mean?

$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$

Overview

- We can find the life_satisfaction value for an unknown GDP per capita
- Find the live_satisfaction value for GDP per capita of 33,000
- By plugging in the value in the equation

$$\text{life_statisfaction} = \underbrace{4.85}_{\theta_0} + \underbrace{4.91 * 10^{-5}}_{\theta_1} * \underbrace{33000}_{\text{GDP per capita}}$$

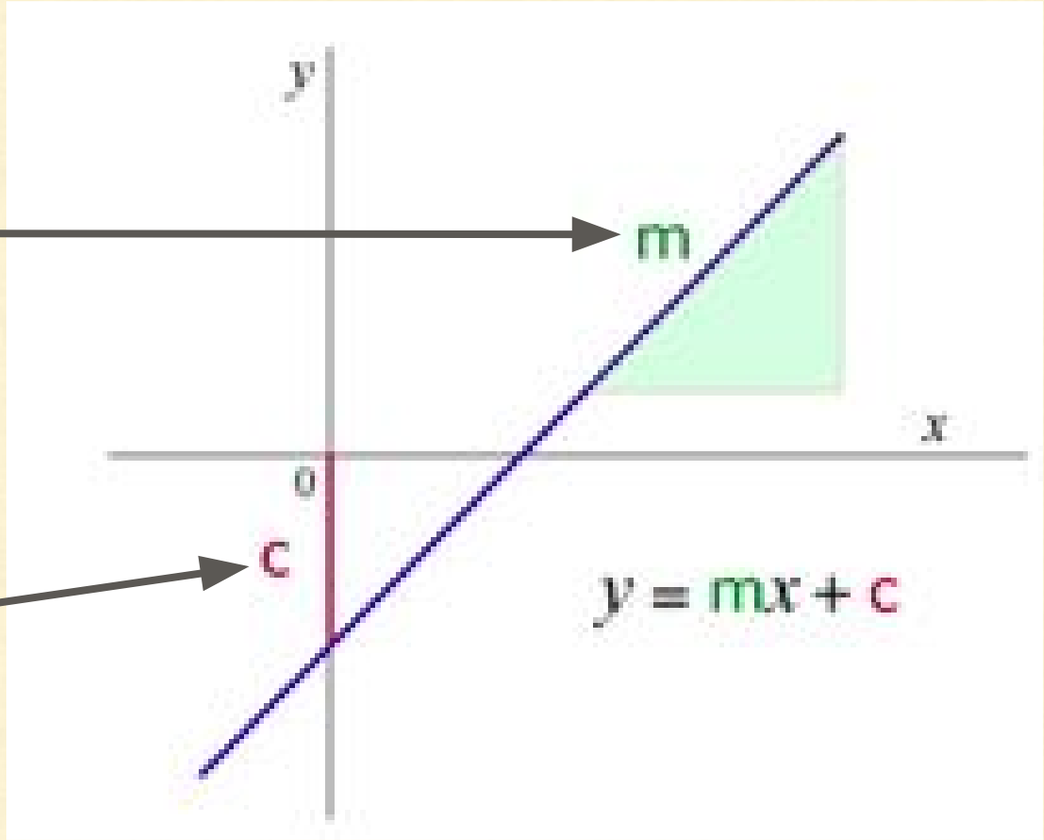
Linear Regression

General Equation of Line

- Equation of line y = mx + c
- Did you notice that previous equation is an equation of a line

Slope (θ_1)

y intercept (θ_0)



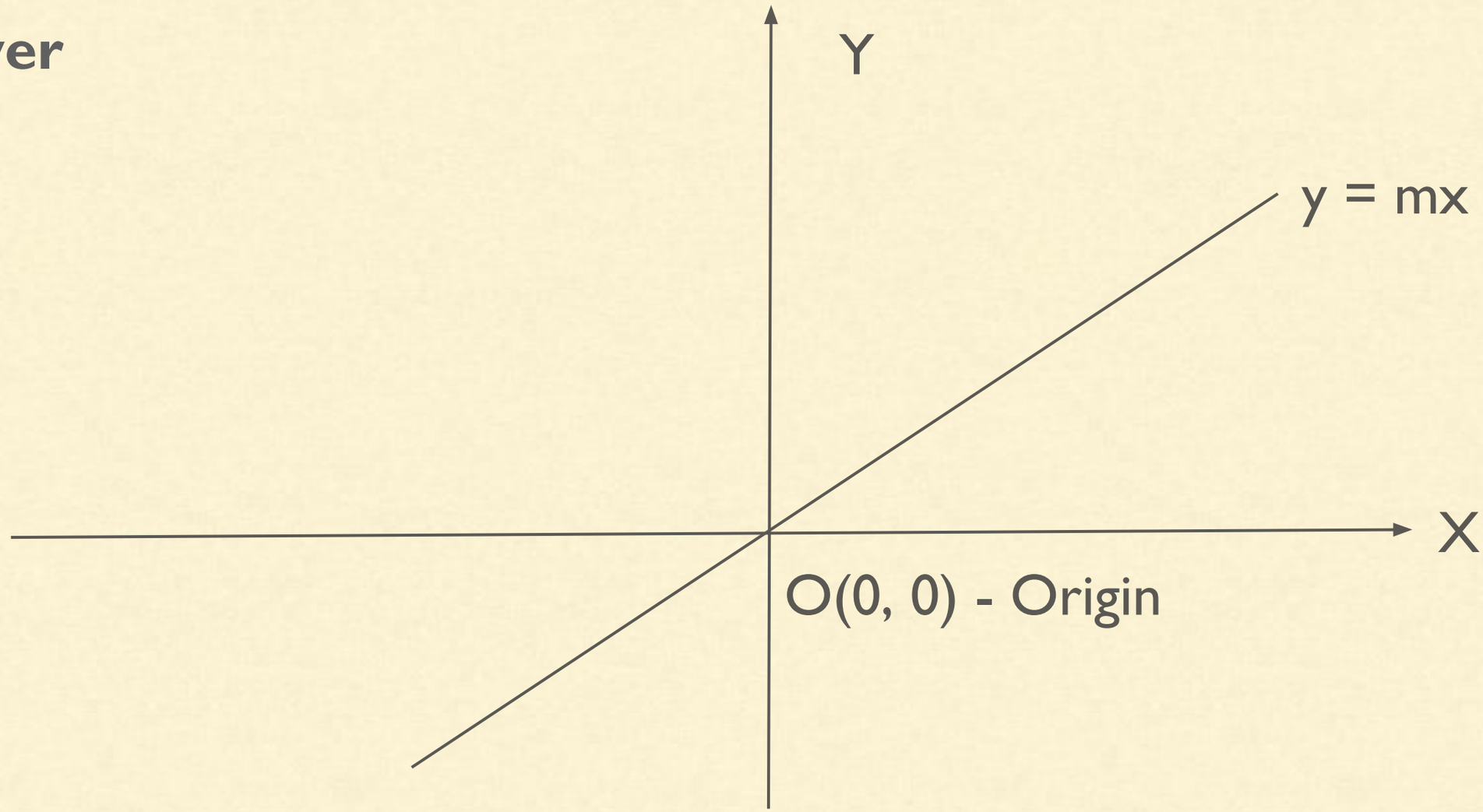
Question

What if y-intercept (θ_0) is zero?

CLOUD x LAB

Linear Regression

Answer



If y-intercept is zero then the line will pass through origin i.e (0, 0)

CLOUD x LAB

Linear Regression

- Let's go back to end-to-end project
- If we build the model for predicting house prices using below attributes
  - longitude
  - latitude
  - inhabitants and
  - median_income

Linear Regression

General Equation of Linear Regression

- Then our trained model will be like below equation

$$\textbf{Predicted_house_price} = \theta_0 + \theta_1 * \textbf{longitude} + \theta_2 * \textbf{latitude} + \theta_3 * \textbf{inhabitants} + \theta_4 * \textbf{median_income}$$

Where

θ_0 - Constant. Also called as bias or intercept term

θ_1 - Weight of longitude feature

θ_2 - Weight of latitude feature

θ_3 - Weight of inhabitants feature

θ_4 - Weight of median_income feature

CLOUD x LAB

Linear Regression

General Equation of Linear Regression

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

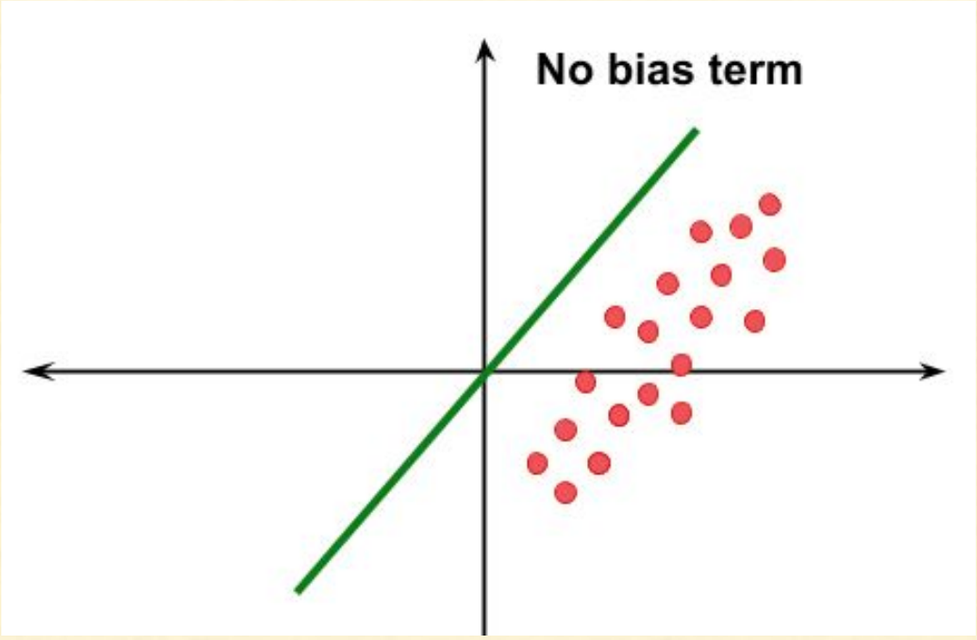
- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the ith feature value
- θ_0 is constant. Also known as bias or intercept term
- θ_j is the jth model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

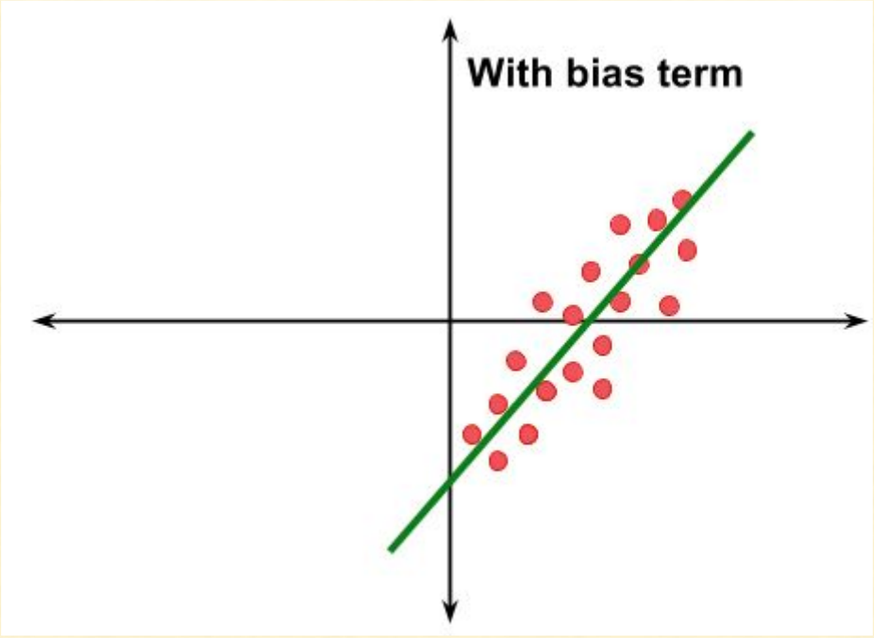
CLOUD x LAB

Linear Regression

Why bias or intercept term?

- Without the bias term our solution may go through the origin
- Let's say if all the features are zero
  - Then predicted value will also be zero
  - However, that may not be the answer the training data suggests

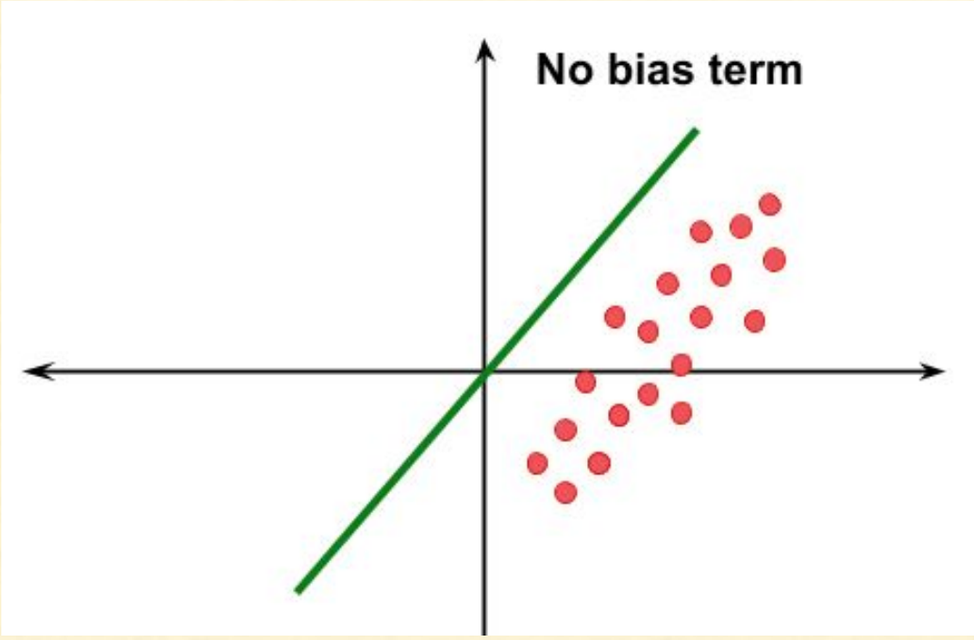


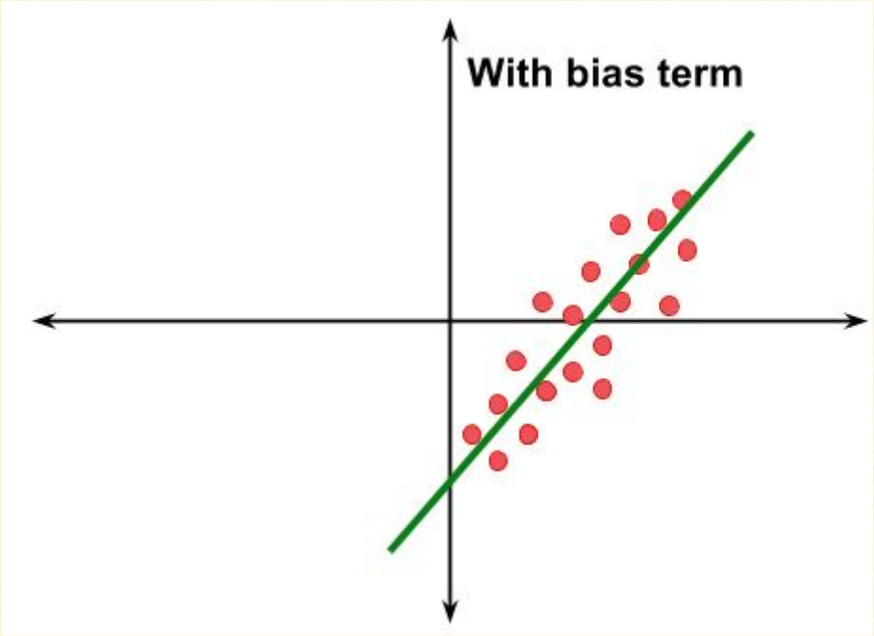


Linear Regression

Why bias or intercept term?

- Bias term is a constant
  - Does not depends on any feature
- Adding a bias weight allows the algorithm to
  - Easily fit data that does not pass through origin





Linear Regression

General Equation of Linear Regression - Vector Form

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

Where

- θ is the model's parameter vector
  - Containing the bias term θ_0 and the feature weights θ_1 to θ_n
- θ^T is the transpose of θ (a row vector instead of a column vector)

$$\theta^T = (\theta_1, \theta_2, \theta_3 \ldots\ldots\ldots\ldots\ldots\ldots \theta_n)$$

CLOUD x LAB

Linear Regression

General Equation of Linear Regression - Vector Form

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

Where

- **x** is the instance's feature vector
  - Containing x_0 to x_n
  - With x_0 always equal to 1

x = (1,
longitude,
Lattitude,

)

CLOUD x LAB

Linear Regression

General Equation of Linear Regression - Vector Form

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

Where

- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and $\mathbf{x}$
- h_θ is the hypothesis function, using the model parameters θ
  - h_θ fits the model and
  - Calculates weights

CLOUD x LAB

Linear Regression

General Equation of Linear Regression - Vector Form

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

- Now we have our linear regression model equation
- **But How do we train it?**

Linear Regression

How Do We Train the Model?

- 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
- As we have seen in end-to-end project
  - Common performance measure of regression model is
    - RMSE - Root Mean Square Error or
    - MSE - Mean Square Error

CLOUD x LAB

Linear Regression

How Do We Train the Model?

- We'll use MSE - Mean Square Error for training the model in this session

Linear Regression

How Do We Train the Model?

- We'll use MSE - Mean Square Error for training the model in this session

Why MSE?

Linear Regression

How Do We Train the Model?

- It is simpler to minimize MSE than RMSE
- Function which minimizes MSE also minimizes its square root so it doesn't matter to use MSE or RMSE

Linear Regression

Let’s have a quick recap of MSE

Linear Regression

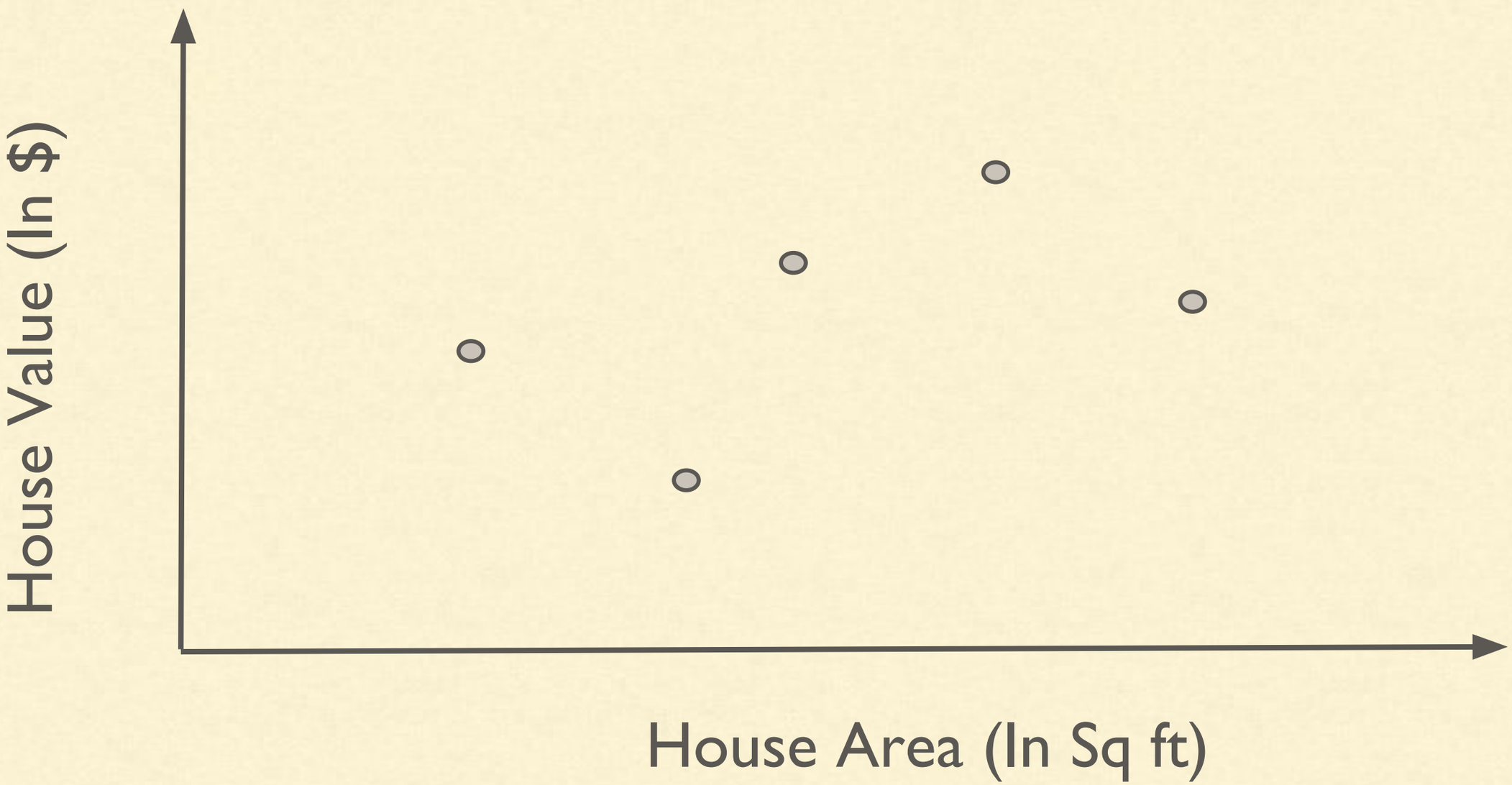
Select a Performance Measure - Mean Square Error

- Let's say we want to predict house value based on house area in square feet
- We will use univariate linear regression
  - Since there is only one feature **house area**

CLOUD x LAB

Linear Regression

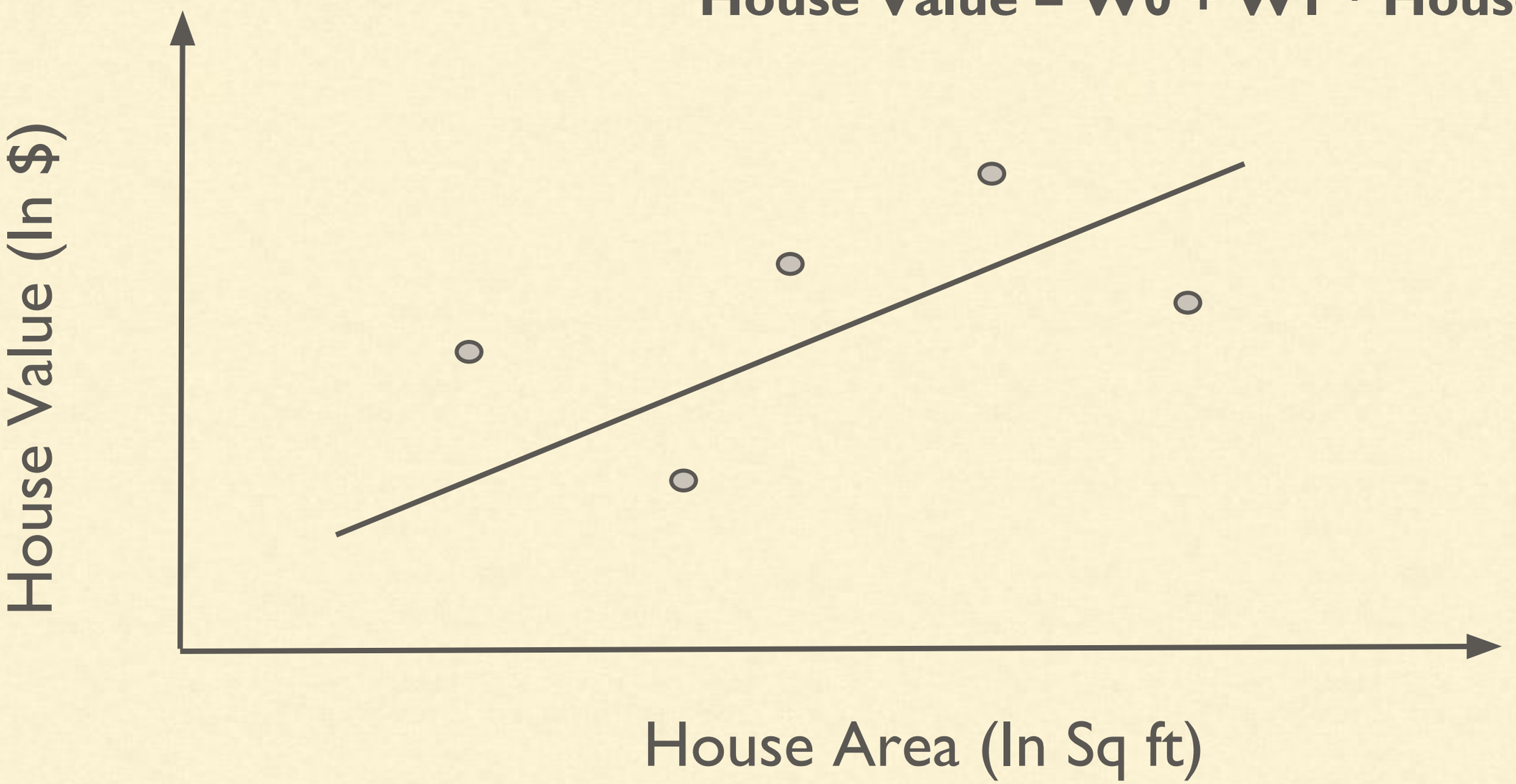
House Value vs House Area - Actual Data



Linear Regression

Fit a Line

House Value = W0 + W1 * House Area

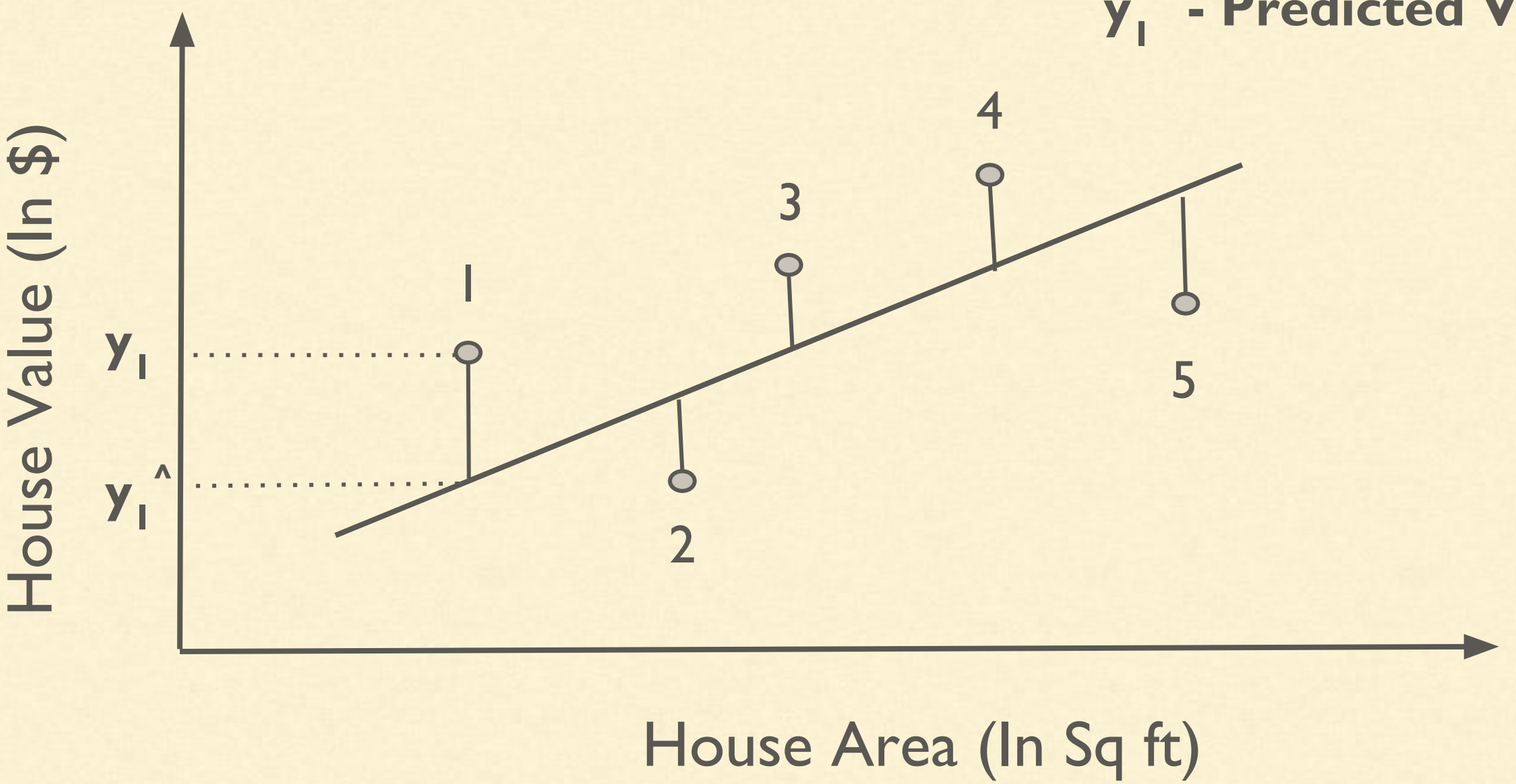


Linear Regression

Fit a Line

y_1 - Actual Value

$\hat{y}_1$ - Predicted Value

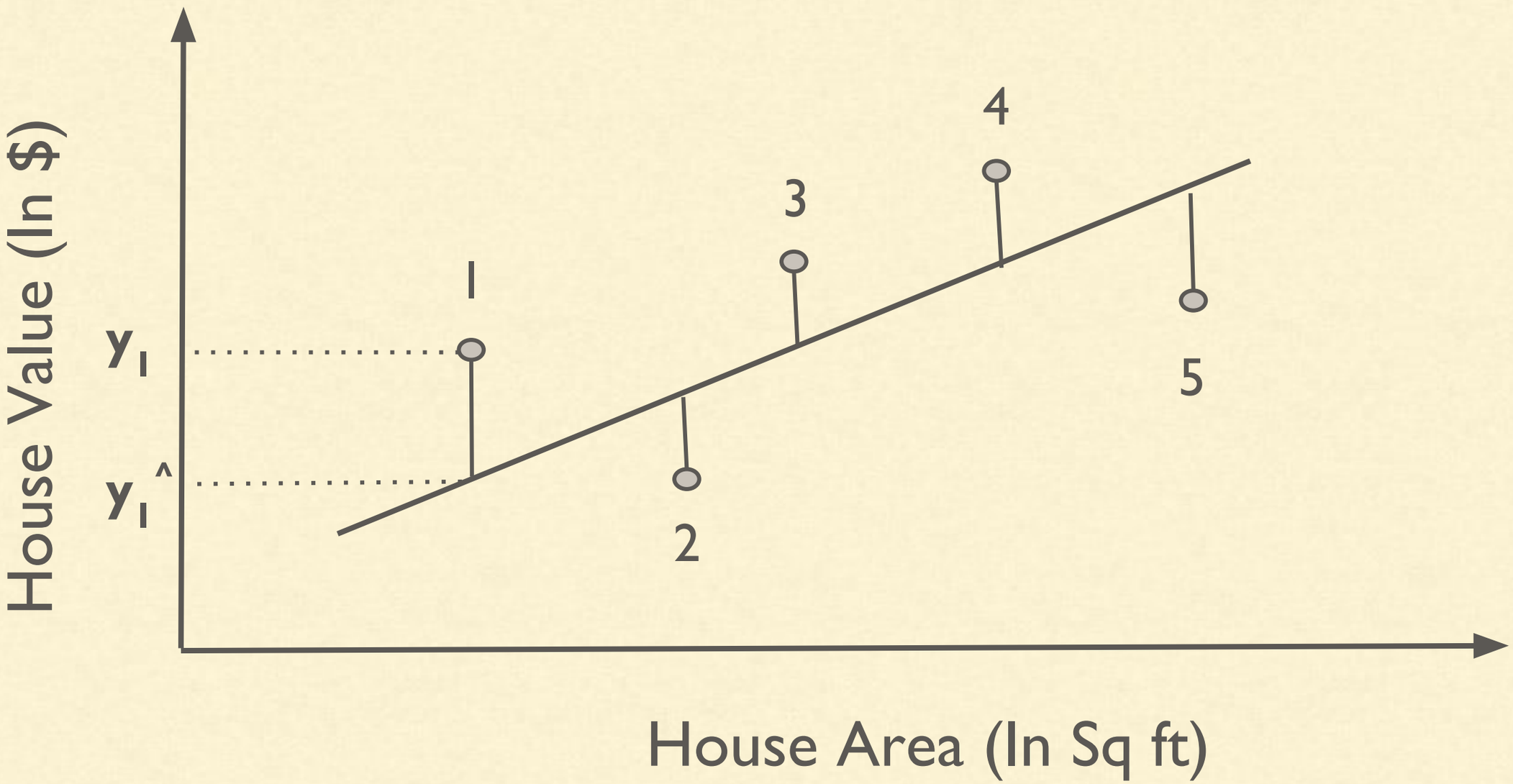


CLOUD x LAB

Linear Regression

Error in Prediction

$$(\hat{y_i} - y_i)$$

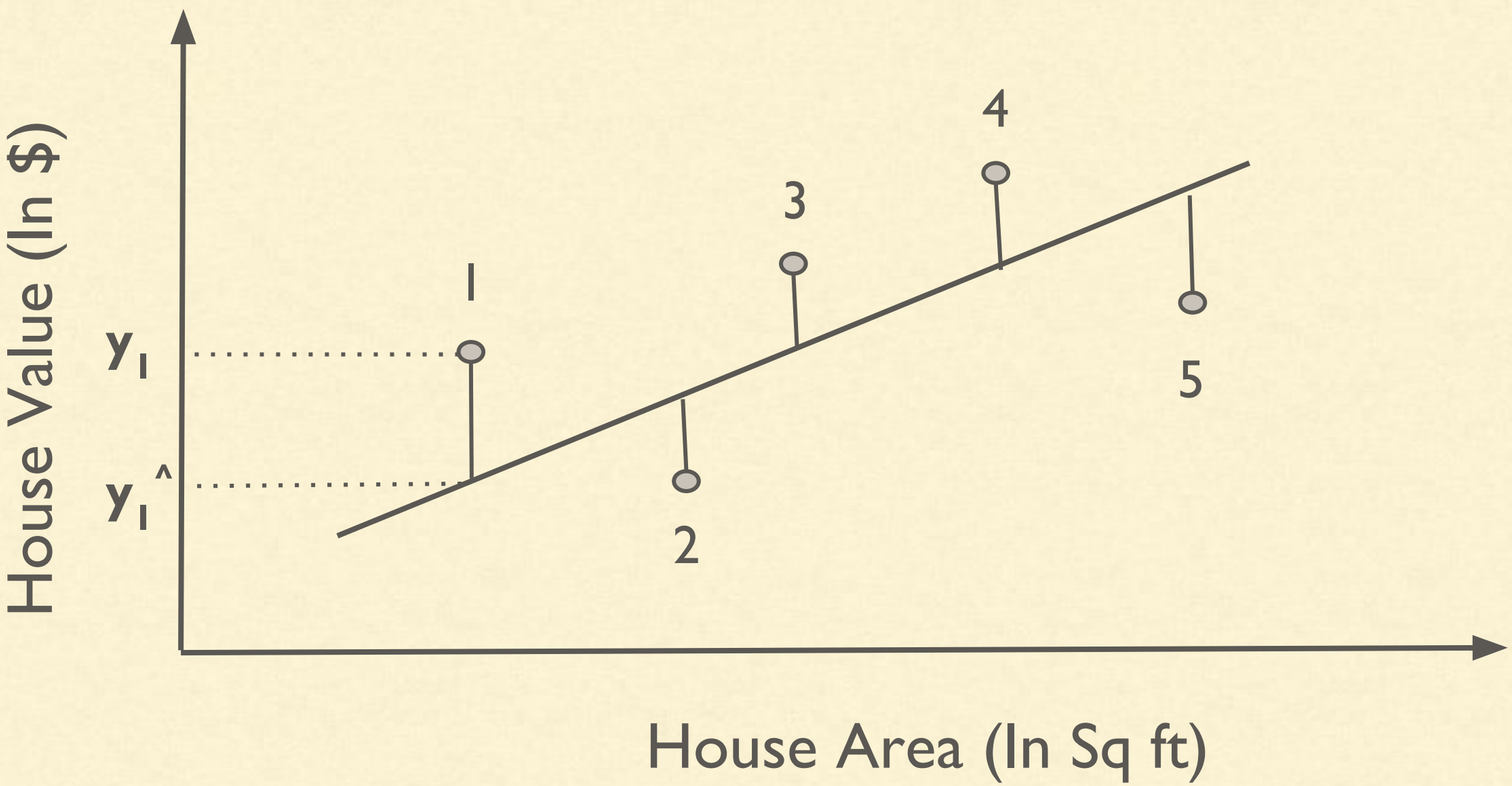


CLOUD x LAB

Linear Regression

Squared Error

$$(\hat{y_i} - y_i)^2$$

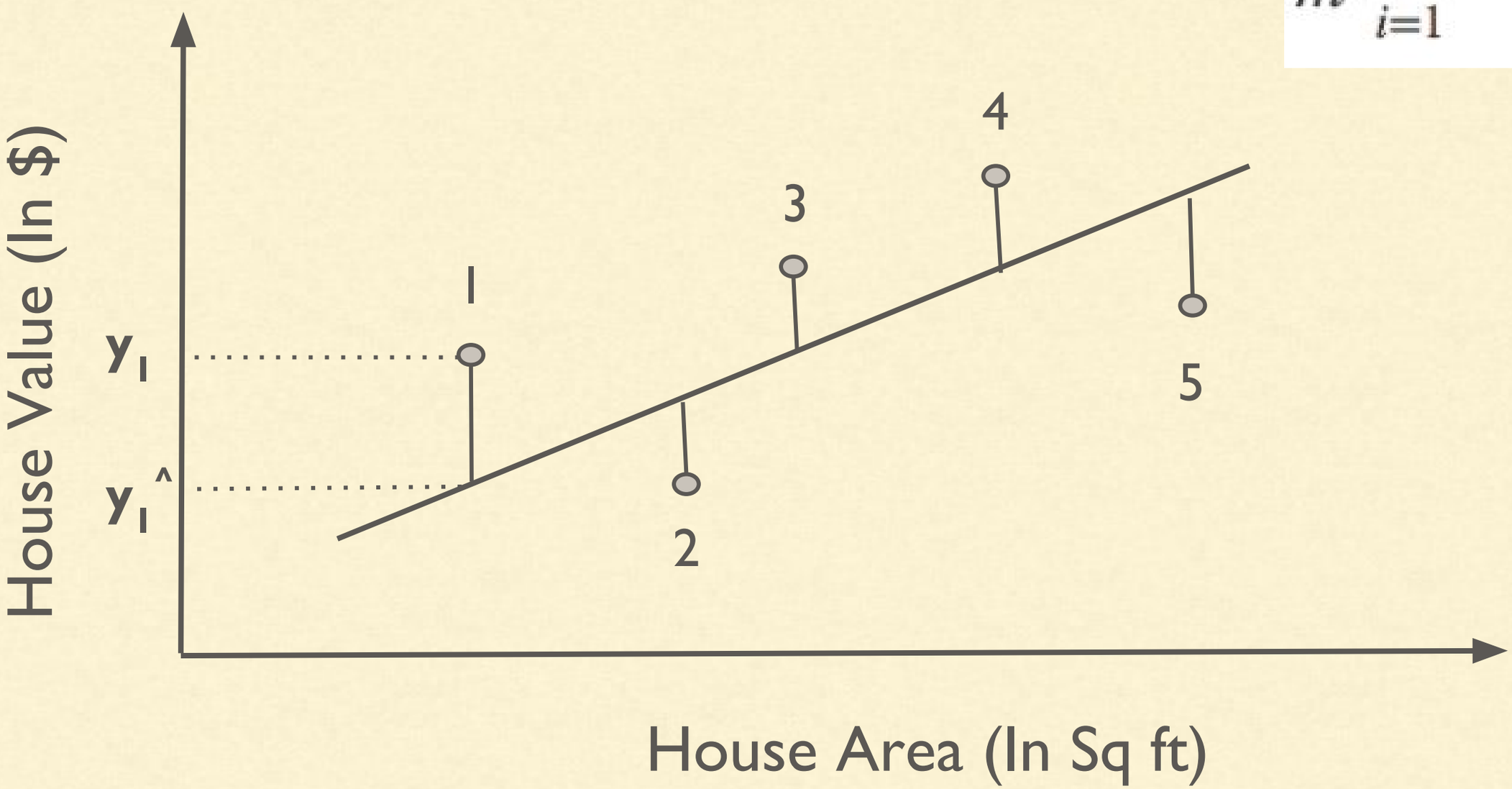


CLOUD x LAB

Linear Regression

Mean Square Error

$$\frac{1}{m}\sum_{i=1}^{m}(\hat{y_i} - y_i)^2$$



CLOUD x LAB

Linear Regression

How Do We Train the Model?

- While training the model
  - We have to find the value of θ (Feature value weights)
  - That minimizes the MSE - Mean Squared Error

Linear Regression

Mean Square Error - MSE

- The MSE of a Linear Regression hypothesis h_θ on a training set $\mathbf{X}$ is calculated using below equation

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

CLOUD x LAB

Linear Regression

Mean Square Error - MSE

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

Where

- $x^{(i)}$ is a vector of all the feature values (excluding the label) of the ith instance in the dataset
- $y^{(i)}$ is its label (the desired output value for that instance)

CLOUD x LAB

Linear Regression

Mean Square Error - MSE

- For example, if the first district in the dataset is located at
  - longitude –118.29°
  - latitude 33.91° and
  - It has 1,416 inhabitants with a
  - Median income of $38,372, and
  - Median house value is $156,400, then

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix} \qquad y^{(1)} = 156,400$$

CLOUD x LAB

Key Thought

While training models we just have to find θ(model parameters or feature weights) which minimizes the RMSE

Linear Regression

Minimise MSE

- Let's understand how the training of linear regression model done internally
- There are two approaches to train the linear regression model
  - The normal "closed-form" equation
  - Gradient Descent (GD) - Iterative optimization approach

The Normal Equation

The Normal Equation

- It is a mathematical equation that gives the result directly
- Hence it is a closed-form solution

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

Value of θ that minimizes the cost function

y is the vector of target values containing $y^{(1)}$ to $y^{(m)}$

More on wikipedia

CLOUD x LAB

The Normal Equation

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

X is training dataset

CLOUD x LAB

The Normal Equation

Let’s test the equation

The Normal Equation

- First, generate some linear looking data

```
>>> import numpy as np

# Uniformly distributed array with dimension (100, 1)
>>> X = 2 * np.random.rand(100, 1)

# Normally distributed array with noise
>>> y = 4 + 3 * X + np.random.randn(100, 1)
```

θ_0 θ_1 Gaussian Noise

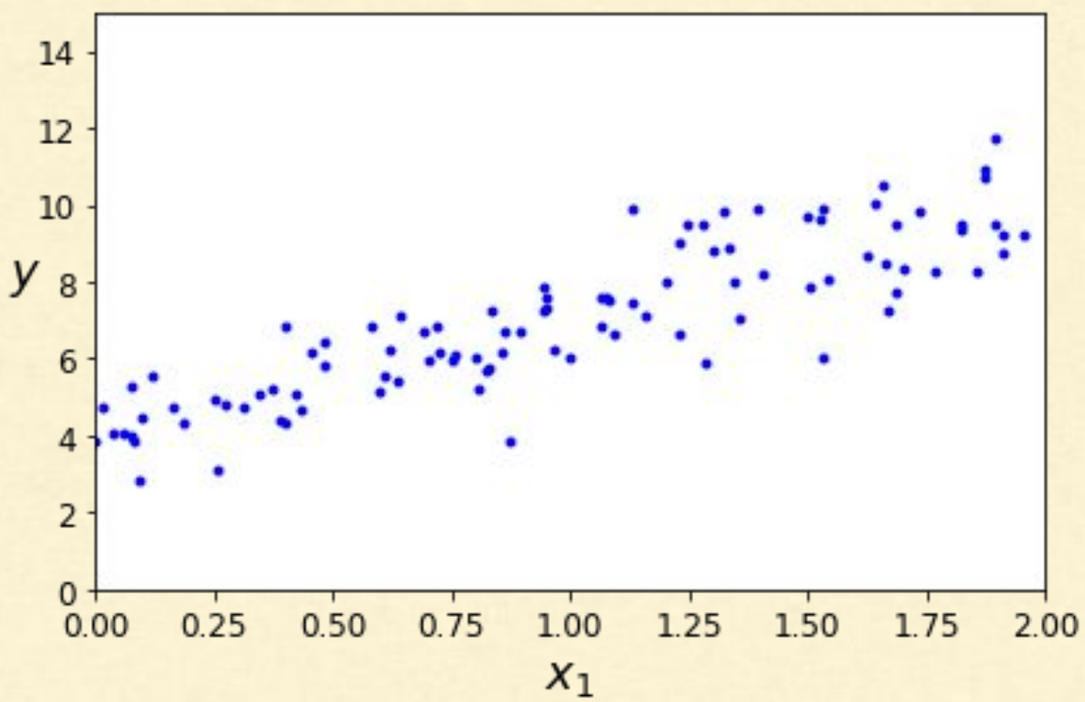
CLOUD x LAB

The Normal Equation

- Let plot the chart to see if X and y have linear relation

```
>>> plt.plot(X, y, "b.")
>>> plt.xlabel("$x_1$", fontsize=18)
>>> plt.ylabel("$y$", rotation=0, fontsize=18)
>>> plt.axis([0, 2, 0, 15])
>>> plt.show()
```

Output



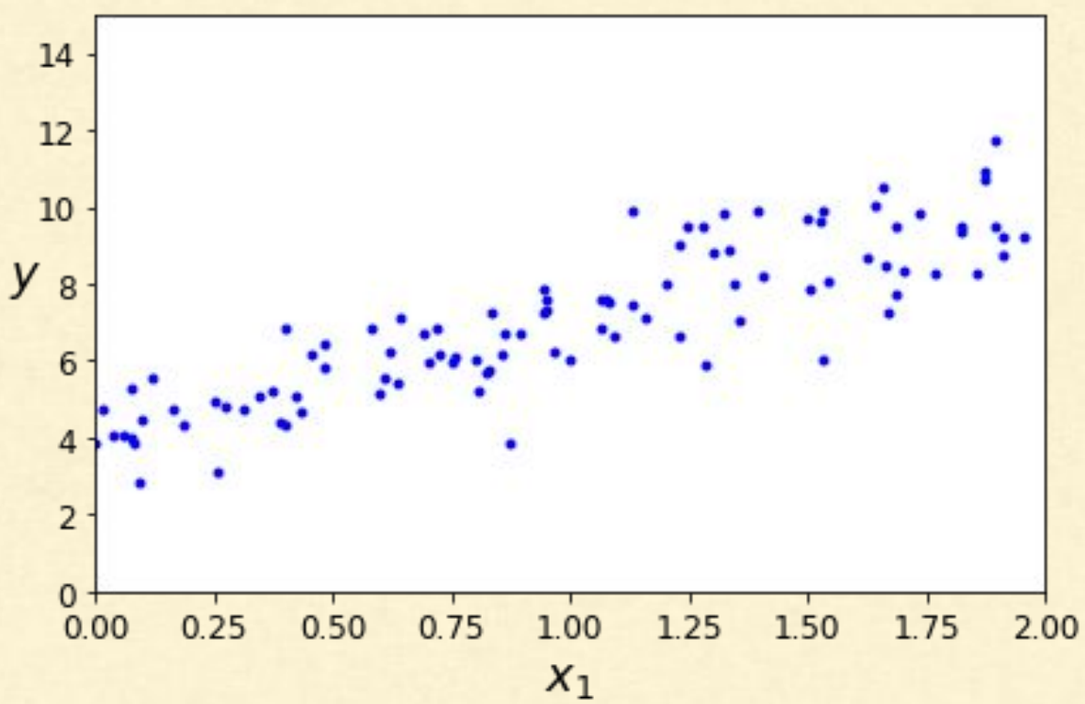
CLOUD x LAB

The Normal Equation

- Let plot the chart to see if X and y have linear relation

```
>>> plt.plot(X, y, "b.")
>>> plt.xlabel("$x_1$", fontsize=18)
>>> plt.ylabel("$y$", rotation=0, fontsize=18)
>>> plt.axis([0, 2, 0, 15])
>>> plt.show()
```

Output



X and y have linear relation

CLOUD x LAB

The Normal Equation

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

- Let's compute theta hat using the Normal Equation

```
# Add Bias term x0 = 1
>>> X_b = np.c_[np.ones((100, 1)), X]

# Apply the normal equation
>>> theta_best =
np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

```
array([[ 4.14766894],
       [ 2.8143552 ]])
```


CLOUD x LAB

The Normal Equation

Weights in Original Equation

```
y = 4 + 3 * X +
np.random.randn(100, 1)
```

Original

$\theta_0 = 4$

$\theta_1 = 3$

Learned Weights from Normal Equation

```
array([[ 4.14766894],
       [ 2.8143552 ]])
```

From Normal Equation

$\theta_0 = 4.14$

$\theta_1 = 2.81$

The Normal Equation

Weights in Original Equation

```
y = 4 + 3 * X +
np.random.randn(100, 1)
```

Learned Weights from Normal Equation

```
array([[ 4.14766894],
       [ 2.8143552 ]])
```

Original

$\theta_0 = 4$

$\theta_1 = 3$

From Normal Equation

$\theta_0 = 4.14$

$\theta_1 = 2.81$

Not bad!

Learned weights are approximately same as original weights

The Normal Equation

- Let's make predictions using theta_best

```
>>> X_new = np.array([[0], [2]])

# add x0 = 1 to each instance
>>> X_new_b = np.c_[np.ones((2, 1)), X_new]

>>> y_predict = X_new_b.dot(theta_best)
```

Output-

```
array([[ 4.21509616],
       [ 9.75532293]])
```

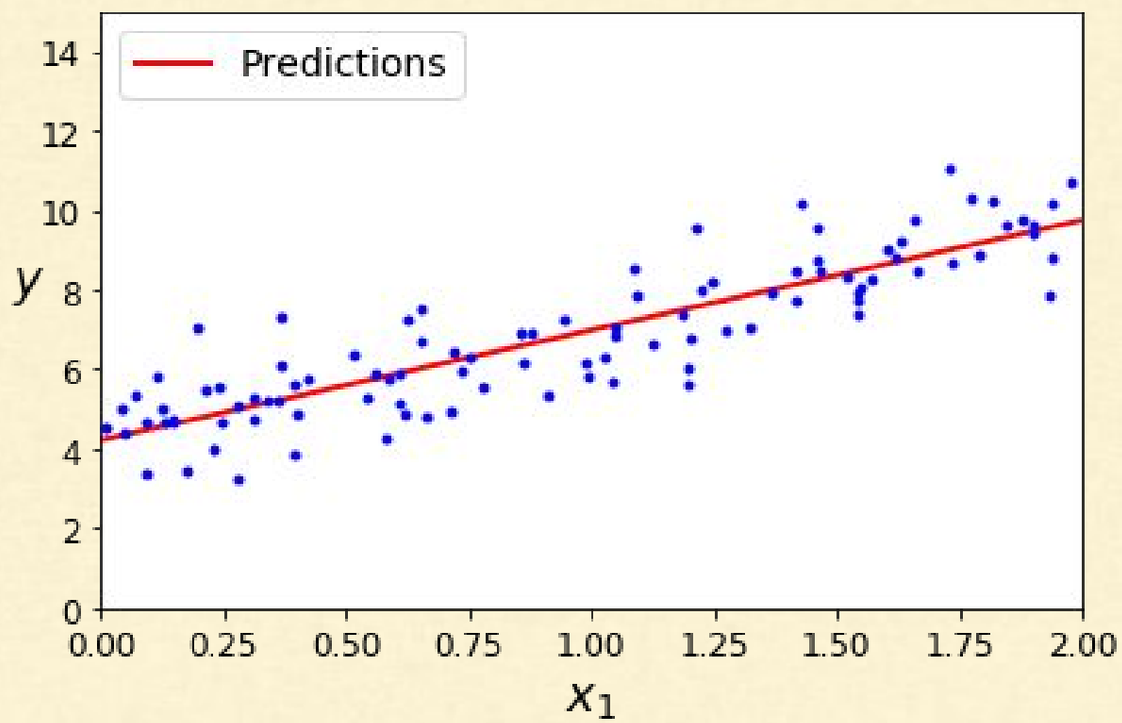

CLOUD x LAB

The Normal Equation

- Let's plot the models predictions

```
>>> plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
>>> plt.plot(X, y, "b.")
>>> plt.xlabel("$x_1$", fontsize=18)
>>> plt.ylabel("$y$", rotation=0, fontsize=18)
>>> plt.legend(loc="upper left", fontsize=14)
>>> plt.axis([0, 2, 0, 15])
>>> plt.show()
```

Output-



CLOUD x LAB

The Normal Equation

- We can also use Scikit-Learn for the previous steps

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
```

Output-

```
(array([ 4.21509616]), array([[ 2.77011339]]))
```


CLOUD x LAB

The Normal Equation

```
>>> lin_reg.predict(X_new)
```

Output-

```
array([[ 4.21509616],
       [ 9.75532293]])
```


CLOUD x LAB

The Normal Equation

Computational Complexity - For Features

- The Normal Equation
  - Computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$
  - Which is an n * n matrix (where n is the number of features)
- Depending on the implementation
  - The computational complexity of inverting such a matrix is
  - Typically about $O(n^{2.4})$ to $O(n^3)$ depending on the implementation

CLOUD x LAB

The Normal Equation

Computational Complexity - For Features

- If we double the number of features
  - Then the computation time gets increased by
  - $2^{2.4}$ to 2^3 = (~= 5.3 to 8) times
- That is why the Normal Equation gets very slow
  - When the number of features grows large

CLOUD x LAB

The Normal Equation

Computational Complexity - For Training Set

- The normal equation is
  - Linear with regards to the number of instances in the training set
  - So it handles large training sets efficiently
  - Provided they can fit in memory

CLOUD x LAB

The Normal Equation

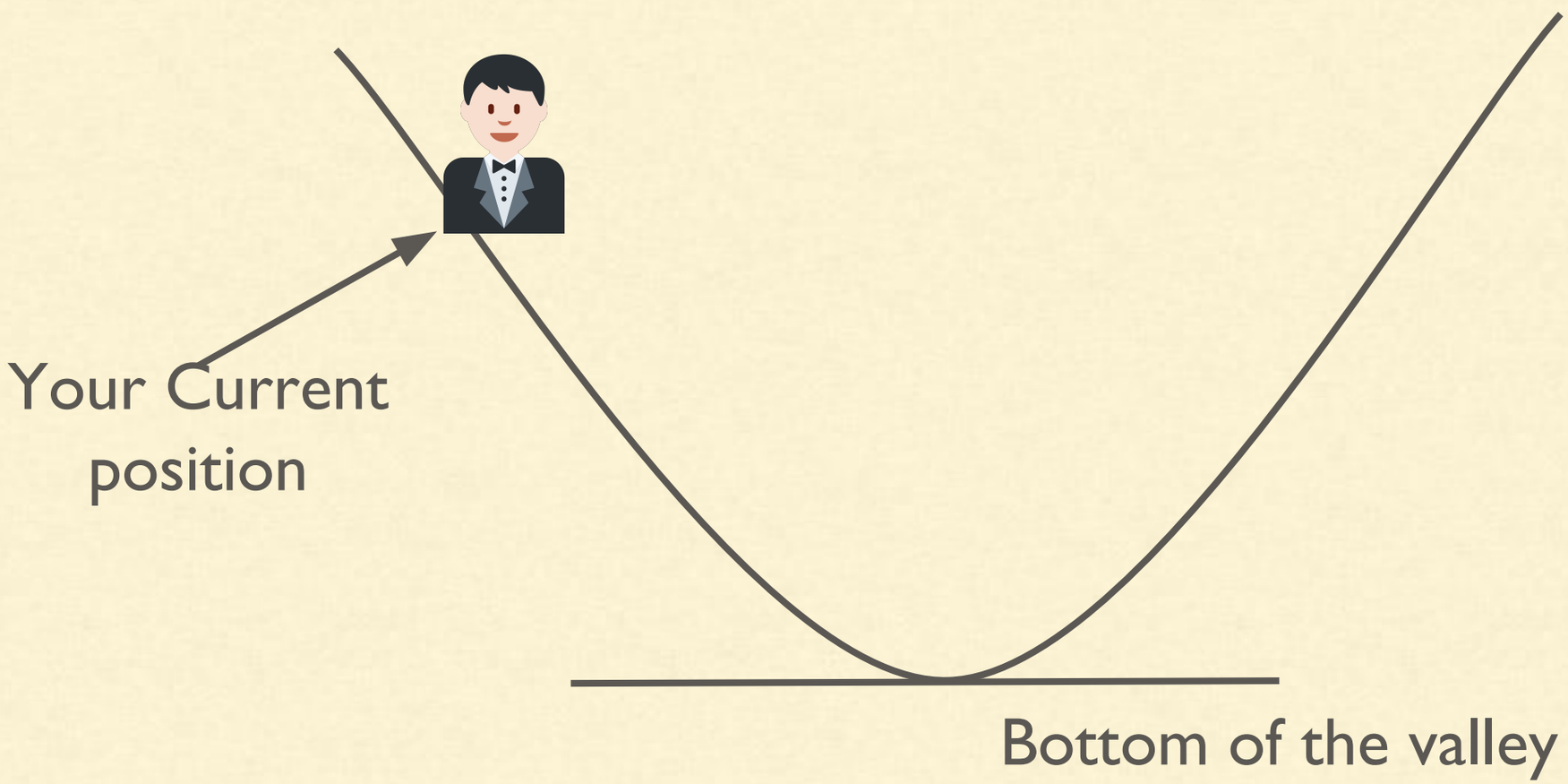
Computational Complexity

- Let's explore different ways to train a Linear Regression model which are
  - Better suited for cases where there are a large number of features
  - Or too many training instances to fit in memory

Gradient Descent

Gradient Descent

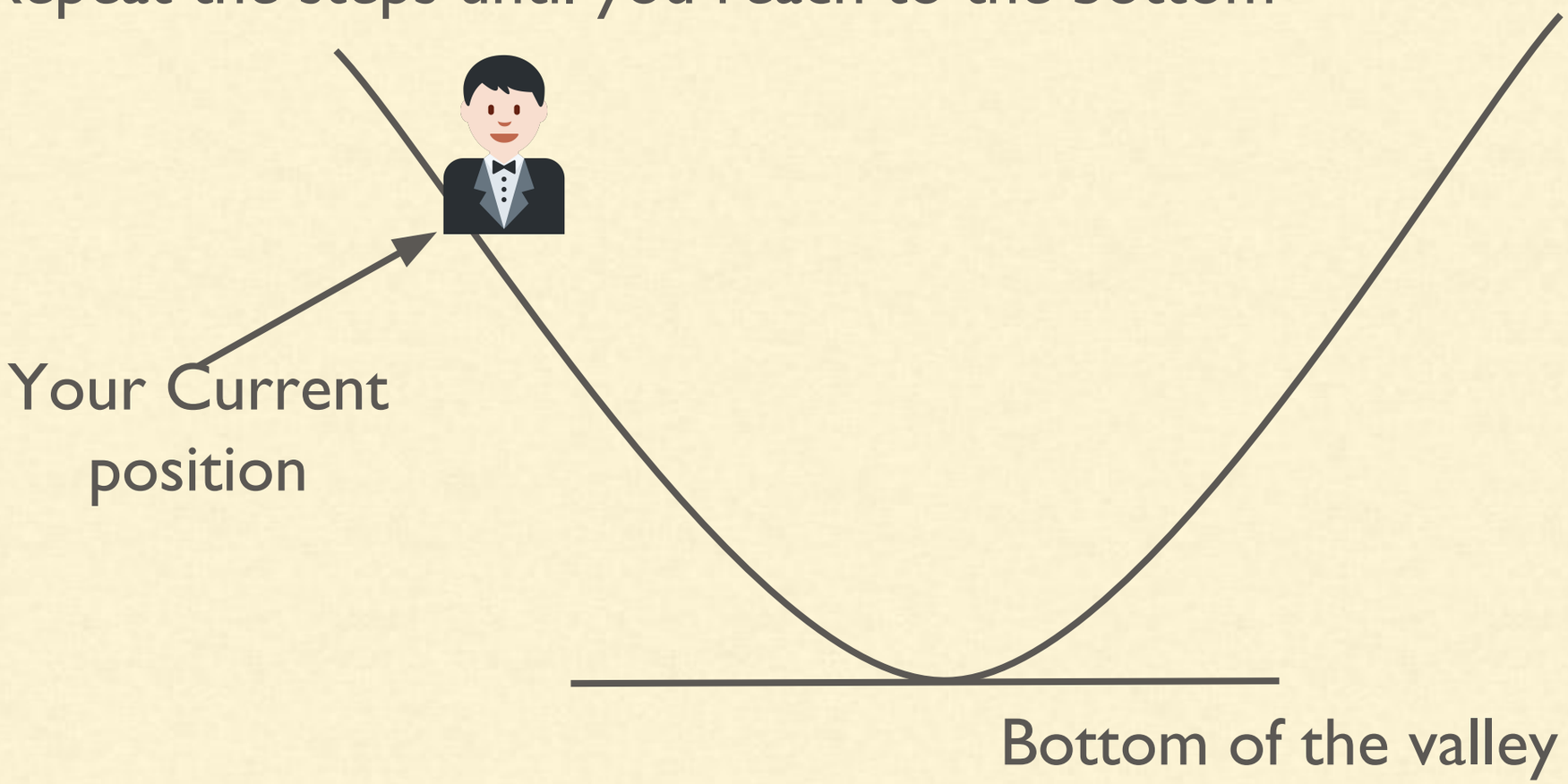
Question - If you are lost in the mountains in a dense fog and you can only feel the slope of the ground below your feet. Which strategy will you take to reach to the bottom of the valley quickly?



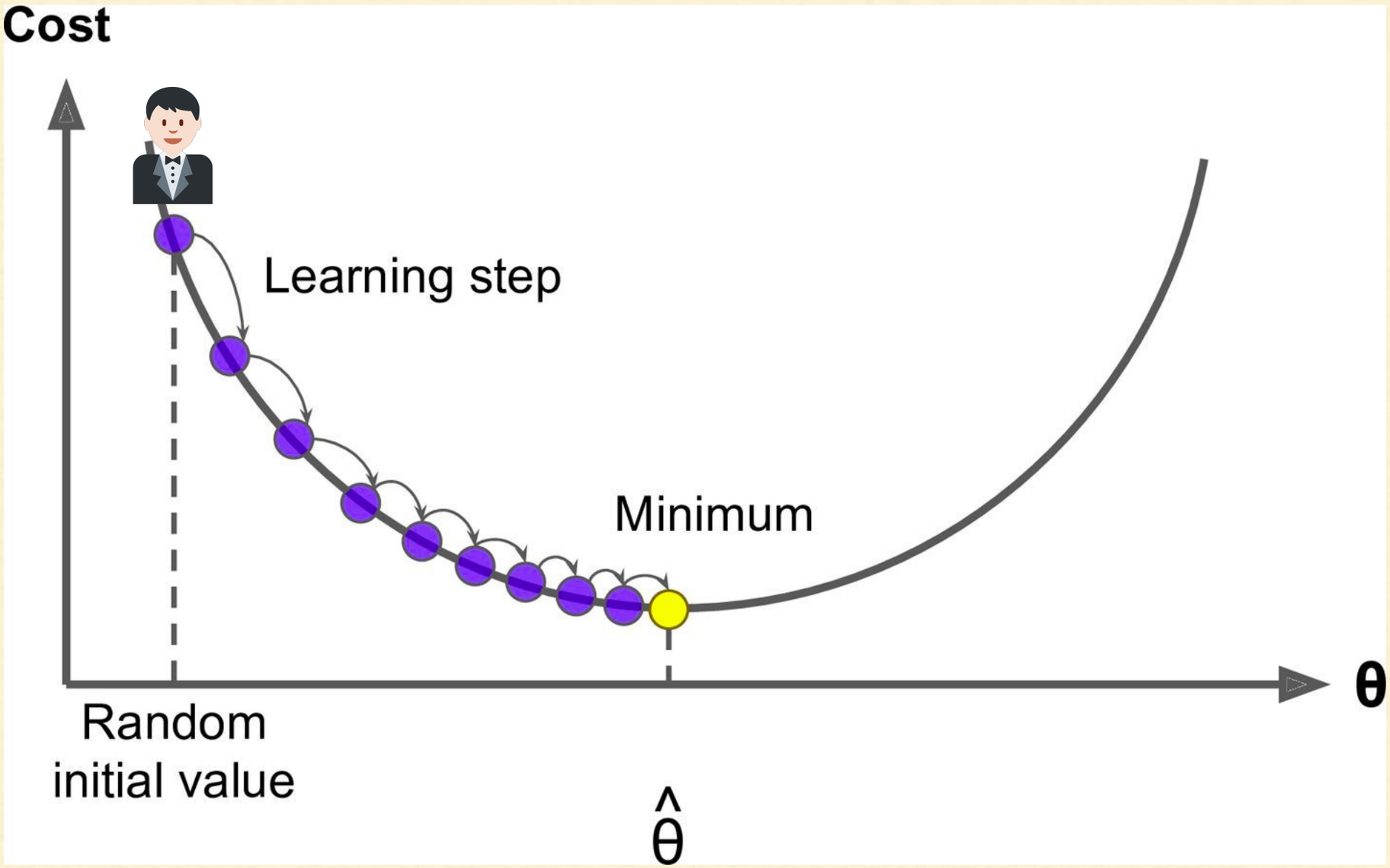
Gradient Descent

Answer

1. Take a step
2. If the step is downhill, keep walking on the same direction
3. Else reverse the direction
4. Repeat the steps until you reach to the bottom

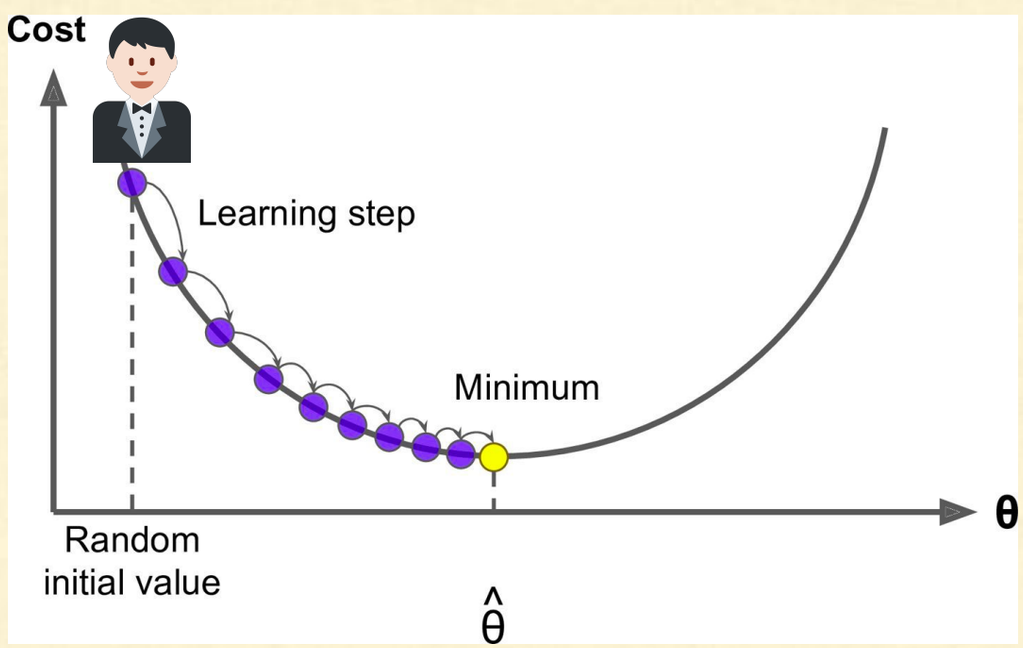


Gradient Descent



Gradient Descent

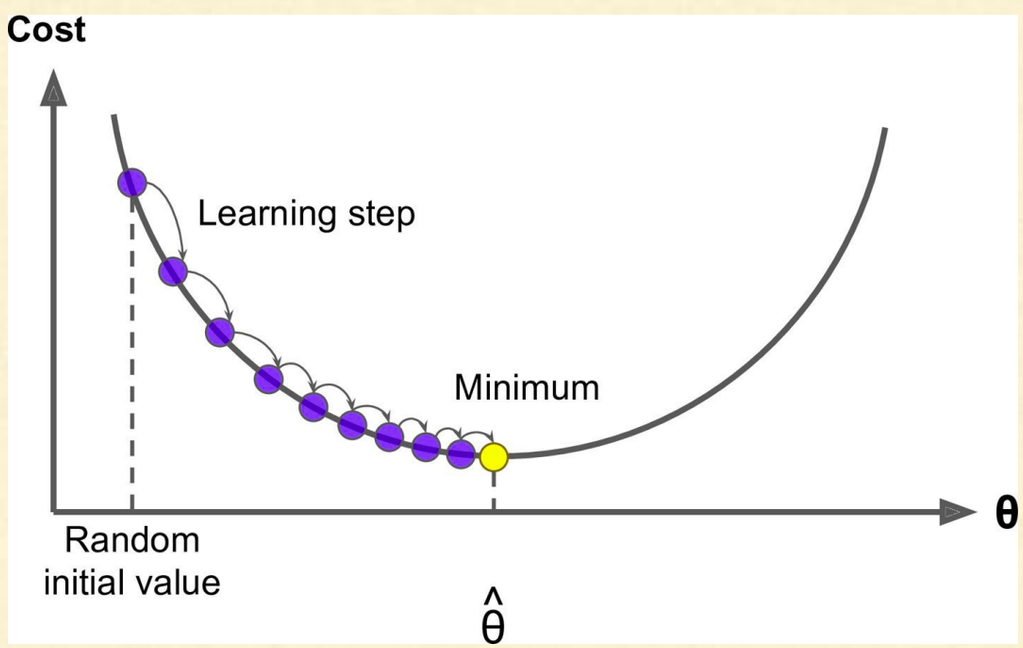
- Gradient Descent
  - Measures the local gradient of the cost function
  - With regards to the parameter vector θ and
  - It goes in the direction of descending gradient
  - Until the gradient becomes zero



Gradient Descent

Random Initialization

- We start by filling θ with random values - Random initialization
- Improve it gradually, taking one baby step at a time
- Each step attempts to decrease the cost function - MSE
- Until the algorithm converges to a minimum



CLOUD x LAB

Gradient Descent

Learning Rate

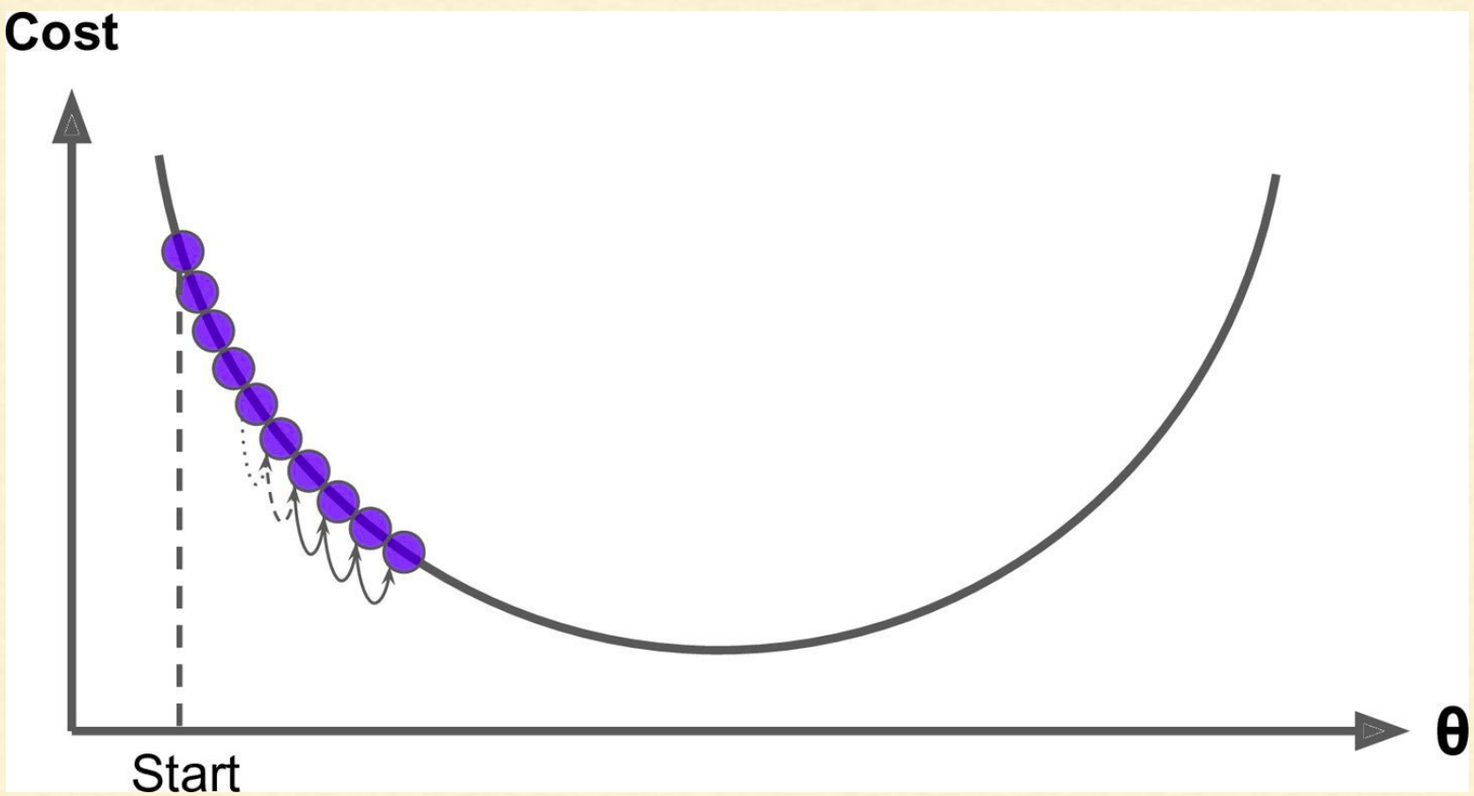
- In Gradient Descent
  - The size of the steps - also called as **learning rate** plays an important role

CLOUD x LAB

Gradient Descent

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 to converge

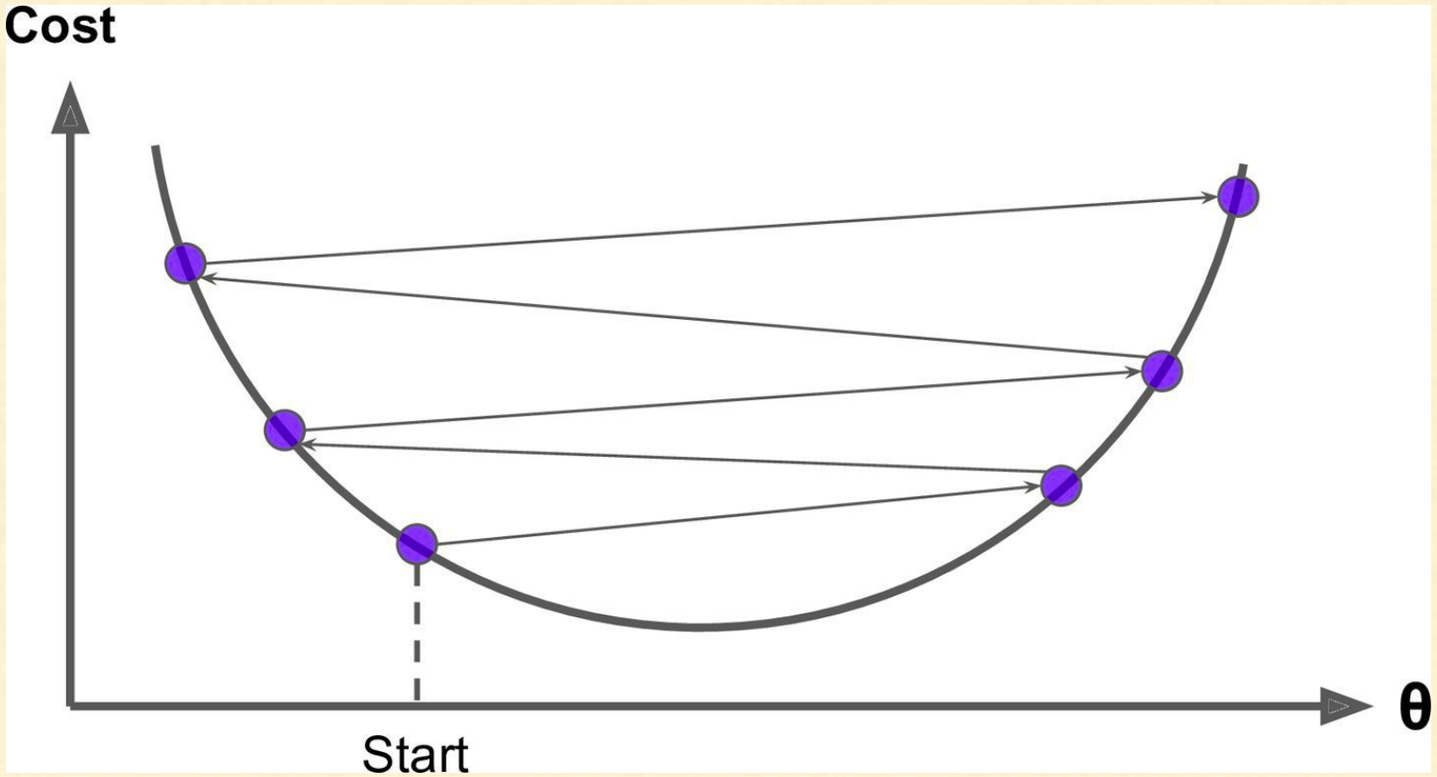


CLOUD x LAB

Gradient Descent

Learning Rate

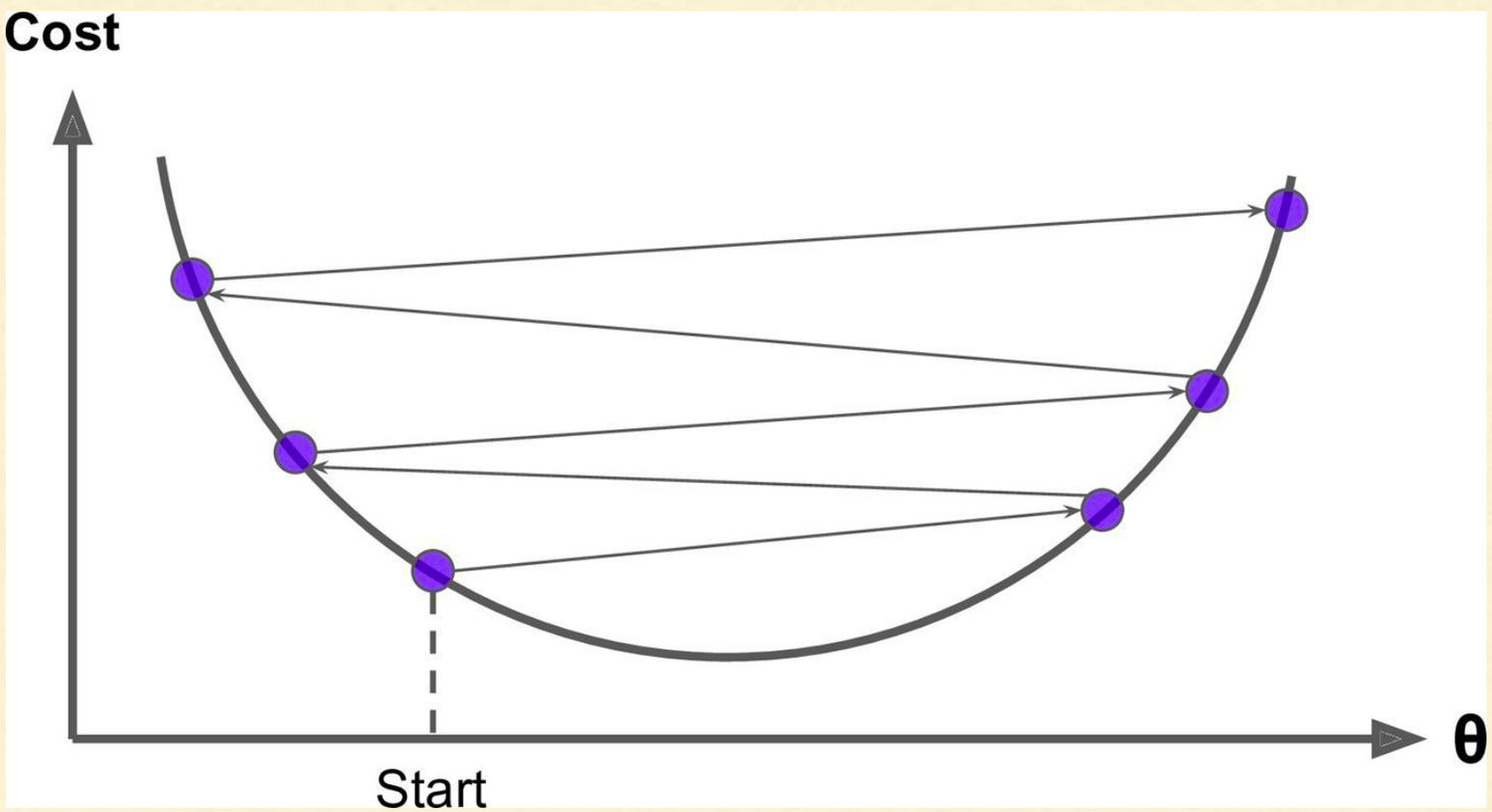
- If the learning rate is too high
  - We might jump across the valley
  - and end up on the other side, possibly even higher up than you were before



Gradient Descent

Learning Rate

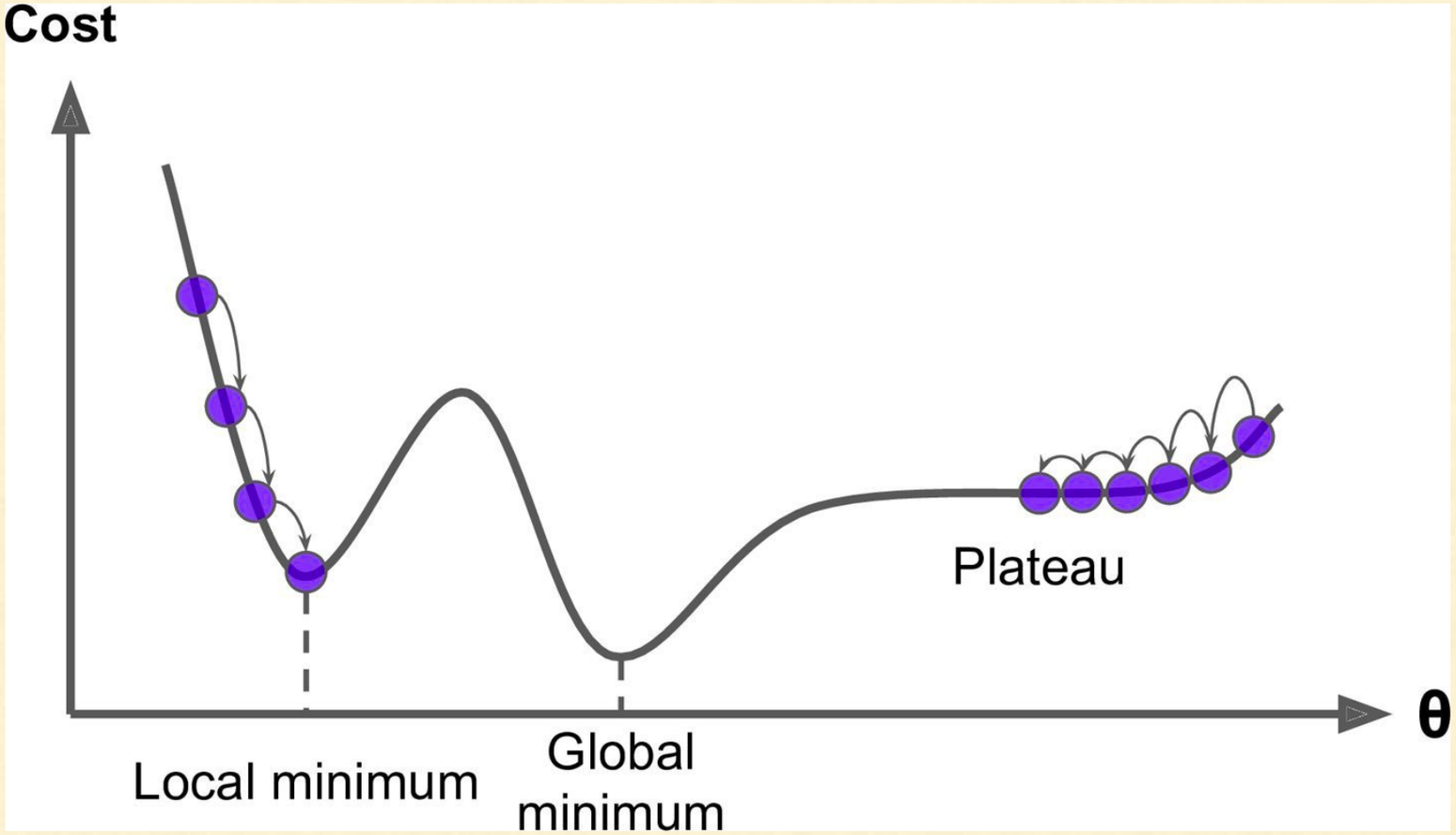
- If the learning rate is too high
  - This might make the algorithm diverge with
  - Larger and larger values, failing to find a good solution



Gradient Descent

Global and Local Minimum

- Not all cost functions are regular bowls
- There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult

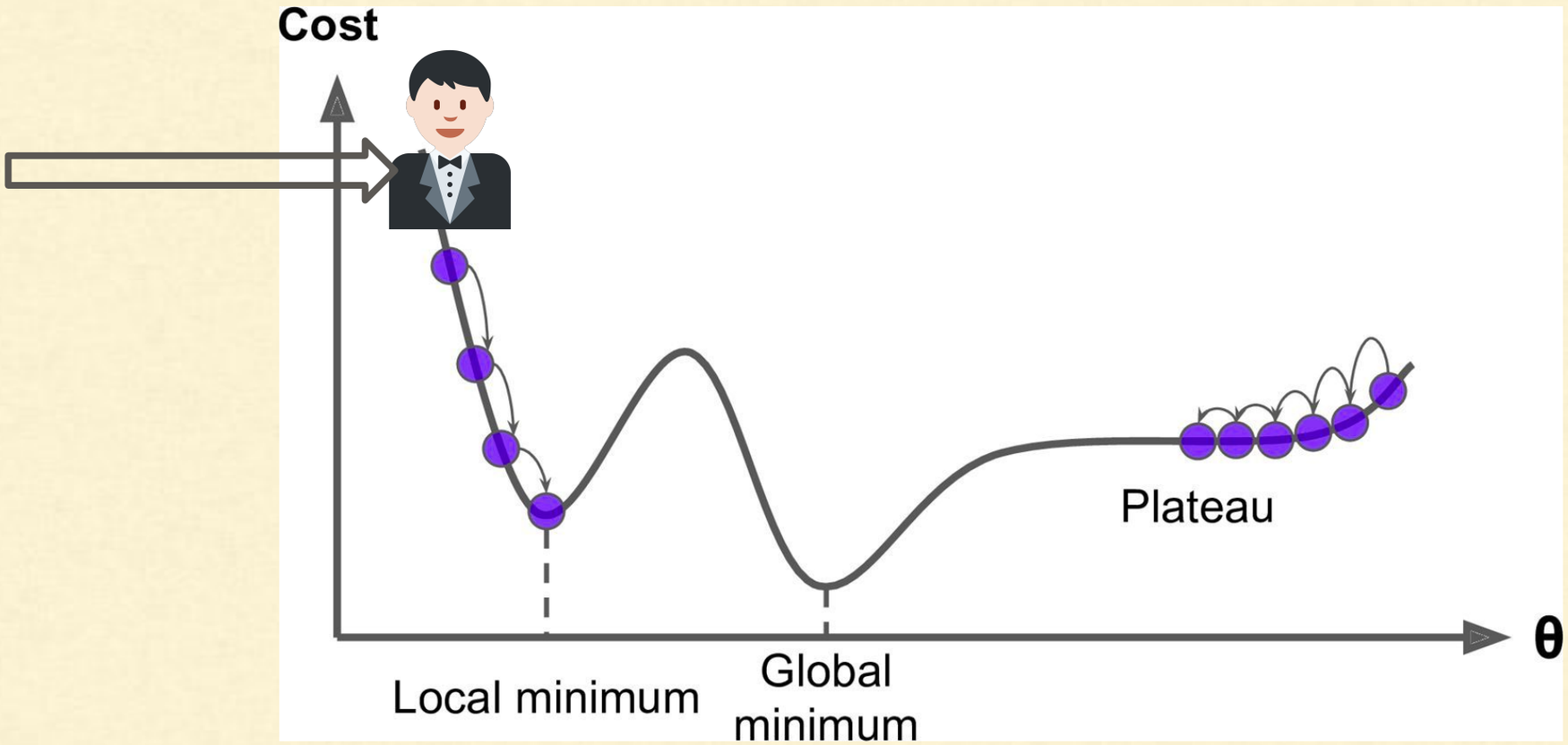


CLOUD x LAB

Gradient Descent

Global and Local Minimum

- If we start from left then Gradient Descent
  - Converges to a local minimum
  - Which is not as good as the global minimum

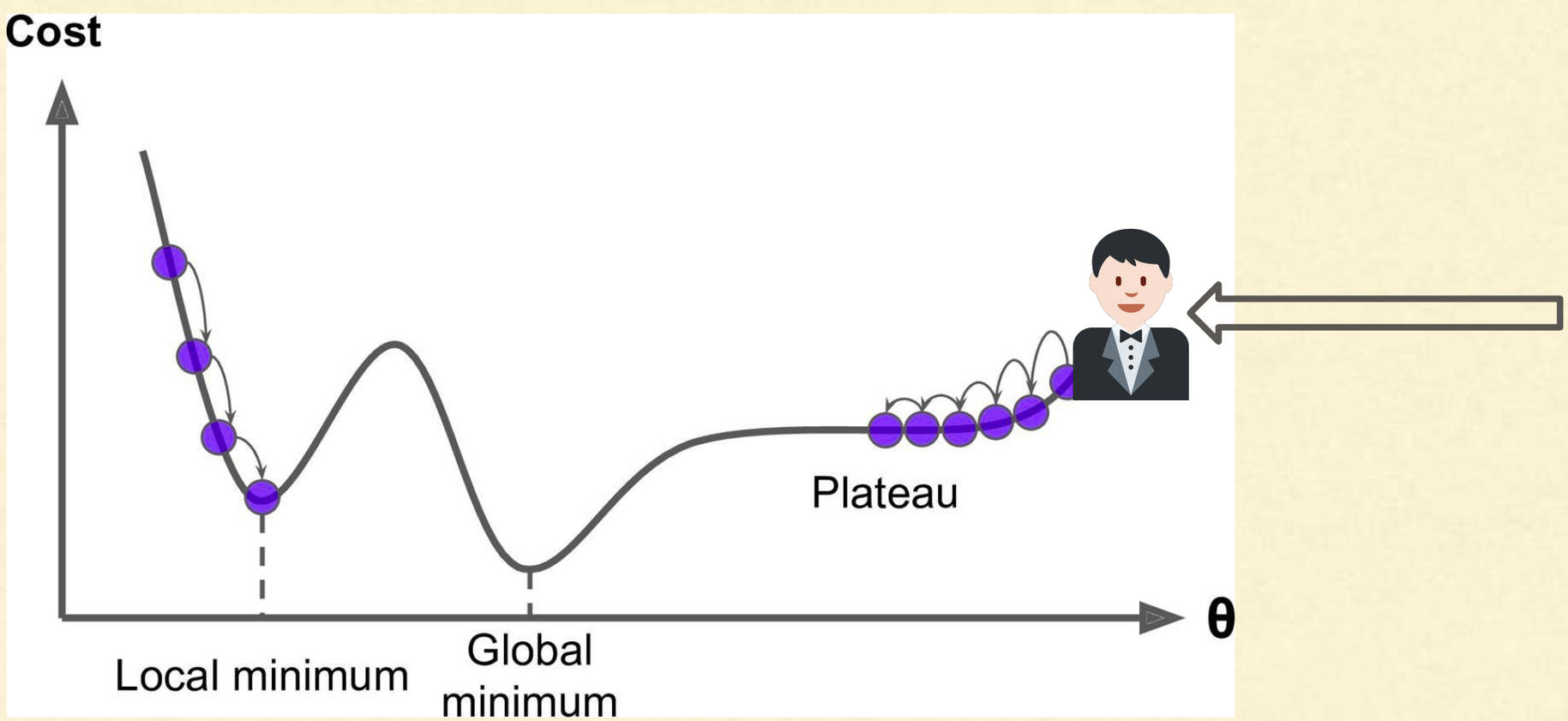


CLOUD x LAB

Gradient Descent

Global and Local Minimum

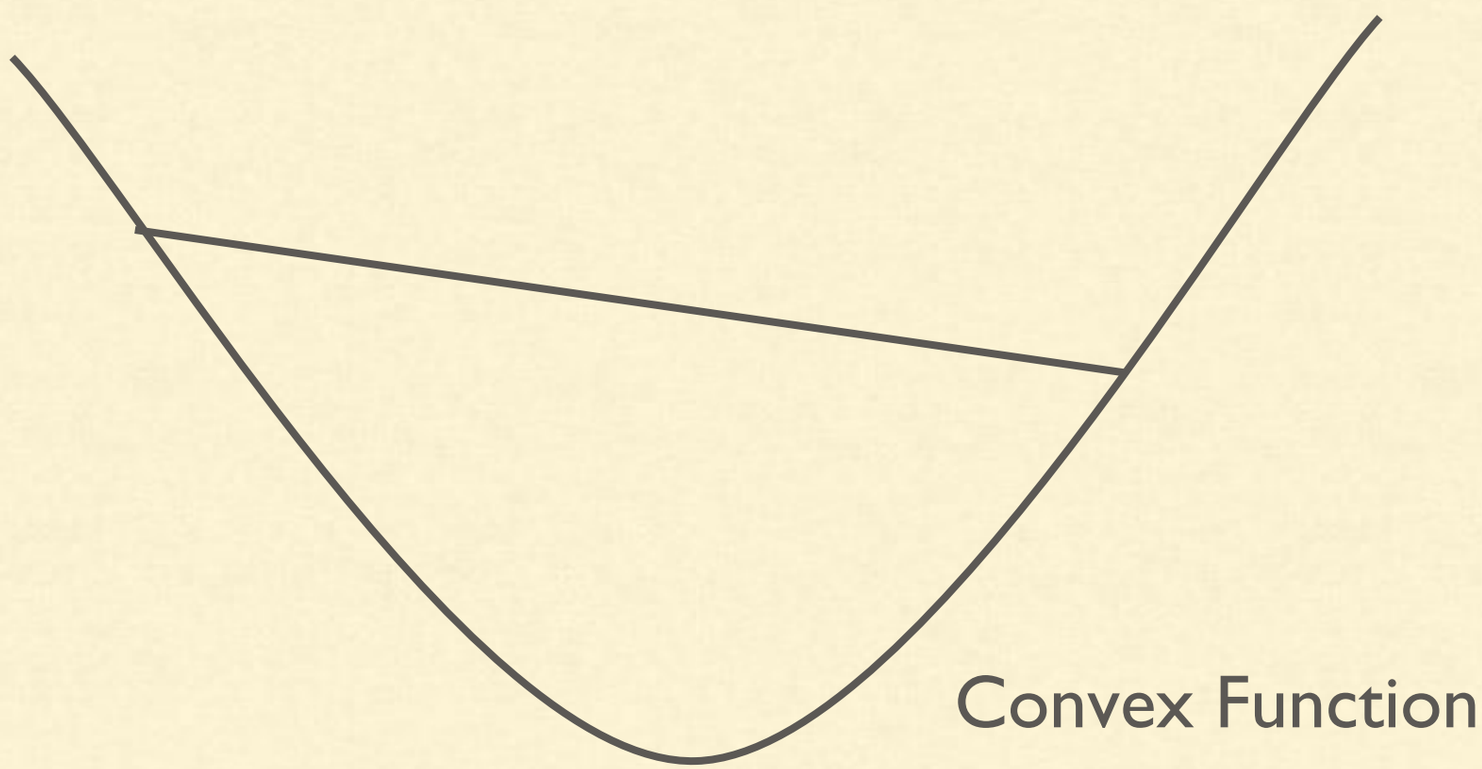
- If we start from right then Gradient Descent
  - Will take a very long time to cross the plateau and
  - If we stop too early then
  - We will never reach the global minimum



Gradient Descent

Global and Local Minimum

- MSE cost function is a convex function
  - If we pick any two points on the curve
  - The line segment joining them never crosses the curve
  - No local minima, just one global minimum



CLOUD x LAB

Gradient Descent

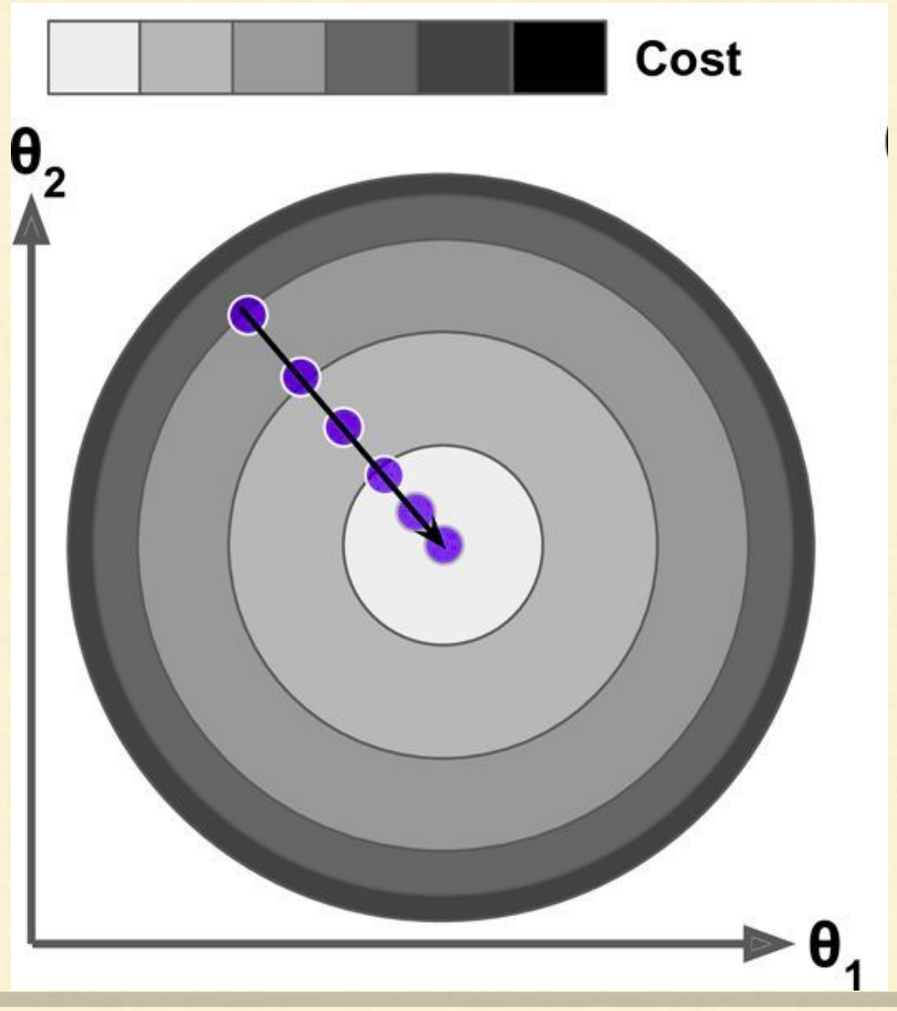
Feature Scaling

- When using Gradient Descent, we should ensure that
  - All features have a similar scale
  - Using Scikit-Learn's **StandardScaler** class
  - Else it will take much longer to converge

Gradient Descent

Feature Scaling - Same Scales

- See the below diagram where feature 1 and 2 have same scales
- Gradient Descent algorithm goes straight toward the minimum
  - Thereby reaching it quickly



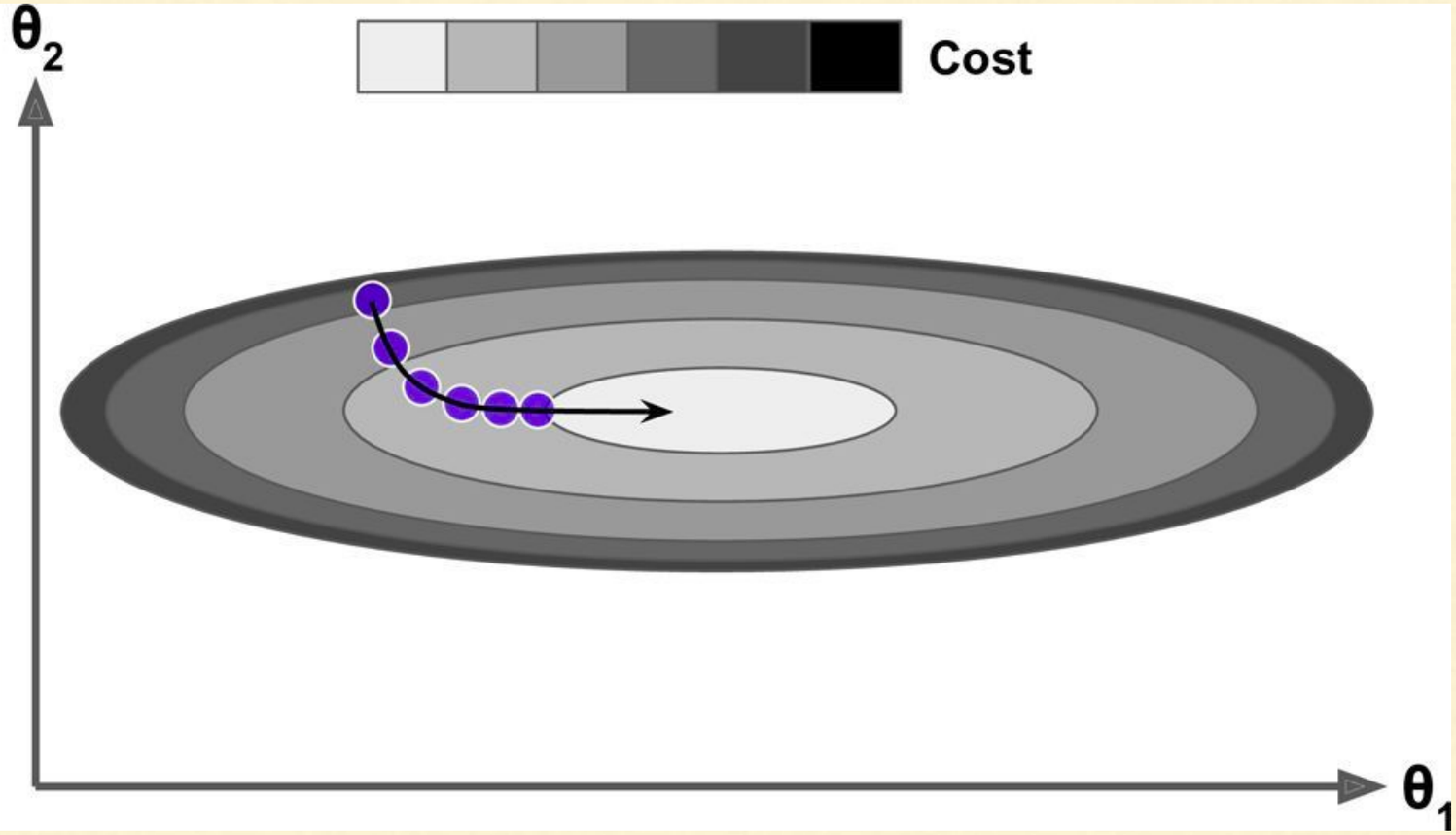
Same Scale

CLOUD x LAB

Gradient Descent

Feature Scaling - Different Scales / Not scaled

- See the below diagram where feature 1 and 2 have different scales



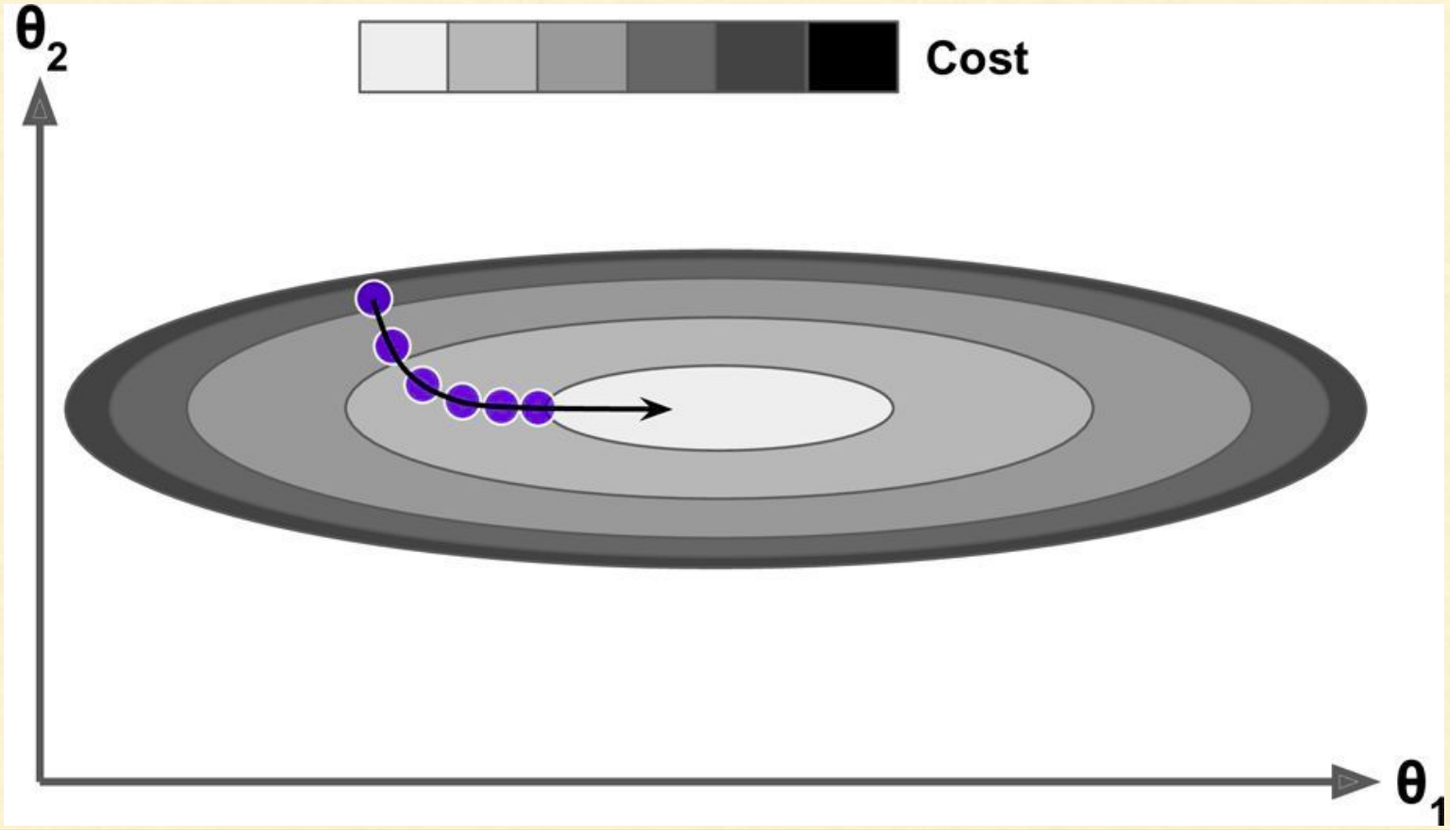
Different Scale

CLOUD x LAB

Gradient Descent

Feature Scaling - Different Scales

- Gradient Descent algorithm goes almost orthogonal to
  - The direction of the global minimum and
  - It ends with a long march down an almost flat valley
  - It will eventually reach the minimum, but it will take a long time

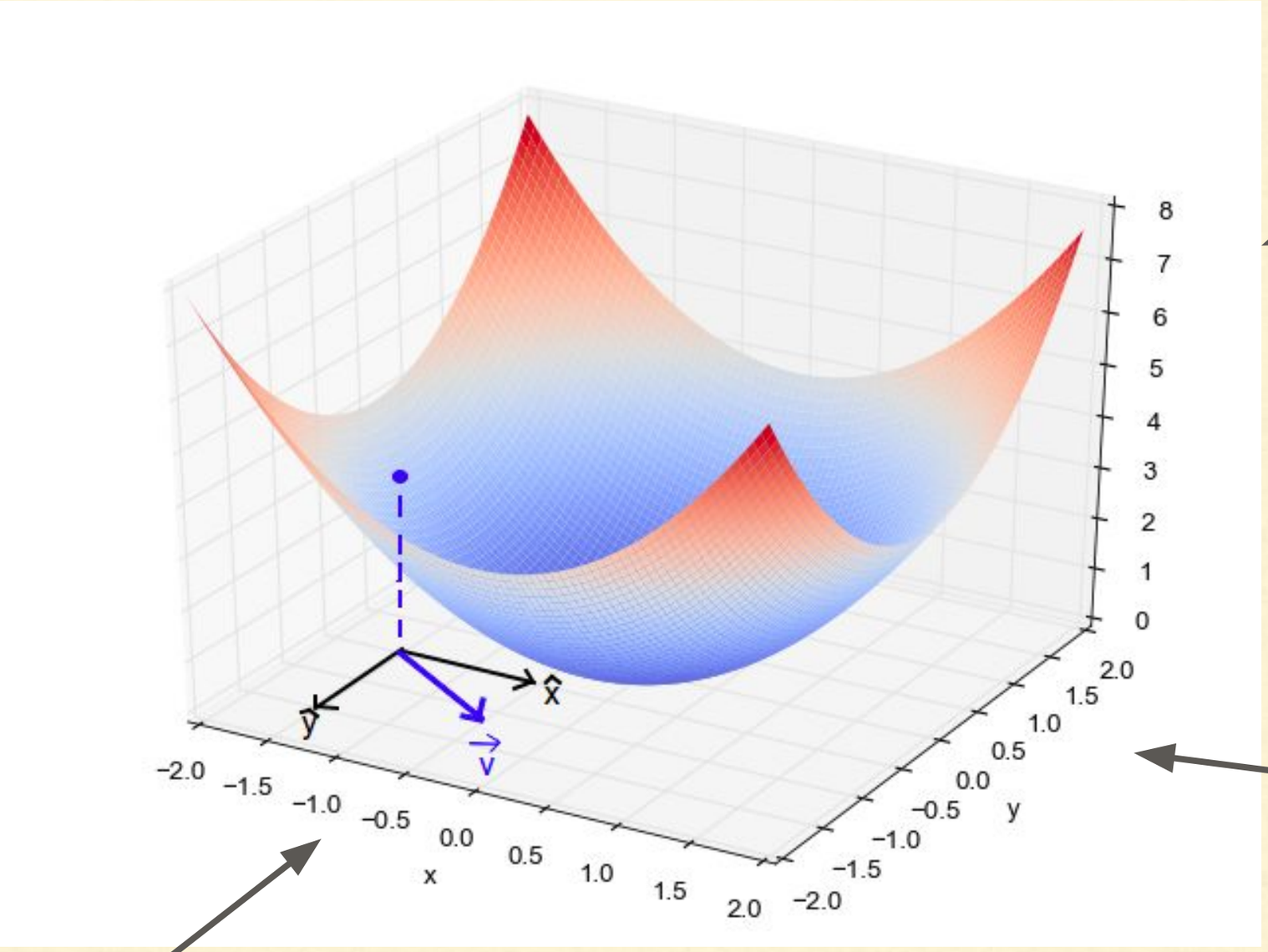


Different Scale

CLOUD x LAB

Gradient Descent

Features and Cost Function



Cost

Feature 2

Feature 1

CLOUD x LAB

Gradient Descent

- Again remember that training a model means
  - Searching for a best possible model's parameters
  - Like θ_0, θ_1 and others
  - That minimizes the cost function like MSE
- Algorithm will have to search in more dimensions if
  - Model has more features
  - And search becomes complex

CLOUD x LAB

Gradient Descent

- Let's implement Gradient descent now using some known algorithms such as
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-batch Gradient Descent

CLOUD x LAB

Batch Gradient Descent

Batch Gradient Descent

Partial Derivative

- To implement Gradient Descent
  - First initialize model parameters(θ_1, θ_2 …..) - Random Initialization
  - For each of the weights, we change one weight and keeping others fixed and calculate the change in cost
  - This is called a **Partial Derivative**

CLOUD x LAB

Batch Gradient Descent

Partial Derivative

- It is like asking
  - What is the slope of the mountain under my feet if I face east?
  - And then asking the same question facing north
  - And so on for all other dimensions

Batch Gradient Descent

Partial Derivative of Cost Function

- In general, we compute the gradient of the cost function
  - With regards to each model parameter θj (Theta j)
- Below is the equation for finding partial derivative of the cost function with respect to parameter θj (Theta j)

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

CLOUD x LAB

Batch Gradient Descent

Partial Derivative of Cost Function

- In this equation
  - We calculate partial derivative of each parameter θj (Theta j) individually
  - Instead we can compute them all in one go

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

CLOUD x LAB

Batch Gradient Descent

Gradient Vector of the Cost Function

- The **gradient vector ∇_θMSE(θ)**
  - Contains all the partial derivatives of the cost function
  - (one for each model parameter)

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

CLOUD x LAB

Batch Gradient Descent

Gradient Vector of the Cost Function

- This formula involves calculations over the full training set **X** at every step
- This is why the name **Batch Gradient Descent**

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

CLOUD x LAB

Batch Gradient Descent

Gradient Vector of the Cost Function

- Since Batch Gradient Descent uses entire training data at every step
  - It is terribly slow on large training sets
- But Gradient Descent scales well with the number of features
- It is faster to train Linear Regression model when there are thousand of features
  - Using Gradient Descent than using the Normal Equation

CLOUD x LAB

Batch Gradient Descent

Gradient Descent Step

- Once we have the gradient vector, which points uphill,
  - Just go in the opposite direction to go downhill
- We can calculate the size of the downhill step using below equation

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

Learning Rate

CLOUD x LAB

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> 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
```

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> 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
```

Let's understand the program

CLOUD x LAB

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> eta = 0.1 # Learning Rate
>>> 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
```

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> eta = 0.1
>>> n_iterations = 1000 # Number of iterations after which algorithm stops
>>> 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
```


CLOUD x LAB

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> eta = 0.1
>>> n_iterations = 1000
>>> m = 100 # Number of elements in training dataset
>>> 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
```

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> eta = 0.1
>>> n_iterations = 1000
>>> m = 100
>>> theta = np.random.randn(2,1) # Random
initialization of model parameters

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

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> 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) # Gradient vector of the cost function
        theta = theta - eta * gradients
```

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> 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 #
    Determine the size of the downhill step
```


CLOUD x LAB

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> 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
```

```
array([[ 4.21509616],
       [ 2.77011339]])
```


CLOUD x LAB

Batch Gradient Descent

Quick Implementation of the Algorithm

```
>>> 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
```

Output

```
array([[ 4.21509616],
       [ 2.77011339]])
```

Theta is same like Normal Equation

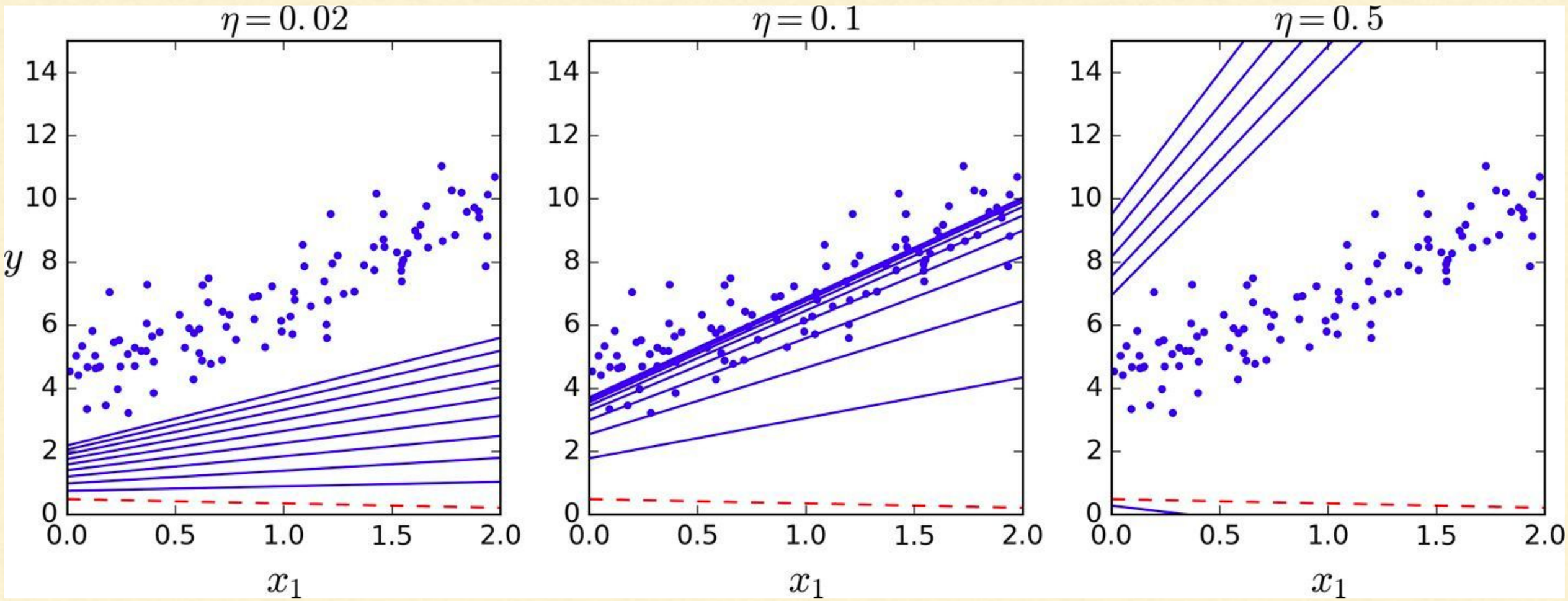
CLOUD x LAB

Batch Gradient Descent

How the learning rate affects Gradient Descent?

Batch Gradient Descent

- First 10 steps of Gradient Descent using three different learning rates
- The dashed line - - - - represents the starting point



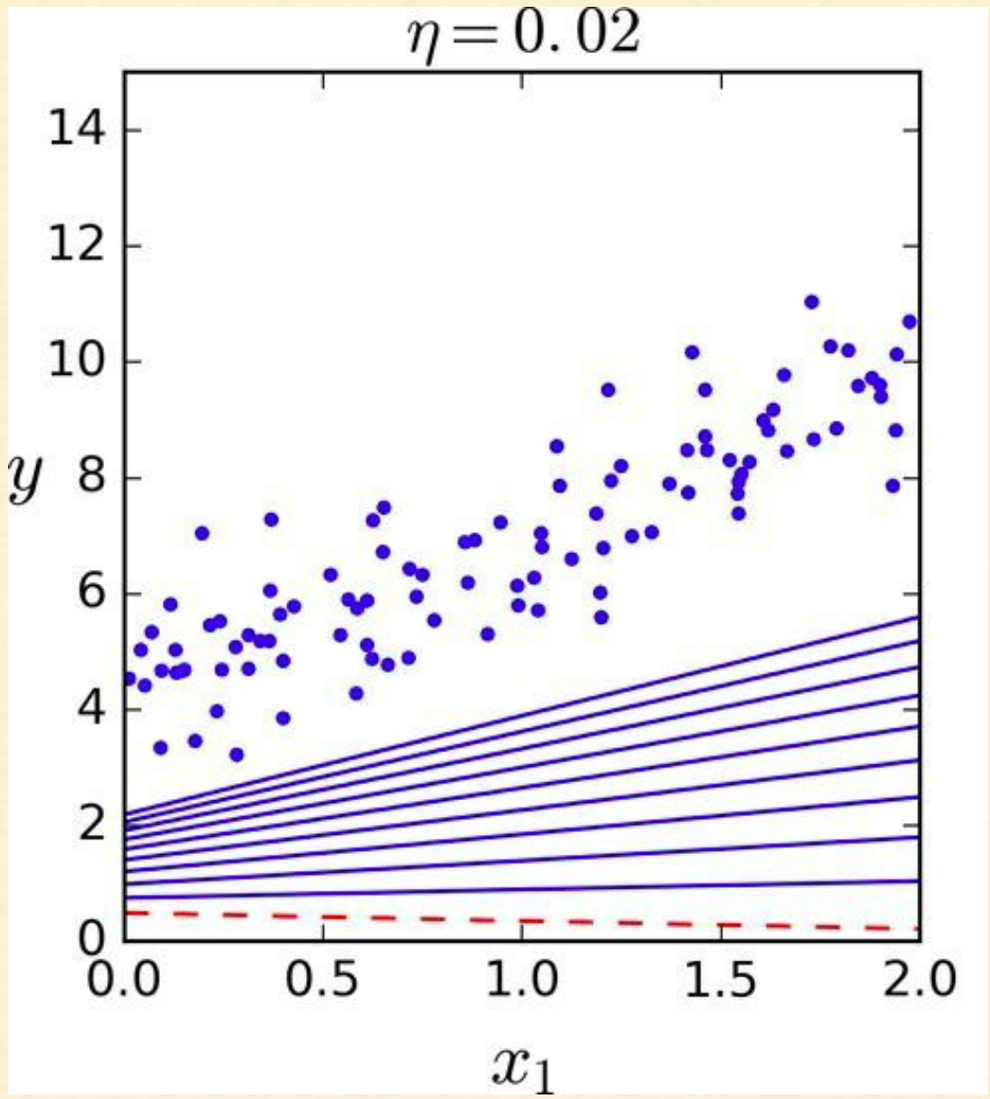
Gradient Descent with Various Learning Rates (See the code in notebook)

CLOUD x LAB

Batch Gradient Descent

Different Learning Rates

- Learning rate is too low (0.02)
- Algorithm will take a lot of time to reach the solution

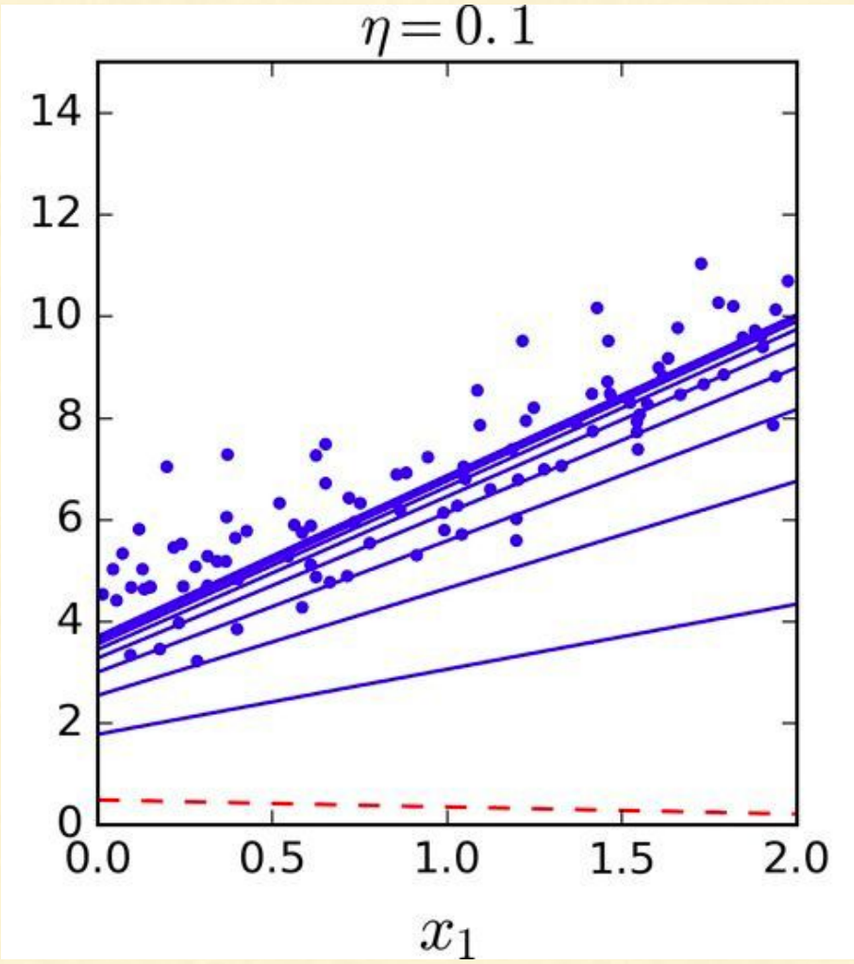


CLOUD x LAB

Batch Gradient Descent

Different Learning Rates

- Learning rate is good (0.1)
- In few iterations only, it converged to good solution

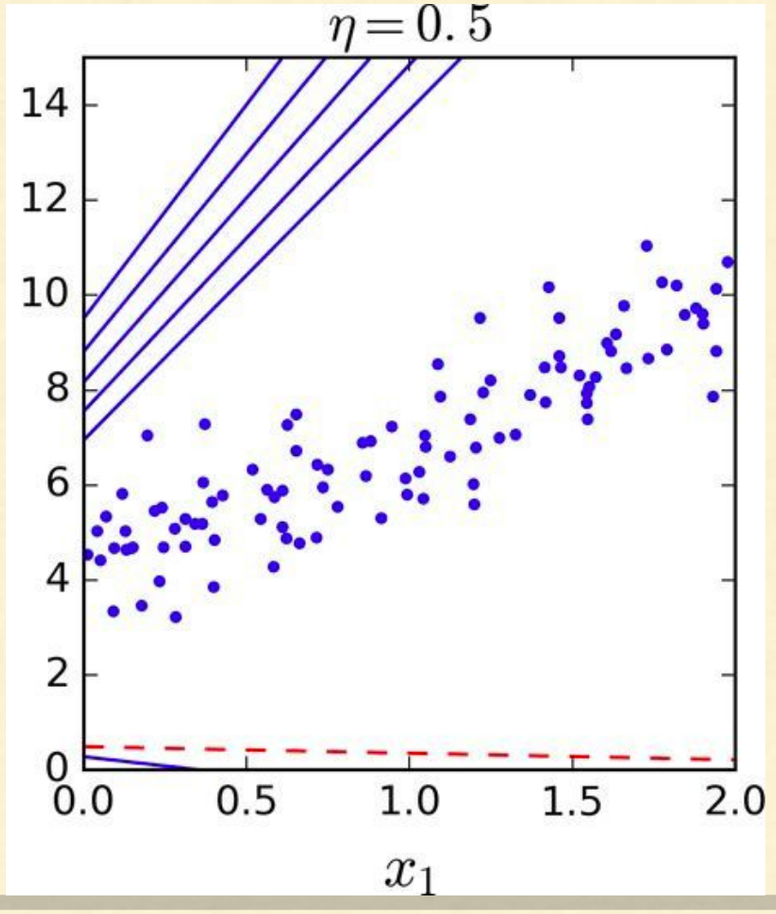


CLOUD x LAB

Batch Gradient Descent

Different Learning Rates

- Learning rate is too high (0.5)
- The algorithm diverges
  - Jumping all over the place and
  - Getting further and further away from the solution at every step



CLOUD x LAB

Batch Gradient Descent

Then how to find good learning rate?

Batch Gradient Descent

Find Good Learning Rate

- Remember grid search from end-to-end project?
- Use grid search to find good learning rate
- But limit the number of iterations
    - So that grid search can eliminate models that take too long to converge

Batch Gradient Descent

Then how to set number of iterations?

Batch Gradient Descent

Find Number of Iterations

- If number of iterations are too low
  - Then we may not get optimal solution when the algorithm stops
- If number of iterations are too high
  - Then our time may be wasted and
  - Algorithm may never converge

CLOUD x LAB

Batch Gradient Descent

Find Good Learning Rate - Solution

- Set a very large number of iterations and
- Interrupt the algorithm when the gradient vector becomes tiny
  - When norm of a gradient vector becomes smaller than ϵ - tolerance

Norm of vector (x1, x2 --- xn)

$$|x| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}\ .$$

Stop when

$$|x| < \epsilon$$

CLOUD x LAB

Batch Gradient Descent

Convergence Rate

- For MSE
  - Batch Gradient Descent with a fixed learning rate has a convergence rate of

$$O\left(\frac{1}{Iterations}\right)$$

- If we divide the tolerance ε by 10 to have a more precise solution
  - Then the algorithm will have to run about 10 times more iterations

CLOUD x LAB

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent - SGD

Problem with Batch Gradient Descent

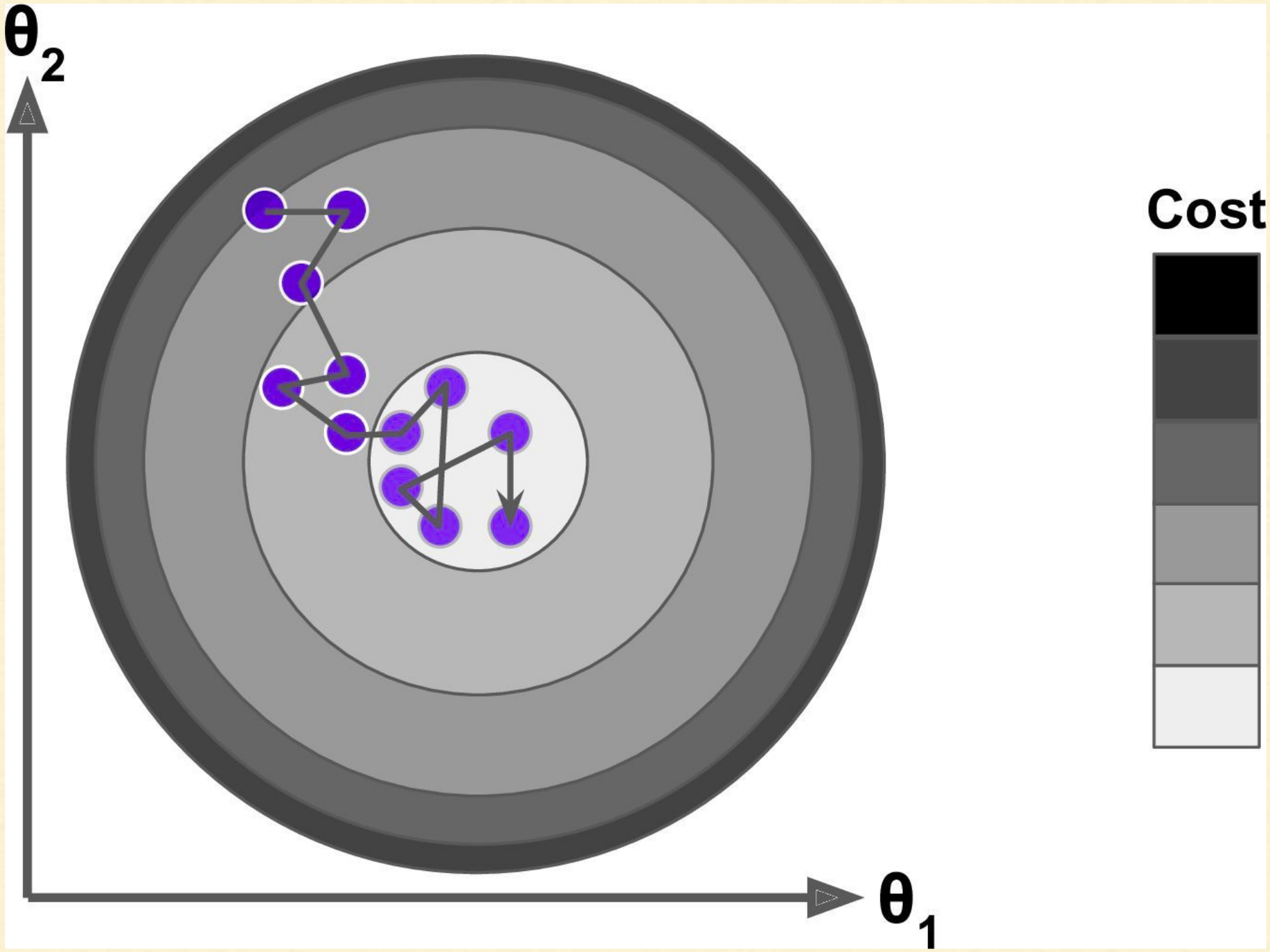
- Batch Gradient Descent
  - Slow when the training set is large
  - As it uses the whole training set to compute the gradient at every step
- Stochastic Gradient Descent
  - Just picks a random instance in the training set at every step and
  - Computes the gradients based only on that single instance

CLOUD x LAB

Stochastic Gradient Descent - SGD

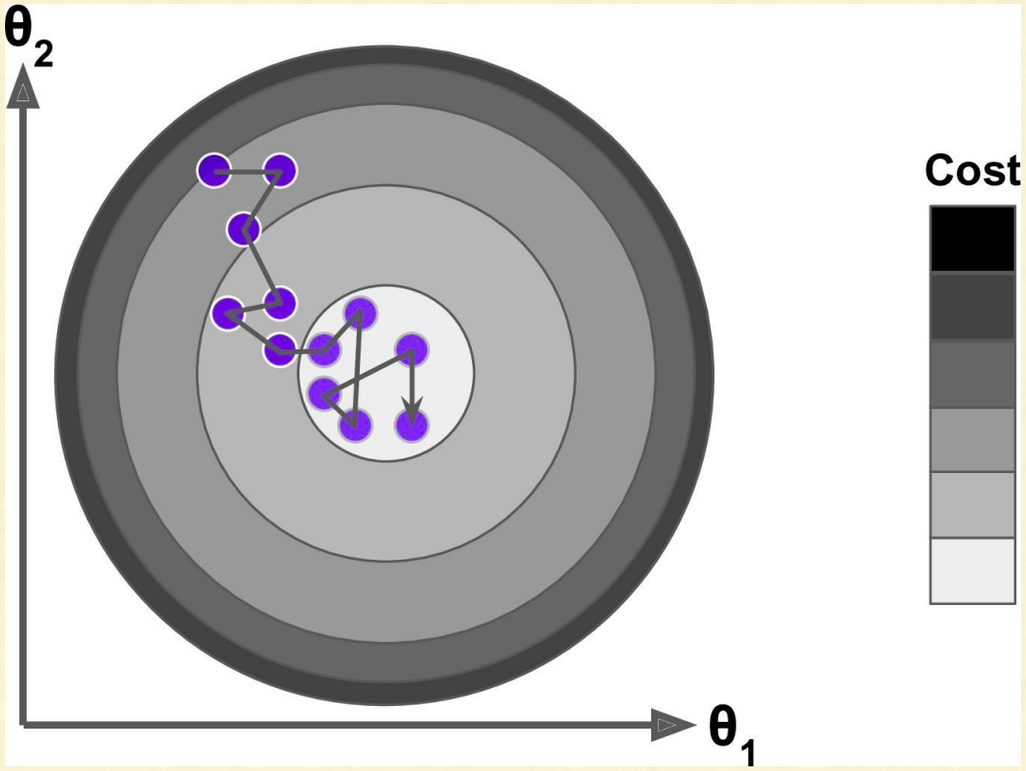
- This makes the algorithm much faster
  - As very little data gets manipulate at every iteration
- Also huge training sets can be trained
  - As only one instance needs to be in memory at each iteration

Stochastic Gradient Descent - SGD



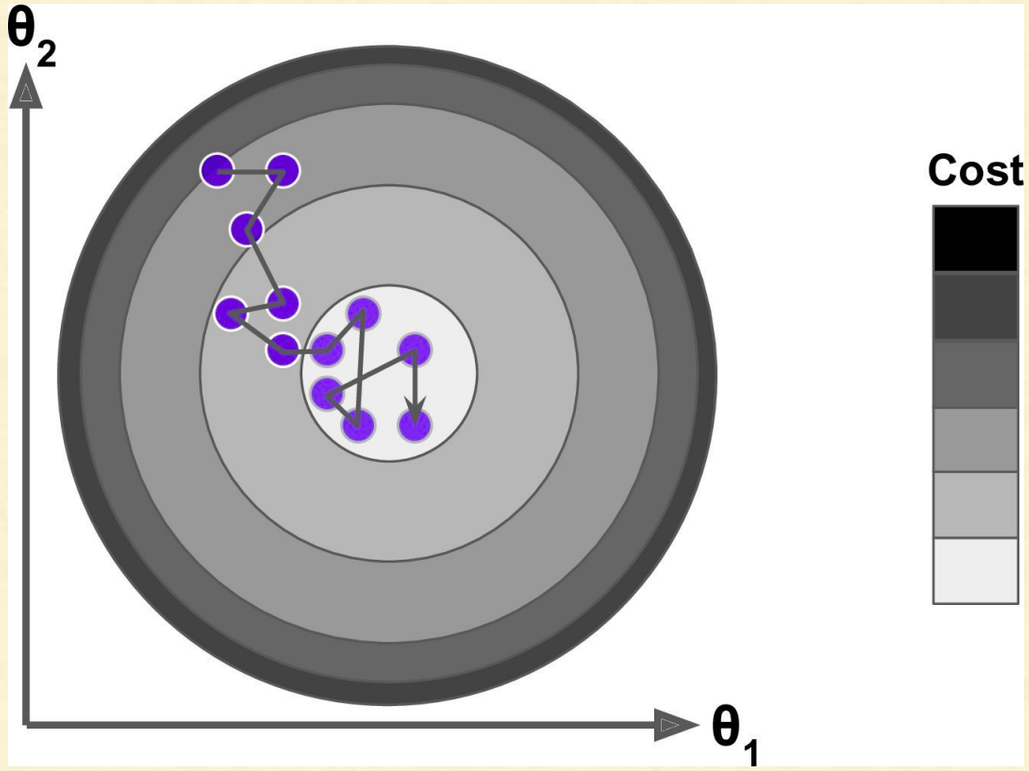
Stochastic Gradient Descent - SGD

- Instead of gently decreasing until it reaches the minimum
  - The cost function bounces up and down, decreasing only on average
  - Over time it ends up very close to the minimum
  - But once it gets there it continues to bounce around, never settling down



Stochastic Gradient Descent - SGD

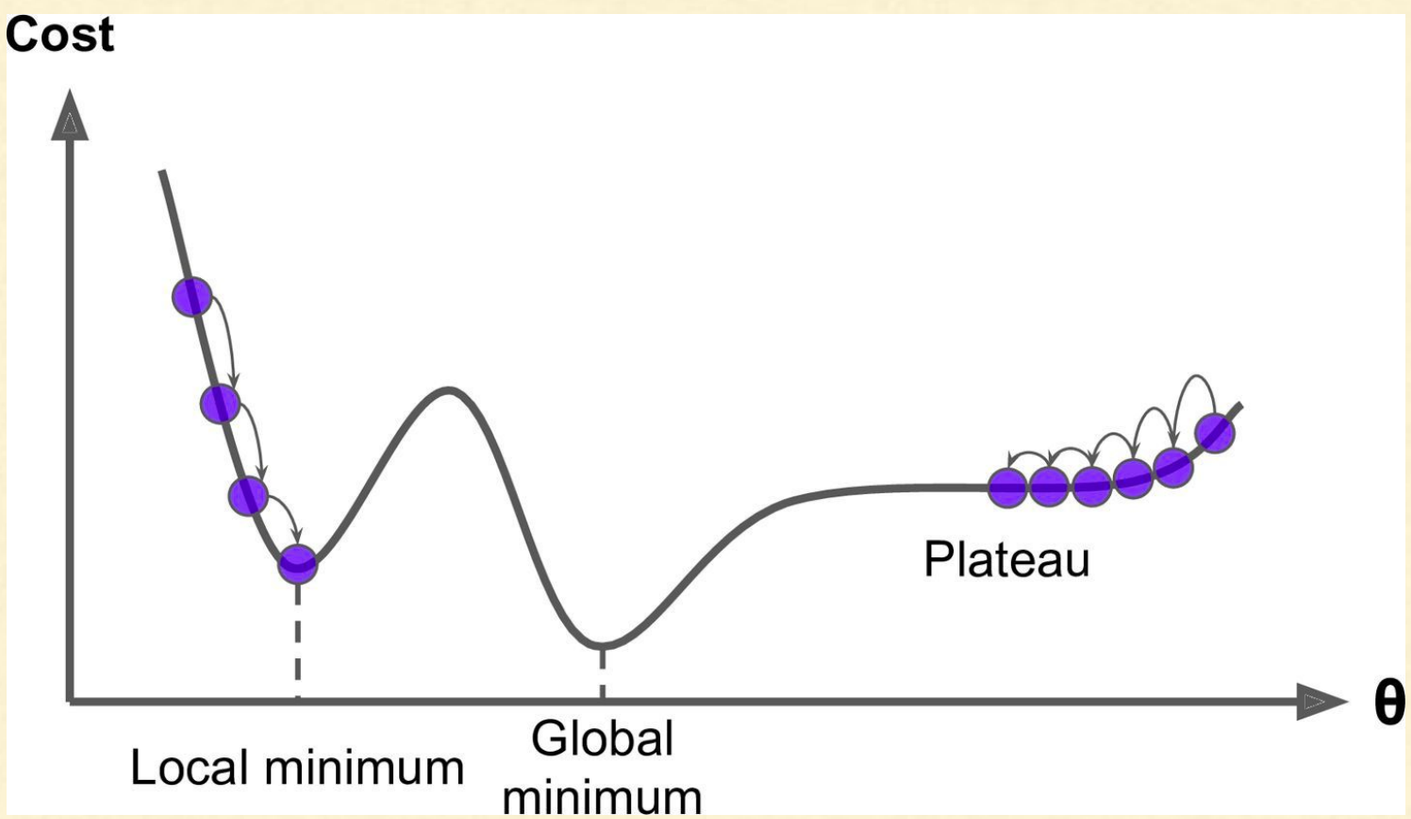
- Once the algorithm stops
  - The final parameter values are good, but not optimal
  - As algorithm never settles down



CLOUD x LAB

Stochastic Gradient Descent - SGD

- When the cost function is very irregular then
  - SGD has a better chance of finding the global minimum than Batch Gradient Descent
  - As randomness may help algorithm to jump out of local minima



CLOUD x LAB

Stochastic Gradient Descent - SGD

- As seen earlier that randomness in SGD
  - Is good to escape from local minimum
  - But bad as algorithm can never settle at minimum

Stochastic Gradient Descent - SGD

- As seen earlier that randomness in SGD
  - Is good to escape from local minimum
  - But bad as algorithm can never settle at minimum

What is the solution?

Stochastic Gradient Descent - SGD

Simulated Annealing

- One solution is to gradually reduce the learning rate
- The steps start out large
  - This helps make quick progress and escape local minima
- Then steps get smaller and smaller
  - This helps the algorithm to settle at the global minimum
- This process is called **Simulated Annealing**
  - Because it resembles the process of annealing in metallurgy where molten metal is slowly cooled down

Stochastic Gradient Descent - SGD

Learning Schedule

- The function that determines the learning rate
  - At each iteration is called the **learning schedule**
- If the learning rate is reduced too quickly
  - We may get stuck in a local minimum Or
  - Even end up frozen halfway to the minimum
- If the learning rate is reduced too slowly
  - We may jump around the minimum for a long time and
  - End up with a suboptimal solution if we halt training too early

CLOUD x LAB

Stochastic Gradient Descent - SGD

Implement SGD

- Let's implement SGD
- By convention we iterate by rounds of *m* iterations
  - Each round is called an **epoch**

m -> number of instances in the training set

Stochastic Gradient Descent - SGD

Implement SGD

- Let's implement SGD
- By convention we iterate by rounds of *m* iterations
  - Each round is called an **epoch**

m -> number of instances in the training set

See Stochastic Gradient Descent section in notebook for the implementation

CLOUD x LAB

Stochastic Gradient Descent - SGD

Implement SGD

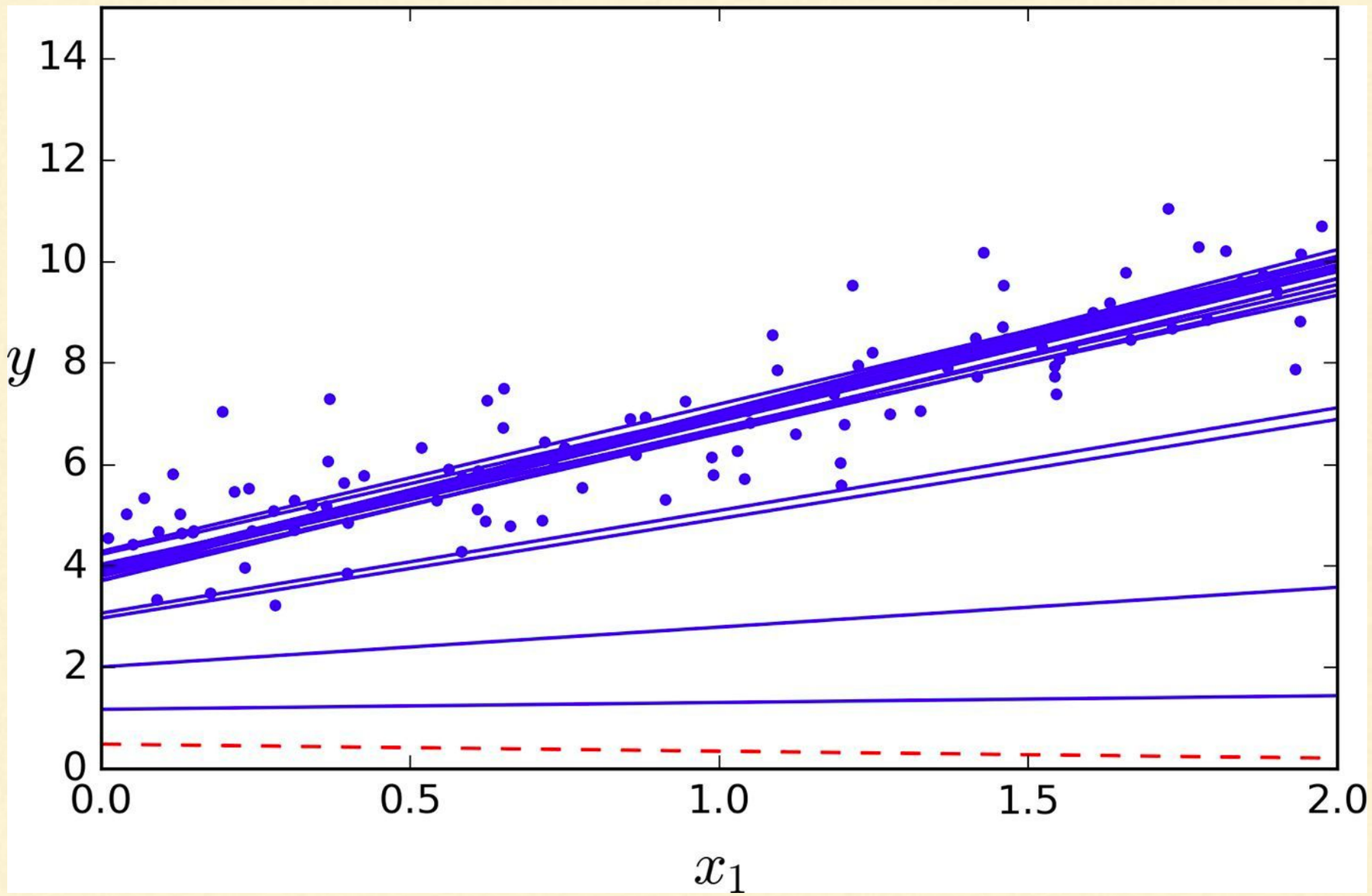
Output

```
array([[ 4.21076011],
       [ 2.74856079]])
```


CLOUD x LAB

Stochastic Gradient Descent - SGD

Implement SGD



SGD first 10 steps. Notice how irregular the steps are

CLOUD x LAB

Stochastic Gradient Descent - SGD

Scikit-Learn SGDRegressor Class

- To perform linear regression using Scikit-Learn
  - Use SGDRegressor class
- SGDRegressor class by default optimizes the squared error cost function

CLOUD x LAB

Stochastic Gradient Descent - SGD

Scikit-Learn SGDRegressor Class

```
>>> from sklearn.linear_model import SGDRegressor
>>> sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.intercept_, sgd_reg.coef_
```

n_iter ---> Number of epochs

eta ---> Learning rate

penalty ---> None means do not use any regularization

CLOUD x LAB

Stochastic Gradient Descent - SGD

Scikit-Learn SGDRegressor Class

```
>>> from sklearn.linear_model import SGDRegressor
>>> sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.intercept_, sgd_reg.coef_
```

Output

```
(array([ 4.16782089]), array([ 2.72603052]))
```

Solution is very close to the Normal Equation

CLOUD x LAB

Mini-batch Gradient Descent

Mini-batch Gradient Descent

- The last Gradient Descent algorithm we will look into
- Quite easy to understand if we know both
  - Batch and
  - Stochastic Gradient Descent

Mini-batch Gradient Descent

- At each step
  - Instead of computing the gradients based on the full training set (as in Batch GD) or
  - Based on just one instance (as in Stochastic GD)
  - Mini-batch GD computes the gradients on small random sets of instances called **mini-batches**

CLOUD x LAB

Mini-batch Gradient Descent

Advantages

- The main advantage of Mini-batch GD over Stochastic GD is that
  - We can get a performance boost
  - From hardware optimization of matrix operations
  - Especially when using GPUs

Mini-batch Gradient Descent

Advantages

- Mini-batch GD algorithm's progress is
  - Less erratic than SGD
  - Especially with fairly large mini-batches
  - As a result it goes more closer to minimum than SGD

Mini-batch Gradient Descent

Disadvantages

- It is harder for Mini-batch GD to escape from local minima

Mini-batch Gradient Descent

Implement Mini-batch GD

See Mini-batch gradient descent section in notebook for the implementation

Mini-batch Gradient Descent

Implement Mini-batch GD

- As we can see
  - For each epoch (as in SGD)
  - We are creating mini-batches from the training dataset (as in Batch GD)

Output

```
array([[ 4.25214635],
       [ 2.7896408 ]])
```

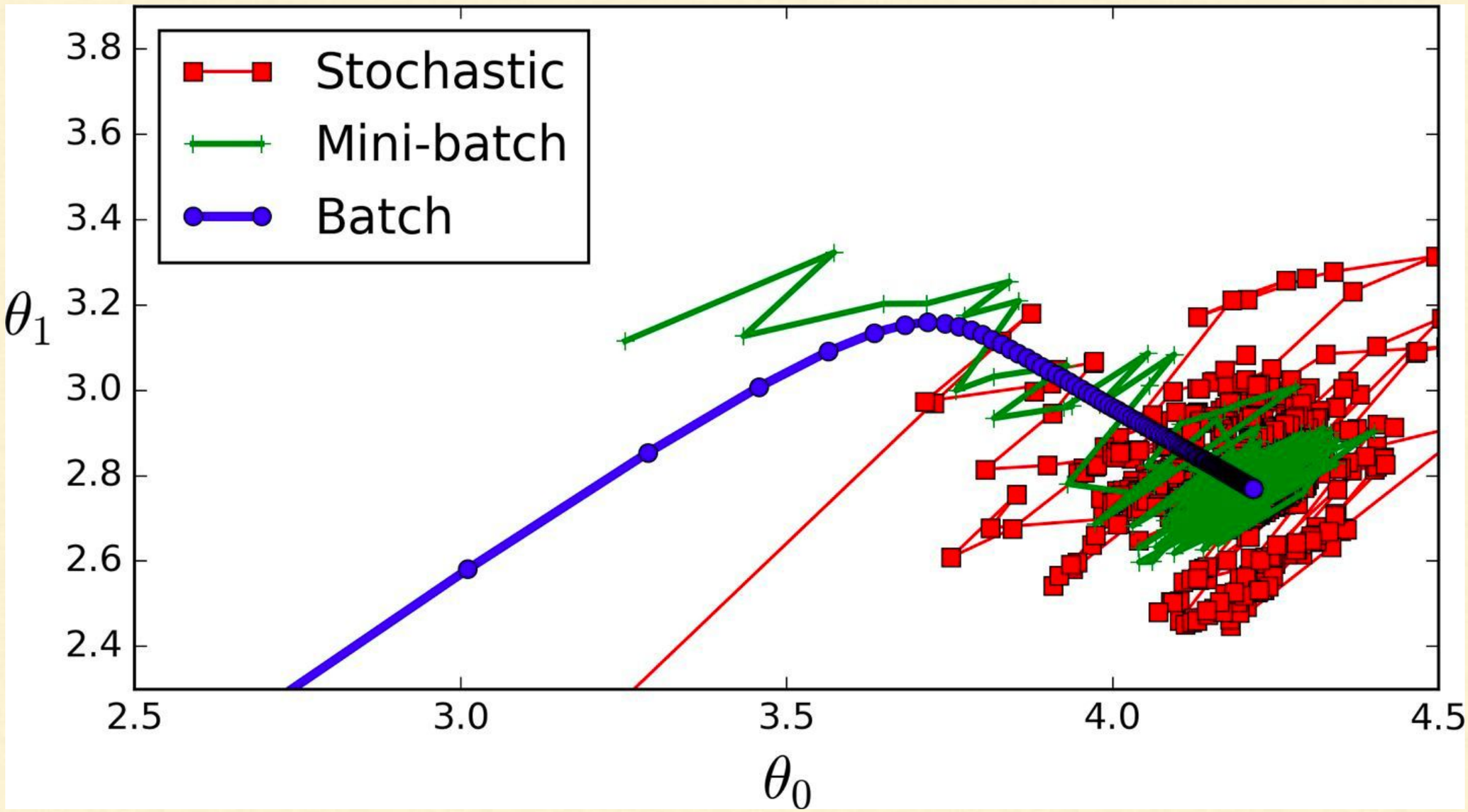

CLOUD x LAB

Mini-batch Gradient Descent

- Let's see the path taken by below algorithms during training
  - Batch GD
  - Stochastic GD
  - Mini-batch GD

For code please see the notebook

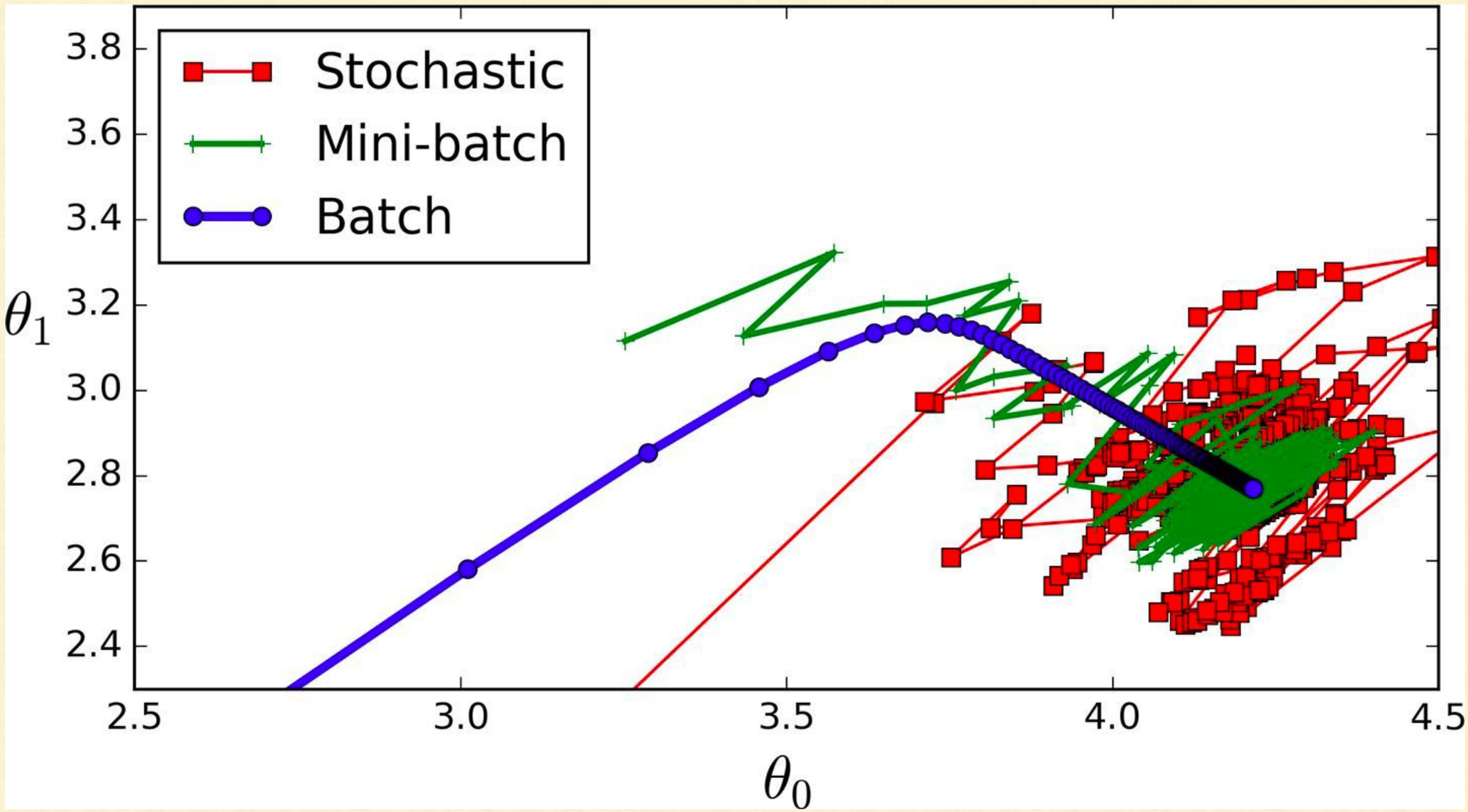
Mini-batch Gradient Descent



Gradient Descent paths in parameter space during training

CLOUD x LAB

Mini-batch Gradient Descent

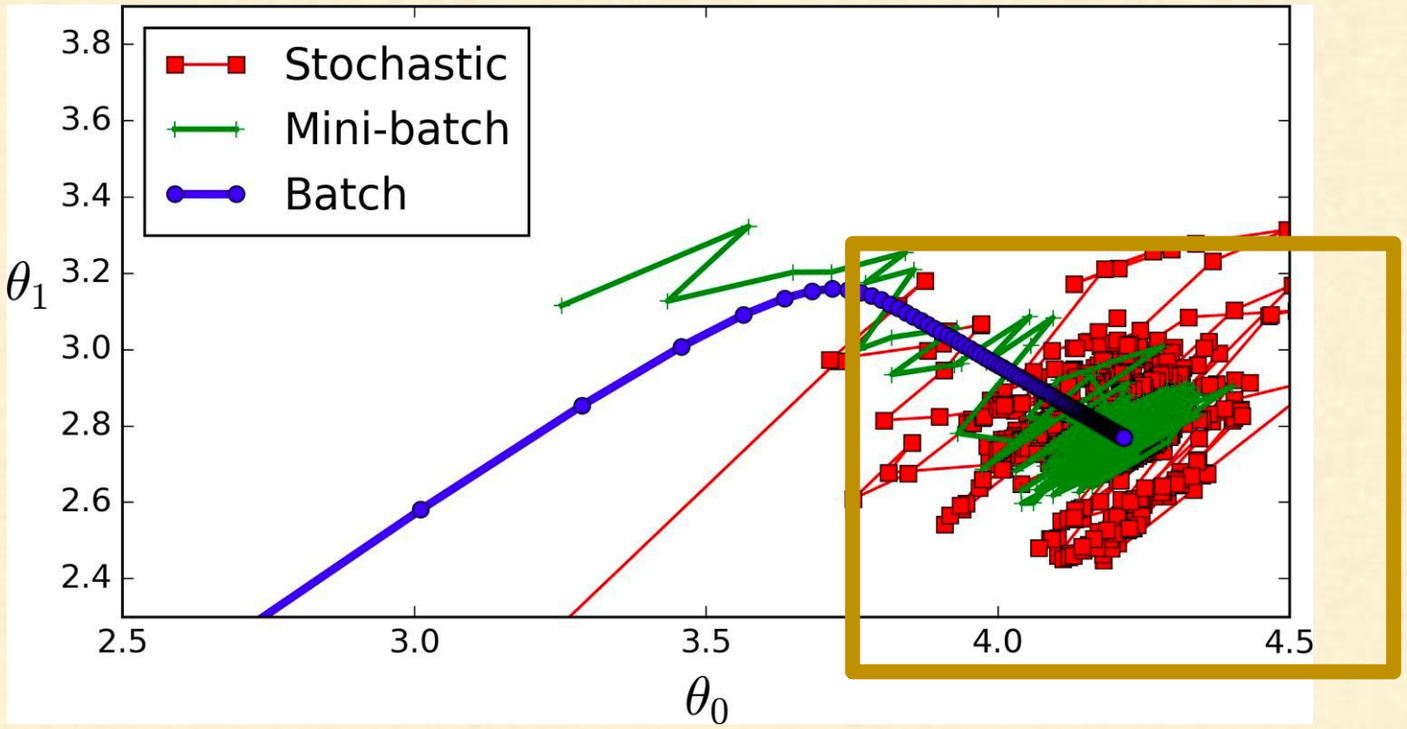


Observations??

Mini-batch Gradient Descent

Observation - 1

All the three algorithms end up near the minimum

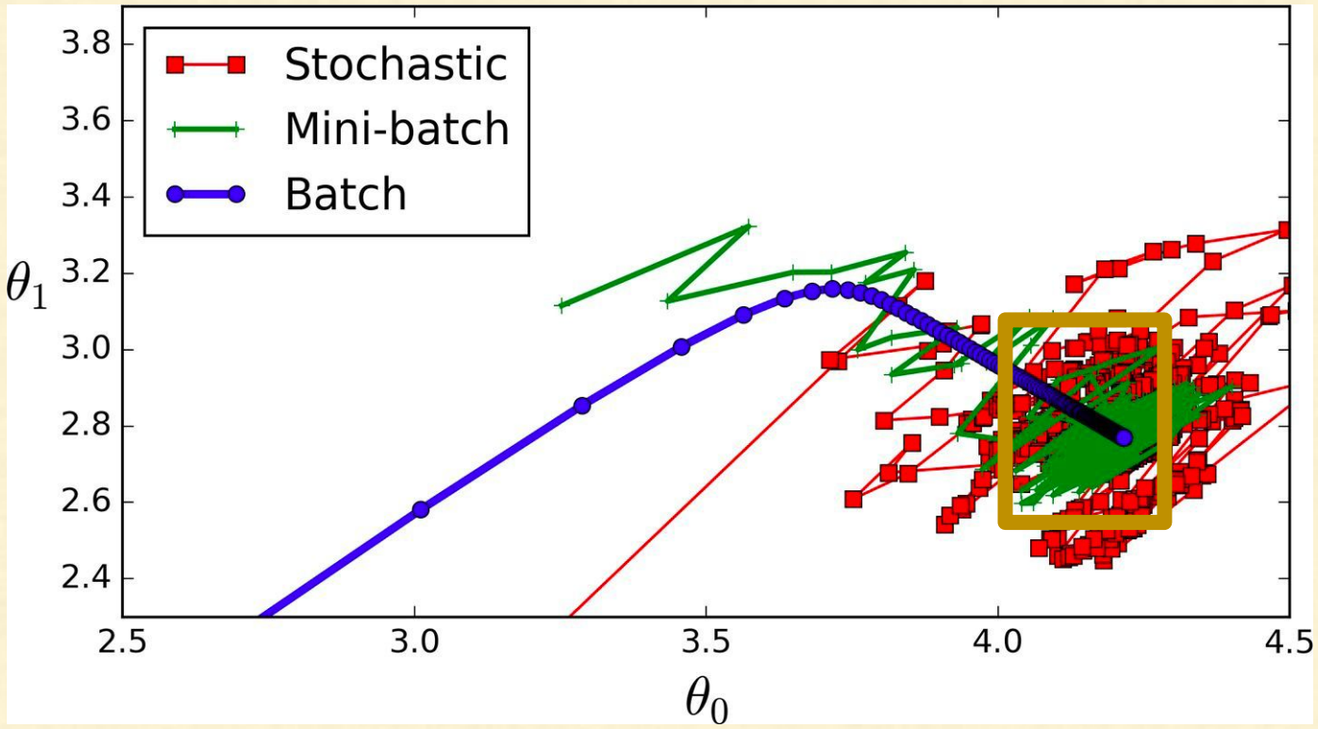


CLOUD x LAB

Mini-batch Gradient Descent

Observation - 2

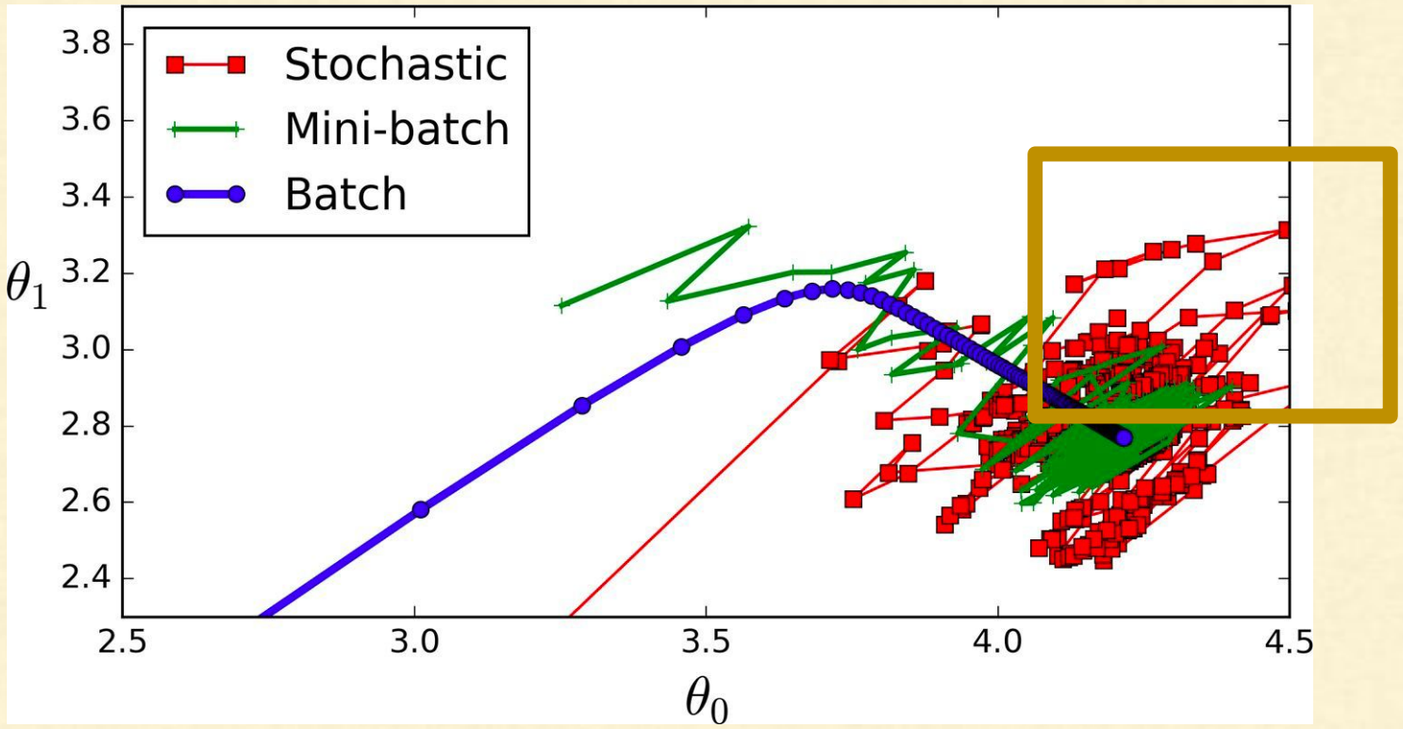
Batch GD's path actually stops at the minimum



Mini-batch Gradient Descent

Observation - 3

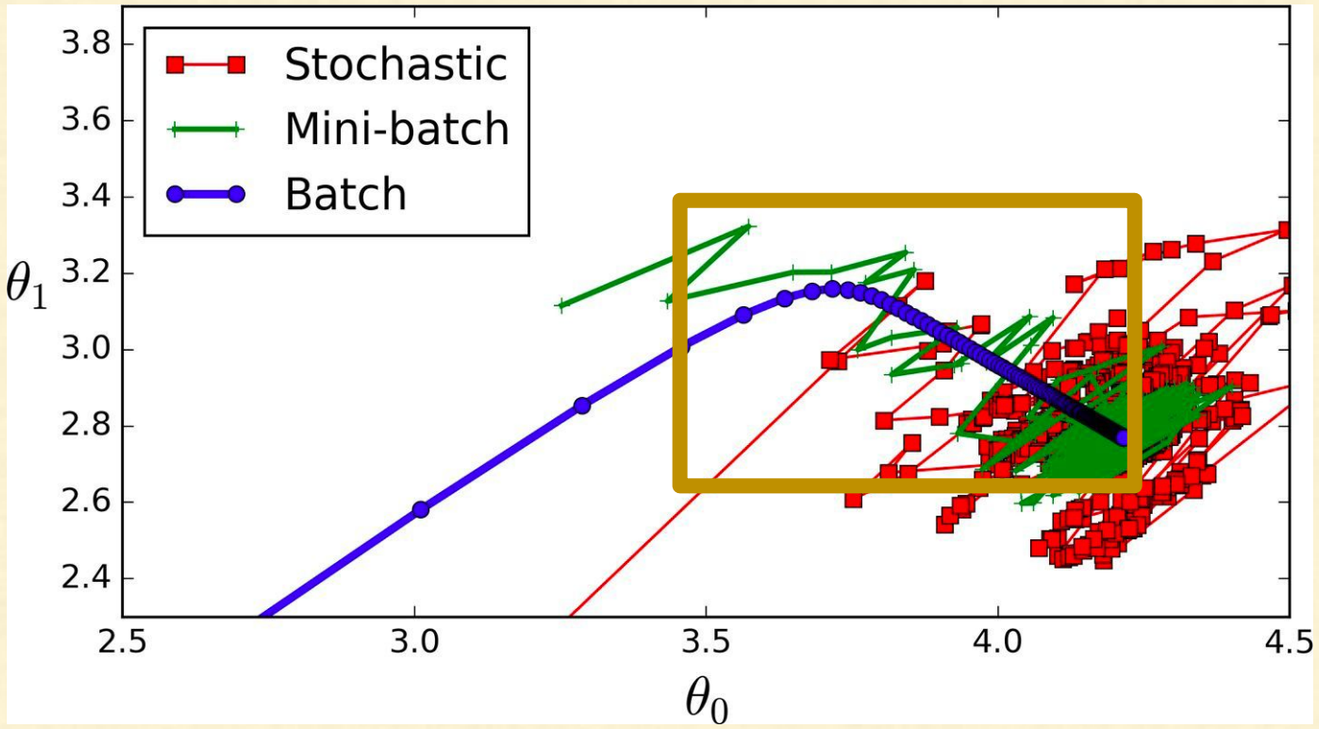
Both Stochastic GD and Mini- batch GD continue to walk around



Mini-batch Gradient Descent

Observation - 4

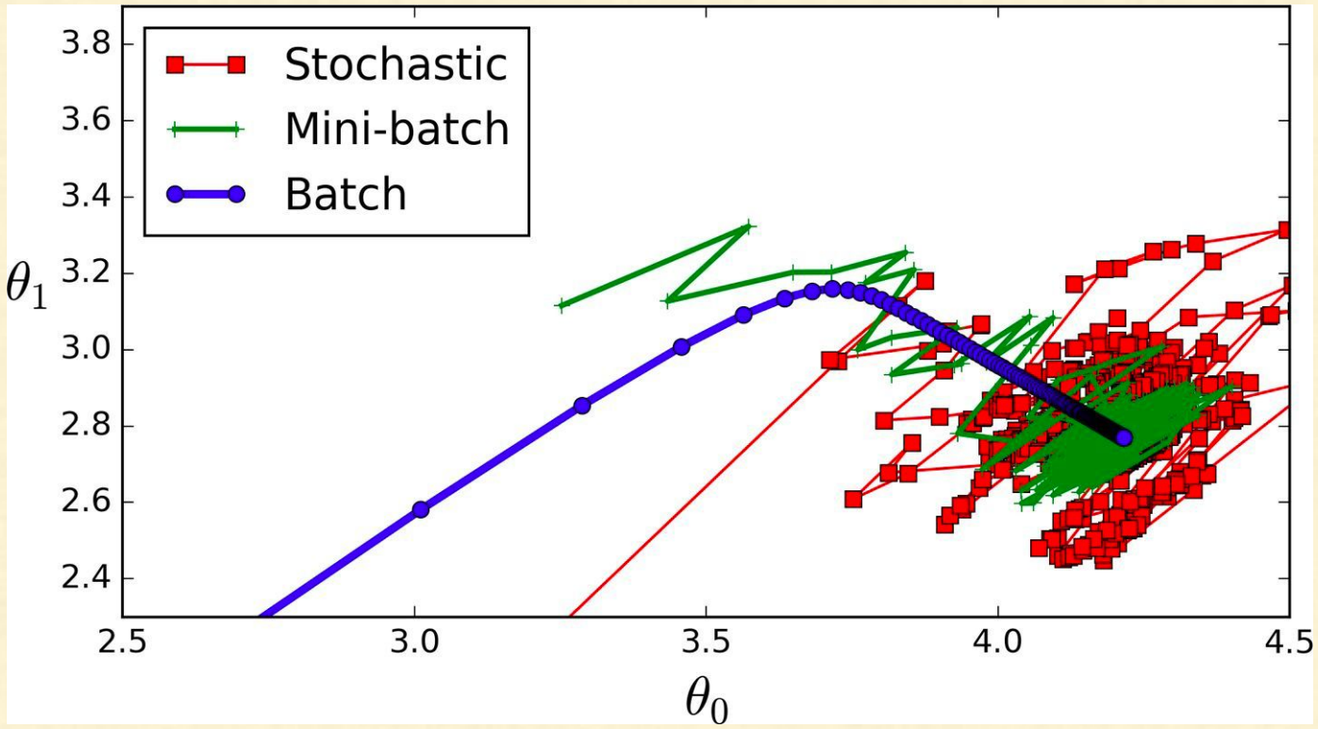
Batch GD takes a lot of time to take each step



Mini-batch Gradient Descent

Note - 1

- While Batch GD takes a lot of time
  - Both Stochastic GD and Mini-batch GD can reach the minimum provided good learning schedule

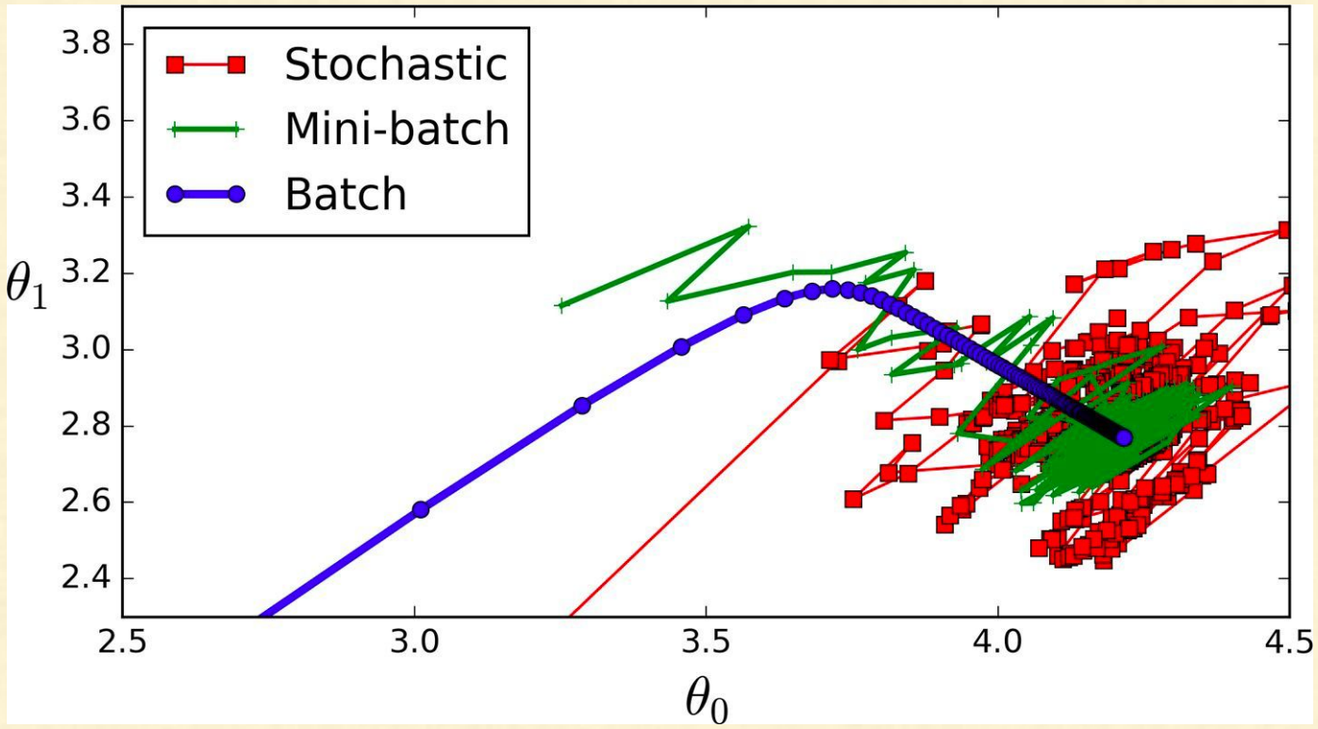


CLOUD x LAB

Mini-batch Gradient Descent

Note - 2

- There is almost no difference after training
  - All these three algorithms end up with very similar models and
  - Make predictions in exactly the same way



CLOUD x LAB

Comparison

Let's compare all the three algorithms for Linear Regression

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

m is the number of training instances
n is the number of features

CLOUD x LAB

Linear Regression

Polynomial Regression

Polynomial Regression

- So far we have just looked the data having
  - A linear relationship or
  - Straight line

Polynomial Regression

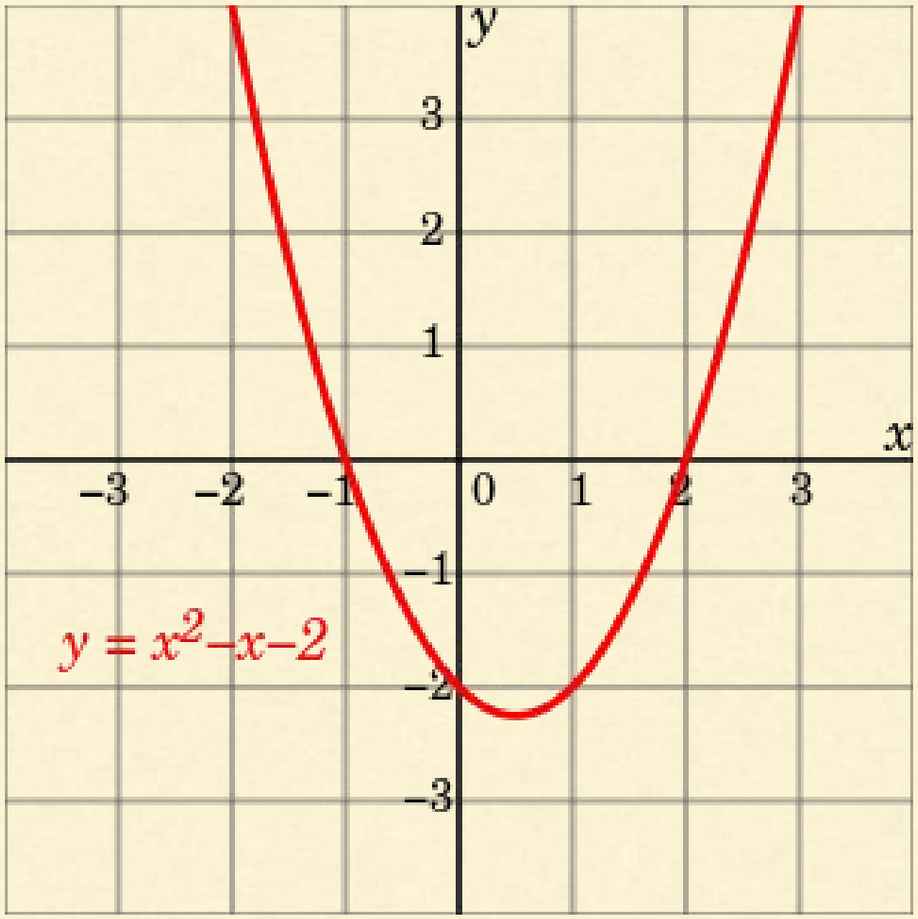
- So far we have just looked the data having
  - A linear relationship or
  - Straight line

What if we have to fit nonlinear data like quadratic equation?

CLOUD x LAB

Polynomial Regression

Quadratic Equation

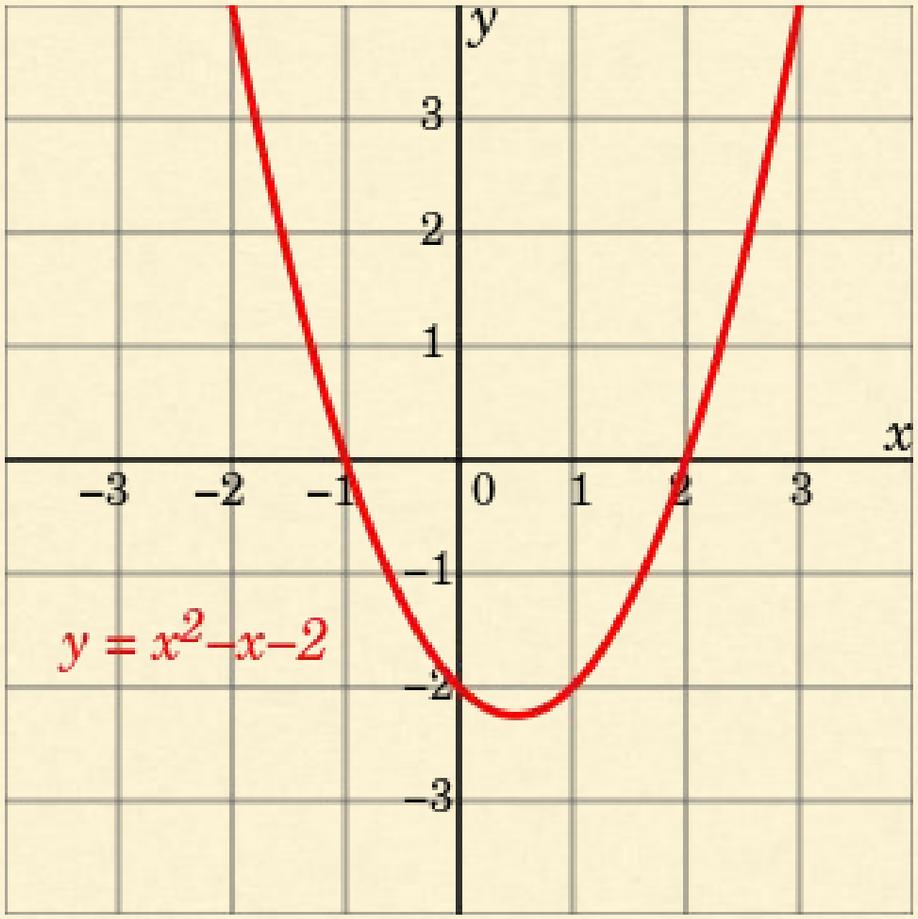


Graph of the quadratic equation y = x^2 - x - 2

CLOUD x LAB

Polynomial Regression

Quadratic Equation

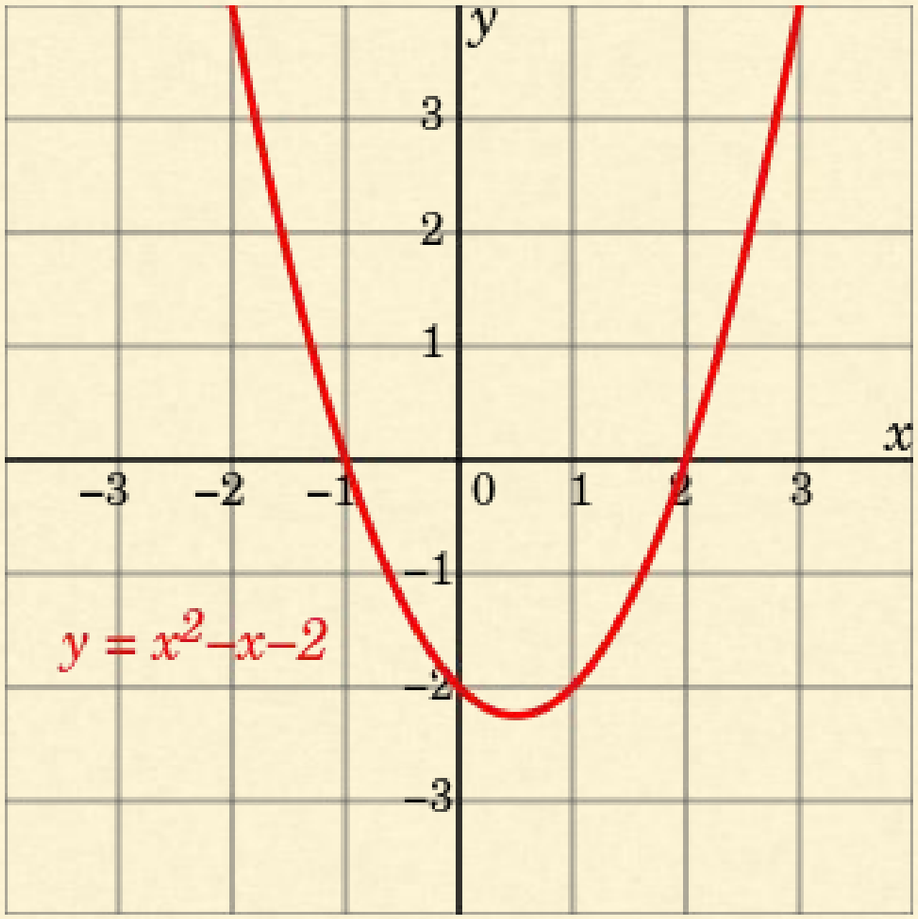


Clearly a straight line will never fit this data properly

CLOUD x LAB

Polynomial Regression

Quadratic Equation



Then how do we fit this kind of nonlinear data?

CLOUD x LAB

Polynomial Regression

- We can use
  - A linear model to fit nonlinear data
  - Add powers of each feature as new features
  - And train a linear model on extended set of features
  - This is called **polynomial regression**

Polynomial Regression

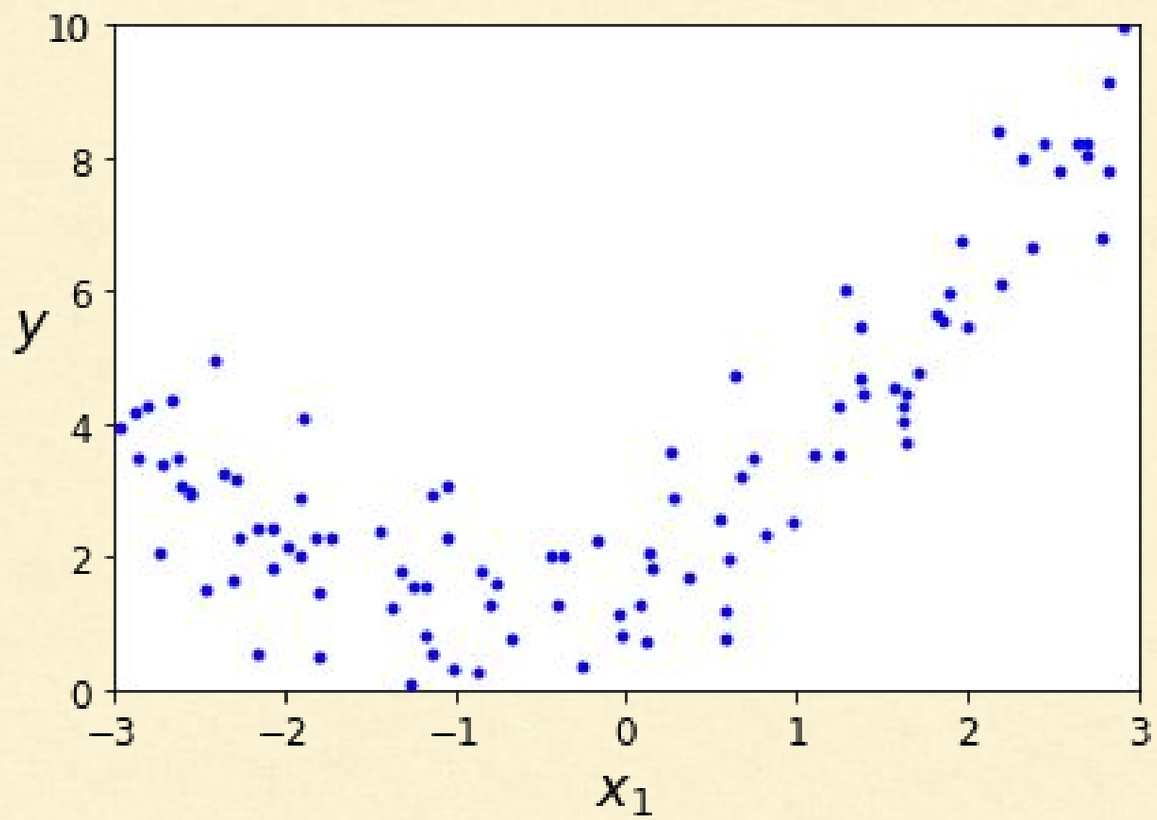
- Let's test it
- First generate some nonlinear data
  - Based on a quadratic equation
  - Plus some noise

$$y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$$

Polynomial Regression

Generate Nonlinear Data with Noise

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



Generated nonlinear and noisy dataset

CLOUD x LAB

Polynomial Regression

Scikit-Learn's PolynomialFeatures Class

- Use PolynomialFeatures class to transform the training data
- Add square of each feature in the training set as new features

```
new features = existing features in training set +
square of each feature in training set
```


CLOUD x LAB

Polynomial Regression

Scikit-Learn's PolynomialFeatures Class

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


CLOUD x LAB

Polynomial Regression

Scikit-Learn's PolynomialFeatures Class

```
>>> X[0]

array([-0.75275929])
```

Original features

```
>>> X_poly[0]

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

X_poly now contains the original feature of X plus the square of this feature

Polynomial Regression

Scikit-Learn's PolynomialFeatures Class

- PolynomialFeatures with degree=d
  - Transforms an array containing n features into an array containing
  - $$\frac{(n+d)!}{d!\,n!}$$ features

CLOUD x LAB

Polynomial Regression

Scikit-Learn's PolynomialFeatures Class

- Polynomial Regression is capable of finding relationships between features
  - Which plain Linear Regression model can not do
- This is because **PolynomialFeatures** adds all combinations of features
  - Up to the given degree

CLOUD x LAB

Polynomial Regression

Scikit-Learn's PolynomialFeatures Class

- For example
  - If there are two features a and b
  - PolynomialFeatures with degree=3 will add these features alse
  - a^2, a^3, b^2, and b^3
  - Also ab, a^2b, and ab^2

CLOUD x LAB

Polynomial Regression

Fit a LinearRegression model

- Now fit a LinearRegression model to this extended training data

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

Polynomial Regression

Fit a LinearRegression model

Output -

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

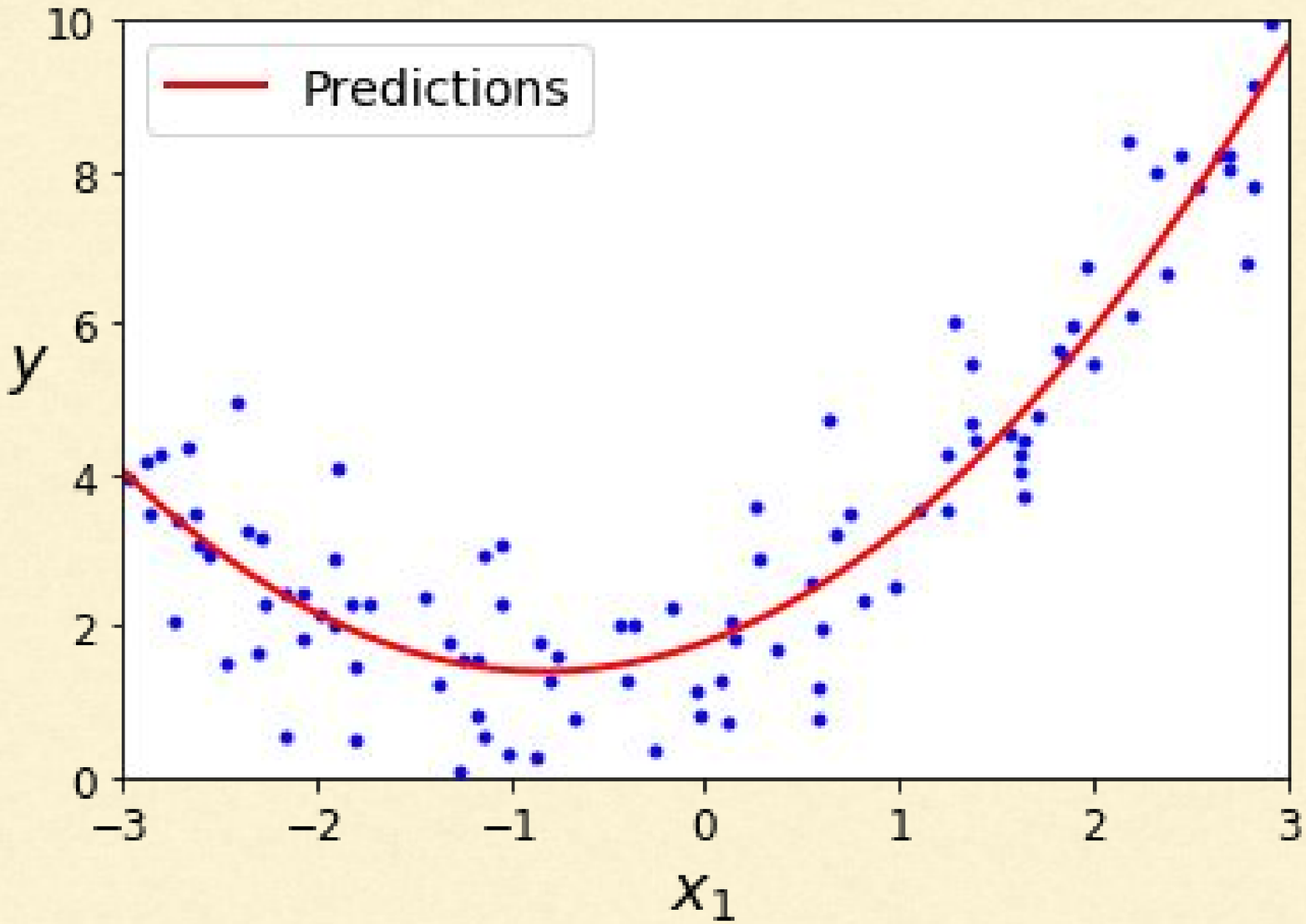
Model Estimation -

$$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$$

CLOUD x LAB

Polynomial Regression

Polynomial Regression model predictions



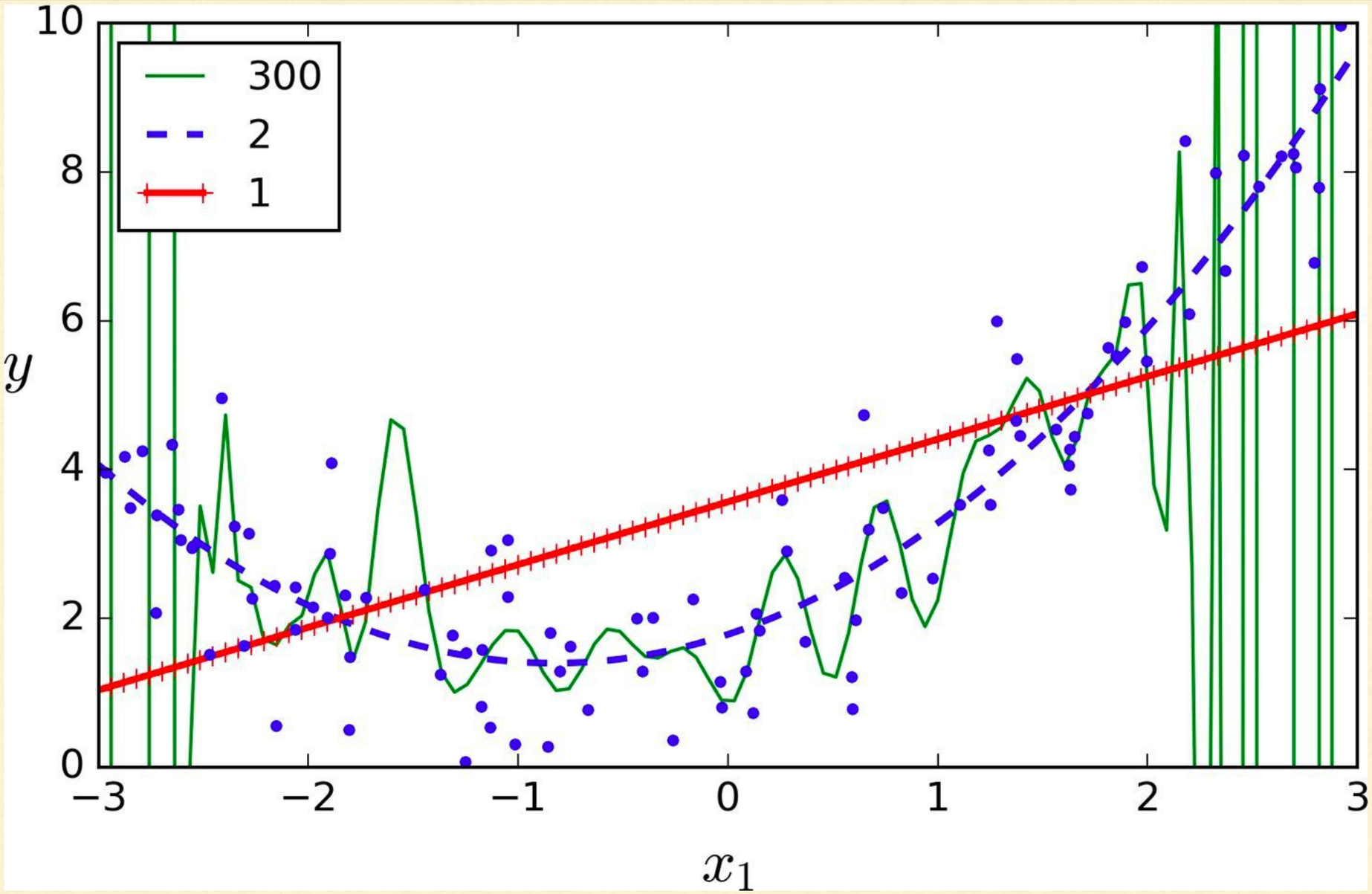
CLOUD x LAB

Learning Curves

Learning Curves

- As we have seen earlier
  - High-degree polynomial regression fits the training data
  - Much better than plain linear regression
- Let's look at the fit of
  - Plain Linear Regression
  - 2nd-degree Polynomial Regression and
  - 300-degree Polynomial Regression
  - See the code in Learning curves section in notebook

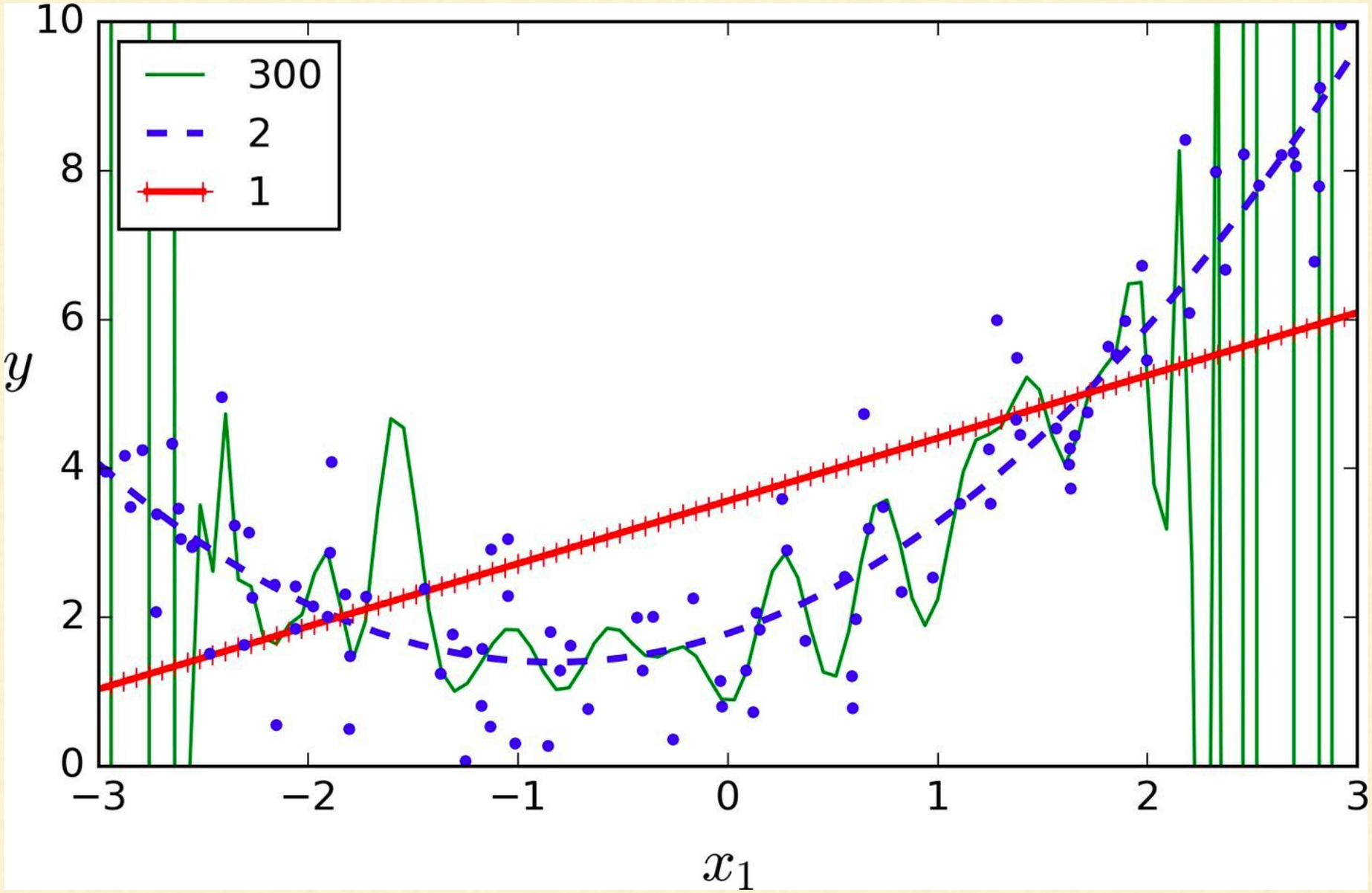
Learning Curves



High-degree Polynomial Regression

CLOUD x LAB

Learning Curves



-- Plain Linear Regression

-- 2nd-degree Polynomial Regression

-- 300-degree Polynomial Regression

CLOUD x LAB

Learning Curves

- As we can see in the previous diagram
  - 300-degree polynomial model
  - Wiggles around to get as close as possible
  - To the training instances
- It is clear that
  - The high-degree Polynomial Regression model is severely overfitting the training data and
  - The linear model is underfitting the training data

CLOUD x LAB

Learning Curves

How do we find if model is overfitting or underfitting the training data?

Learning Curves

One approach is to use cross-validation

Learning Curves

- If a model
  - Performs well on the training data
  - But generalizes poorly according to the cross-validation
  - Then model is **overfitting**
  - In this case model is too complex
- If a model
  - Performs poorly on both
  - Training data and cross-validation
  - Then model is **underfitting**
  - In this case model is too simple

Learning Curves

Another approach is to look at learning curves

Learning Curves

- Learning curves are
  - Plots of the model's performance on
  - The training set and the validation set
  - As a function of the training set size
- To generate the plots
  - Simply train the model
  - Several times on different sized subsets of the training set

Learning Curves

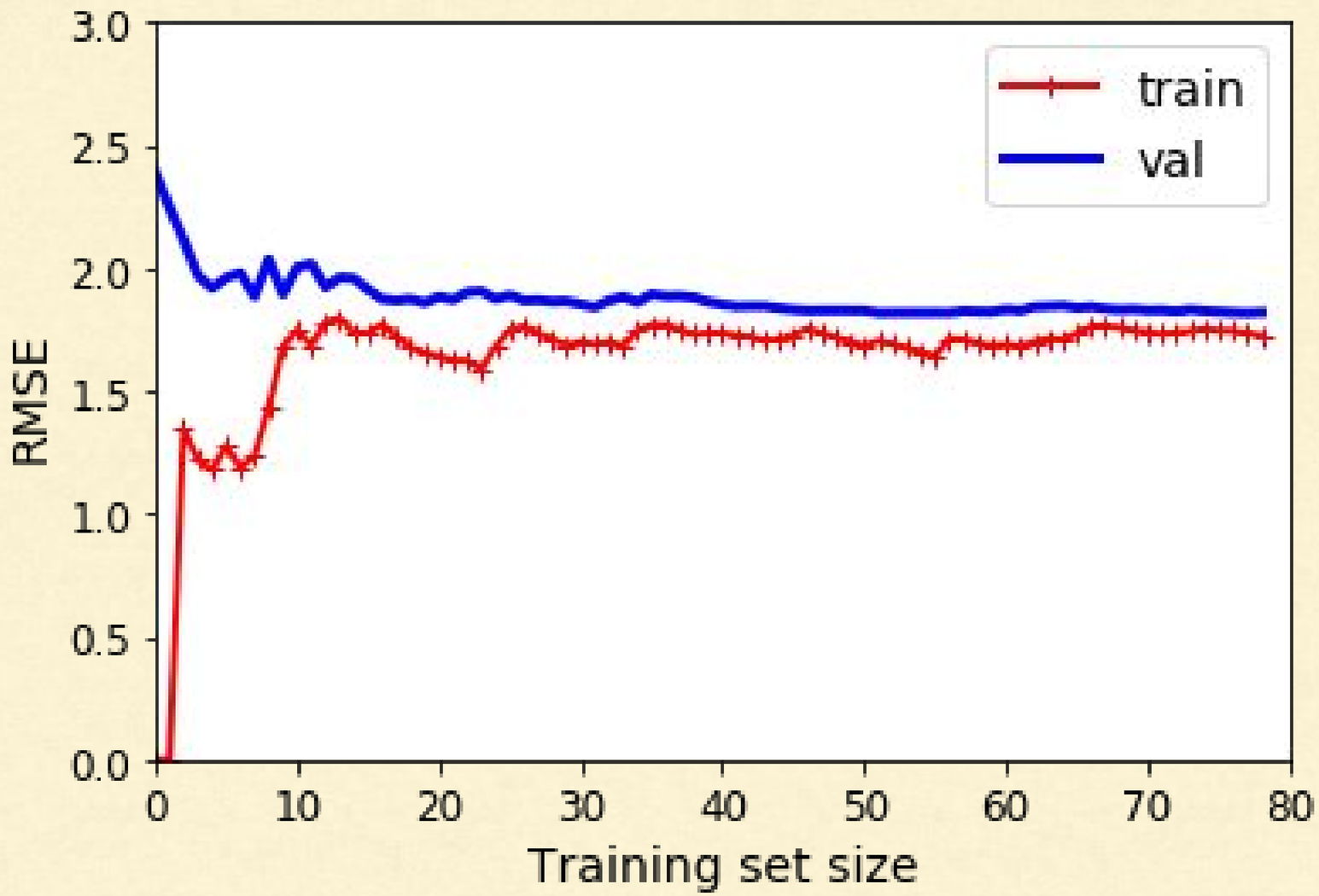
Learning curves of the plain Linear Regression model

- Let's plot the learning curves
- We'll train the model on different subsets of training set
- And plot the RMSE and Training set size

Switch to the notebook for the code

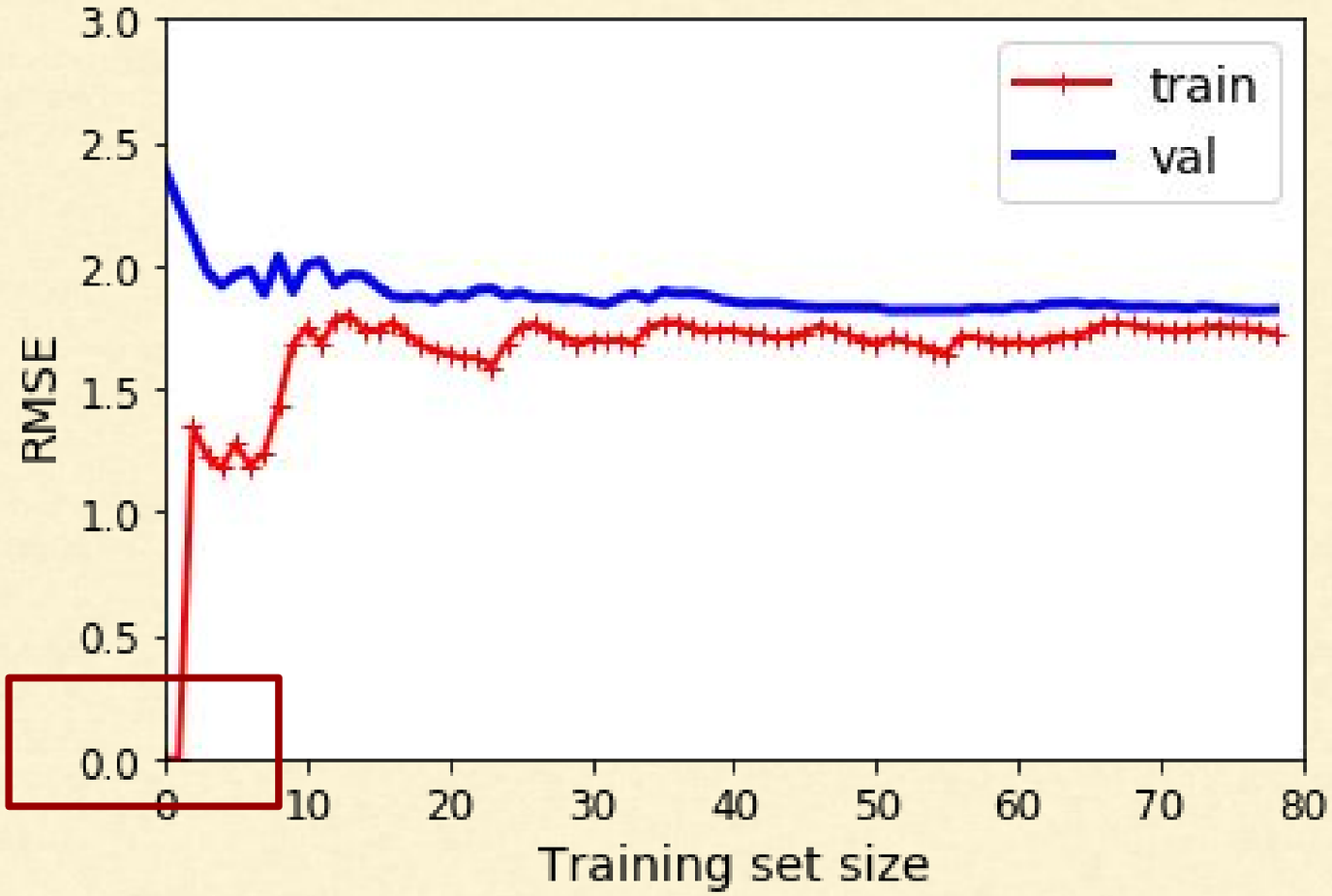
Learning Curves

Learning curves of the plain Linear Regression model



Learning Curves

Let's look at performance of the model on the training set

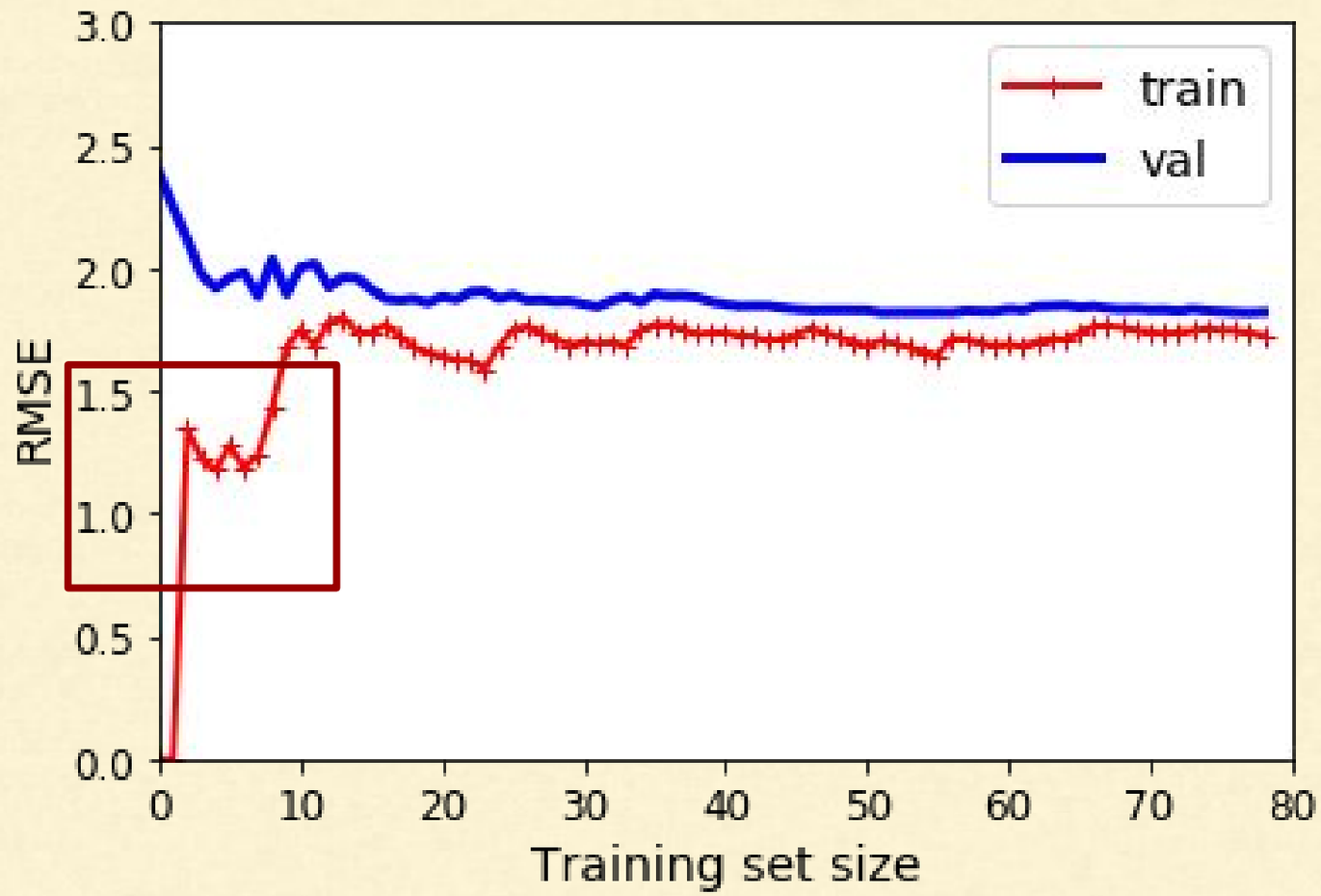


- When there are just one or two instances in the training set
  - The model can fit them perfectly
  - Which is why the curve starts at zero

CLOUD x LAB

Learning Curves

Let's look at performance of the model on the training set

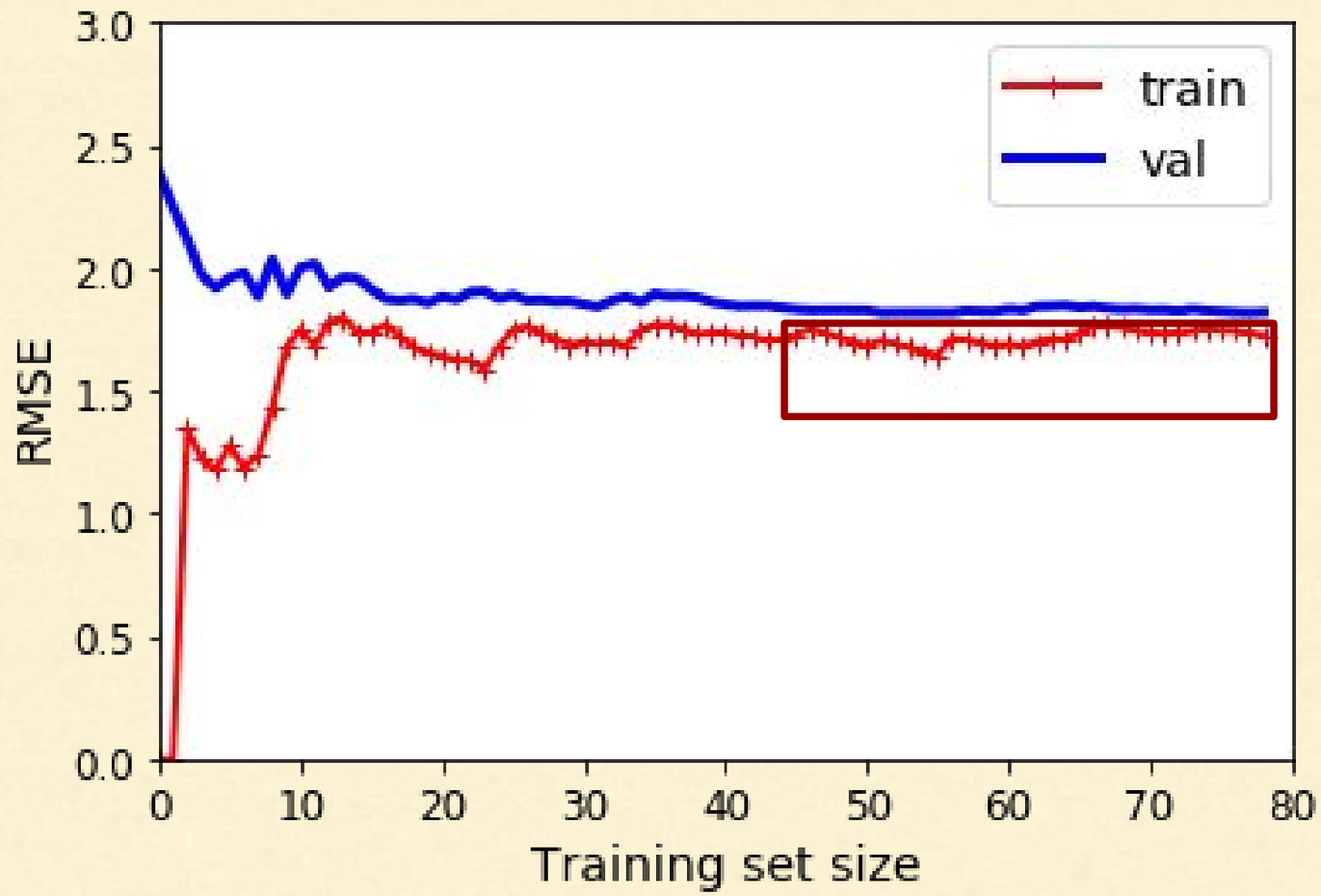


- But as new instances are added to the training set
  - It becomes impossible for the model to fit the training data perfectly
  - Both because the data is noisy and
  - Because it is not linear at all

CLOUD x LAB

Learning Curves

Let's look at performance of the model on the training set

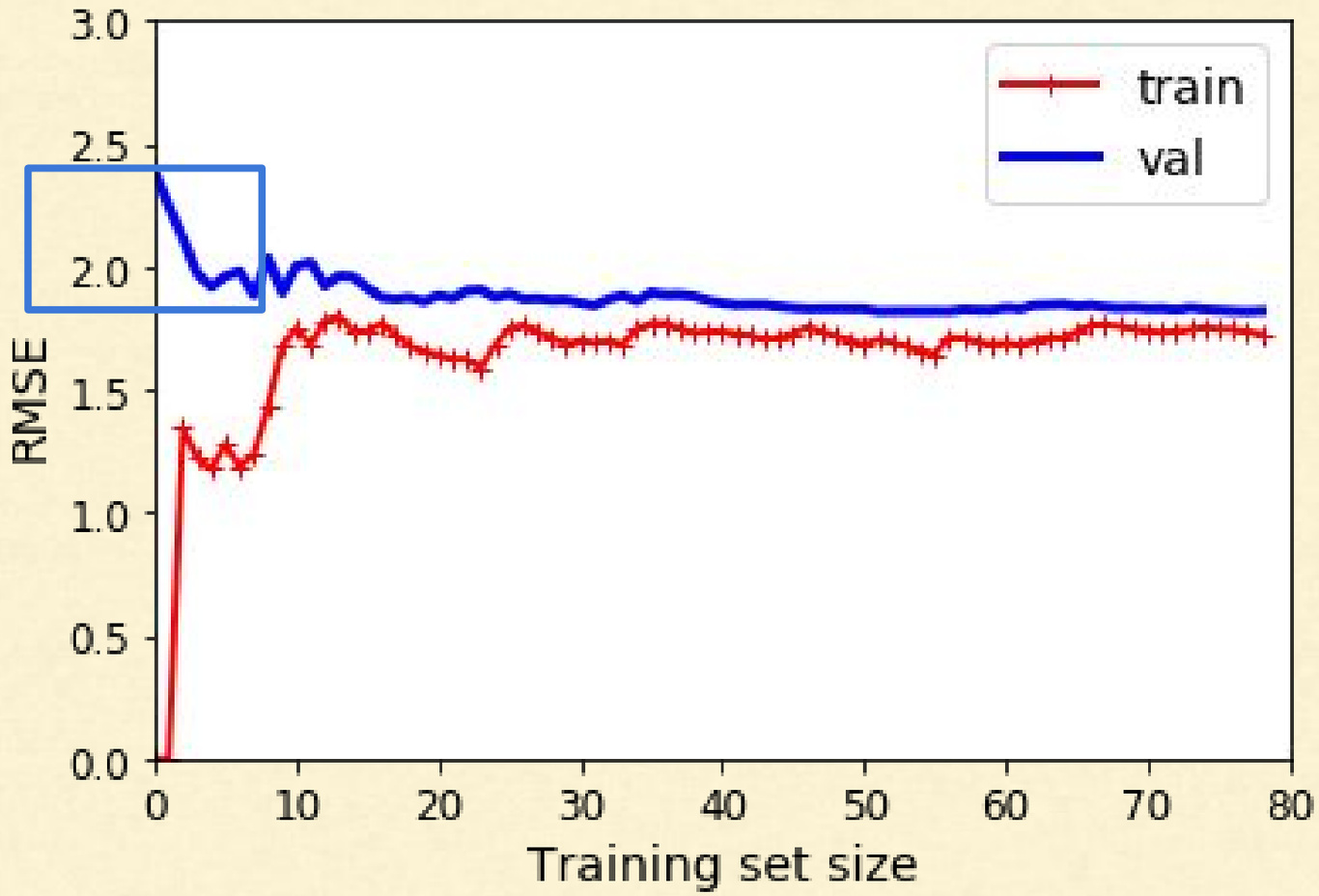


- The error on the training data goes up until it reaches a plateau
  - At that point adding new instances to the training set
  - Doesn't make the error much better or worse

CLOUD x LAB

Learning Curves

Let's look at performance of the model on the validation set

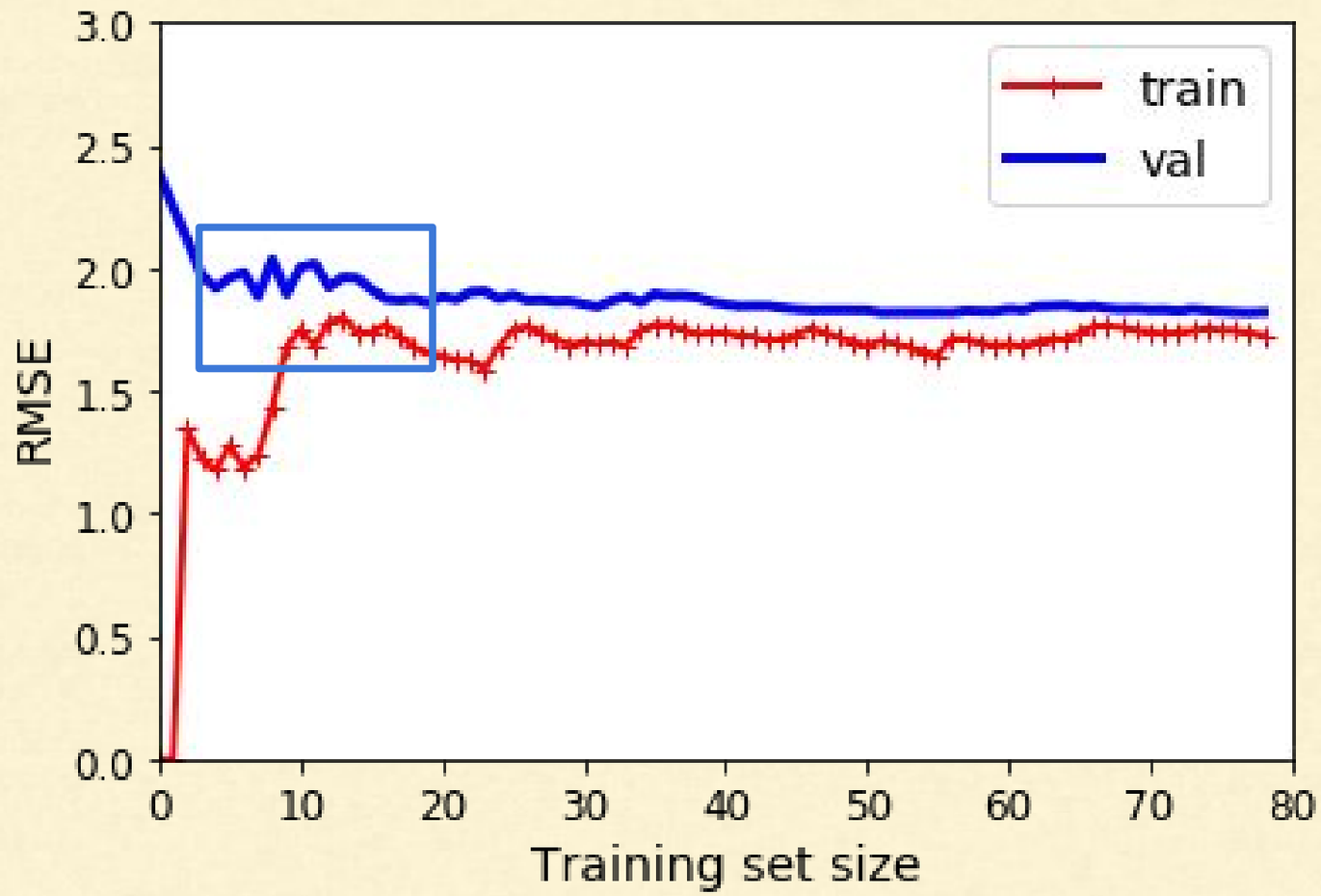


- When the model is trained on very few training instances
  - It is incapable of generalizing properly
  - Which is why the validation error is initially quite big

CLOUD x LAB

Learning Curves

Let's look at performance of the model on the validation set

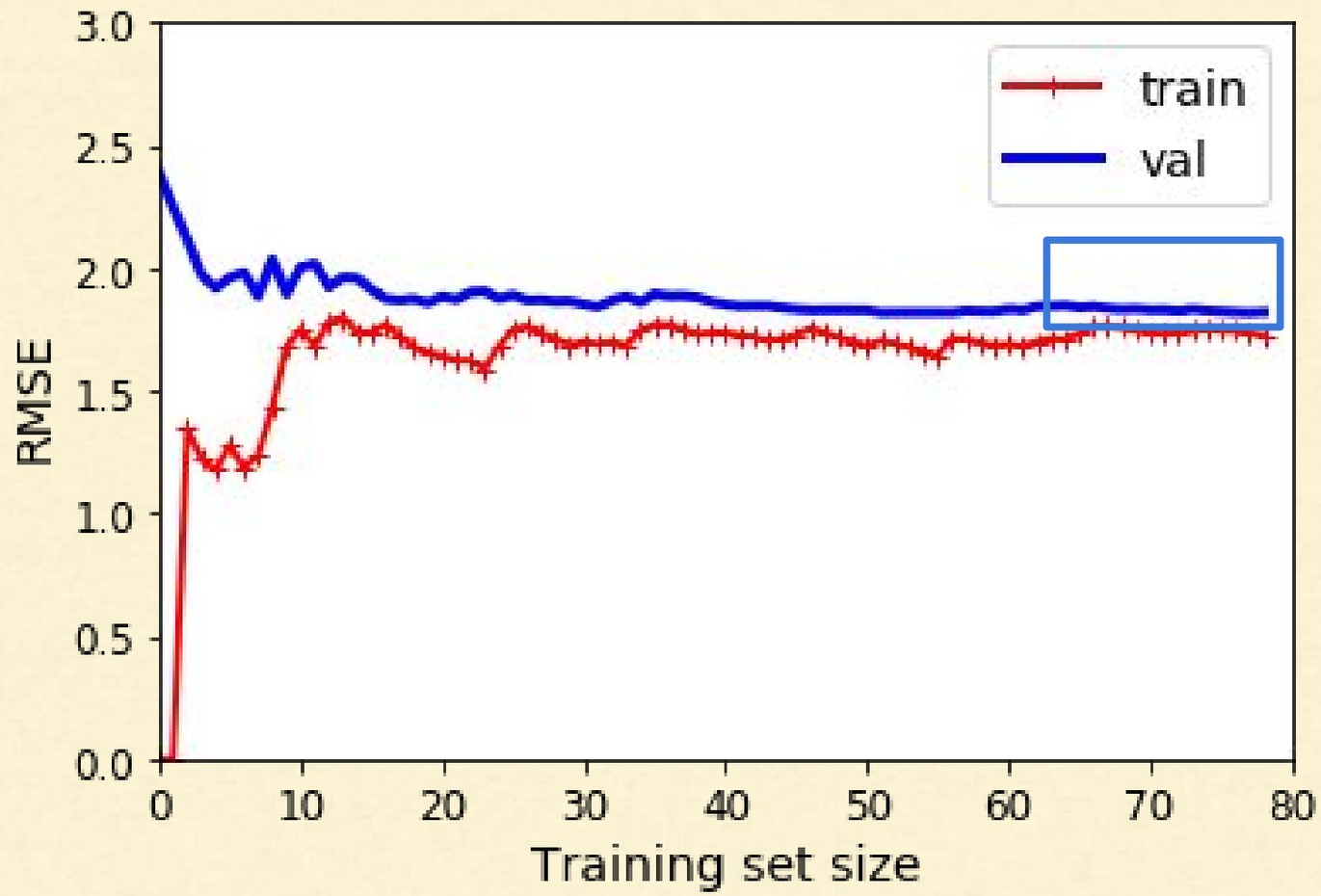


- As the model is shown more training examples
  - It learns and thus the validation error slowly goes down

CLOUD x LAB

Learning Curves

Let's look at performance of the model on the validation set

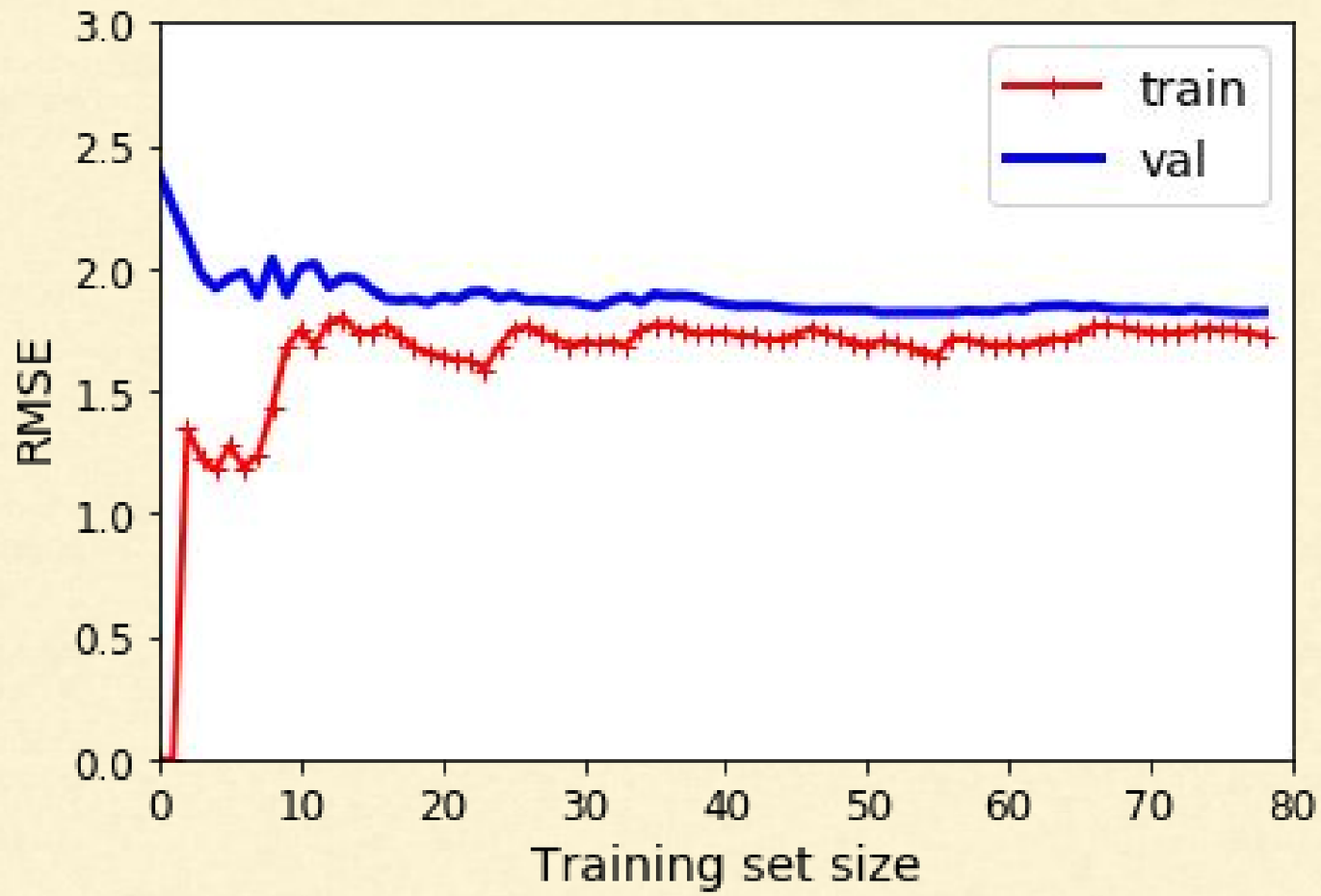


- Since straight line can not do a good job modeling the data
  - So the error ends up at a plateau
  - Very close to the other curve

CLOUD x LAB

Learning Curves

Tip

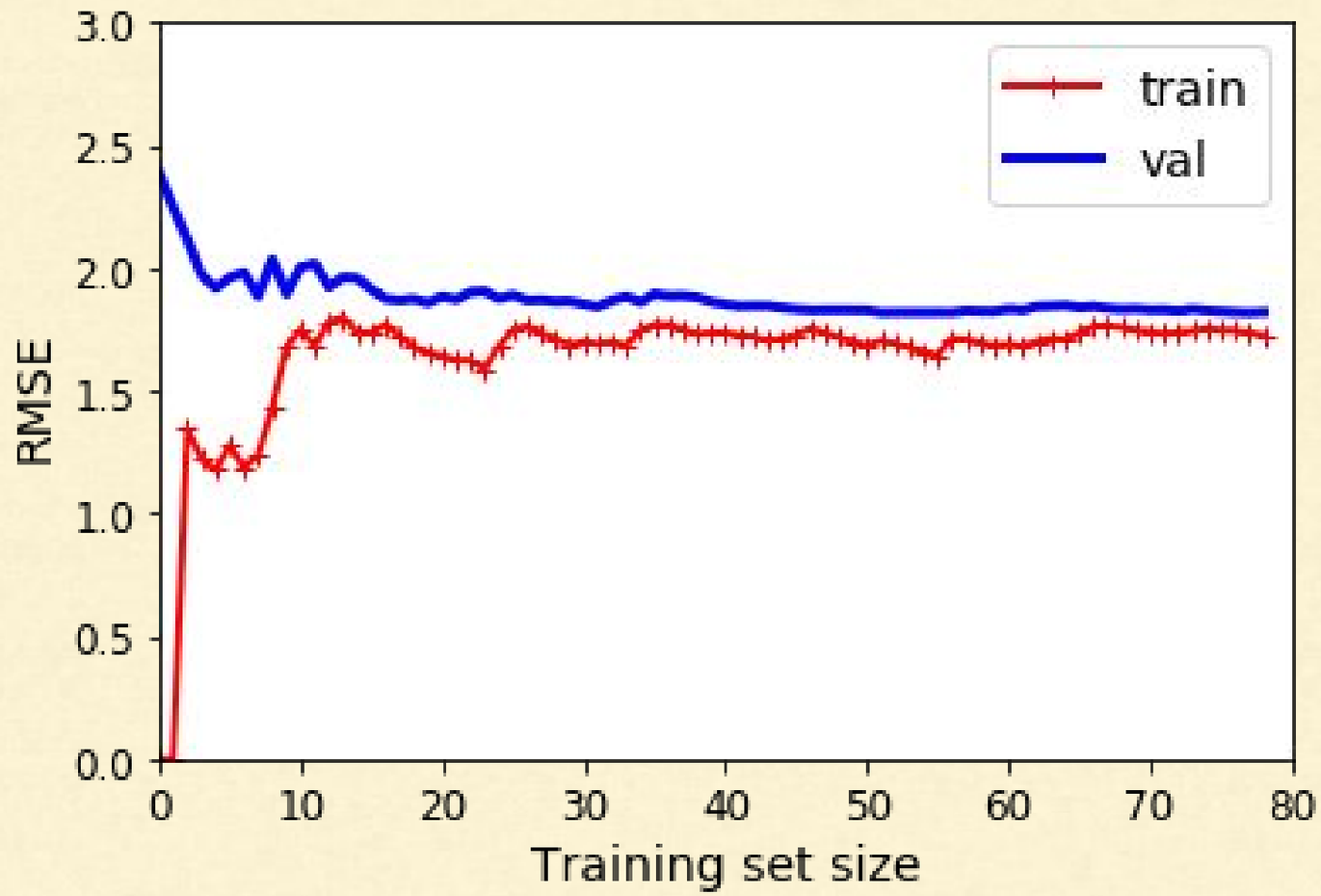


- These learning curves are examples of an underfitting model
  - Both curves have reached a plateau
  - They are close and RMSE is fairly high

CLOUD x LAB

Learning Curves

Tip



- If model is **underfitting** the training data
  - Then adding more training data will not help
  - We need to use a more complex model or
  - Come up with better features

CLOUD x LAB

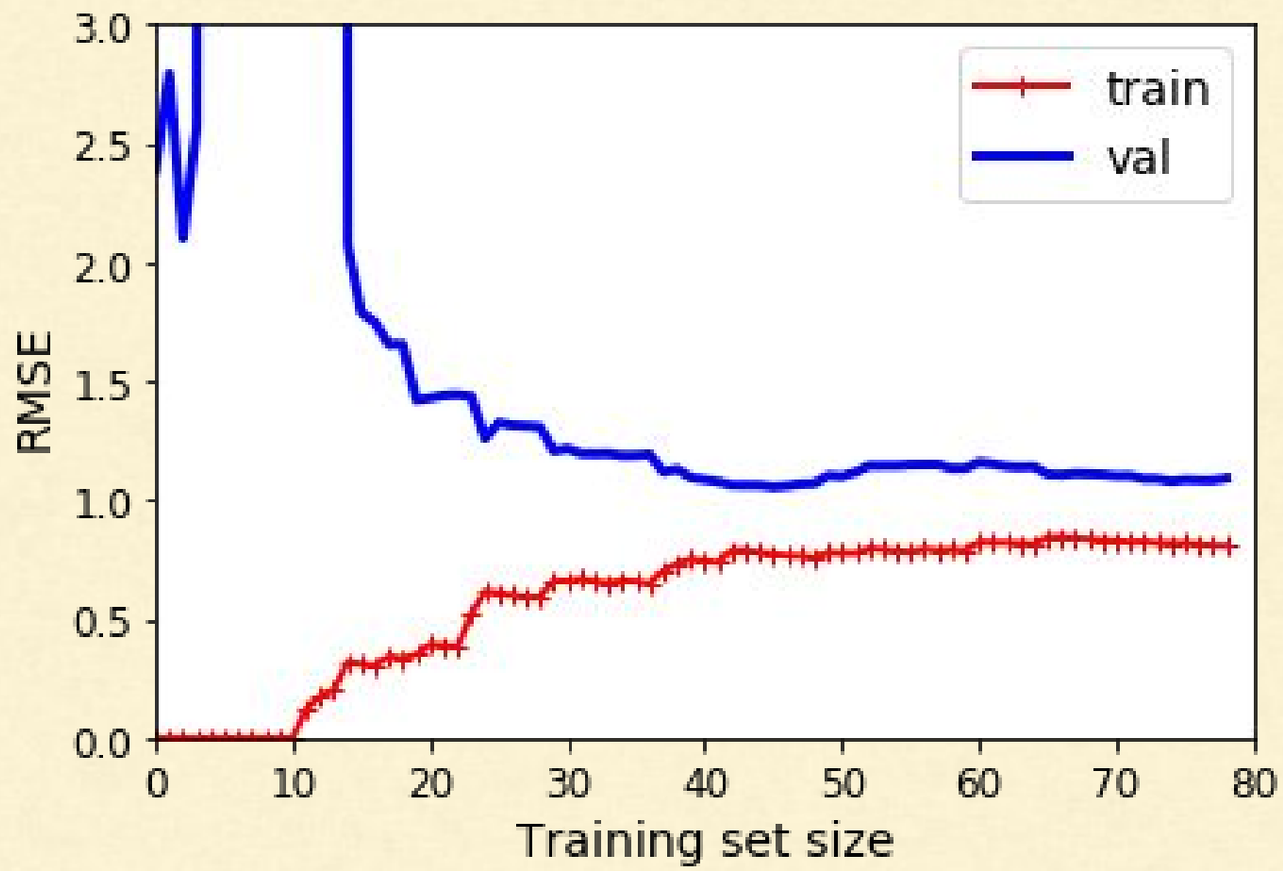
Learning Curves

- Let's plot the learning curves of a 10th-degree polynomial model on the same data

Switch to the notebook for the code

Learning Curves

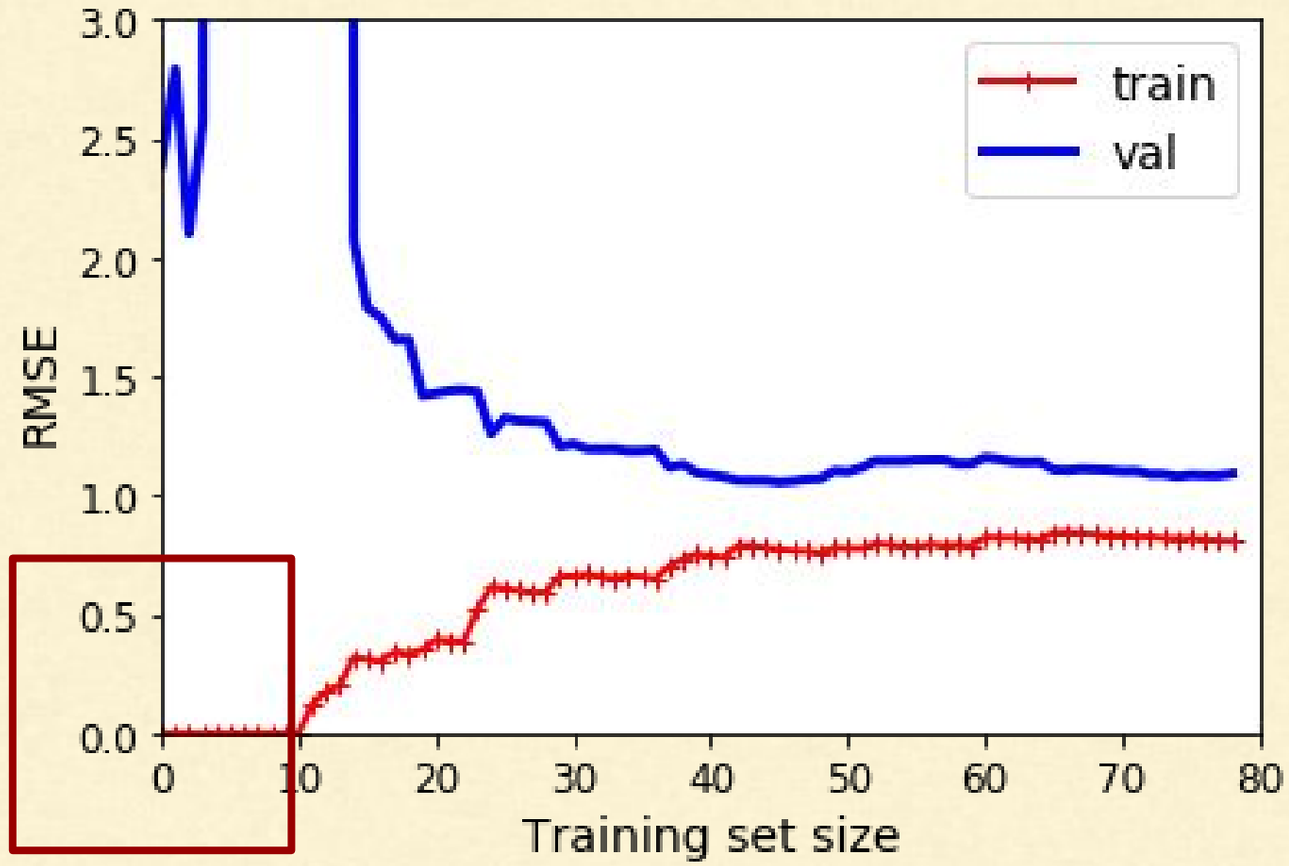
Learning curves of a 10th-degree polynomial model



CLOUD x LAB

Learning Curves

Observations

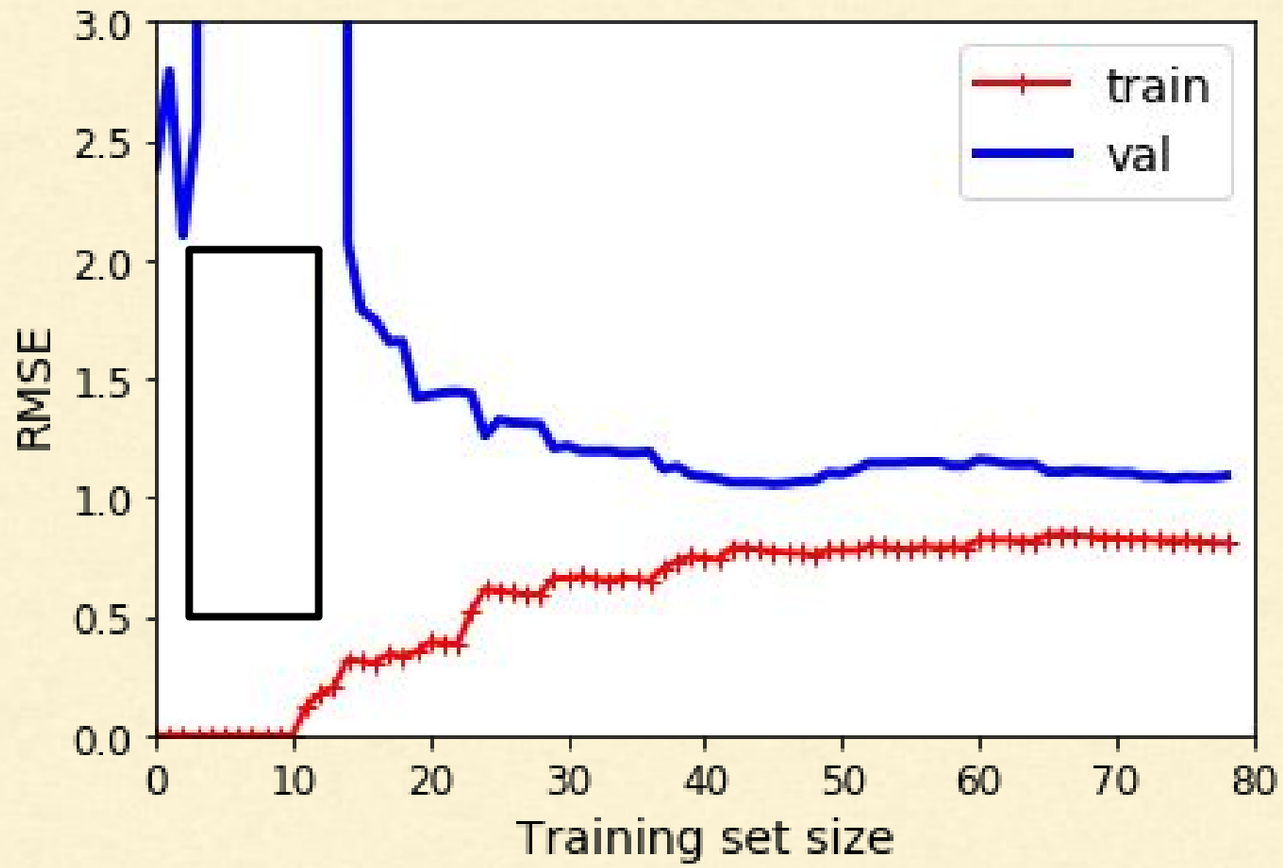


- The error on the training data (~0.5 RMSE) is much lower than with the Linear Regression model (~1.5 - 2 RMSE)

CLOUD x LAB

Learning Curves

Observations

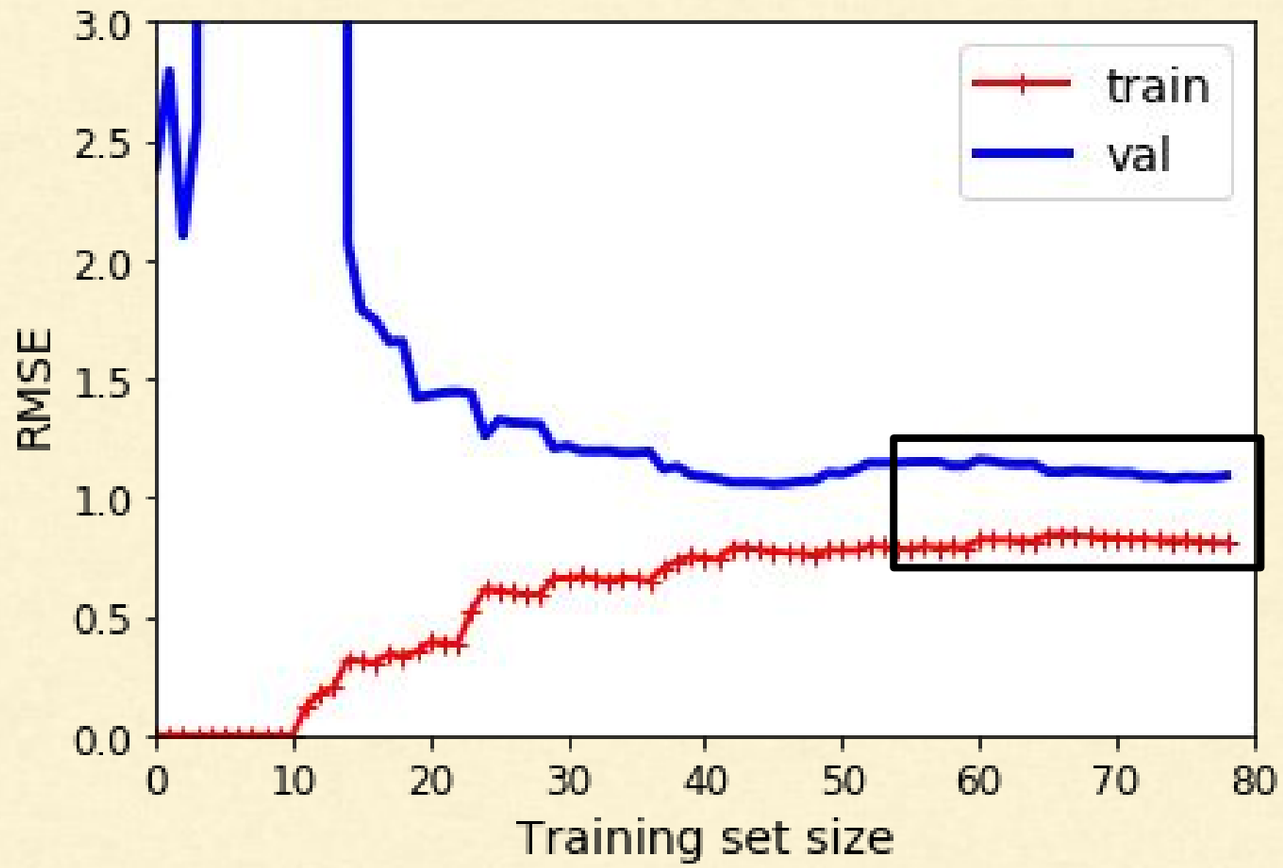


- There is a gap between the curves
  - This means that the model performs significantly better on the
  - Training data than on the validation data
  - Which shows model is overfitting the training set

CLOUD x LAB

Learning Curves

Observations

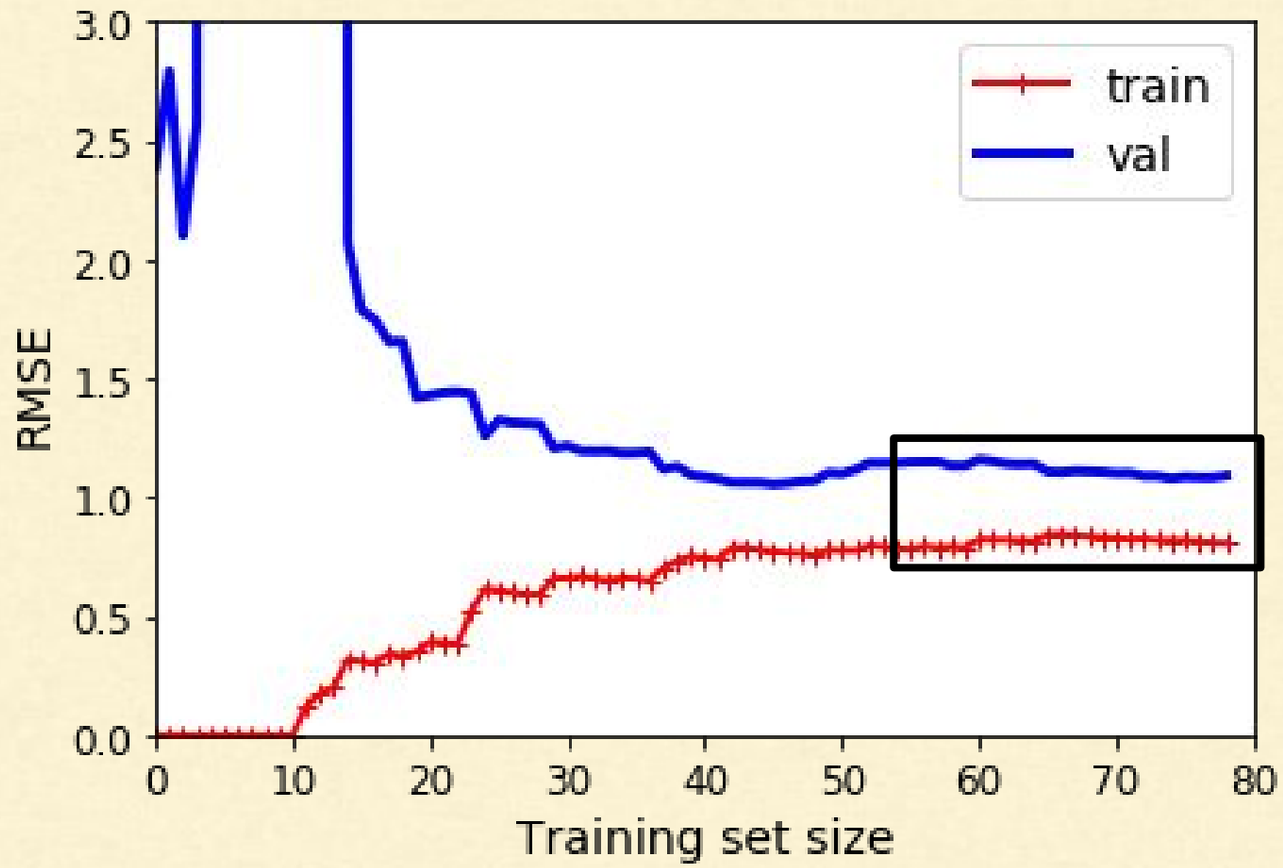


- On a larger training set
  - The two curves continue to get closer

CLOUD x LAB

Learning Curves

Tip



- One way to improve an **overfitting** model is to
  - Feed it more training data
  - Until the validation error reaches the training error

CLOUD x LAB

Learning Curves

The Bias / Variance Tradeoff

- A model's generalization error can be expressed as the sum of three very different errors
  - Bias
  - Variance and
  - Irreducible error

Learning Curves

The Bias / Variance Tradeoff - Bias error

- This 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

Learning Curves

The Bias / Variance Tradeoff - Variance

- A model with many degrees of freedom
  - Such as a high-degree polynomial model is likely to have
  - High variance and thus overfits the training data

Learning Curves

The Bias / Variance Tradeoff - Irreducible error

- This error 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
  - Such as fix
    - The data sources
    - Broken sensors or
    - Detect and remove outliers

CLOUD x LAB

Learning Curves

The Bias / Variance Tradeoff

- Increasing a model's complexity
  - Increases 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

Regularized Linear Models

- As we saw earlier, a good way to reduce overfitting is to
  - Regularize the model - Constraint it
- It is hard for a model to overfit data if it has less degree of freedom
- Example
  - To regularize a polynomial model, reduce the number of polynomial degrees

CLOUD x LAB

Regularized Linear Models

- For a linear model
  - Regularization is achieved by
  - Constraining the weights of the model
- There are different ways to constrain the weights like
  - Ridge Regression
  - Lasso Regression and
  - Elastic Net

CLOUD x LAB

Regularized Linear Models

Let's understand L1 and L2 norms first

Regularized Linear Models

Different Performance Measures

- We've already looked into RMSE (Root Mean Square Error) as a performance measure
- RMSE is generally the preferred performance measure for regression tasks
- But in some cases we may want to use another performance measure
  - Like MAE - **Mean Absolute Error**
  - Also called the **Average Absolute Deviation**
- MAE is particularly useful when there are many outliers

Regularized Linear Models

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m} \left| h(\mathbf{x}^{(i)}) - y^{(i)} \right|$$

Mean Absolute Error

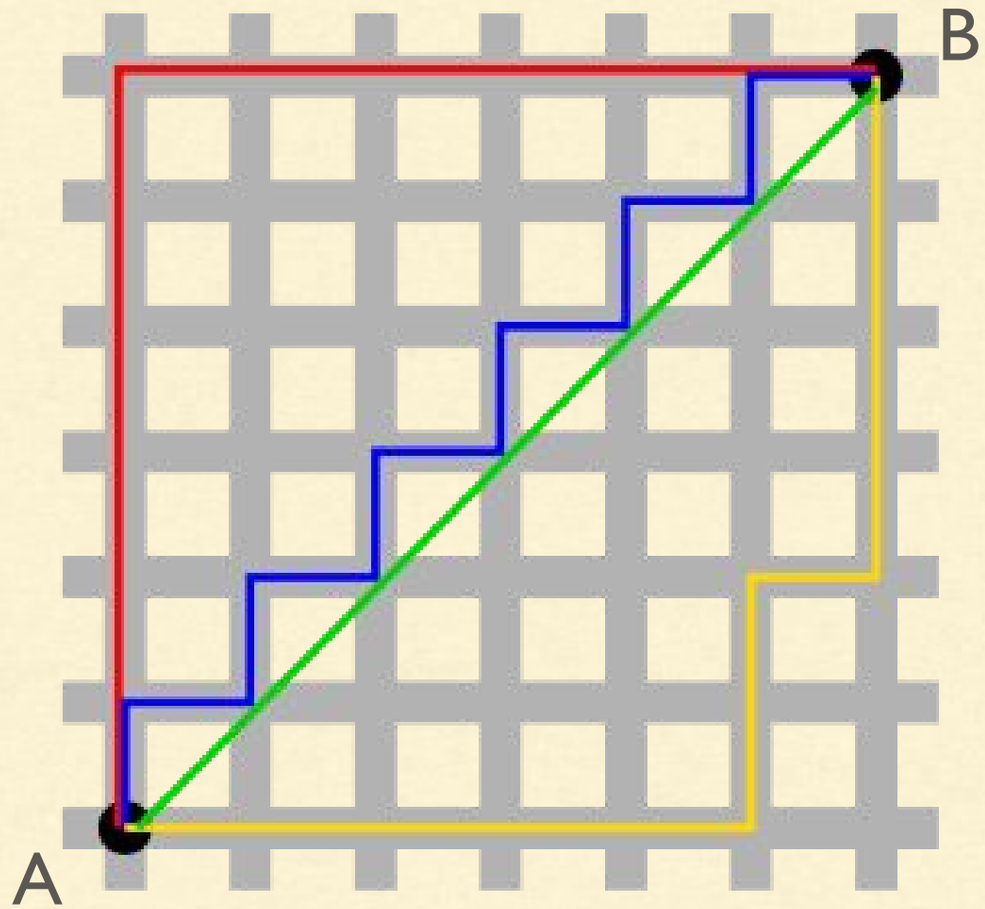
CLOUD x LAB

Regularized Linear Models

- Both RMSE and MAE are ways to measure distances
  - Between two vectors
  - Predictor values vector and target values vector
- There are various distance measures or norm to calculate distance between two vectors
  - Manhattan Norm
  - Euclidean Norm

CLOUD x LAB

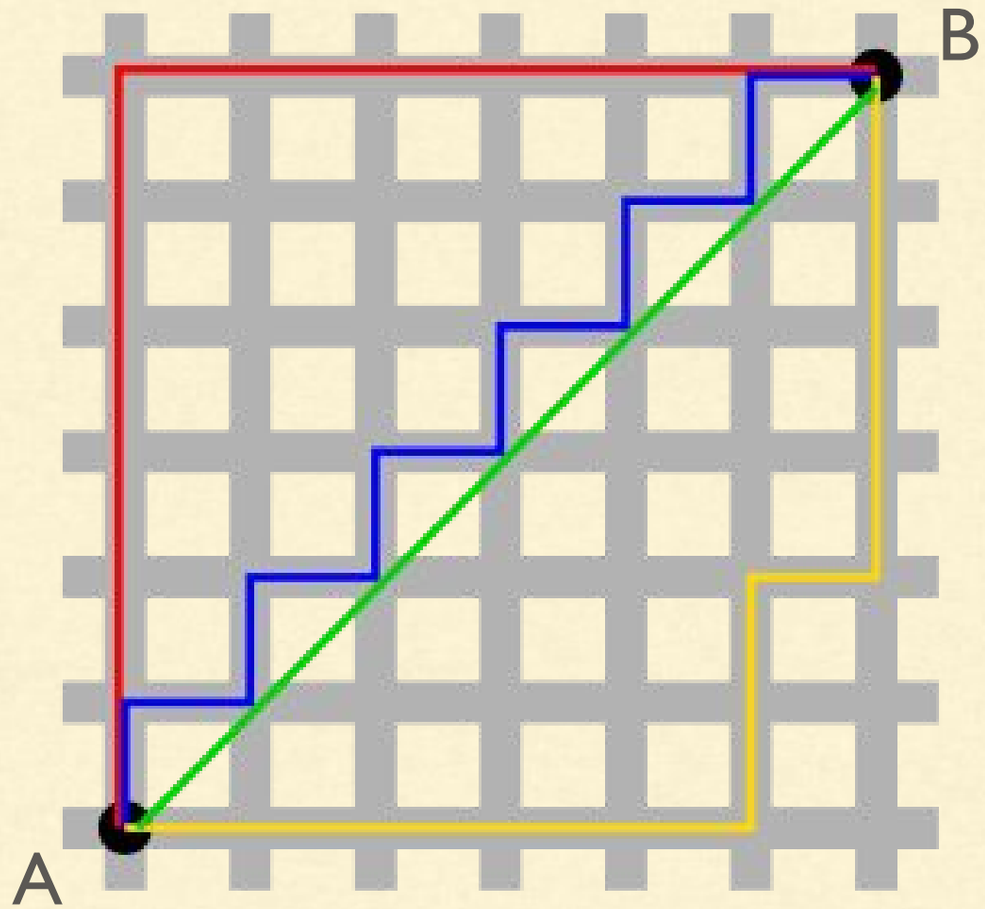
Regularized Linear Models



Different paths between point A and B

CLOUD x LAB

Regularized Linear Models



Different paths between point A and B

- Red, Yellow and Blue paths (If we can only travel along orthogonal city blocks) - Manhattan Distance
- Green path - Euclidean Distance

CLOUD x LAB

Regularized Linear Models

L1 norm

- Computing the sum of absolutes (MAE) corresponds to L1(ℓ_1) norm
- Denoted as
  - $\| \cdot \|_1$
- Also called as Manhattan norm

CLOUD x LAB

Regularized Linear Models

L2 norm

- Computing the root of a sum of squares(RMSE) corresponds to L2(ℓ_2) norm
- Denoted as
  - $\| \cdot \|_2$ or just $\| \cdot \|$
- Also called as Euclidean norm

CLOUD x LAB

Regularized Linear Models

ℓ_k norm

- The ℓ_k norm of a vector **v** containing n elements is defined as

$$\|\mathbf{v}\|_k = (|v_0|^k + |v_1|^k + \cdots + |v_n|^k)^{\frac{1}{k}}$$

- ℓ_0 just gives the cardinality of the vector
  - i.e., the number of elements
- And ℓ_∞ gives the maximum absolute value in the vector

CLOUD x LAB

Regularized Linear Models

ℓ_k norm

- The higher the norm index
  - The more it focuses on large values and
  - Neglects small ones
- This is why the RMSE is
  - More sensitive to outliers than the MAE
  - But when outliers are exponentially rare (like in a bell-shaped curve)
  - The RMSE performs very well and is generally preferred

CLOUD x LAB

Regularized Linear Models

L1 vs L2 Regularization

- L1 can yield sparse models while L2 doesn't
- Sparse models help in feature selection
  - Which features are important and
  - Which features are not or redundant

Regularized Linear Models

Let's understand Ridge, Lasso and Elastic Net now

Regularized Linear Models

Ridge Regression

Ridge Regression

- Ridge Regression is a regularized version of Linear Regression
- A regularization term is added to the cost function

$$\alpha \sum_{i=1}^{n} \theta_i^2$$

Regularization term

CLOUD x LAB

Ridge Regression

- Adding a regularization term to cost function
  - Forces the learning algorithm to not only fit the data
  - But also keep the model weights as small as possible
- Note that
  - Regularization term should only be added to the cost function during training
  - Once the model is trained we evaluate the model's performance using the unregularized performance measure

CLOUD x LAB

Ridge Regression

Question

Why add the regularization term only during the training not while evaluating the model performance during testing?

Answer

- Cost function should have optimization- friendly weights during training
- While the performance measure used for testing should be as close as possible to the final objective
- This is why we evaluate the model's performance using the unregularized performance measure

Ridge Regression

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

Ridge Regression Cost Function

- The hyperparameter "α" controls
  - How much you want to regularize the model
- If α = 0
  - Then Ridge Regression is just Linear Regression

CLOUD x LAB

Ridge Regression

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

Ridge Regression Cost Function

- Note that the bias term θ_0 is not regularized
  - As the sum starts at i = 1, not 0
- We are just constraining the weights of features hence
  - Regularization of bias term is not required

CLOUD x LAB

Ridge Regression

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

Ridge Regression Cost Function

- If **w** is the vector of feature weights (θ_1 to θ_n)
  - Then the regularization term is equal to

$$\frac{1}{2}(\| w \|_2)^2$$

Where $// \cdot //_2$ represents the l_2 (L2) norm of the weight vector

CLOUD x LAB

Ridge Regression

Important point

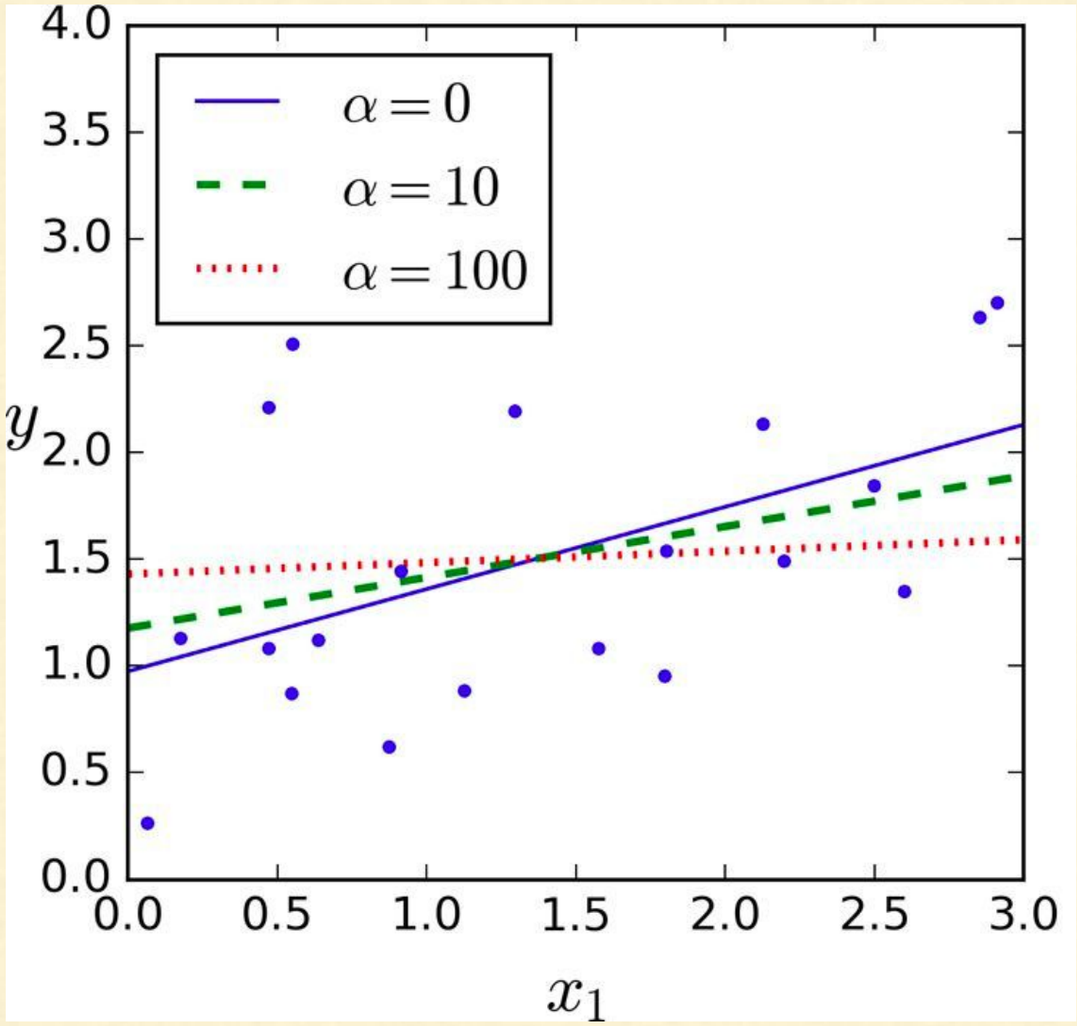
- Please note to
  - Scale the data (e.g., using a StandardScaler)
  - Before performing Ridge Regression
  - As Ridge Regression is sensitive to the scale of input features

CLOUD x LAB

Ridge Regression

Let's train models using Ridge Regression on some linear data using different α value

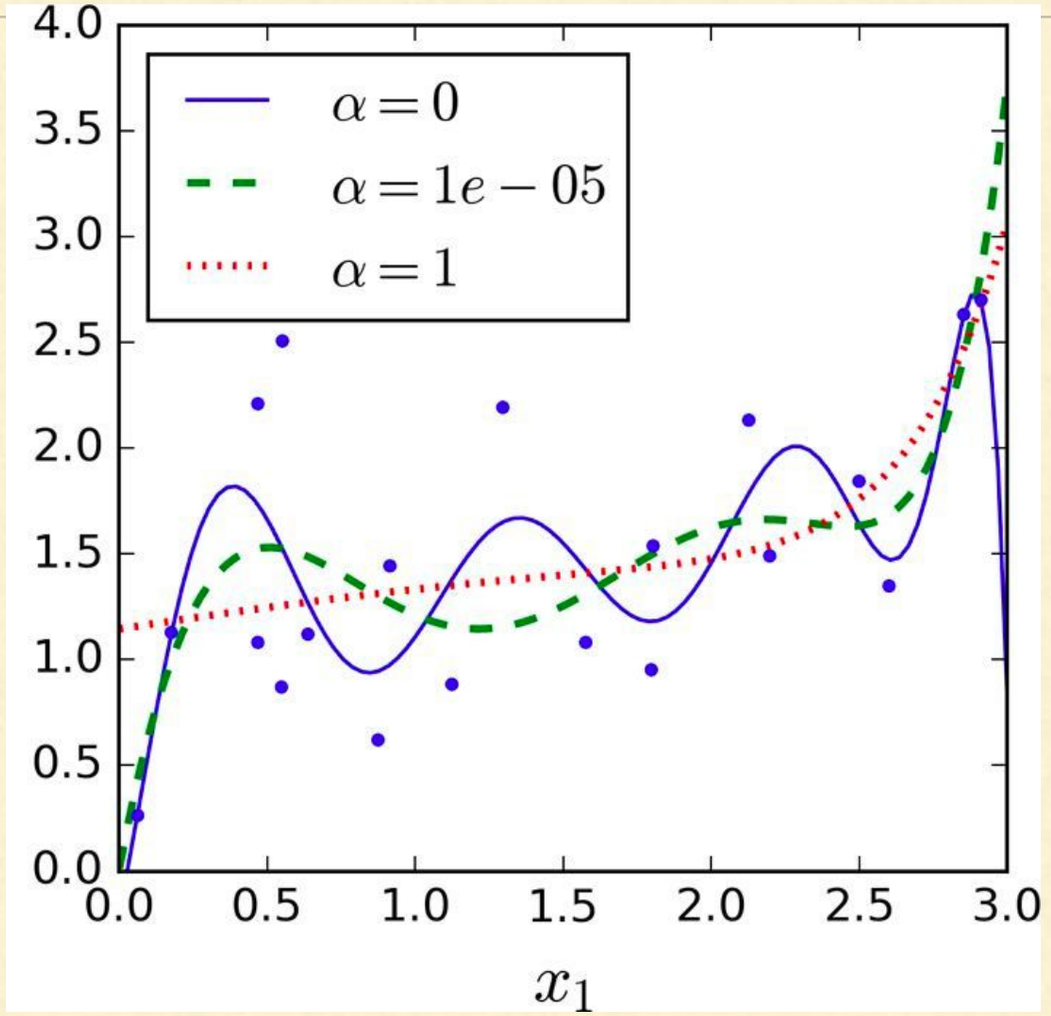
Ridge Regression



Plain Ridge models, leading to linear predictions

CLOUD x LAB

Ridge Regression

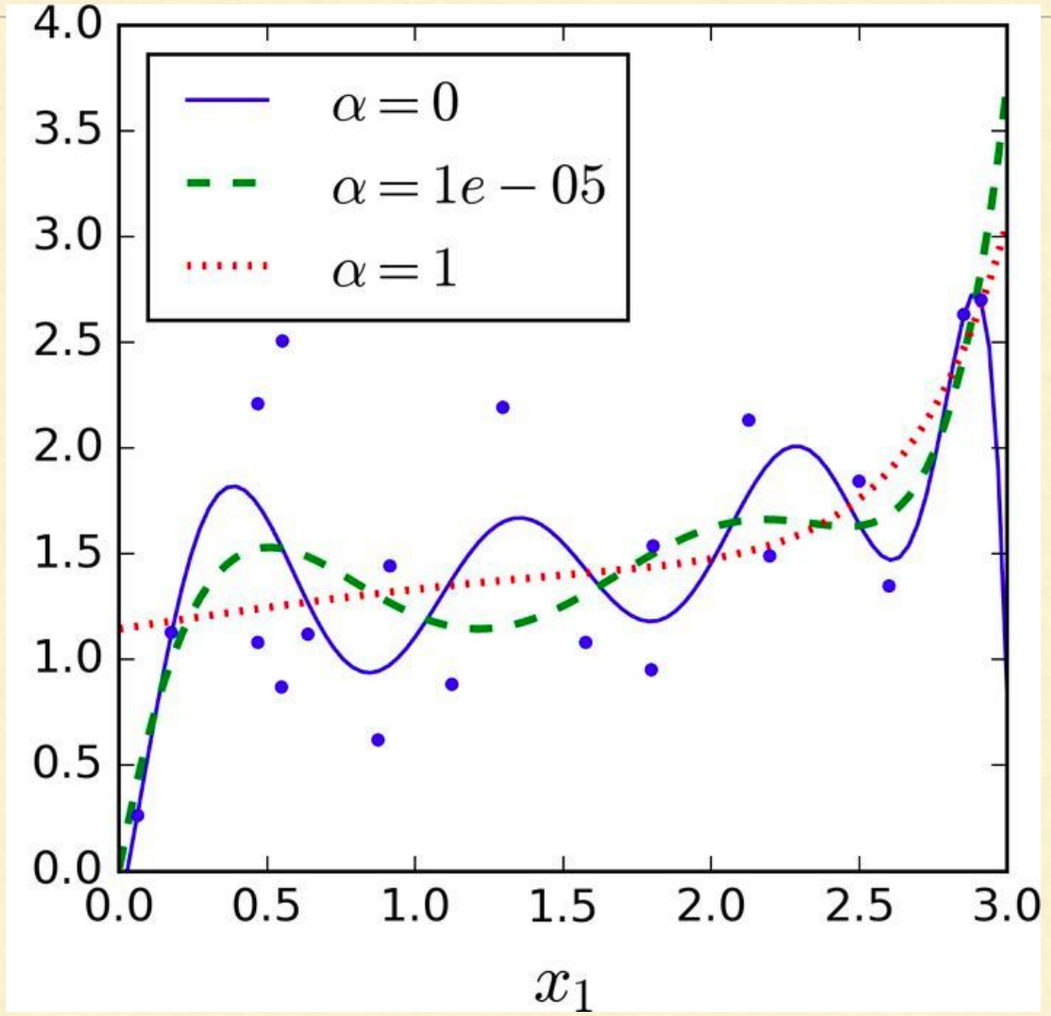


Polynomial Regression with Ridge regularization (See the code in notebook)

- Data is first expanded using **PolynomialFeatures**(degree=10)
- Then scaled using a **StandardScaler**
- Finally the Ridge models are applied to the resulting features

CLOUD x LAB

Ridge Regression



Polynomial Regression with Ridge regularization

- Increasing α leads to
  - flatter - less extreme, more reasonable predictions
  - This reduces the model's variance but increases its bias

CLOUD x LAB

Ridge Regression

Implementing Ridge Regression

- Like Linear Regression we can perform Ridge Regression either
  - By computing a closed-form equation or
  - By performing Gradient Descent

Ridge Regression

Implementing Ridge Regression - closed-form Solution

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

- Where A is the
  - n * n identity matrix
  - except with a 0 in the top-left cell - The bias term

CLOUD x LAB

Ridge Regression

Implementing Ridge Regression - closed-form Solution

$$I_1 = [1],\ I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},\ \cdots,\ I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Identity matrices with ones on the main diagonals and zeros elsewhere

CLOUD x LAB

Ridge Regression

Ridge Regression with Scikit-Learn using a closed-form solution

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
```

Output-

```
array([[ 1.55071465]])
```


CLOUD x LAB

Ridge Regression

Ridge Regression with Scikit-Learn using Stochastic Gradient Descent

```
>>> sgd_reg = SGDRegressor(penalty="l2", random_state=42)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
```

- The penalty hyperparameter sets the type of regularization term to use
- Specifying "l2" (L2) means
  - We want SGD to add a regularization term to the cost function
  - Equal to half the square of the l2 norm of the weight vector

CLOUD x LAB

Ridge Regression

Ridge Regression with Scikit-Learn using Stochastic Gradient Descent

```
>>> sgd_reg = SGDRegressor(penalty="l2", random_state=42)
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
```

Output-

```
array([ 1.13500145])
```


CLOUD x LAB

Regularized Linear Models

Lasso Regression

Lasso Regression

- Least Absolute Shrinkage and Selection Operator Regression
  - Like Ridge regression, a regularization term is added to the cost function
  - But it uses the L1 norm of the weight vector instead of half the square of the L2 norm

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

Lasso Regression cost function

CLOUD x LAB

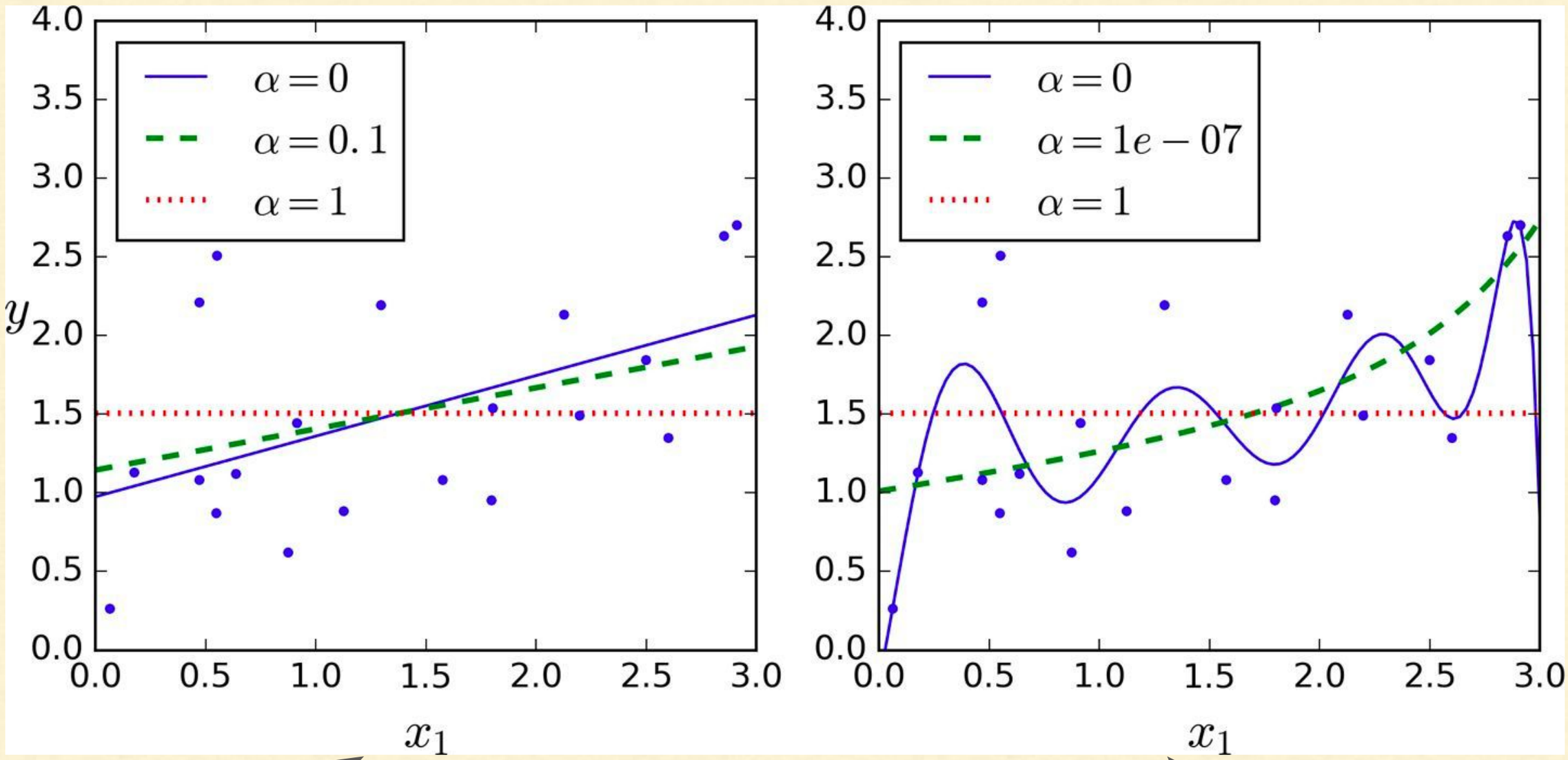
Lasso Regression

Let's train models using Lasso Regression on some linear data using different α value

(See the code in notebook)

Lasso Regression

Lasso Regression for Different Values of Alpha

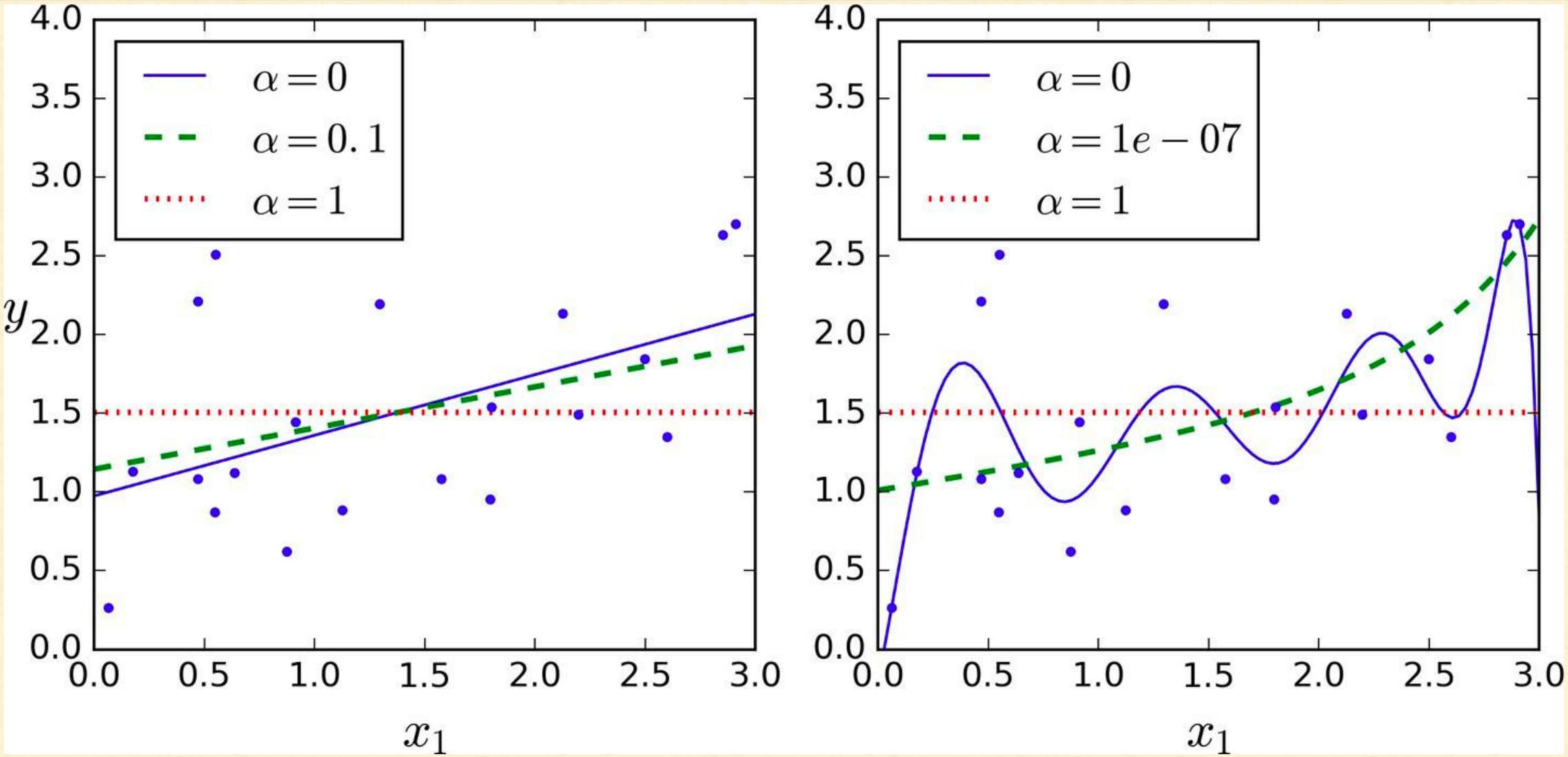


Plain Linear Regression

Polynomial Regression

CLOUD x LAB

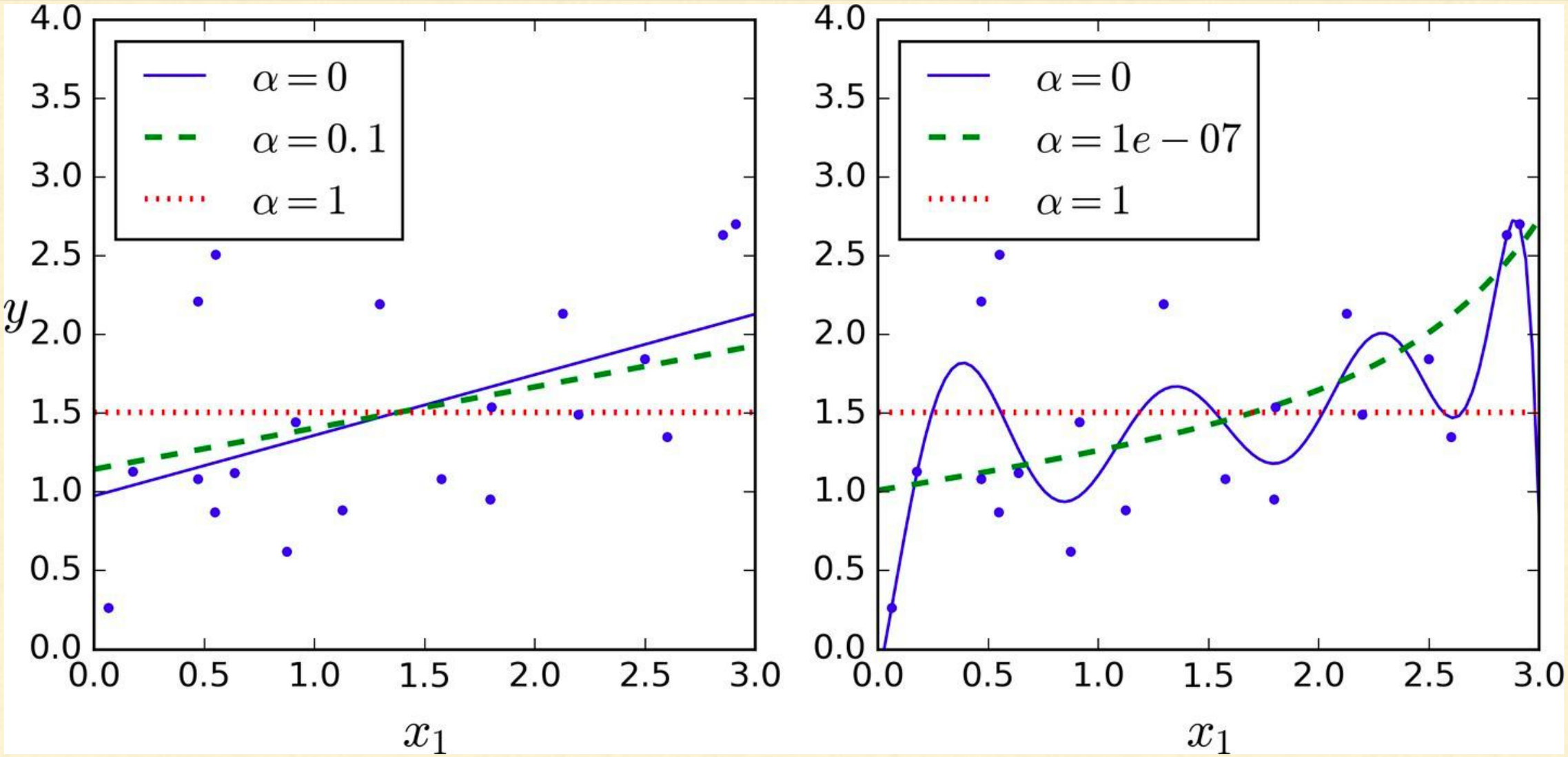
Lasso Regression



- An important characteristic of Lasso Regression
  - It completely eliminates the weights of the least important features
  - Sets them to zero

CLOUD x LAB

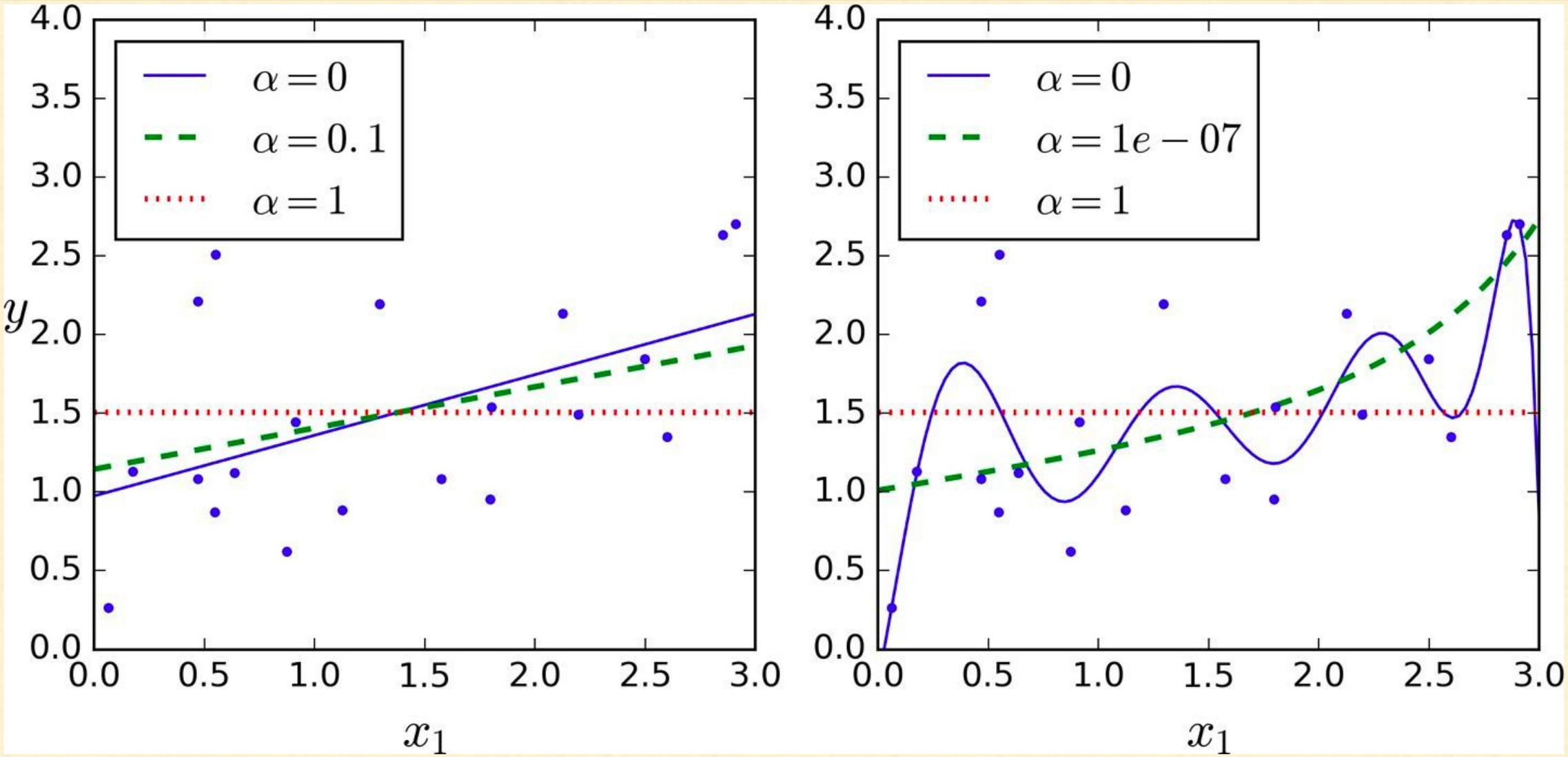
Lasso Regression



- Lasso Regression automatically performs feature selection and
  - Outputs a sparse model
  - i.e, the model with few nonzero feature weights

CLOUD x LAB

Lasso Regression



- The dashed green line (with $\alpha = 10^{-7}$) in the right plot
  - Looks quadratic, almost linear
  - All the weights for the high-degree polynomial features are equal to zero

CLOUD x LAB

Lasso Regression

Lasso Regression using Scikit-Learn

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
```

Output-

```
array([ 1.53788174])
```


CLOUD x LAB

Regularized Linear Models

Elastic Net

Elastic Net

- Elastic Net is middle ground
  - Between Ridge Regression and Lasso Regression
- The regularization term is a
  - Simple mix of both Ridge and Lasso's regularization terms

Elastic Net cost function

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

Lasso's regularization term

Ridge regularization term

CLOUD x LAB

Elastic Net

- When r = 0
  - Elastic Net is equivalent to Ridge Regression

Elastic Net cost function

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

When r=0

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

Ridge Regression cost function

CLOUD x LAB

Elastic Net

- When r = 1
  - Elastic Net is equivalent to Lasso Regression

Elastic Net cost function

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

When r=1

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

Lasso Regression cost function

CLOUD x LAB

Elastic Net

So when should we use Linear Regression, Ridge, Lasso, or Elastic Net?

Elastic Net

Linear Regression, Ridge, Lasso, or Elastic Net

- We should avoid plain Linear Regression
  - Always preferable to have at least
  - A little bit of regularization
- Ridge is a good default
- But if we think **only few features** are useful
  - We should prefer Lasso or
  - Elastic Net
  - As they reduce the useless features' weights down to zero

CLOUD x LAB

Elastic Net

Linear Regression, Ridge, Lasso, or Elastic Net

- 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

Elastic Net

Implement Elastic Net using Scikit-Learn

```
>>> from sklearn.linear_model import ElasticNet
>>> # l1_ratio corresponds to the mix ratio r
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
```

Output-

```
array([ 1.54333232])
```


CLOUD x LAB

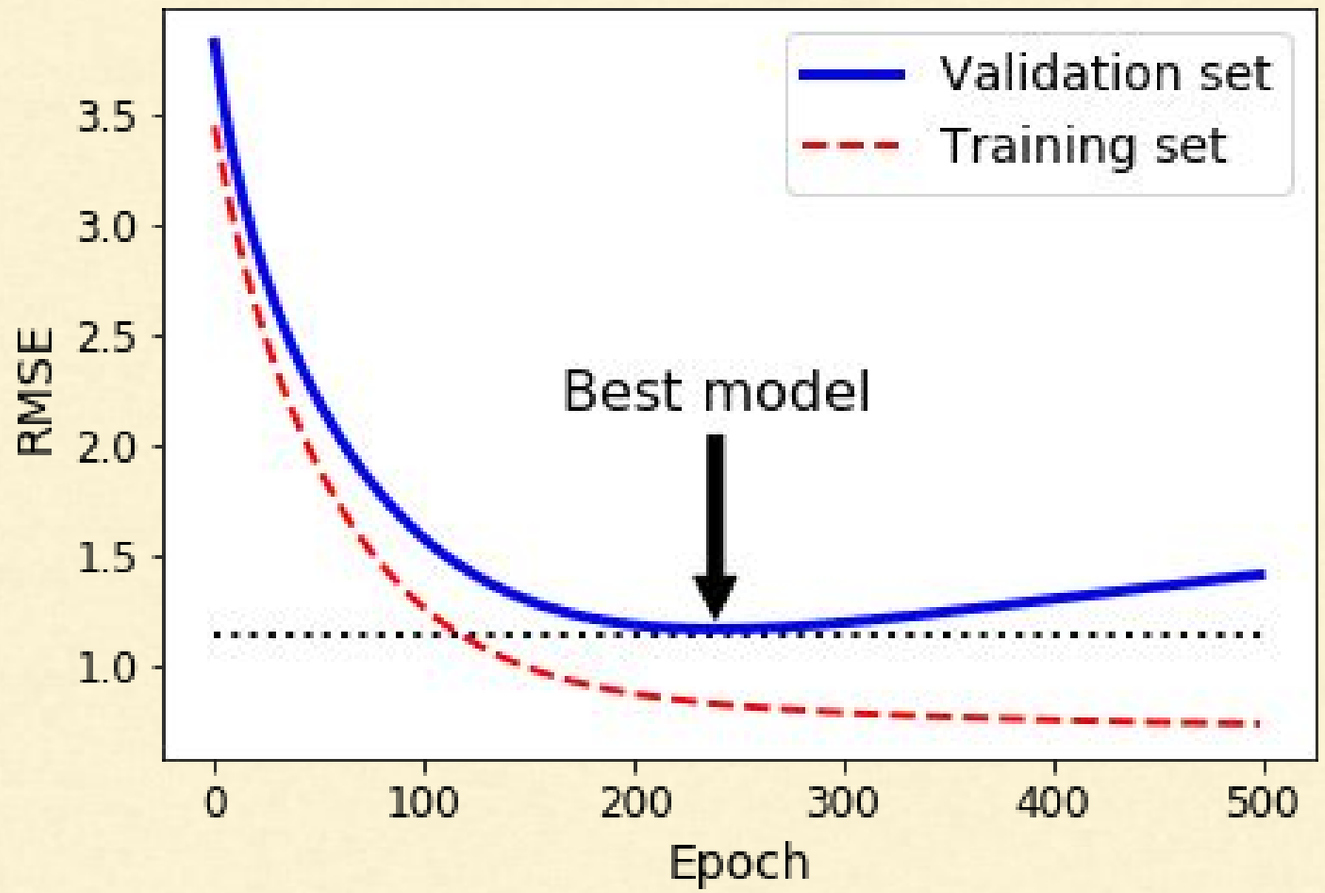
Regularized Linear Models

Early Stopping

Early Stopping

- A 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**

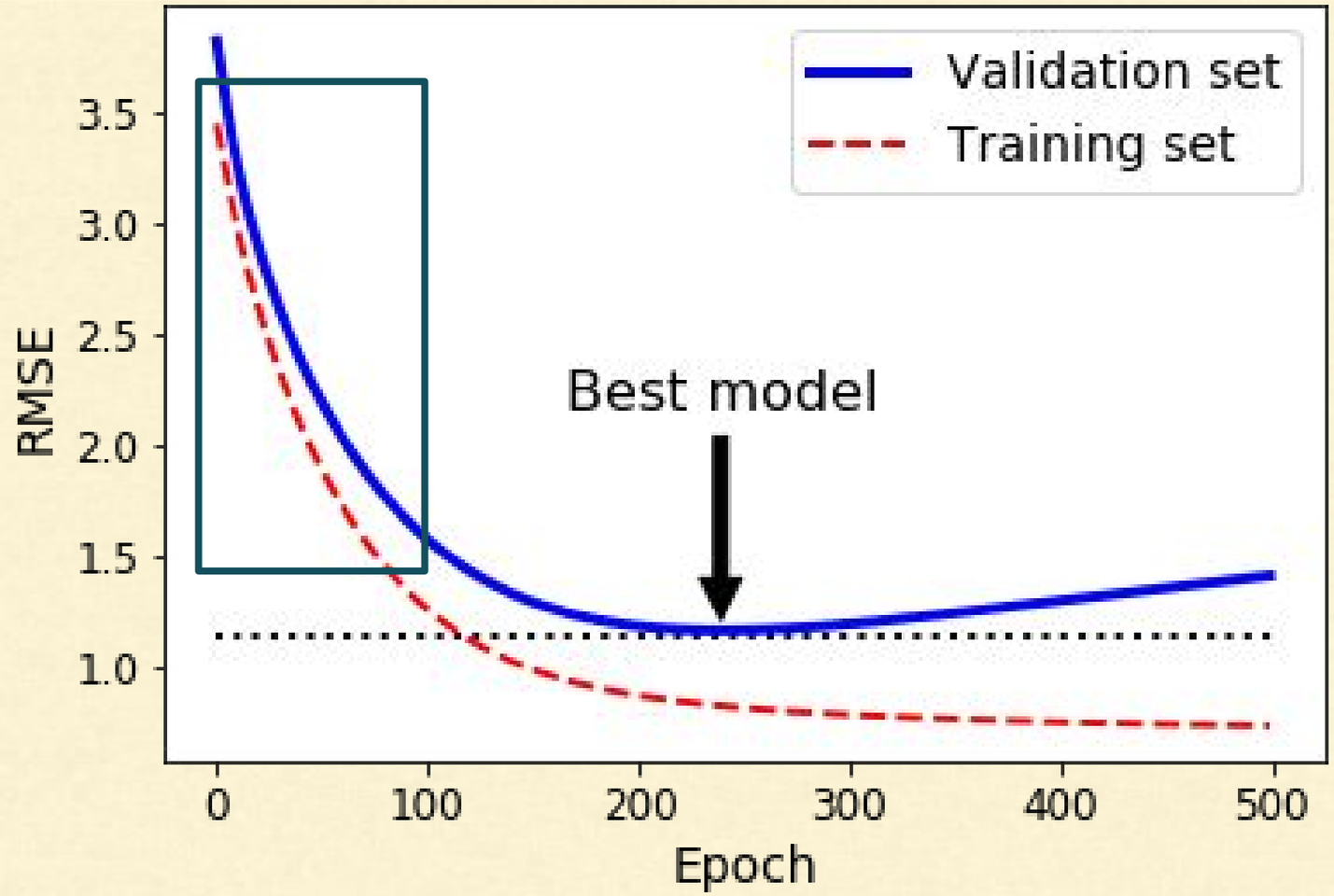
Early Stopping



High-degree Polynomial Regression model trained using Batch Gradient Descent
(See the code in the notebook)

CLOUD x LAB

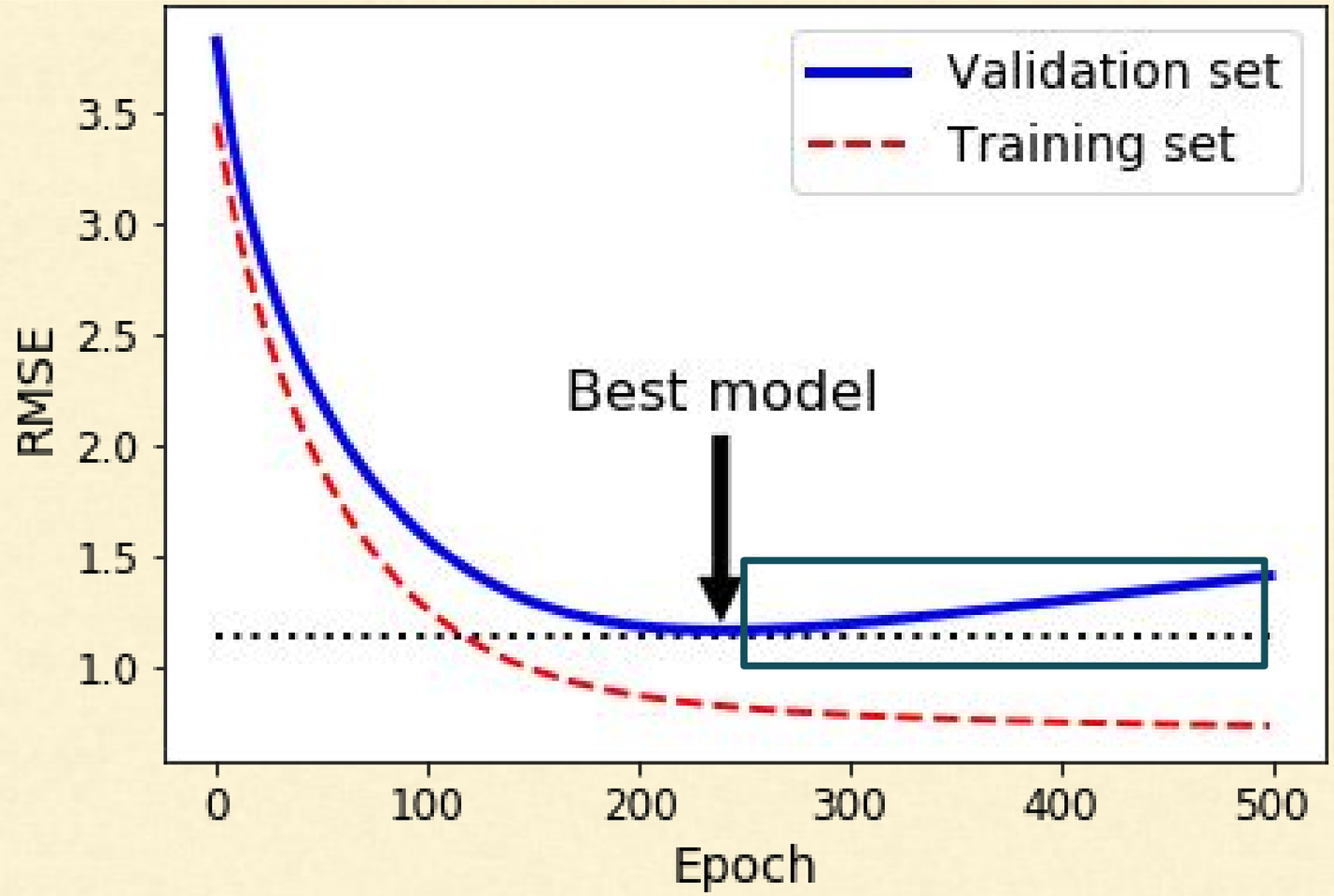
Early Stopping



- As the epochs increases the algorithm learns and
  - Its prediction error (RMSE) on the training set goes down and
  - So does its prediction error on the validation set

CLOUD x LAB

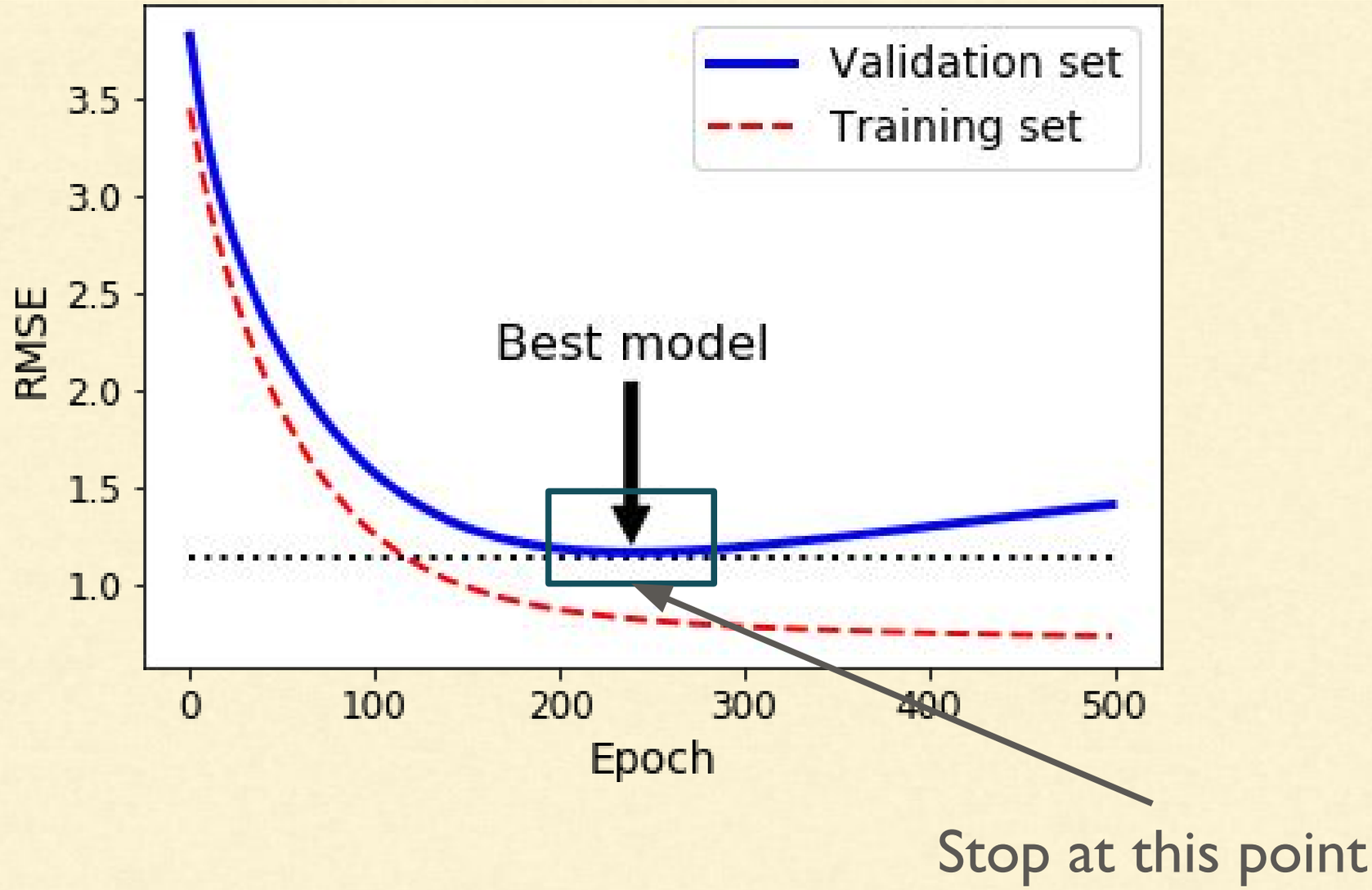
Early Stopping



- After a while the
  - Validation error starts increasing
- This indicates that the
  - Model has started to overfit the training data

CLOUD x LAB

Early Stopping



- With early stopping
  - We just stop training as soon as the
  - Validation error reaches the minimum

CLOUD x LAB

Early Stopping

Tip

- In previous example of Batch Gradient Descent
  - The curve was smooth
  - That is why we could locate the minimum easily
- With Stochastic and Mini-batch Gradient Descent
  - The curves are not so smooth and
  - It may be hard to know whether
  - We have reached the minimum or not

CLOUD x LAB

Early Stopping

Tip

- One solution is to
  - Stop only after the validation error
  - Has been above the minimum for some time
  - Then roll back the model parameters to the point
  - Where the validation error was at a minimum

Early Stopping

See the code in notebook for basic implementation of early stopping

Logistic Regression

Logistic Regression

- Logistic Regression is a binary classifier
- It is used to
  - Estimate the probability that
  - An instance belongs to a particular class

Logistic Regression

Positive and Negative Class

- Example
  - What is the probability that this email is spam?
- If the estimated probability is
  - Greater than 50%
  - Then the instance belong to that class
  - Positive class is labeled as “1”
- Else it predicts that
  - Instance does not belongs to that class
  - Negative class is labeled as “0”

CLOUD x LAB

Logistic Regression

Estimating Probabilities

CLOUD x LAB

Estimating Probabilities

- Just like a Linear Regression model, a Logistic Regression model
  - Computes the weighted sum of
  - The input features plus a bias term
- Unlike Linear Regression model, a Logistic Regression model
  - Outputs the logistic of result

Estimating Probabilities

Logistic Regression Model Estimated Probability

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

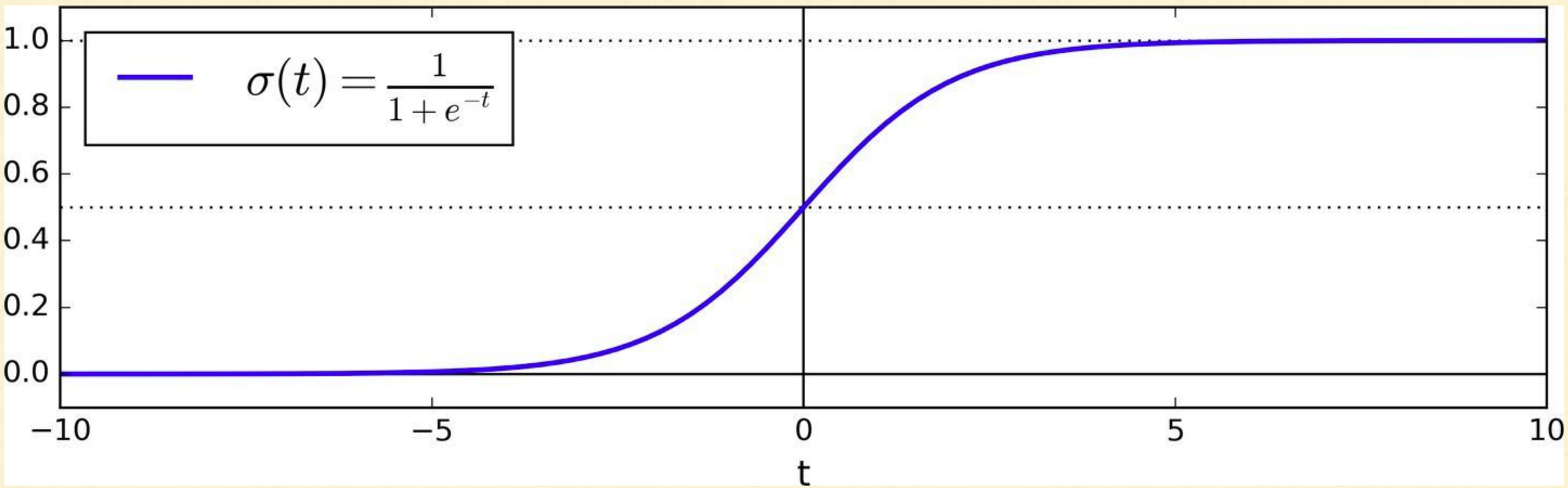
σ -The logistic or Sigmoid function

CLOUD x LAB

Estimating Probabilities

Logistic or Sigmoid Function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



Outputs a number between 0 and 1

CLOUD x LAB

Estimating Probabilities

Logistic Regression Model Estimated Probability

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

- Once the probability $\hat{p}$ (p-hat) is estimated
  - Make predictions $\hat{y}$ (y-hat) using below equation

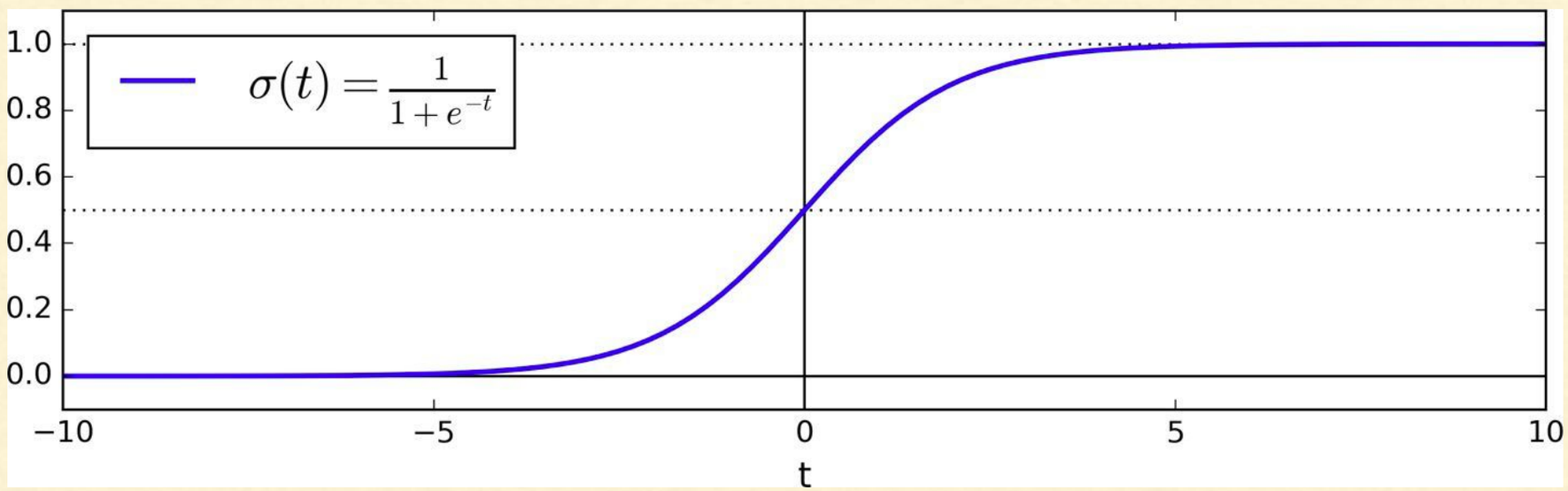
$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$

CLOUD x LAB

Estimating Probabilities

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

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$



- $\sigma(t) < 0.5$ when $t < 0$ and
- $\sigma(t) \geq 0.5$ when $t \geq 0$
- So a Logistic Regression model predicts
  - 1 if $\theta^T \cdot \mathbf{X}$ is positive and
  - 0 if it is negative

CLOUD x LAB

Logistic Regression

Training and Cost Function

CLOUD x LAB

Training and Cost Function

- Let's understand how to train a Logistic Regression model
- The objective of training is to
  - Set the parameter vector θ so that model estimates
  - High probabilities for positive instances (y = 1) and
  - Low probabilities for negative instances (y = 0)

Training and Cost Function

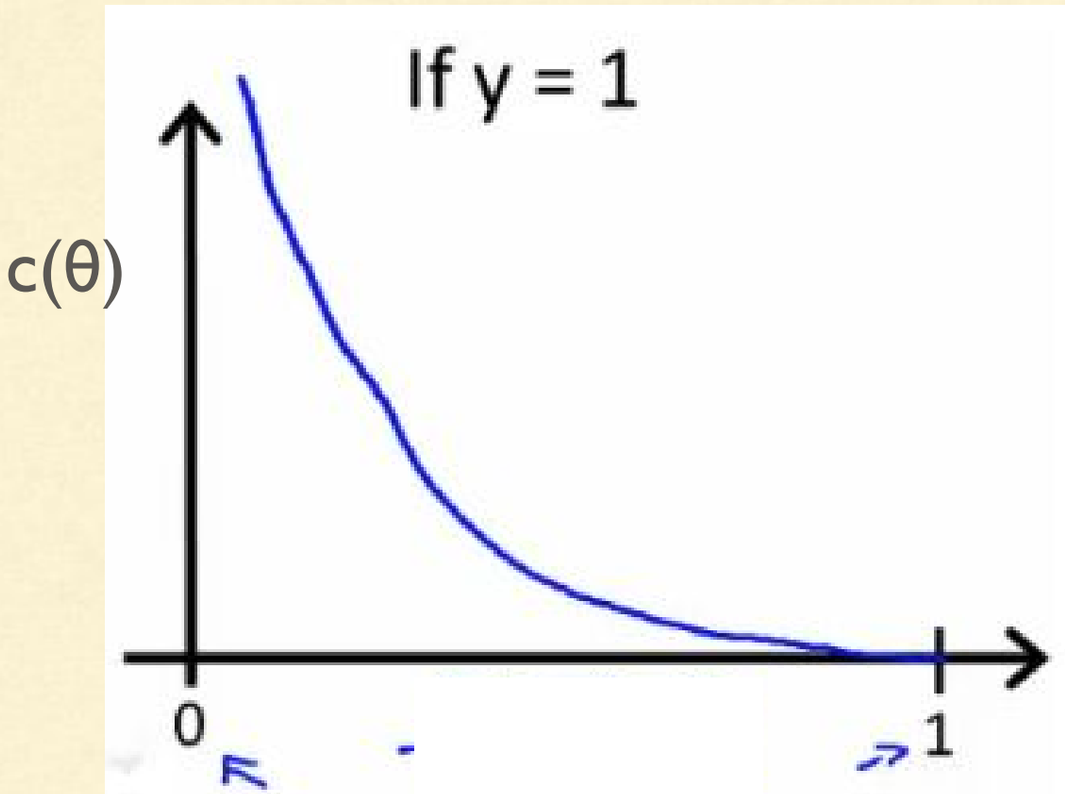
$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1 - \hat{p}) & \text{if } y = 0. \end{cases}$$

Cost function of a single training instance

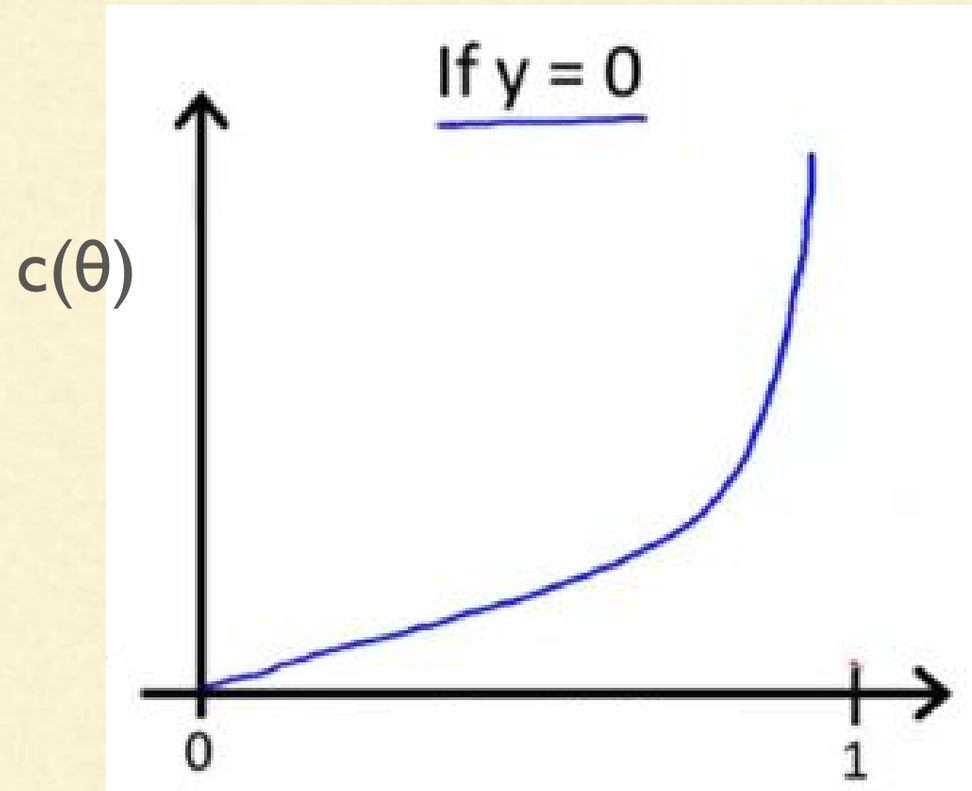
Note that in case of classification y can have only two values 0 and 1

CLOUD x LAB

Training and Cost Function



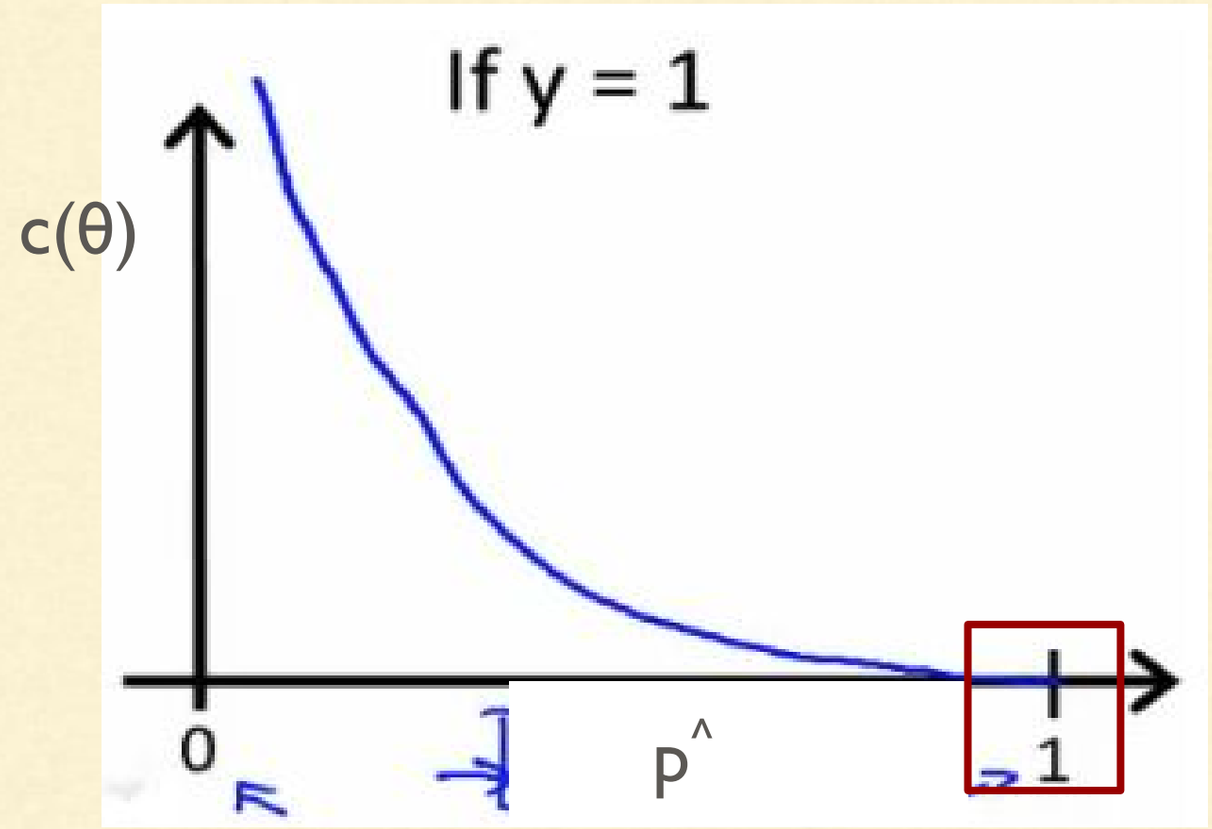
y = -log(p^)



y = -log(1-p^)

CLOUD x LAB

Training and Cost Function

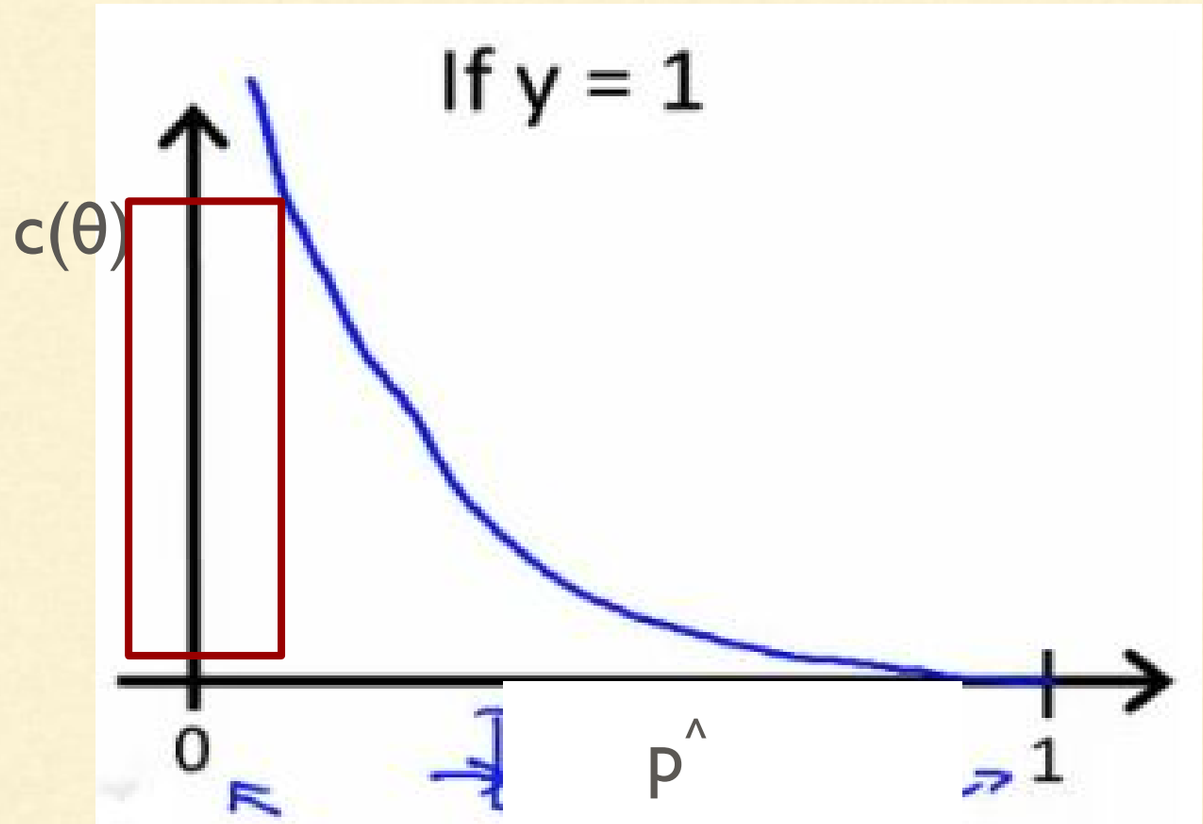


$y = -\log(\hat{p})$

- If y = 1 and p^ = 1 (both actual and predicted probability are one)
  - Then Cost = 0

CLOUD x LAB

Training and Cost Function

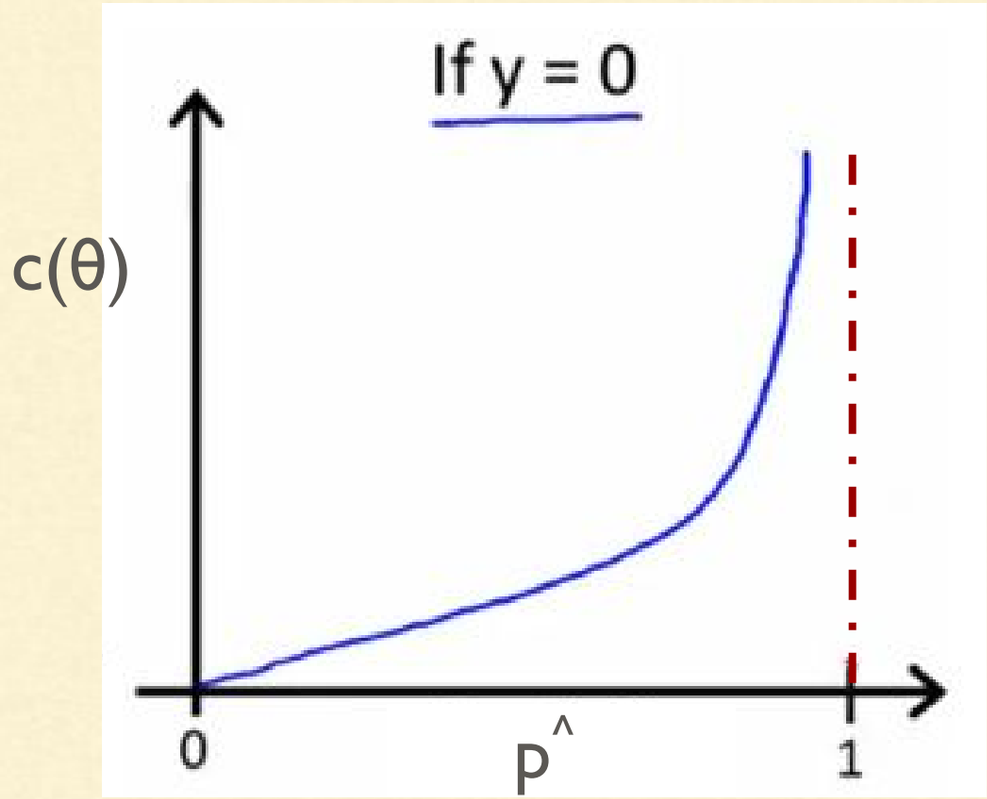


$y = -\log(\hat{p})$

- If y = 1 and p^ = 0 (Predicted probability is different from actual value)
  - As p^ -> 0, cost -> infinity
  - We penalize learning algorithm by a very large cost

CLOUD x LAB

Training and Cost Function

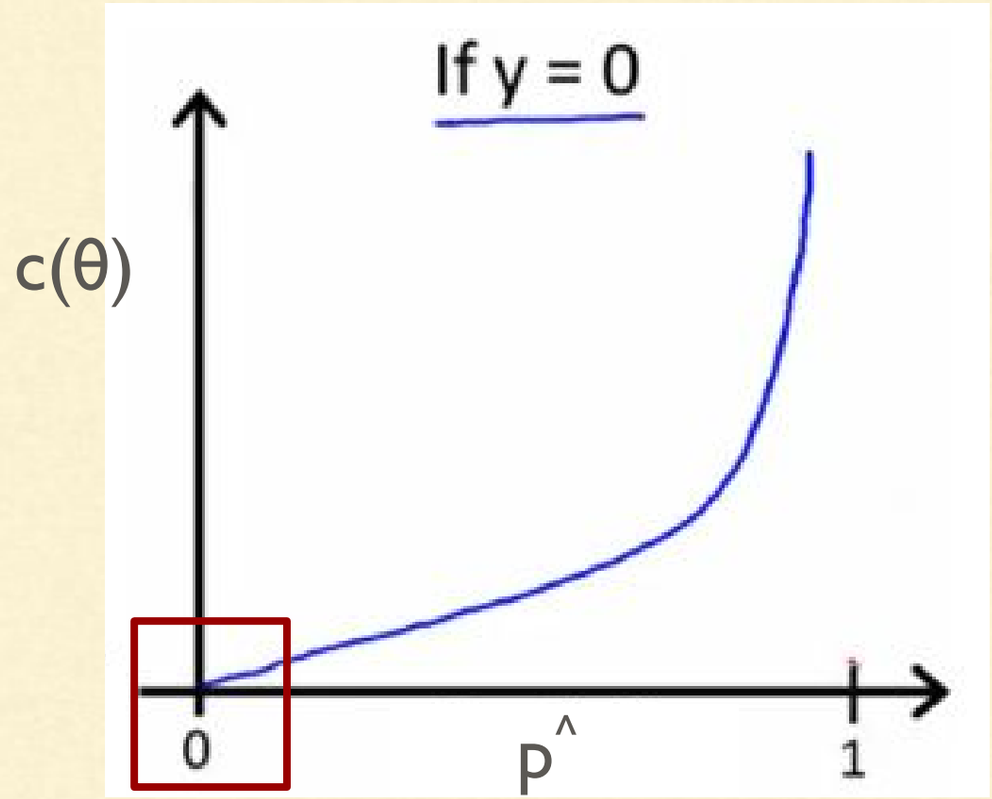


y = -log(1-p^)

- If y = 0, p^ = 1 (Predicted probability is different from actual value)
  - Then cost -> infinity
  - We penalize learning algorithm by a very large cost

CLOUD x LAB

Training and Cost Function



y = -log(1-p^)

- If y = 0, p^ = 0 (both actual and predicted probability are zero)
  - Then cost = 0

CLOUD x LAB

Training and Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(\hat{p}^{(i)}) + (1 - y^{(i)})log(1 - \hat{p}^{(i)})]$$

Logistics Regression Cost Function

CLOUD x LAB

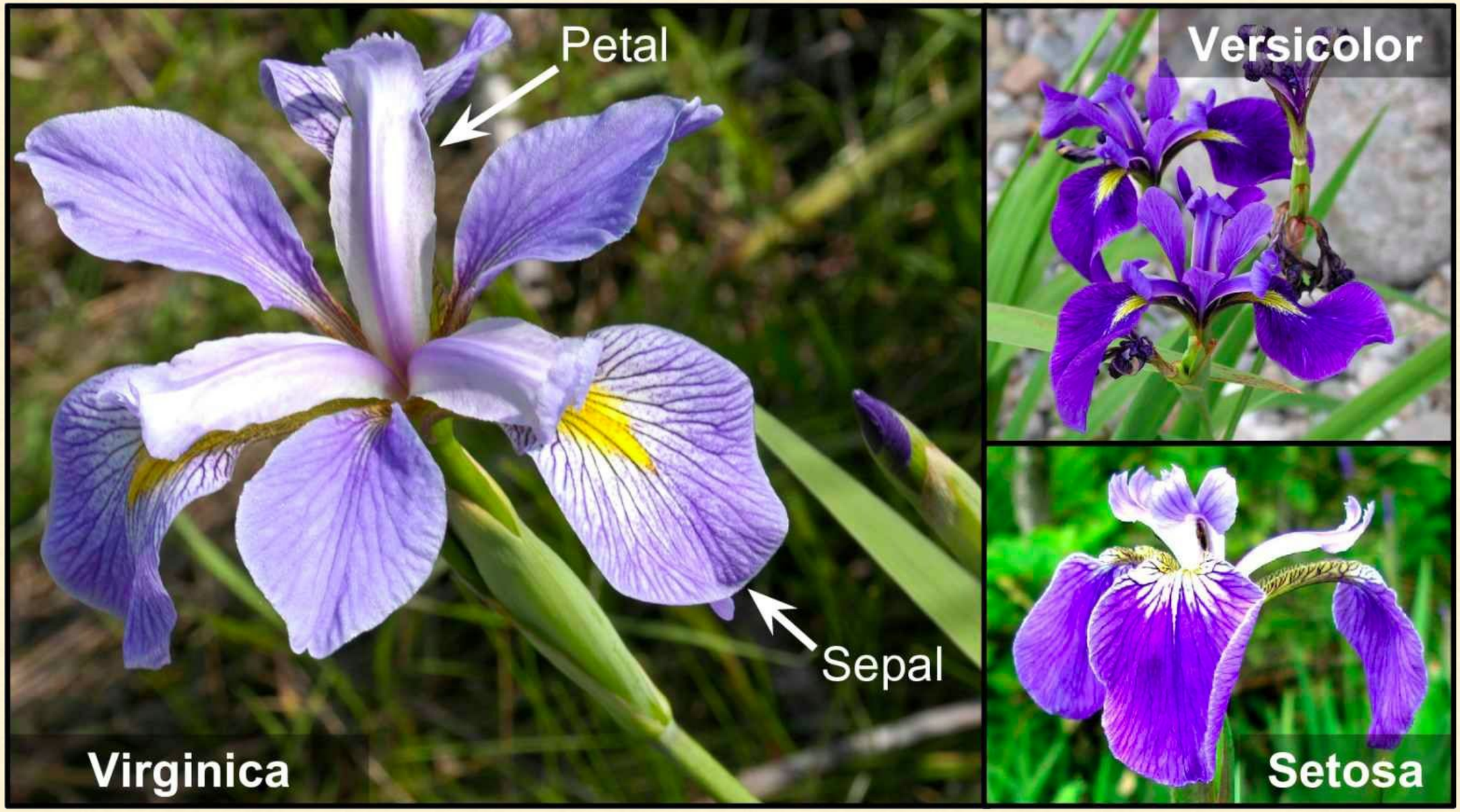
Decision Boundaries

Decision Boundaries

- Let's use the iris dataset to illustrate Logistic Regression
- This is the famous dataset
  - That contains the sepal and petal length and width of
  - 150 iris flowers of three different species
    - Iris-Setosa
    - Iris-Versicolor and
    - Iris-Virginica

CLOUD x LAB

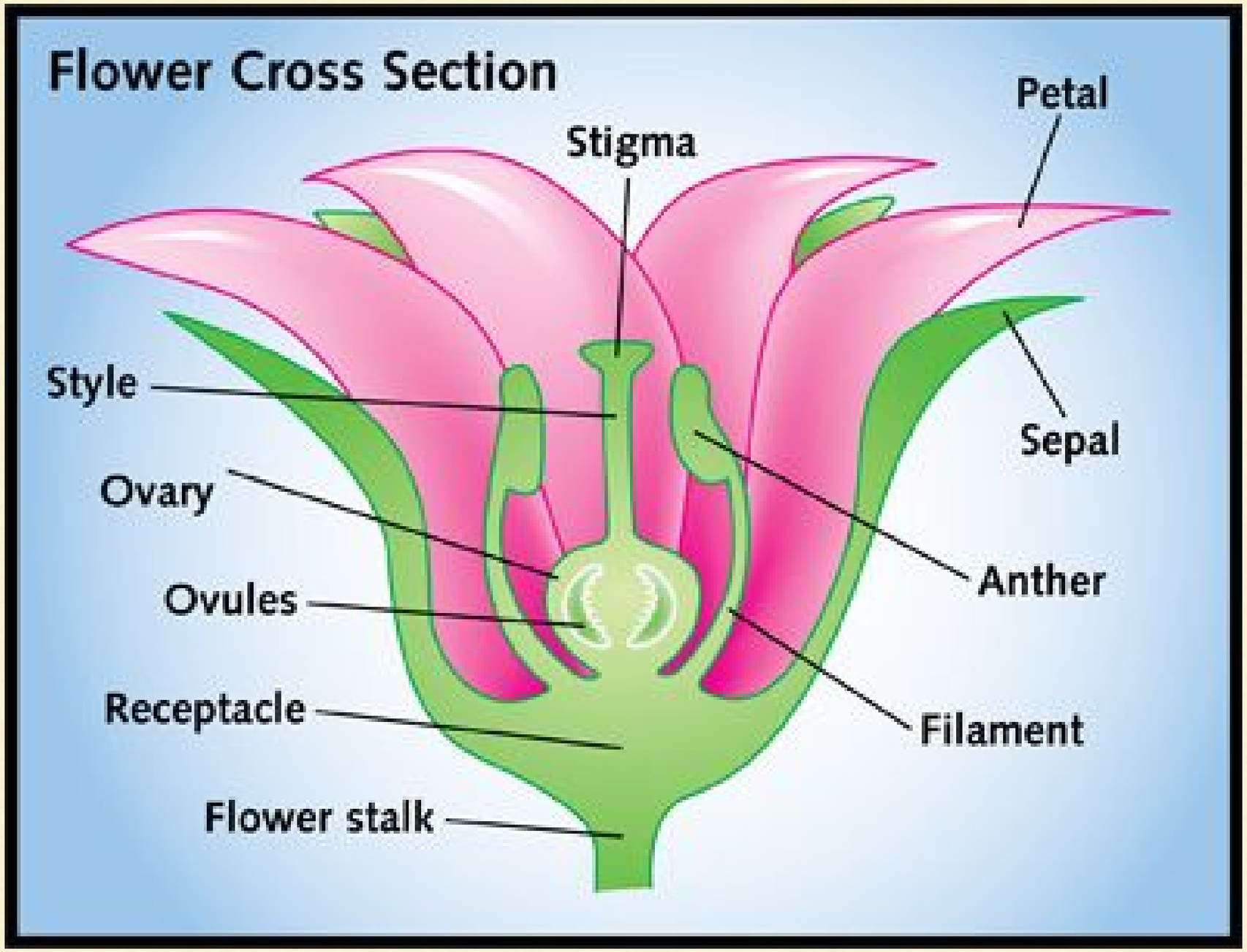
Decision Boundaries



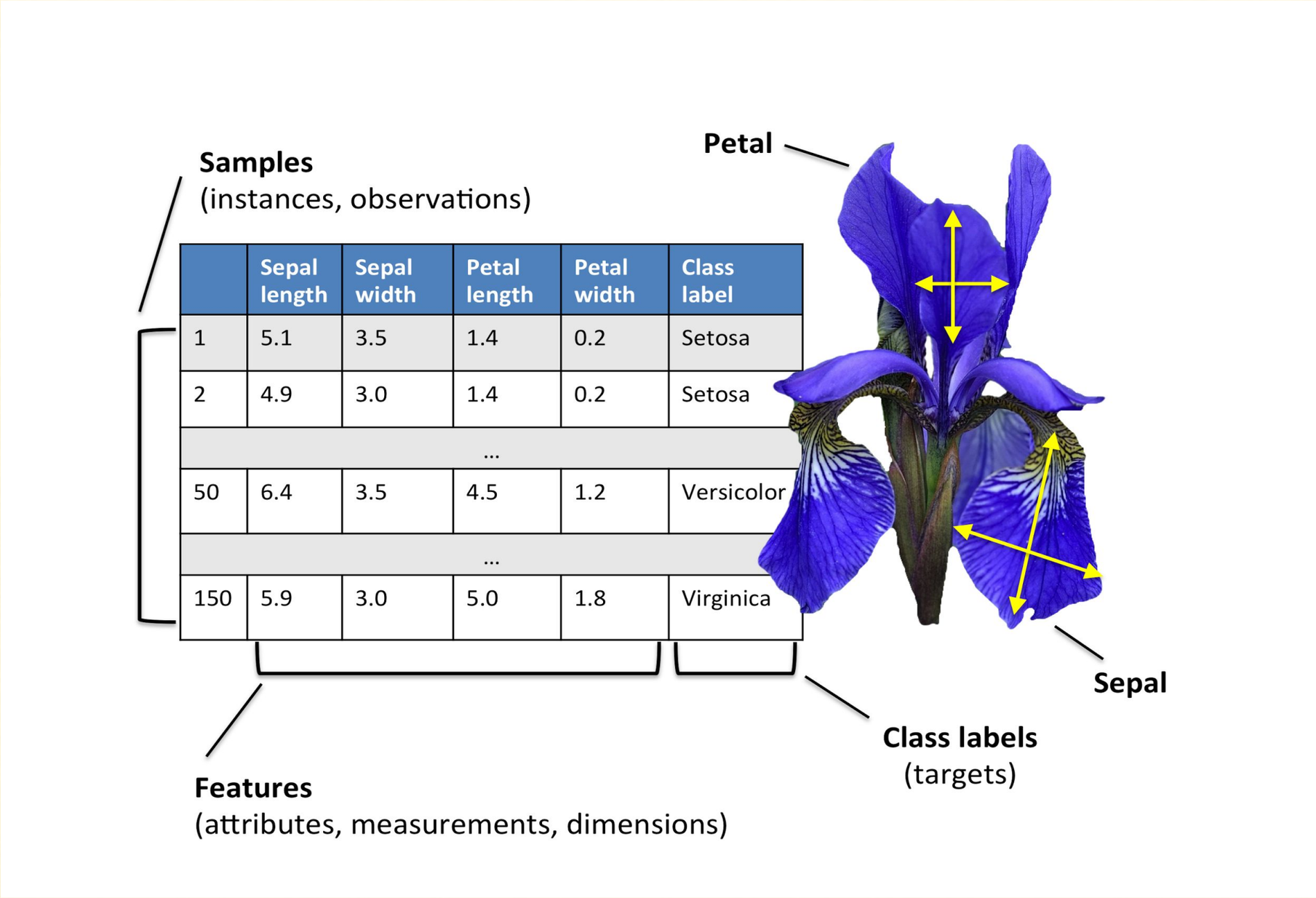
Flowers of three iris plant species

CLOUD x LAB

Decision Boundaries



Decision Boundaries



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

IRIS Dataset

CLOUD x LAB

Decision Boundaries

- Let's look at the dataset

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
```

Output -

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']
```

Decision Boundaries

- Let’s look at the dataset

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
```

Output -

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']
```

Let’s look at the keys

Decision Boundaries

- Check description of dataset

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']

>>> print(iris.DESCR)
```

Check the description in notebook

Decision Boundaries

- Check data

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']

>>> print(iris.data)

Output -
```

Sepal Width (in cm)

Sepal Length (in cm)

```
[[ 5.1   3.5   1.4   0.2]
 [ 4.9   3.    1.4   0.2]
 [ 4.7   3.2   1.3   0.2]
 [ 4.6   3.1   1.5   0.2]
```

Petal Length (in cm)

Petal Width (in cm)

Decision Boundaries

- Check features_names

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']

>>> print(iris.feature_names)

Output -

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
width (cm)']
```


CLOUD x LAB

Decision Boundaries

- Check target

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']

>>> print(iris.target)

Output -
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

- There are 150 targets with 50 0s, 50 1s and 50 2s
- What does 0, 1 and 2 means here?

CLOUD x LAB

Decision Boundaries

- Check target_names

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']

>>> print(iris.target_names)

Output -
```

```
['setosa' 'versicolor' 'virginica']
```

0 1 2

CLOUD x LAB

Decision Boundaries

Now let's build a classifier to detect the **Iris-Virginica** type based only on the petal width feature

CLOUD x LAB

Decision Boundaries

- Train a Logistic Regression model

```
>>> X = iris["data"][:, 3:]  # petal width
>>> y = (iris["target"] == 2).astype(np.int)  # 1 if Iris-Virginica,
else 0
>>> from sklearn.linear_model import LogisticRegression
>>> log_reg = LogisticRegression(random_state=42)
>>> log_reg.fit(X, y)
```


CLOUD x LAB

Decision Boundaries

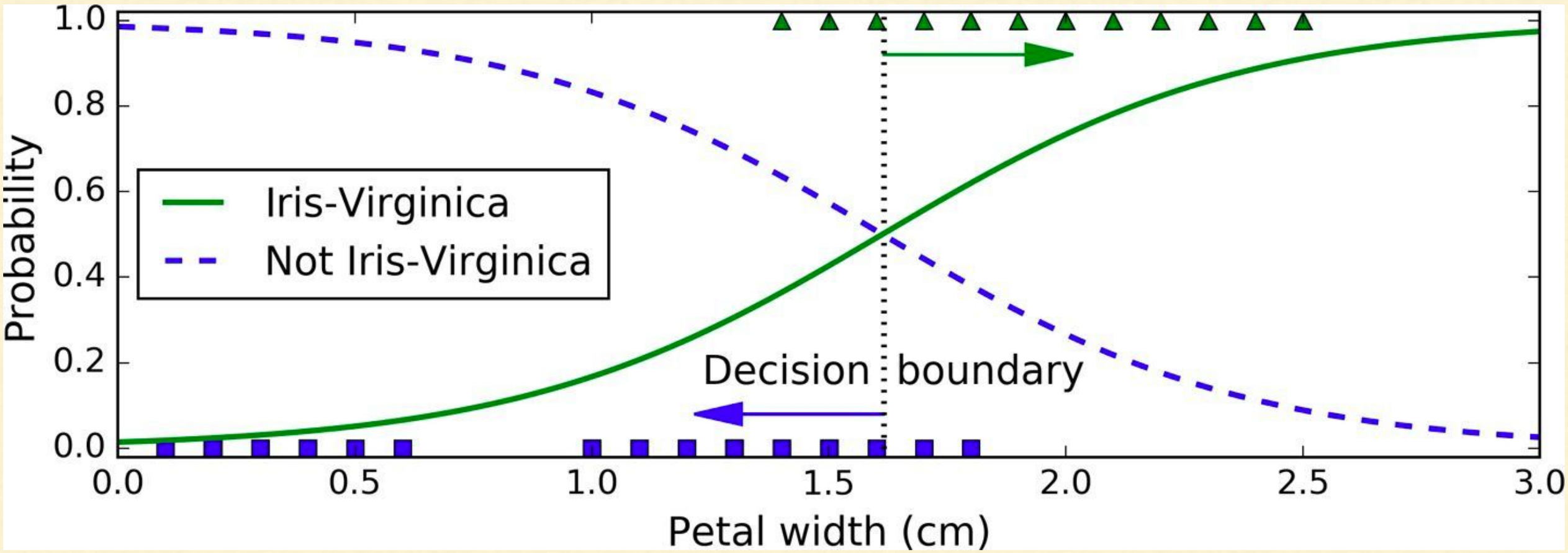
- Now let's look at the model's estimated probabilities
  - For flowers with petal widths varying from 0 to 3 cm

```
>>> X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
>>> y_proba = log_reg.predict_proba(X_new)
>>> plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2,
label="Iris-Virginica")
>>> plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not
Iris-Virginica")
```


CLOUD x LAB

Decision Boundaries

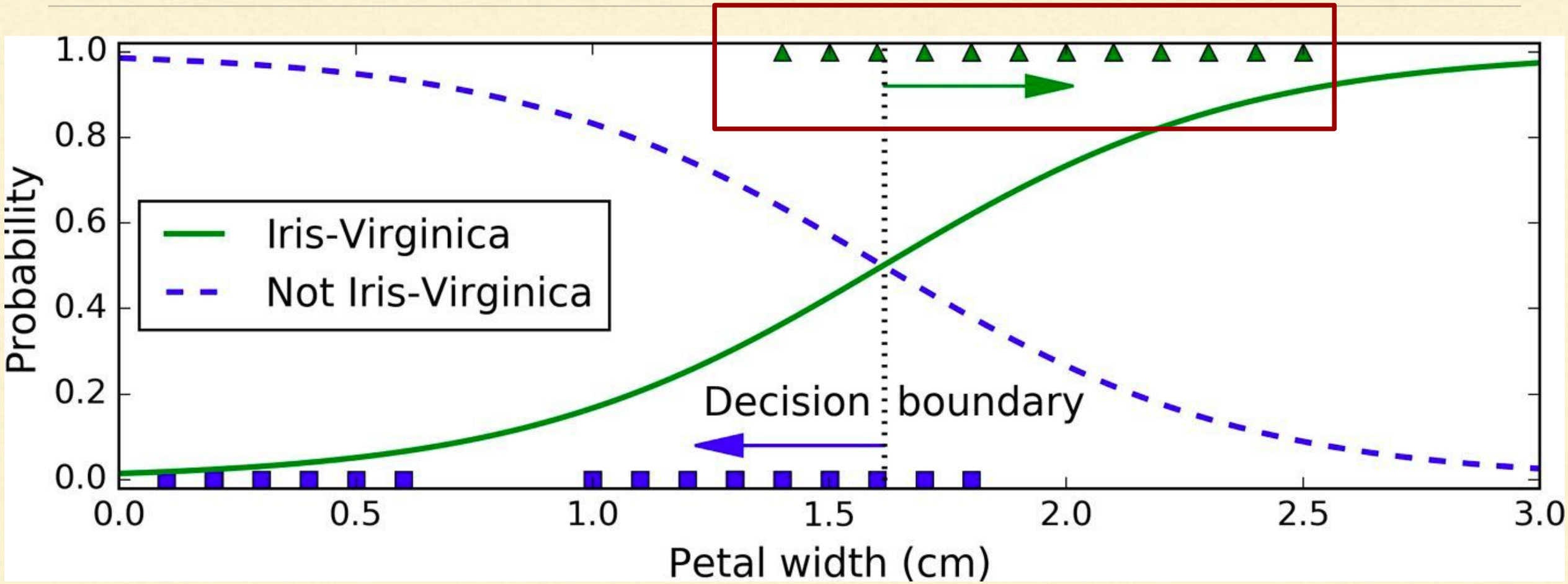
Output-



Estimated probabilities and decision boundary

CLOUD x LAB

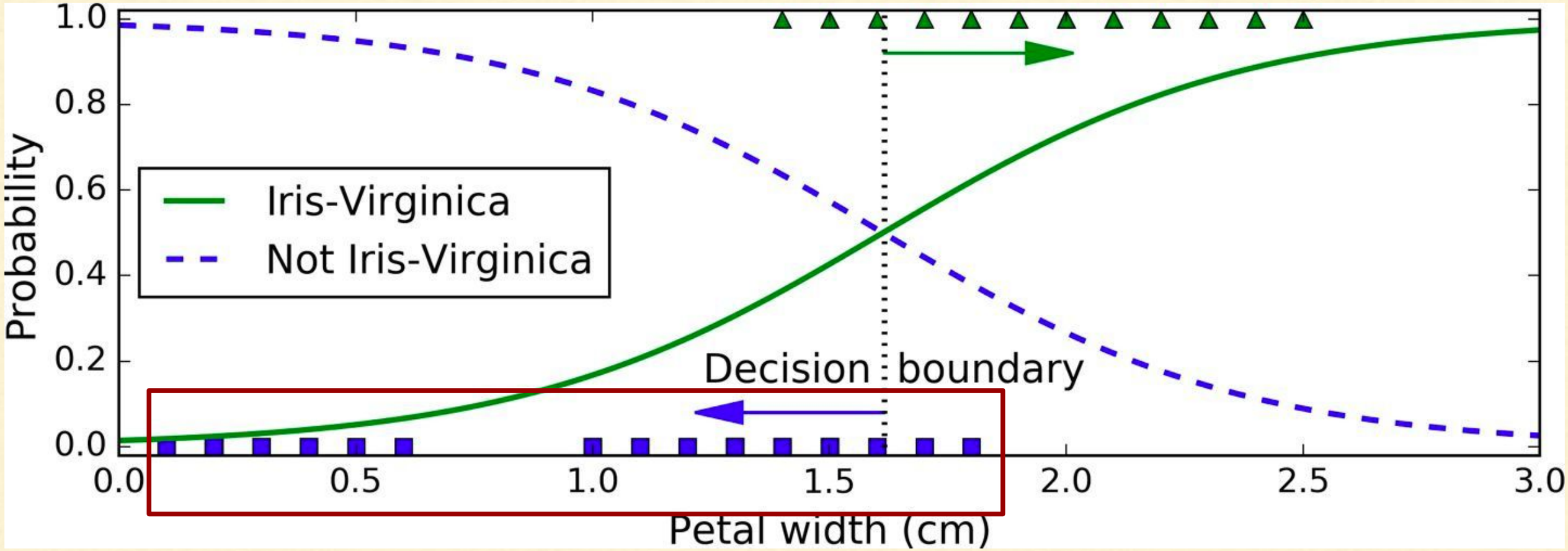
Decision Boundaries



- The petal width of Iris-Virginica flowers
  - Represented by triangles
  - Ranges from 1.4 cm to 2.5 cm

CLOUD x LAB

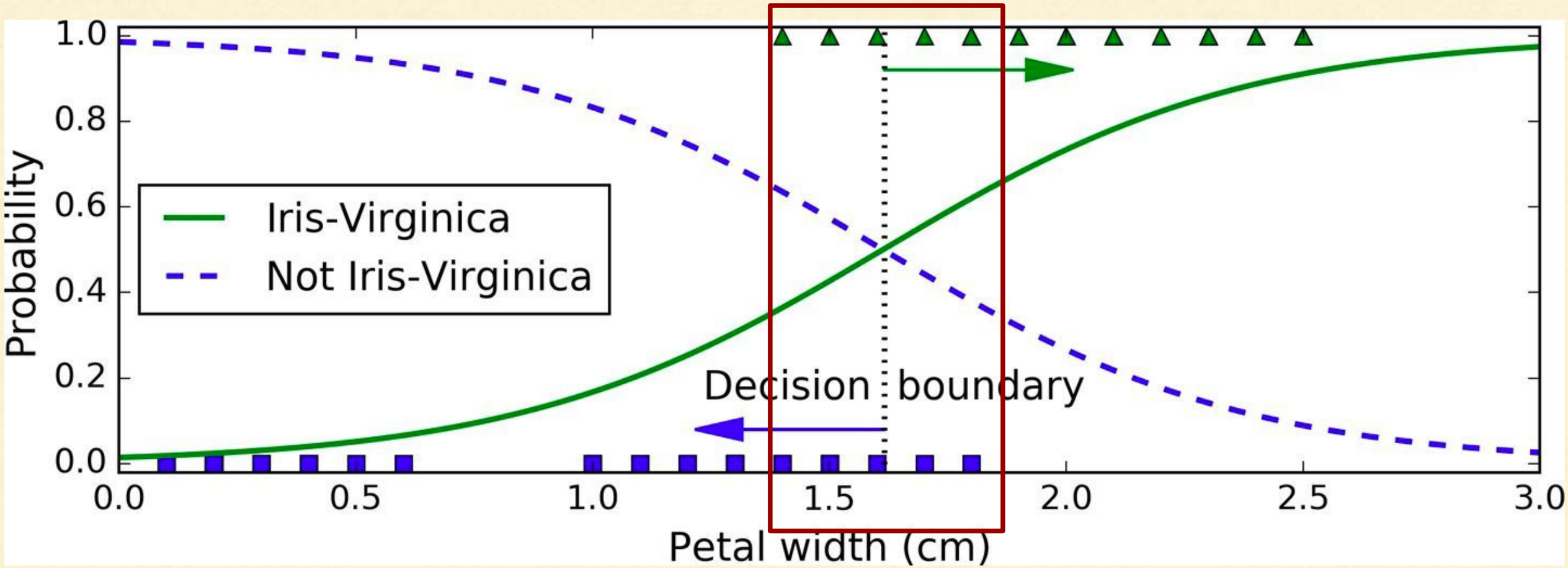
Decision Boundaries



- Other Iris flowers
  - Represented by Squares
  - Have a smaller petal width
  - Ranging from 0.1 cm to 1.8 cm

CLOUD x LAB

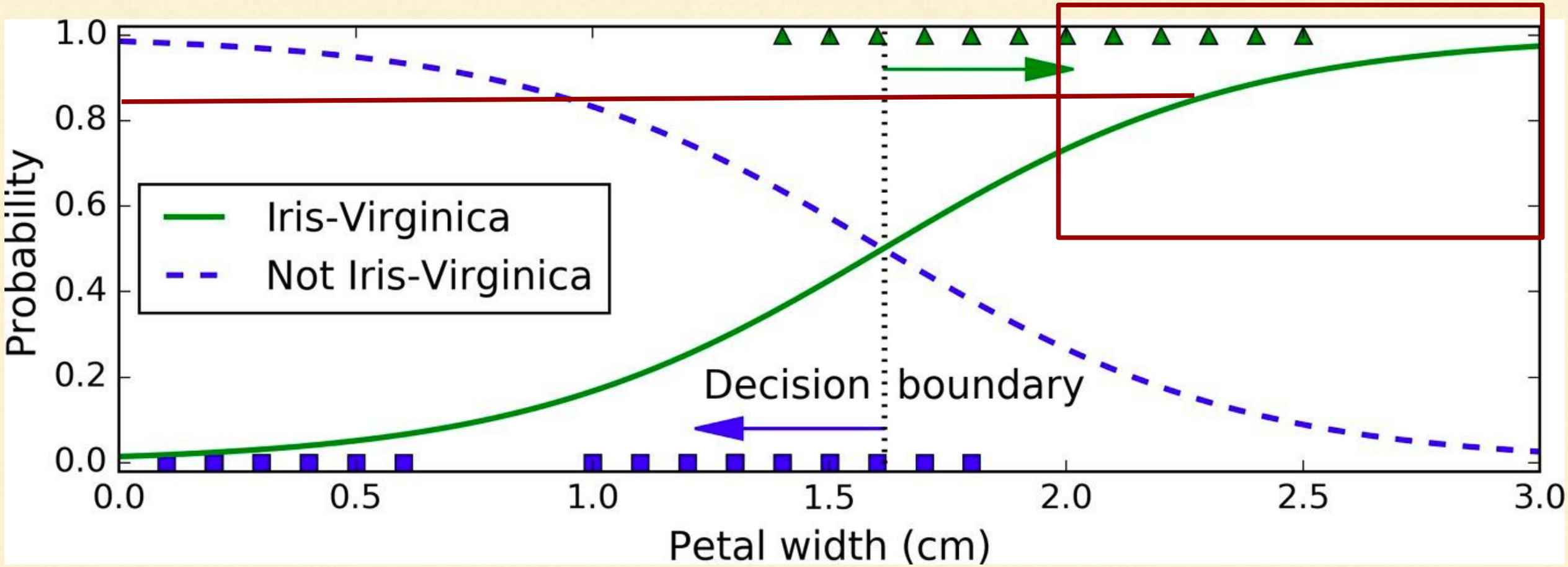
Decision Boundaries



- Notice there is a bit of overlap
- How does classifier predicts then?

CLOUD x LAB

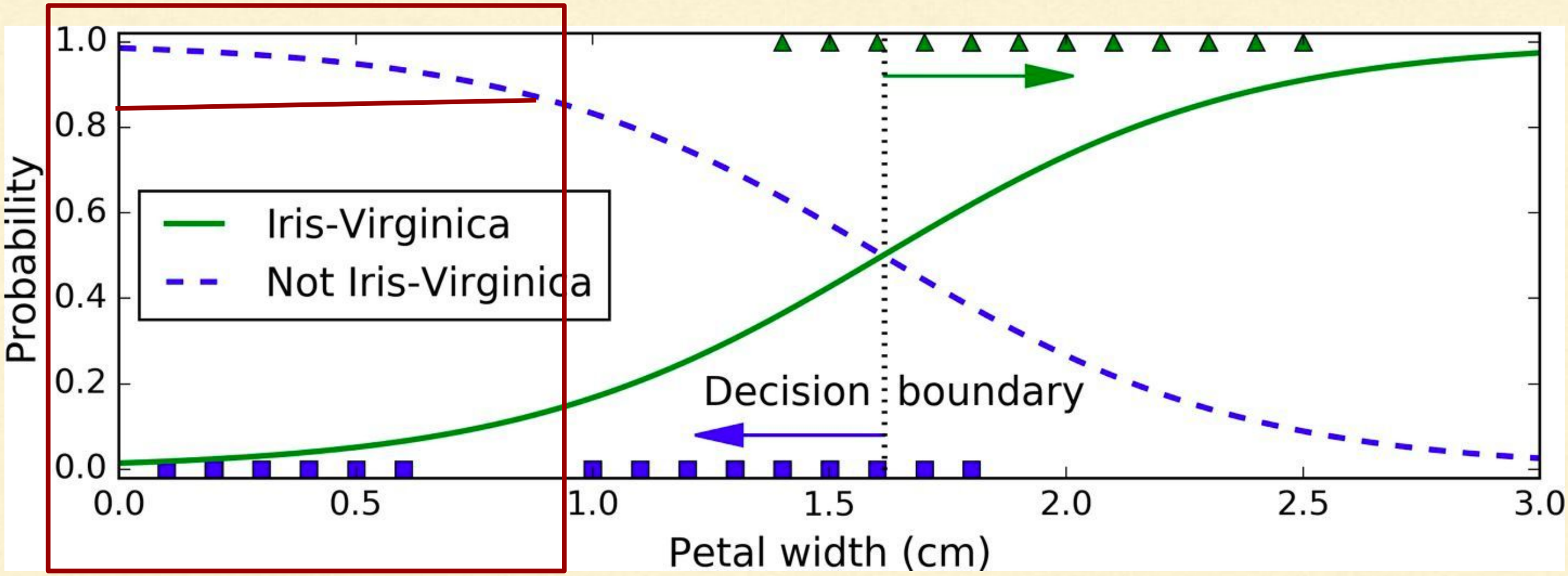
Decision Boundaries



- Above about 2 cm
  - The classifier is highly confident that the
  - Flower is an Iris-Virginica
  - As it outputs a high probability to that class

CLOUD x LAB

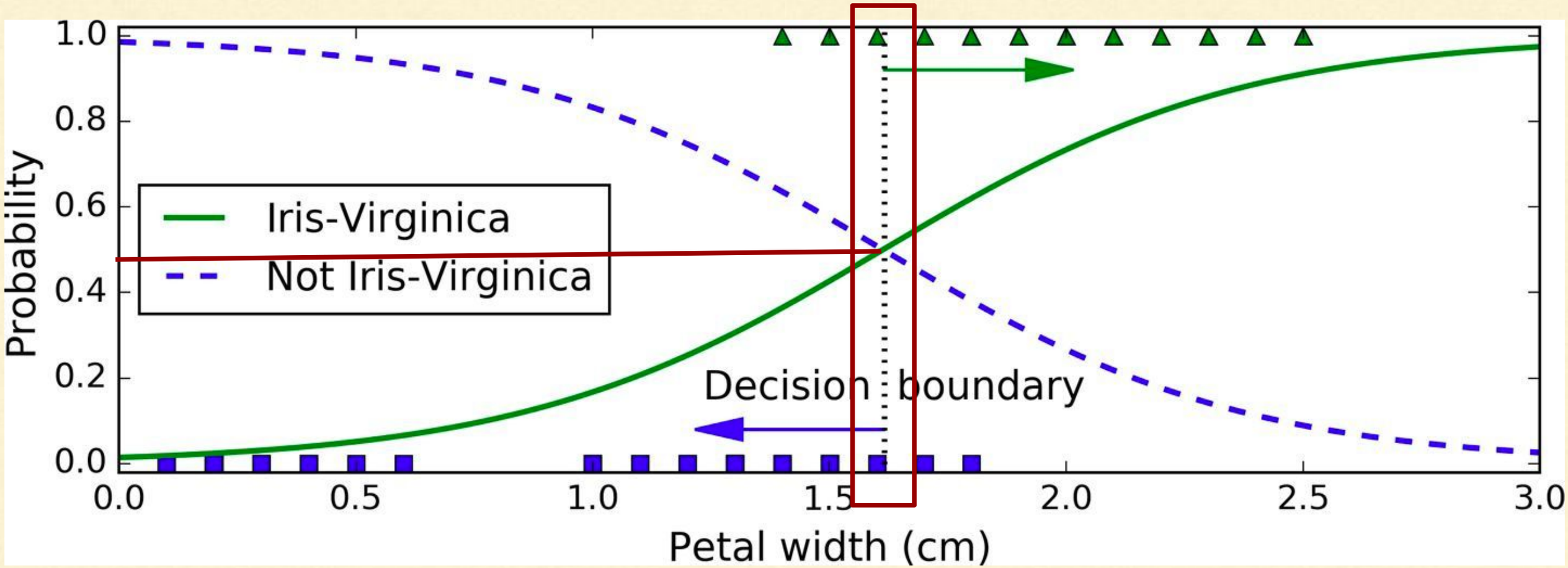
Decision Boundaries



- While below 1(one) cm
- The classifier is highly confident that the
- Flower is not an Iris-Virginica
- As it outputs a high probability for the "Not Iris-Virginica" class

CLOUD x LAB

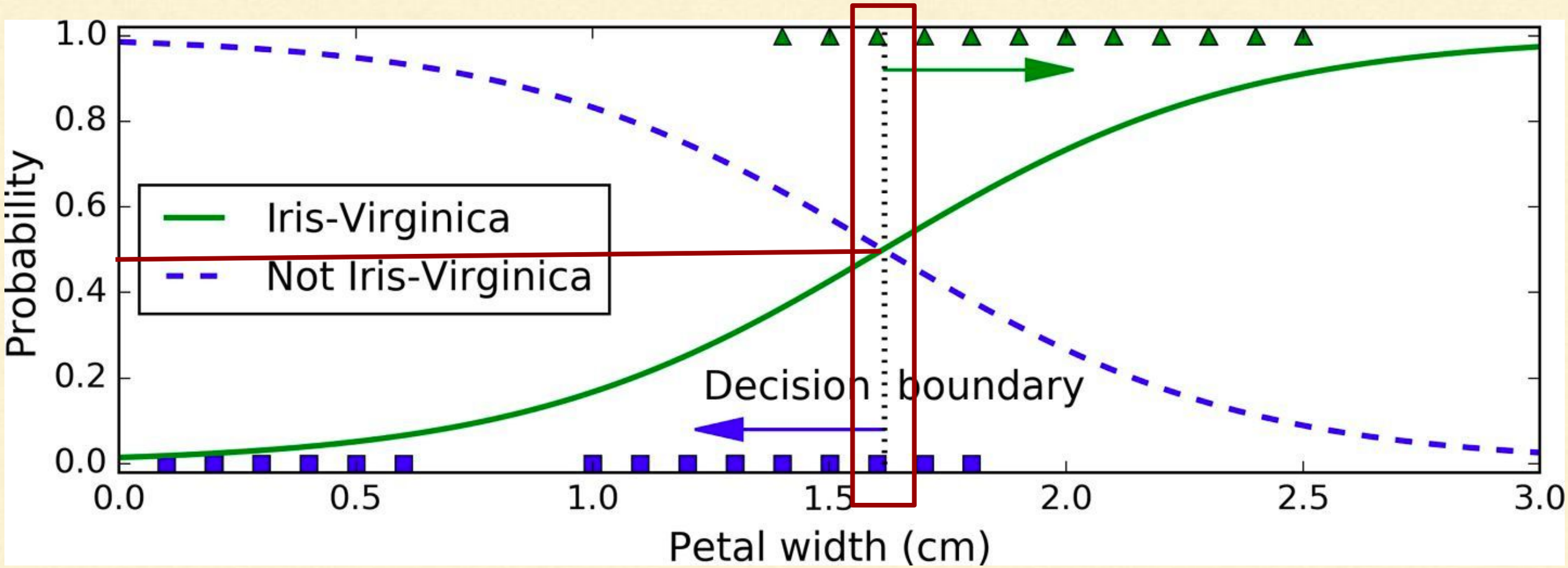
Decision Boundaries



- There is a decision boundary at around 1.6 cm
  - Where both probabilities are equal to 50%

CLOUD x LAB

Decision Boundaries



- If the petal width is higher than 1.6 cm
  - The classifier will predict that the flower is an Iris-Virginica
  - Or else it will predict that it is not

CLOUD x LAB

Decision Boundaries

- Let's check the predictions

```
>>> log_reg.predict([[1.7], [1.5]])

Output-

array([1, 0])
```

Decision Boundaries

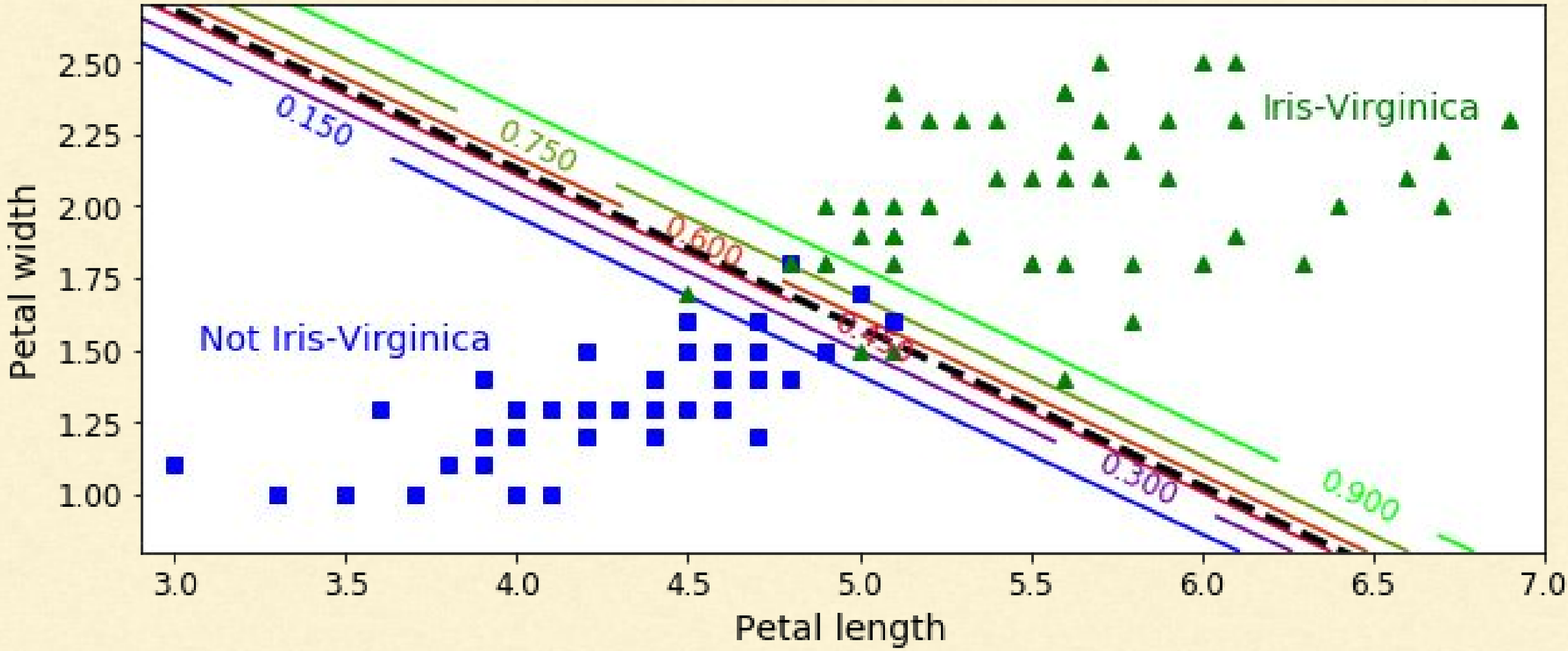
Now let's build a classifier to detect the Iris-Virginica type based on petal width and length

Decision Boundaries

- Now the the Logistic Regression classifier
  - Estimates the probability that a
  - New flower is an Iris-Virginica based on
    - Petal width and
    - Petal length features

Check out the code in Jupyter notebook

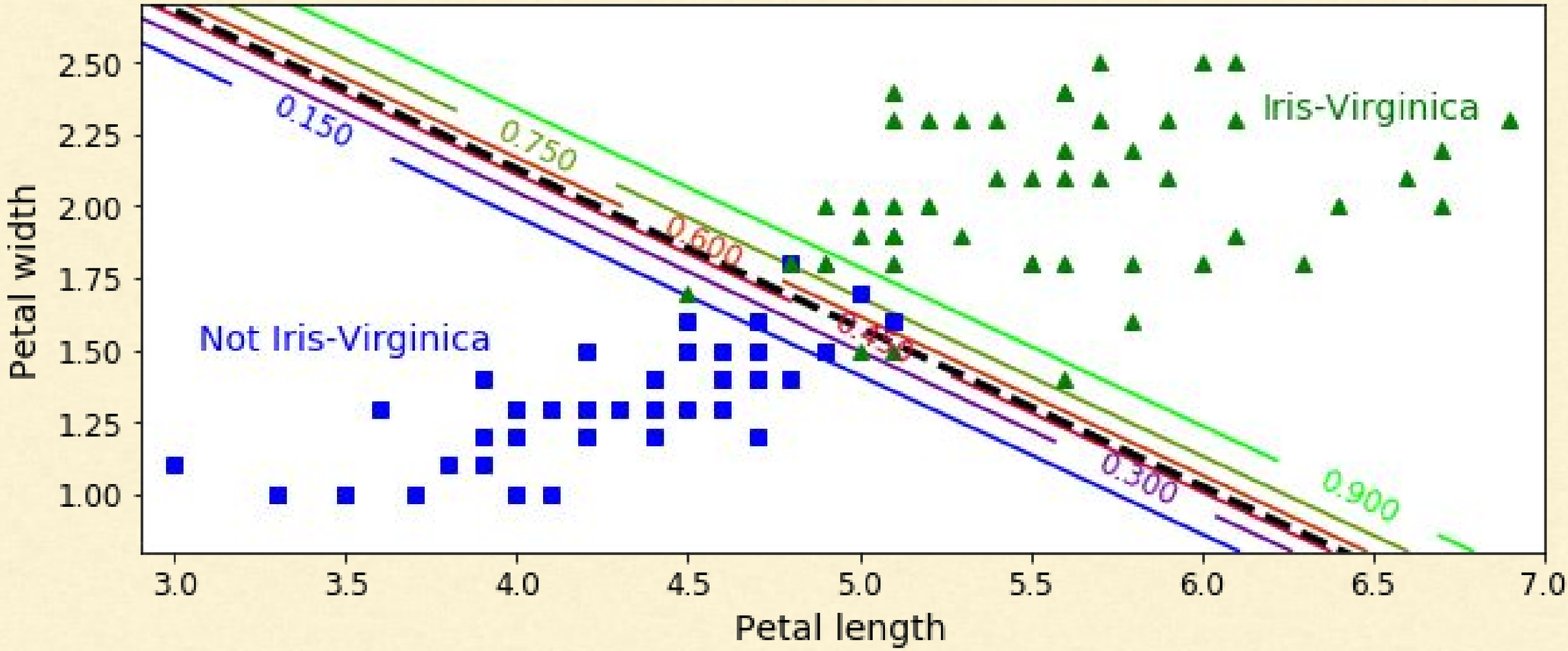
Decision Boundaries



Linear decision boundary

CLOUD x LAB

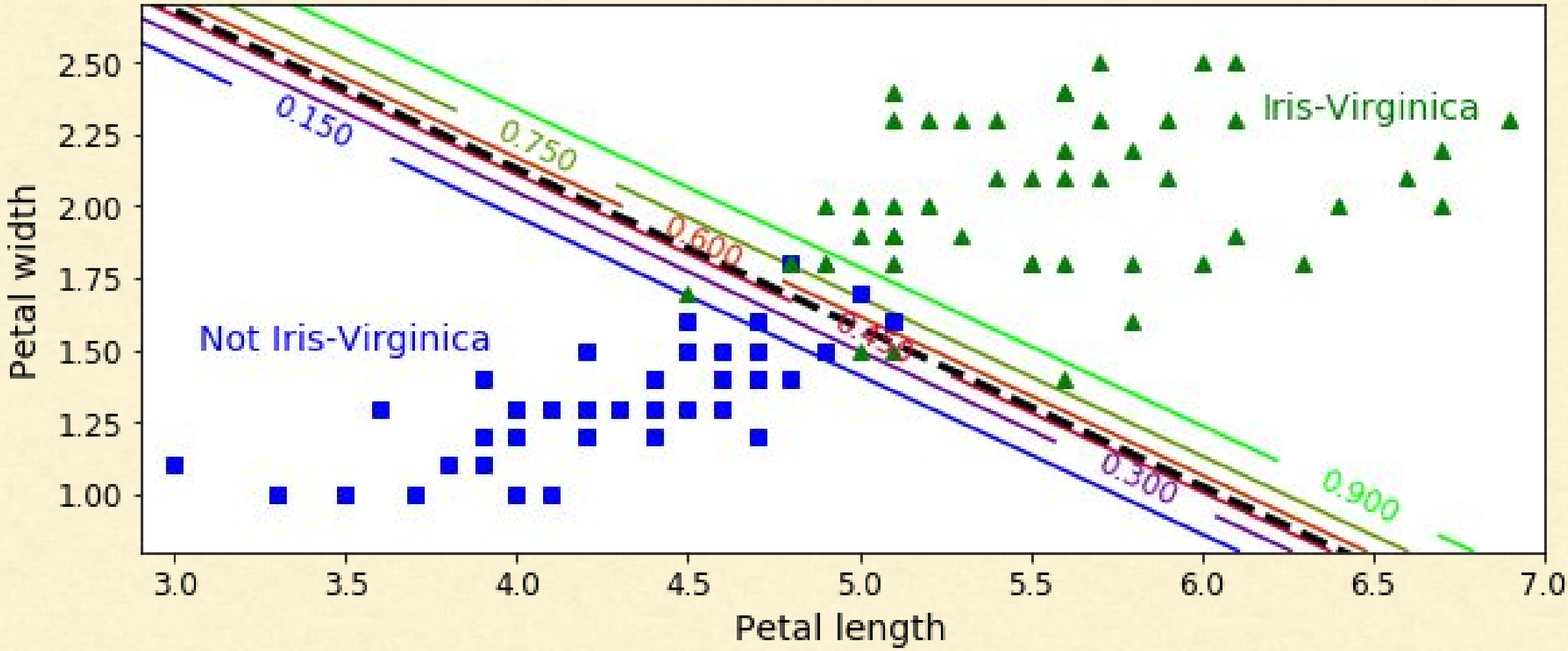
Decision Boundaries



- The dashed line represents the points
  - Where the model estimates a 50% probability
  - This is the model's decision boundary

CLOUD x LAB

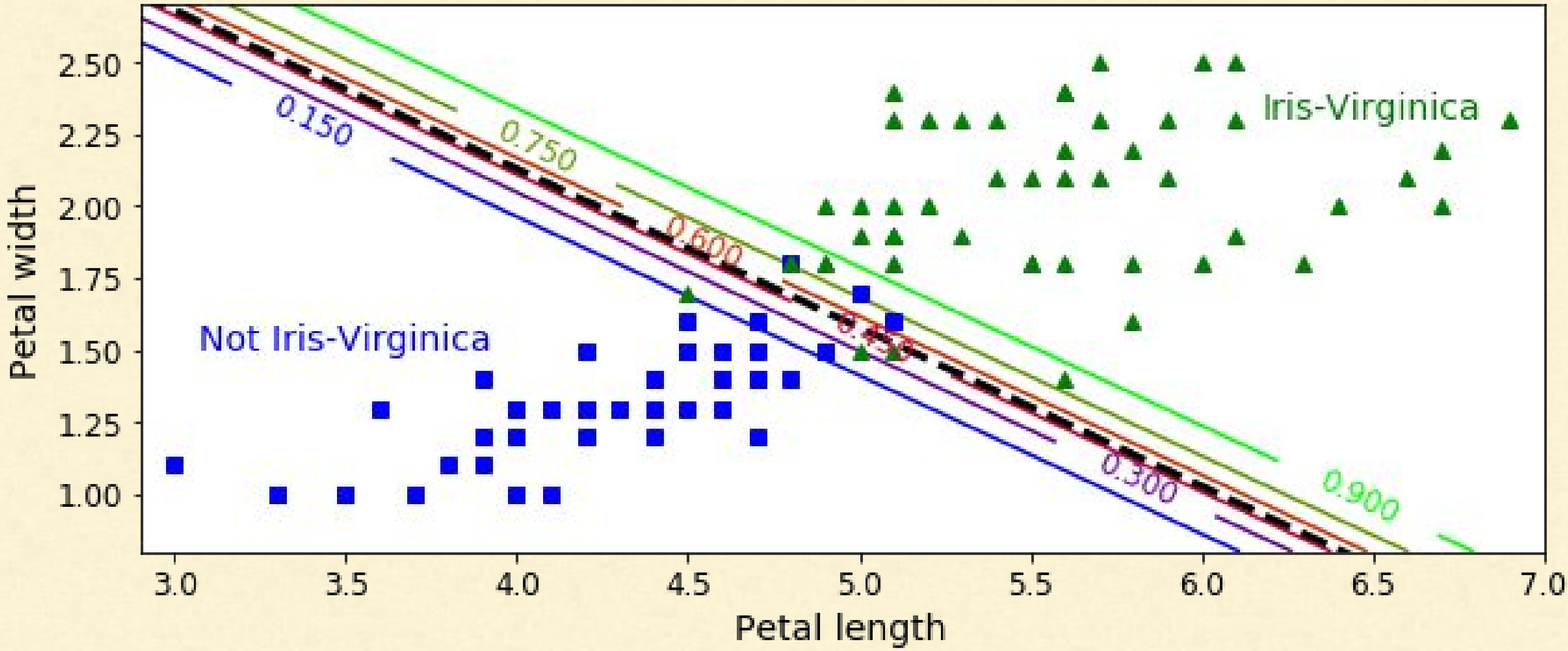
Decision Boundaries



- Each parallel line represents the points
  - Where the model outputs a specific probability
  - from 15% (bottom left) to 90% (top right)

CLOUD x LAB

Decision Boundaries



- All the flowers beyond the top-right line
  - Have an over 90% chance of being Iris-Virginica

CLOUD x LAB

Decision Boundaries

Regularization

- Like linear models, Logistic Regression models can be regularized
  - Using L1 or L2 penalties
  - Scikit-Learn adds an L2 penalty by default

Logistic Regression

Softmax Regression

Softmax Regression

- The Logistic Regression model
  - Can be generalized to support multiple classes
  - Without having to train and combine multiple binary classifiers
  - This is called Softmax Regression or Multinomial Logistic Regression
- Let's do a quick hands-on on Softmax Regression before going into theory

CLOUD x LAB

Softmax Regression

- Let's classify the iris flowers into all three classes

```
>>> # Softmax Regression
>>> X = iris["data"][:, (2, 3)]  # petal length, petal width
>>> y = iris["target"]
>>> softmax_reg = LogisticRegression(
        multi_class="multinomial",
        solver="lbfgs",
        C=10,
        random_state=42
    )
>>> softmax_reg.fit(X, y)
```

Softmax Regression

- `lbfgs` - Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm
  - Is an iterative method for solving unconstrained nonlinear optimization problems
  - C - Hyperparameter for regularizing L2

CLOUD x LAB

Softmax Regression

- Predict the type of iris having 5 cm long and 2 cm wide petal

```
>>> softmax_reg.predict([[5, 2]])
```

```
Output -
array([2])
```

Iris-Virginica

CLOUD x LAB

Softmax Regression

- Predict the probability

```
>>> softmax_reg.predict_proba([[5, 2]])
```

```
Output -
array([[6.33134077e-07, 5.75276067e-02, 9.42471760e-01]])
```

Iris- Versicolor with 5.8% probability

Iris-Virginica (class 2) with 94.2% probability

Softmax Regression

- In Softmax Regression
  - Given an instance **x**
  - Softmax model first computes a score $s_k(x)$ for each class k
  - Then applies the softmax function (Discussed in next slide) to the scores

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}$$

Softmax score for class k

- Each class has its own parameter vector θ_k

CLOUD x LAB

Softmax Function

Softmax Function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

- Once we have score of every class for the instance **x**
- Then we can estimate the probability $\hat{p}_k$ using softmax function
- $\hat{p}_k$ - probability that the instance belongs to class k
- It computes the exponential of every score
  - Then normalizes them (dividing by the sum of all the exponentials)

CLOUD x LAB

Softmax Function

Softmax Function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

- K is the number of classes
- s(x) is a vector containing the
  - Scores of each class for the instance x.
- $\sigma(s(x))_k$ is the estimated probability that the
  - Instance x belongs to class k
  - Given the scores of each class for that instance.

CLOUD x LAB

Softmax Function

- Softmax function
  - Calculates the probabilities of each target class
  - Over all possible target classes
- It returns the
  - Probabilities of each class and
  - The target class will have the high probability

Softmax Function

Softmax Function Properties

- The calculated probabilities will be in the range of 0 to 1
- The sum of all the probabilities is equals to 1

Softmax Function

Softmax Function Usages

- Used in multiple classification logistic regression model
- In building neural networks softmax functions used in different layers

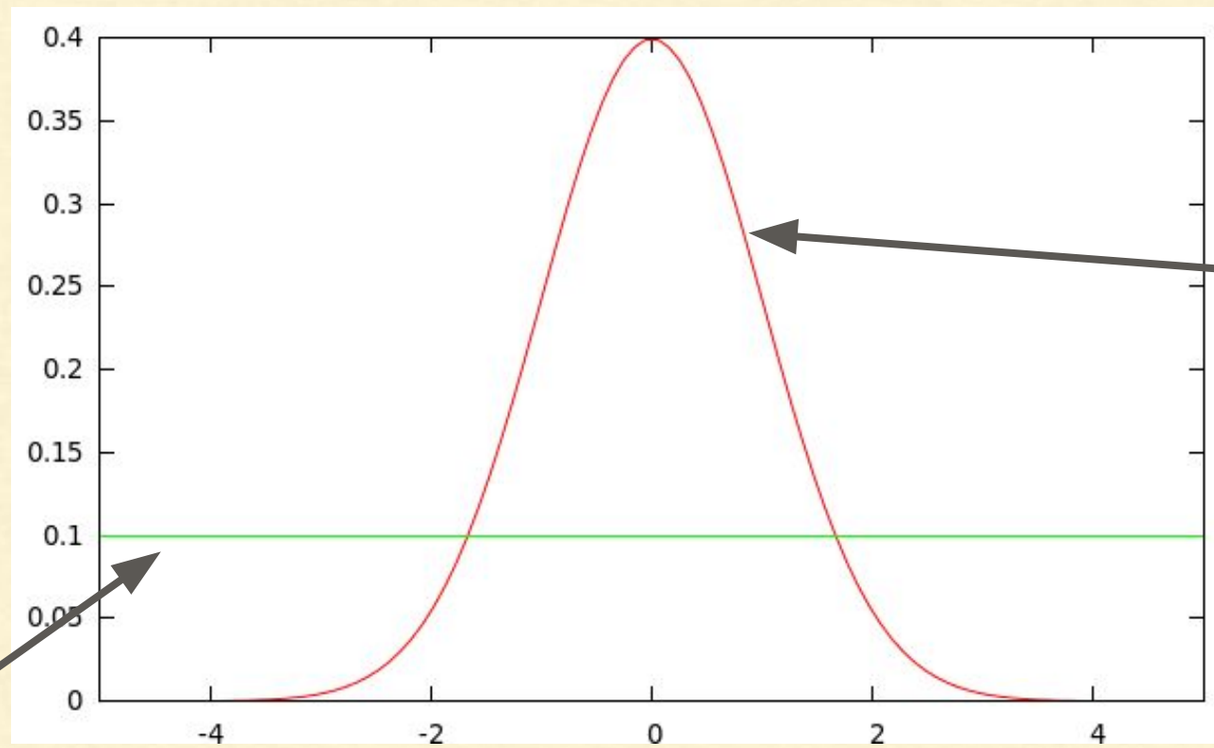
Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com

Linear Regression

Normal vs Uniform Distribution Example



Normal distribution

Uniform distribution over the range of [-5, 5]

CLOUD x LAB