CITS5508 Machine Learning

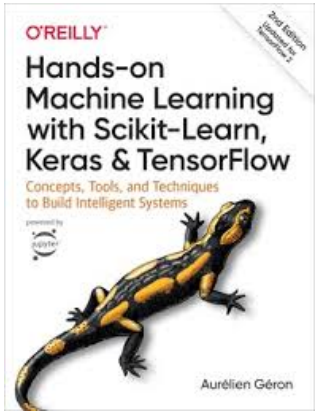
Débora Corrêa (Unit Coordinator and Lecturer)

2023

Today

Chapter 4.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Chapter Four

Regression Models

We start to look at how Machine Learning algorithms work, starting with some regression models. We will cover:

- Linear Regression
- Gradient Descent
- Polynomial Regression
- Learning Curves
- Logistic Regression

Linear Regression

This is a linear model which makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias* term (also called the *intercept* term),

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

where

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the i^{th} feature value
- θ_j is the j^{th} model parameter

Linear Regression: vectorized form of model

...Or in vectors ...

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x}$$

where

- $\hat{y}$ is the predicted value
- $\boldsymbol{\theta}$ is the model's parameter vector, containing the bias term θ_0 and the feature weights θ_1 to θ_n. i.e., $\boldsymbol{\theta} = (\theta_0, \theta_1, \cdots, \theta_n)^\top$.
- $\mathbf{x}$ is the instance's feature vector, containing x_0 to x_n, with x_0 always equal to 1. That is, $\mathbf{x} = (x_0, x_1, \cdots, x_n)^\top$.
- $\boldsymbol{\theta}^\top \mathbf{x}$ is the matrix multiplication of $\boldsymbol{\theta}$ and $\mathbf{x}$.
- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.

Linear Regression: training the model

- Training a model means setting its parameters so that the model best fits the training set.
- Thus, we first need a measure of how well (or poorly) the model fits the training data. A common performance measure of a regression model is the *Root Mean Square Error* (*RMSE*).
- Therefore, to train a Linear Regression model, you need to find the value of $\boldsymbol{\theta}$ that minimizes the RMSE.
- In practice, it is simpler to minimize the *Mean Square Error* (*MSE*) than the RMSE. It leads to the same result (the value that minimizes a function also minimizes its square root).

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

Linear Regression: the Normal Equation

To find the value of $\boldsymbol{\theta}$ that minimizes the cost function, there is a closed-form solution—in other words, a mathematical equation that gives the result directly. This is called the **Normal Equation**.

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

where

- $\mathbf{X}$ is a $m \times (n+1)$ matrix – the i^{th} row of $\mathbf{X}$ is the training instance $\mathbf{x}^{(i)}$;
- $\hat{\boldsymbol{\theta}}$ is the value that minimizes the cost function;
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Once the optimal model parameter $\hat{\boldsymbol{\theta}}$ is known, given a new test instance $\mathbf{x}_{\text{test}}$, we can predict y_{test} using the formula:

$$y_{\text{test}} = \hat{\boldsymbol{\theta}}^\top \mathbf{x}_{\text{test}}.$$

A random dataset to use as an example

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

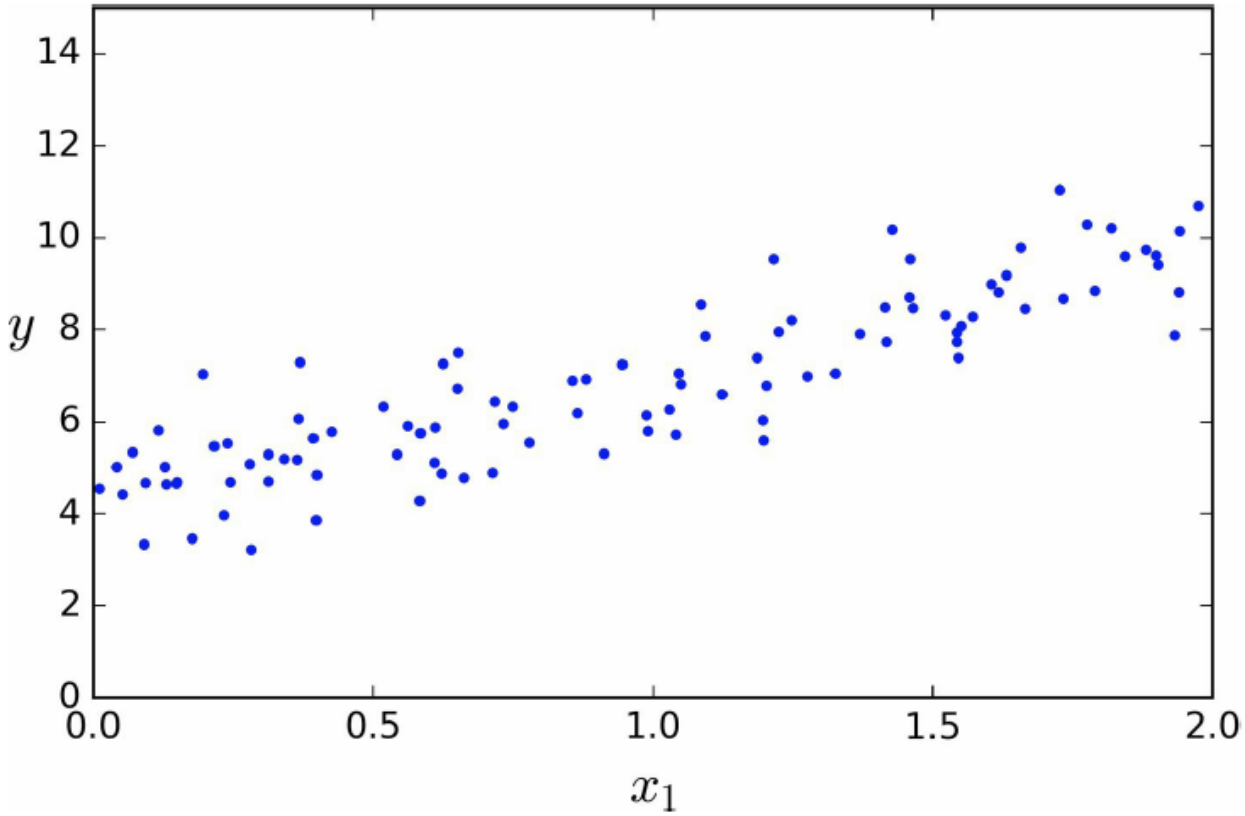


Figure 4-1. Randomly generated linear dataset

Example: Fitting the model

Now let's compute $\hat{\boldsymbol{\theta}}$ using the Normal Equation.

```
X_b = np.c_[np.ones((100, 1)), X]    # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Making Predictions

Just using basic numpy python:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Let's plot this model's predictions:

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

Plot of Predictions

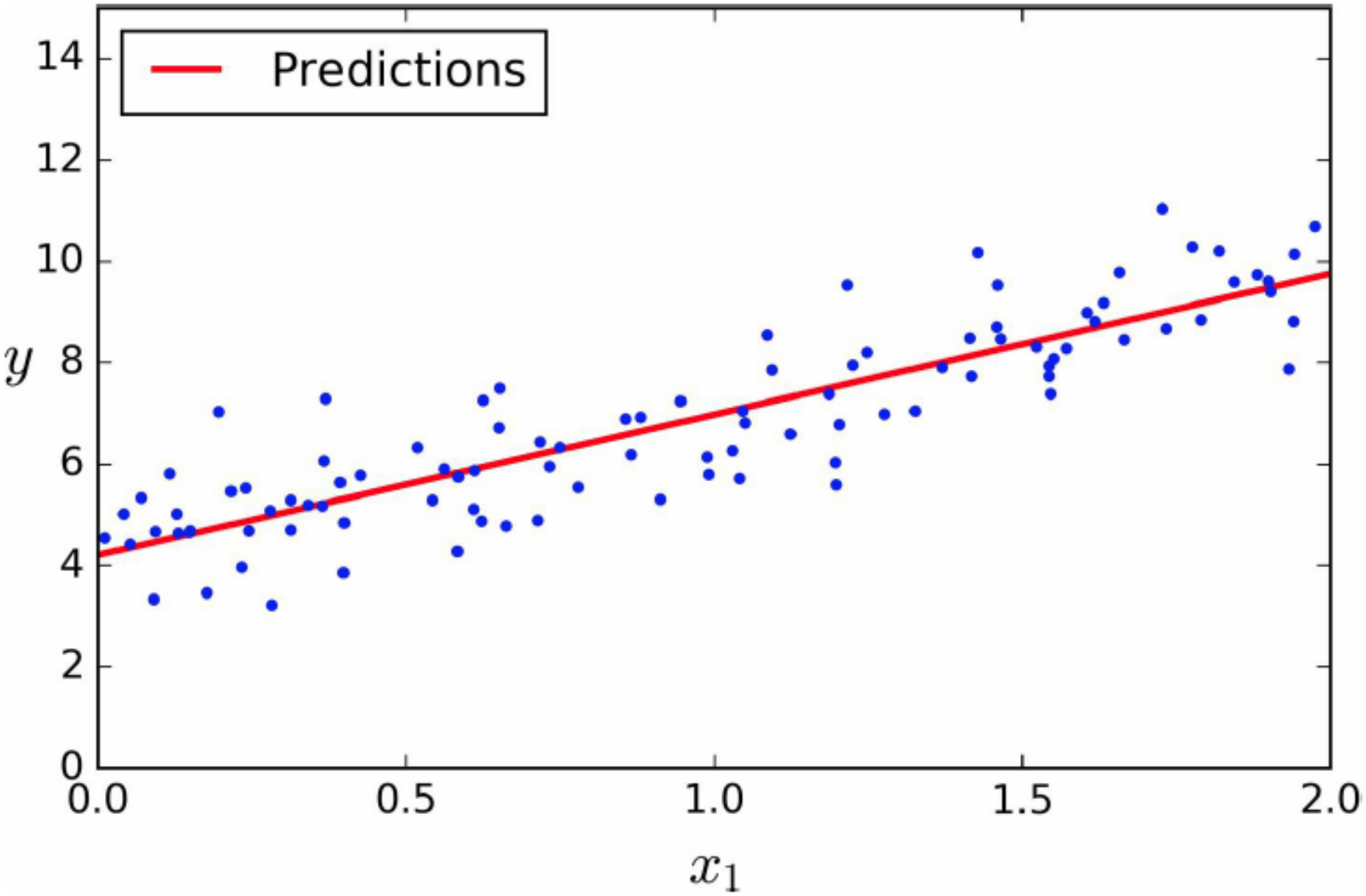


Figure 4-2. Linear Regression model predictions

The equivalent code using Scikit-Learn looks like this:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

Beware: running time of normal equation can be cubic in number of features but linear in data.

Gradient Descent

Gradient Descent is a very generic optimization algorithm for which the general idea is to tweak parameters iteratively in order to minimize a cost function.

It measures the local gradient of the error function with regards to the parameter vector $\boldsymbol{\theta}$, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

It starts by filling $\boldsymbol{\theta}$ with random values (this is called random initialization), and then improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum.

Gradient Descent

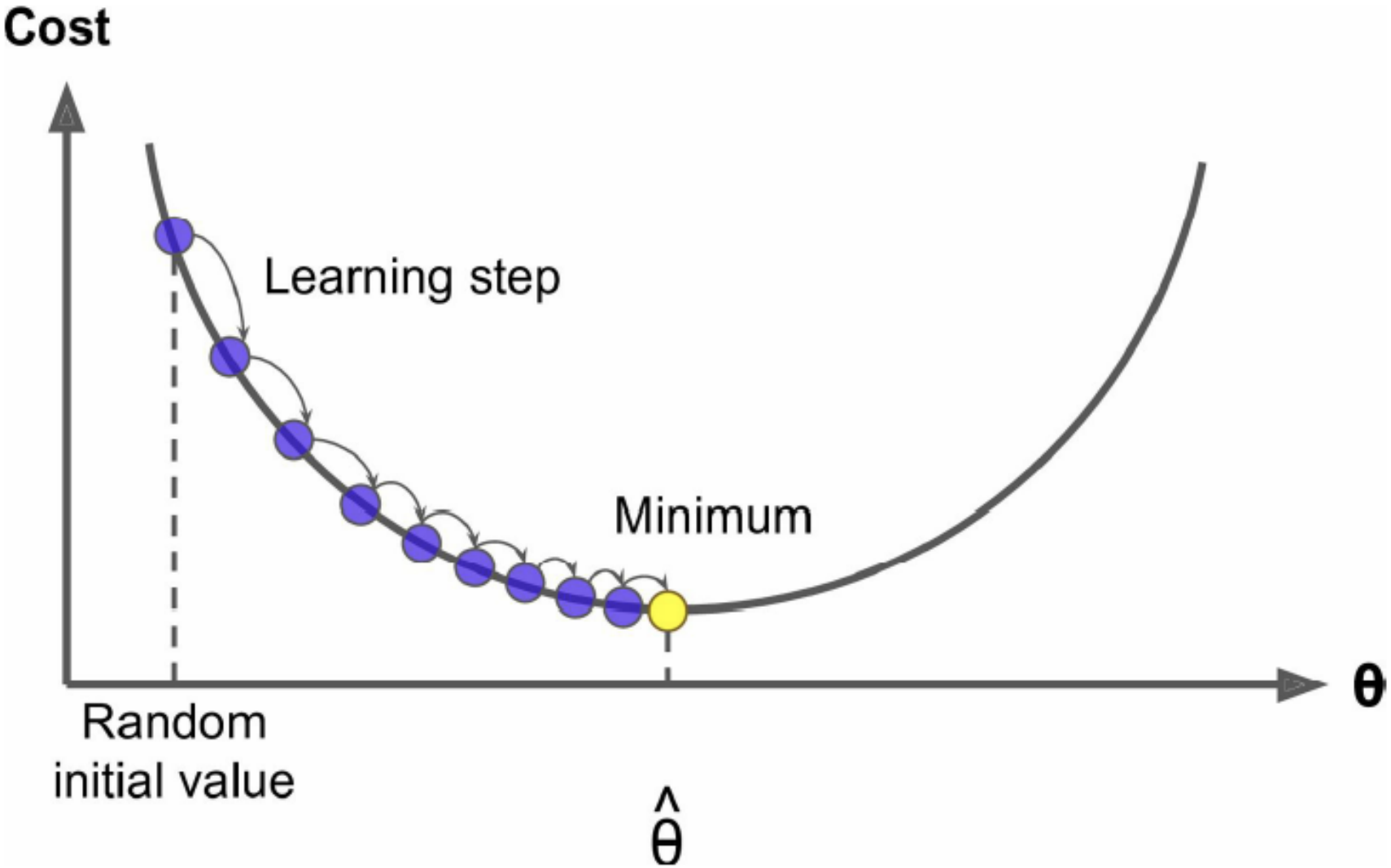


Figure 4-3. Gradient Descent

Learning rate.

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

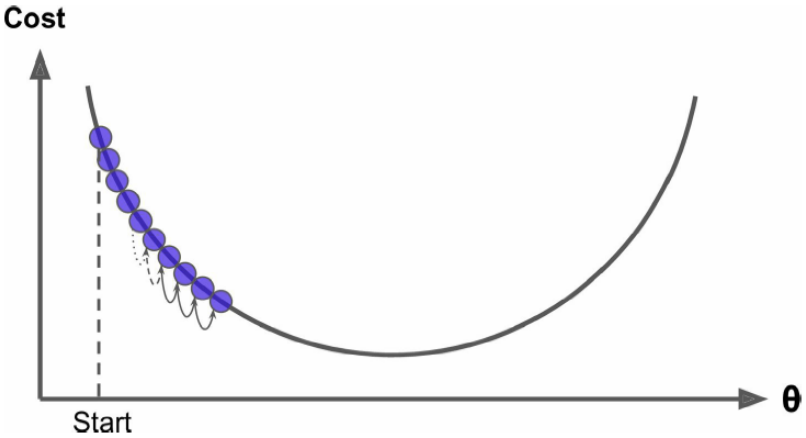


Figure 4-4. Learning rate too small

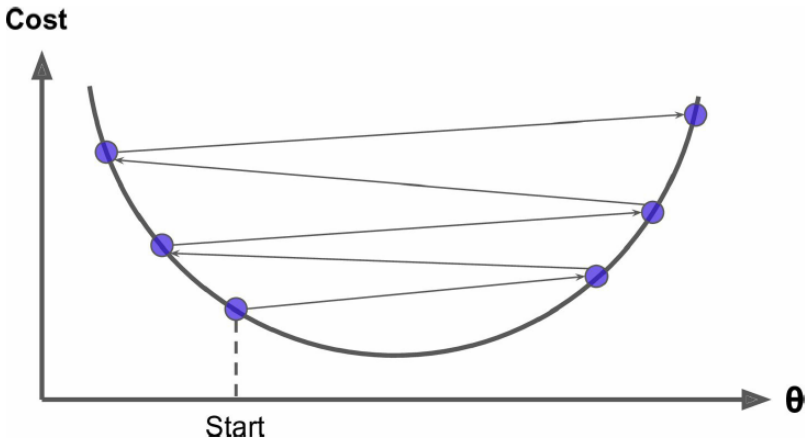


Figure 4-5. Learning rate too large

Gradient Descent: pitfalls

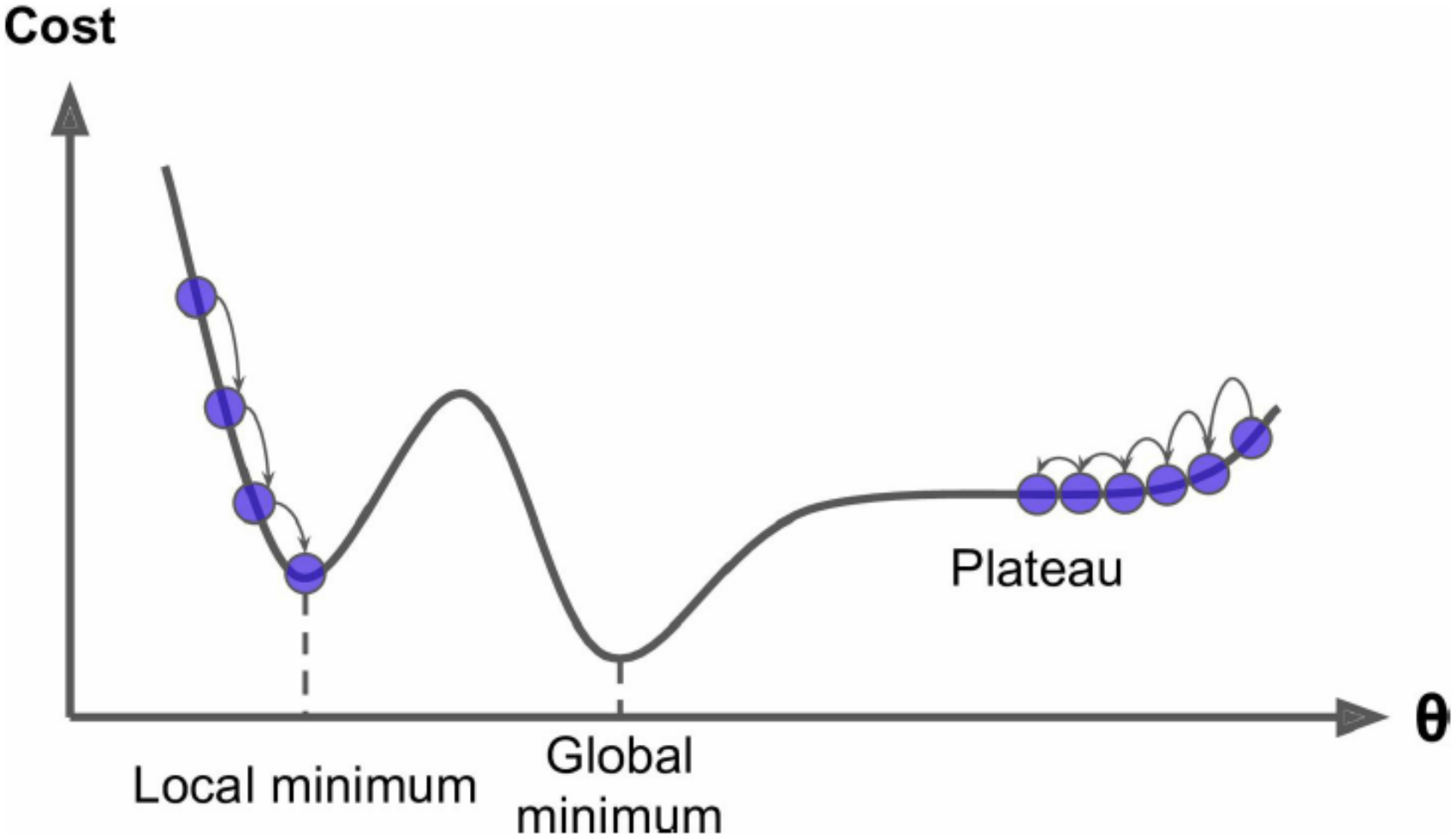


Figure 4-6. Gradient Descent pitfalls

Gradient Descent for a Regression with an MSE cost function

Will be a bowl shape (good) but may be elongated if there are different scales.

How does parameter space help ?? And how is it generated ??

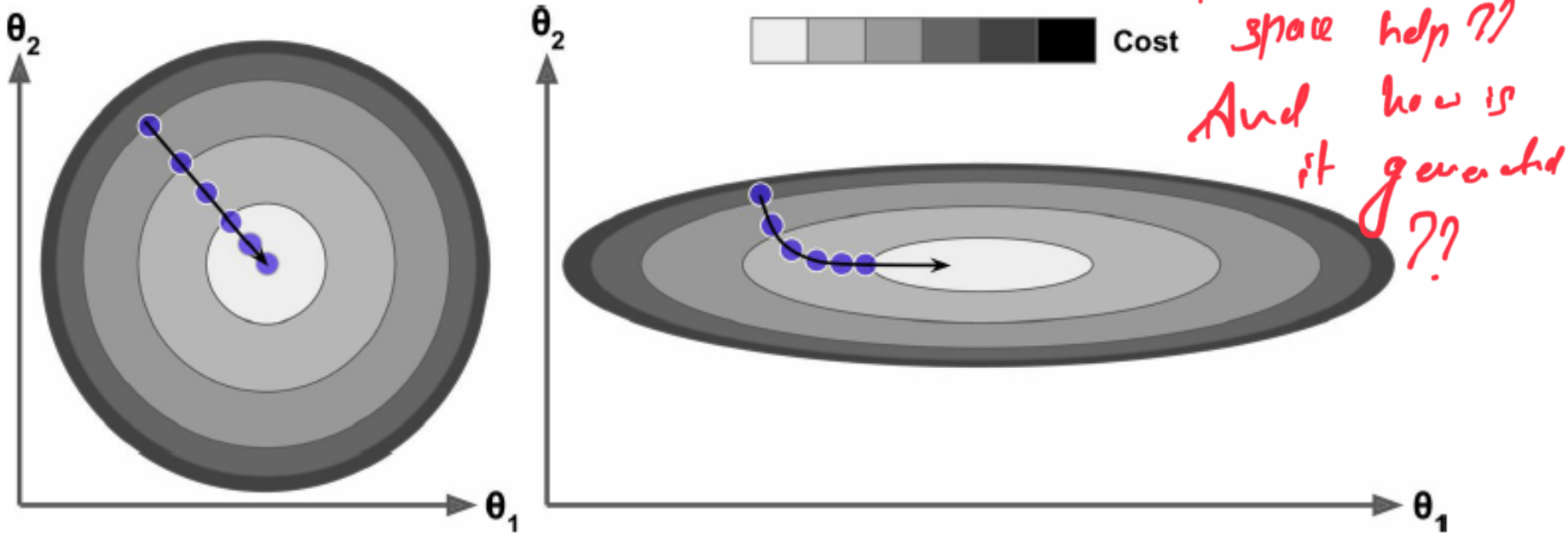


Figure 4-7. Gradient Descent with and without feature scaling

Partial Derivatives and the gradient vector

To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θ_j. In other words, you need to calculate how much the cost function will change if you change θ_j just a little bit. This is called a *partial derivative*.

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

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

GD – implementation

```
eta = 0.1                 # learning rate
n_iterations = 1000       # number of iterations
m = 100                   # number of training instances

theta = np.random.randn(2,1)   # random initialization

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

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

GD with various learning rates

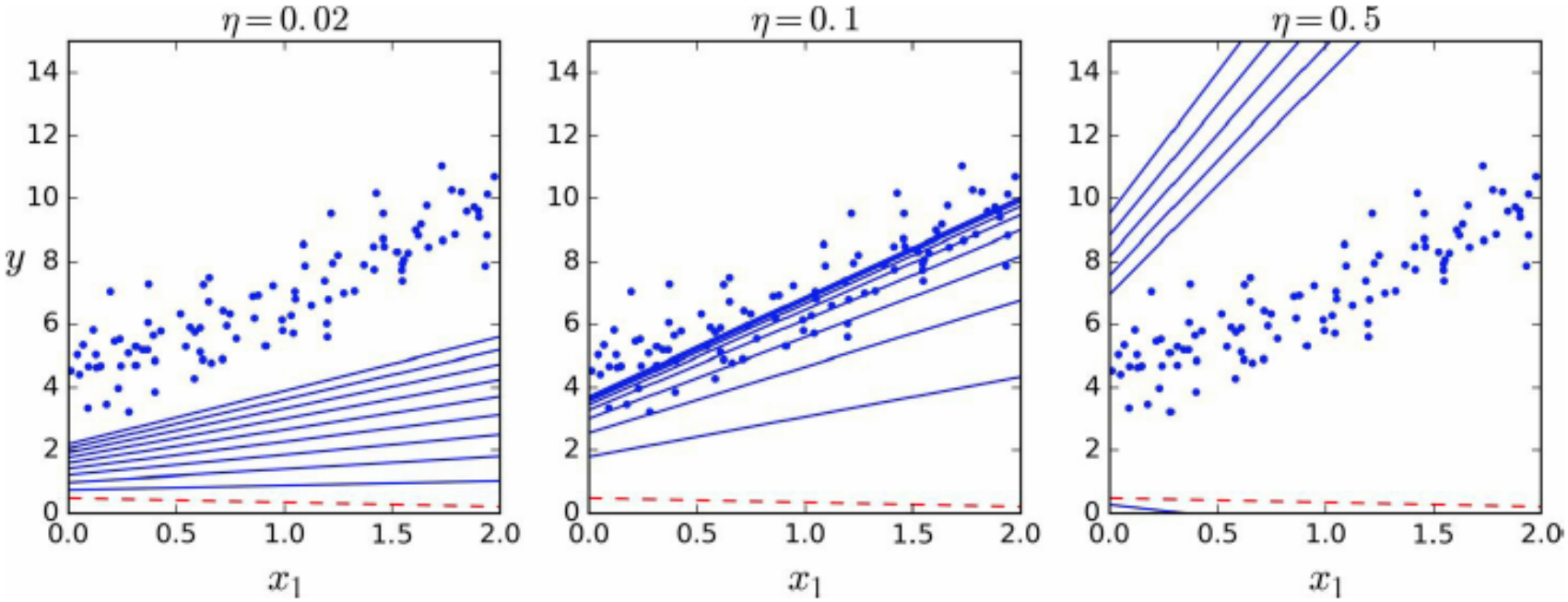
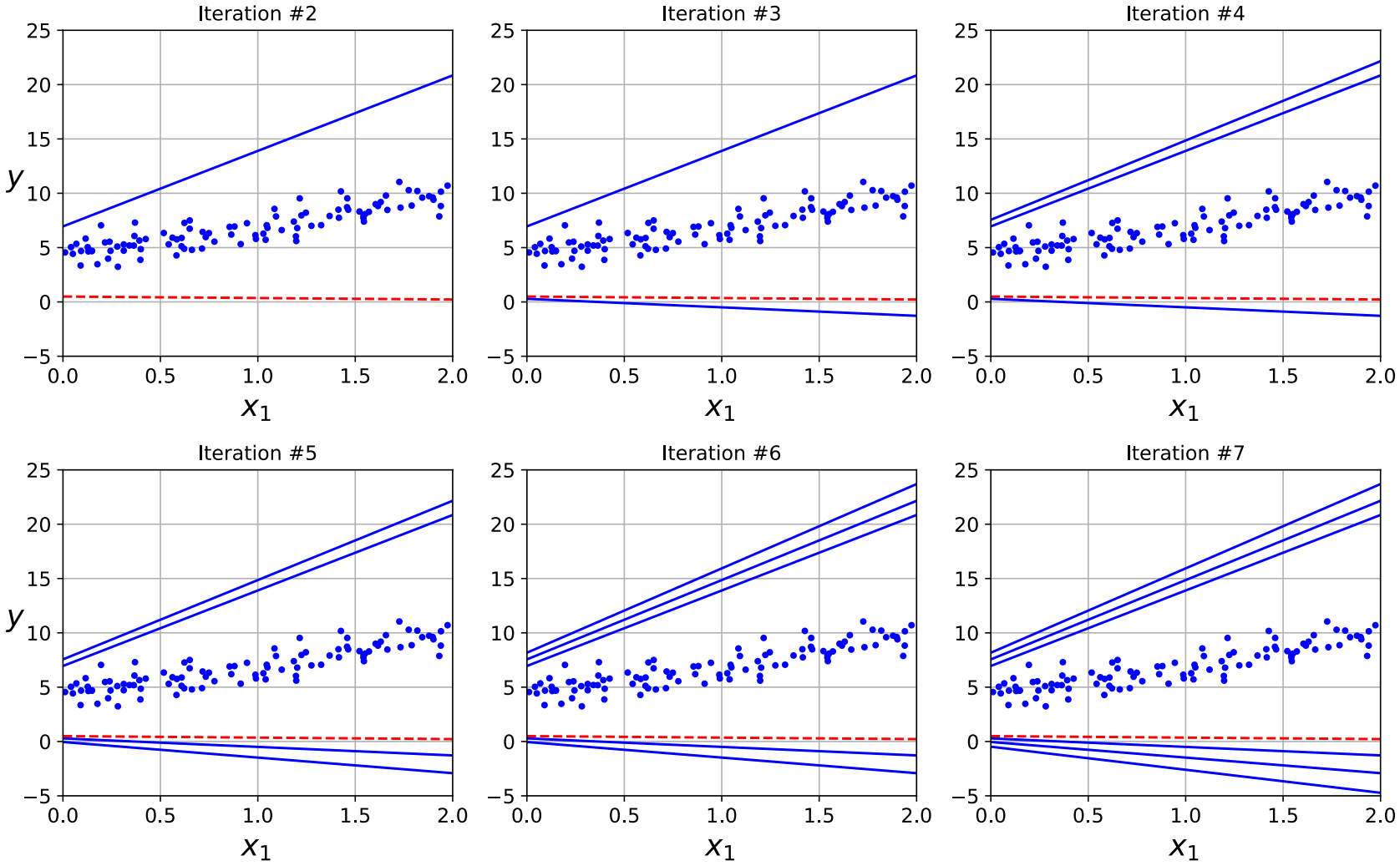


Figure 4-8. Gradient Descent with various learning rates

GD with various learning rates

A divergence example -- when the learning rate is too high ($\eta = 0.5$ in this plot)



3 Ways to do Gradient Descent

Batch GD: Uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.

Stochastic GD: Just picks a random instance in the training set at every step and computes the gradients based only on that single instance. Makes the algorithm much faster since it has very little data to manipulate at every iteration. Also possible to train on huge training sets.

Mini-batch GD: 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), computes the gradients on small random sets of instances (*mini-batches*). Get a performance boost from hardware optimization of matrix operations (cf GPUs).

Stochastic Gradient Descent (SGD) jumps around:

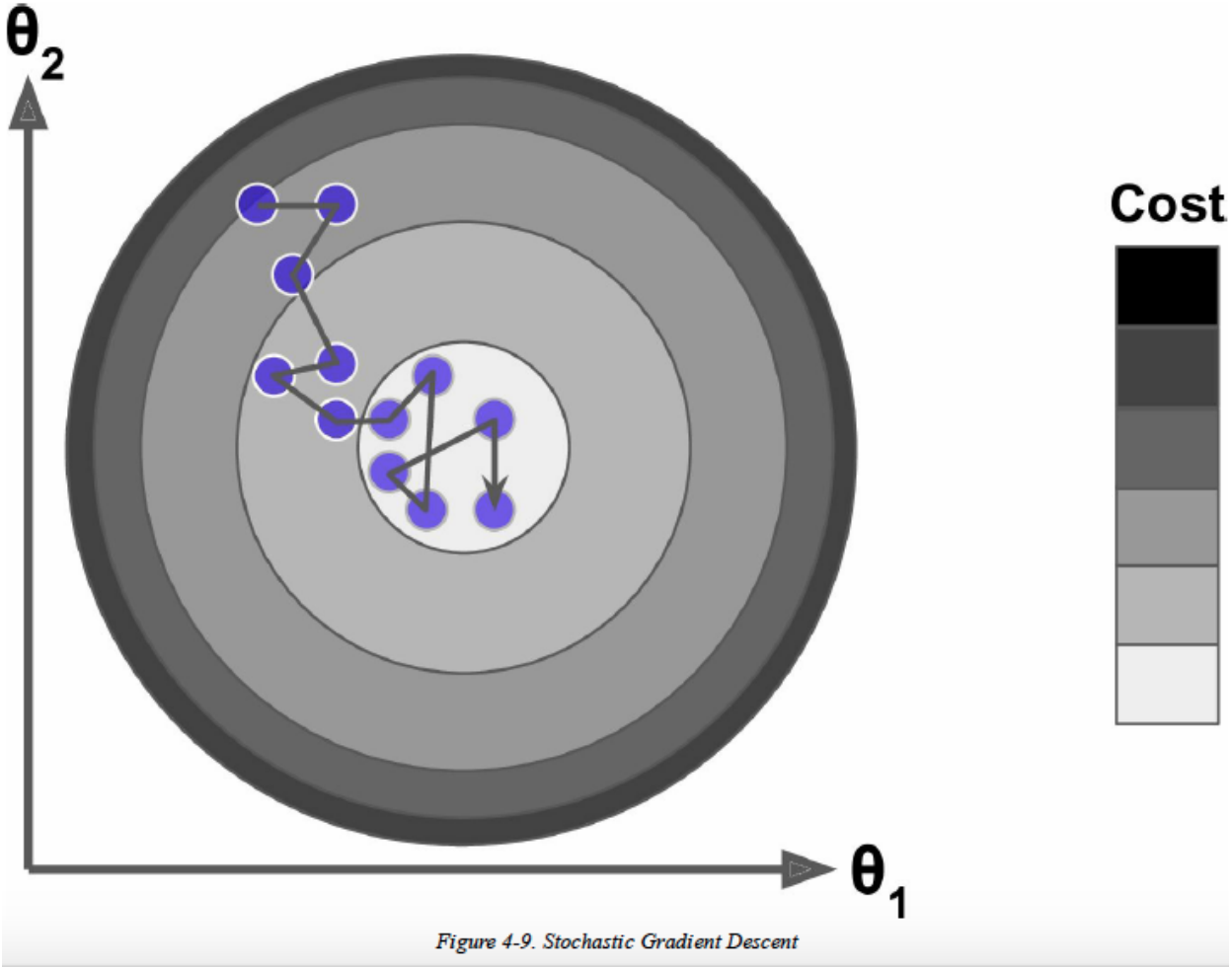


Figure 4-9. Stochastic Gradient Descent

One option is to gradually reduce the learning rate to allow the algorithm to settle to the global minimum.

Stochastic Descent

Reducing the learning rate gradually

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization
for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

Should explain a bit?

(m is the size of the training set)

Stochastic Descent in Action

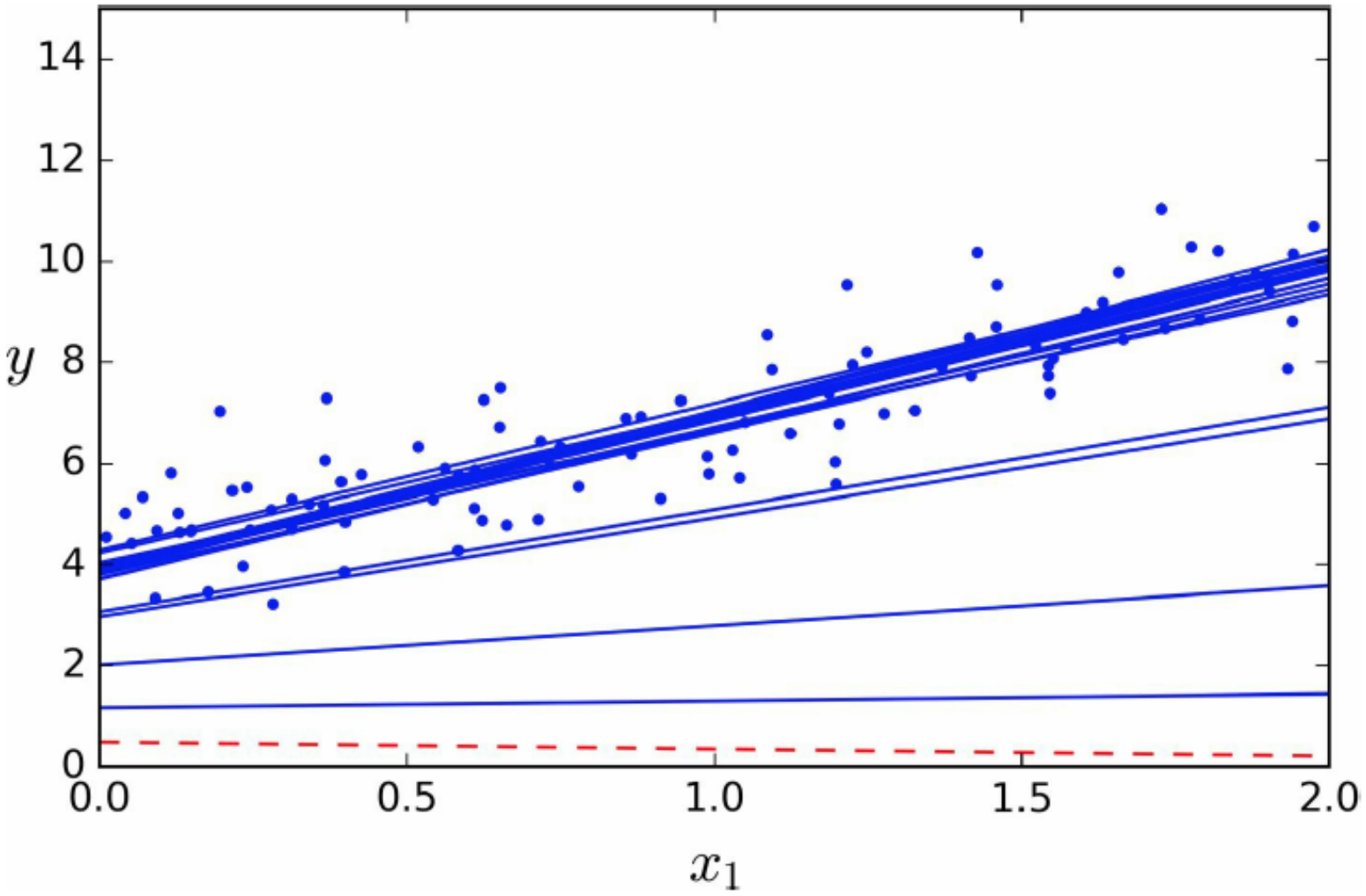


Figure 4-10. Stochastic Gradient Descent first 10 steps

Paths in Parameter Space

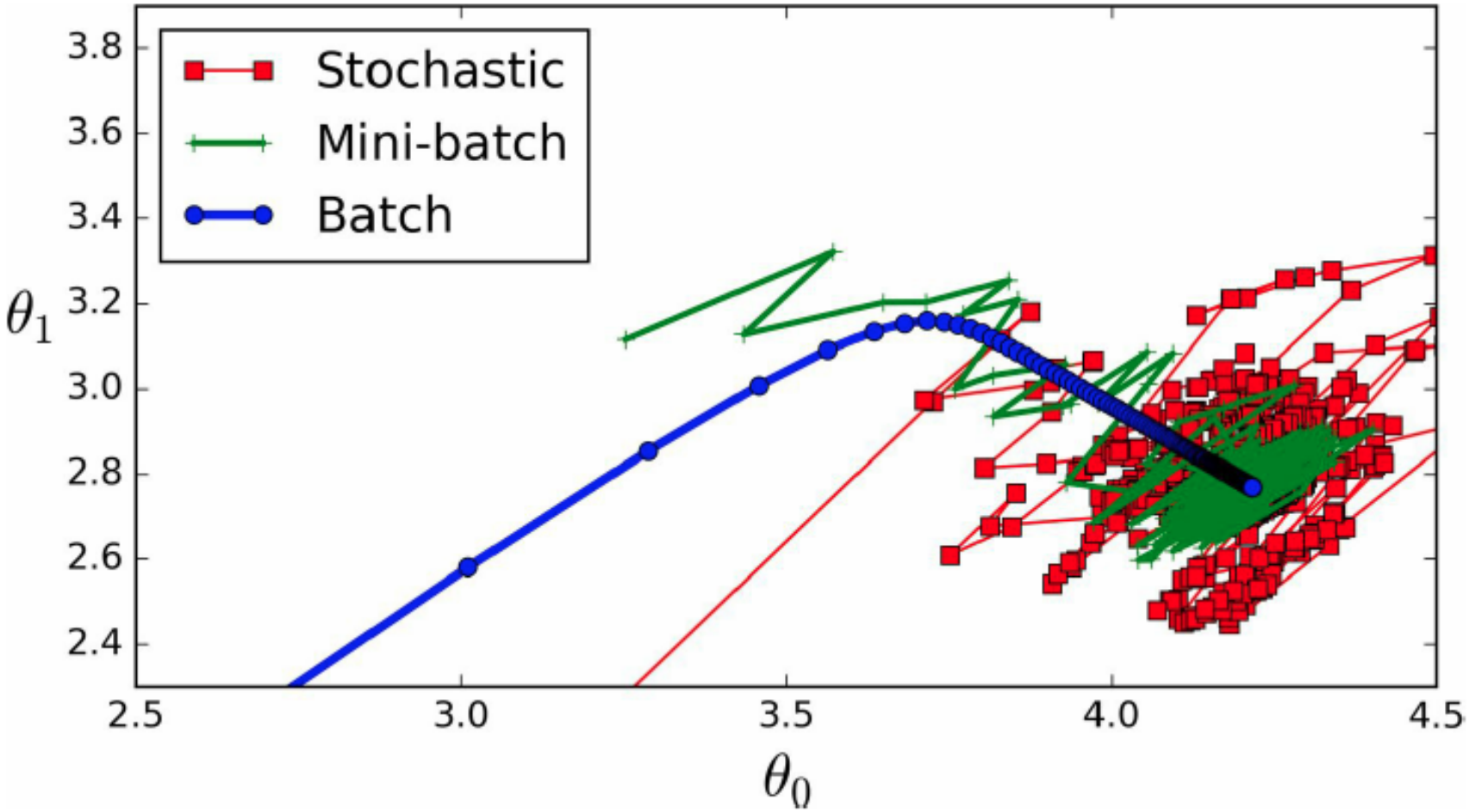


Figure 4-11. Gradient Descent paths in parameter space

Polynomial Regression

What if your data is actually more complex than a simple straight line?

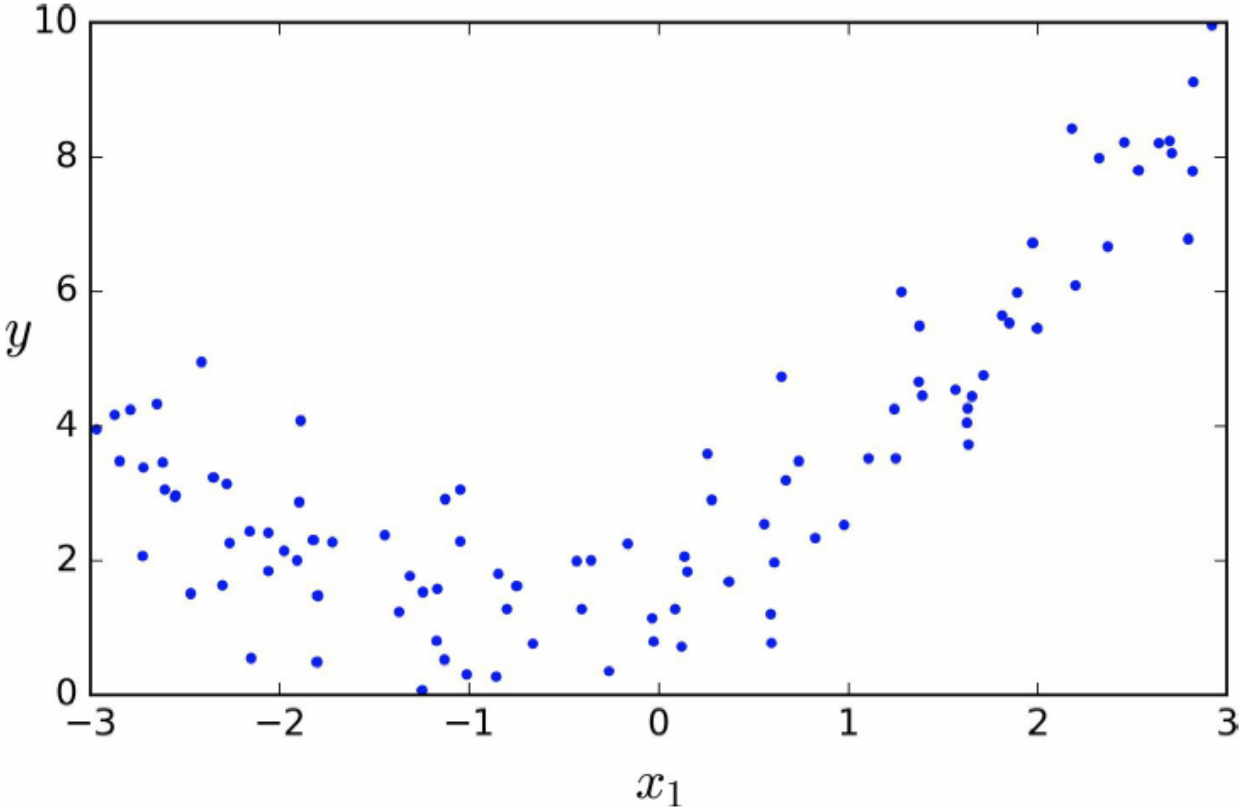


Figure 4-12. Generated nonlinear and noisy dataset

In the above example, $y = 0.5x^2 + x + 2 +$ Gaussian noise.

... Just use linear regression

But introduce a new feature which is x_1^2.

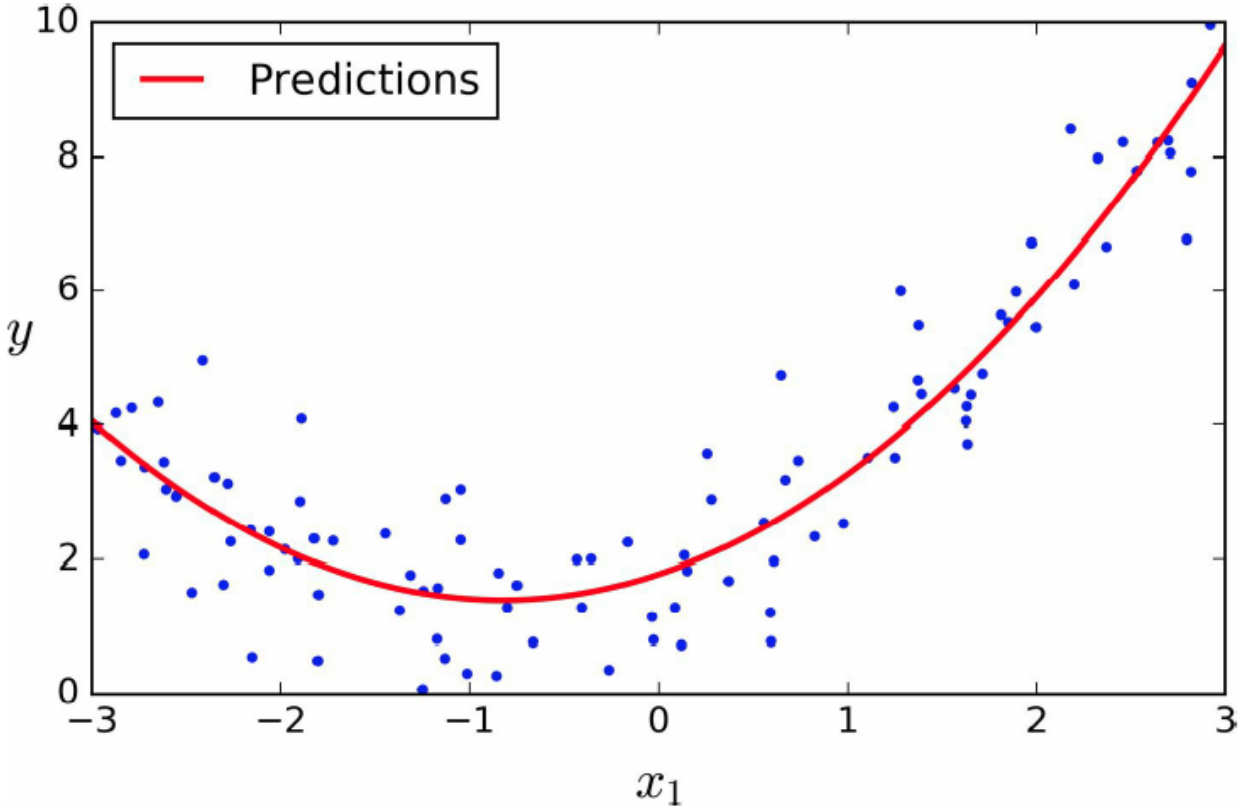


Figure 4-13. Polynomial Regression model predictions

The fitted curve (red) is: $y = 0.56x^2 + 0.93x + 1.78 = 0$. Not bad at all!

See Scikit-Learn's `PolynomialFeatures` class. Easy!

Learning Curves

A polynomial of a large degree can fit data better

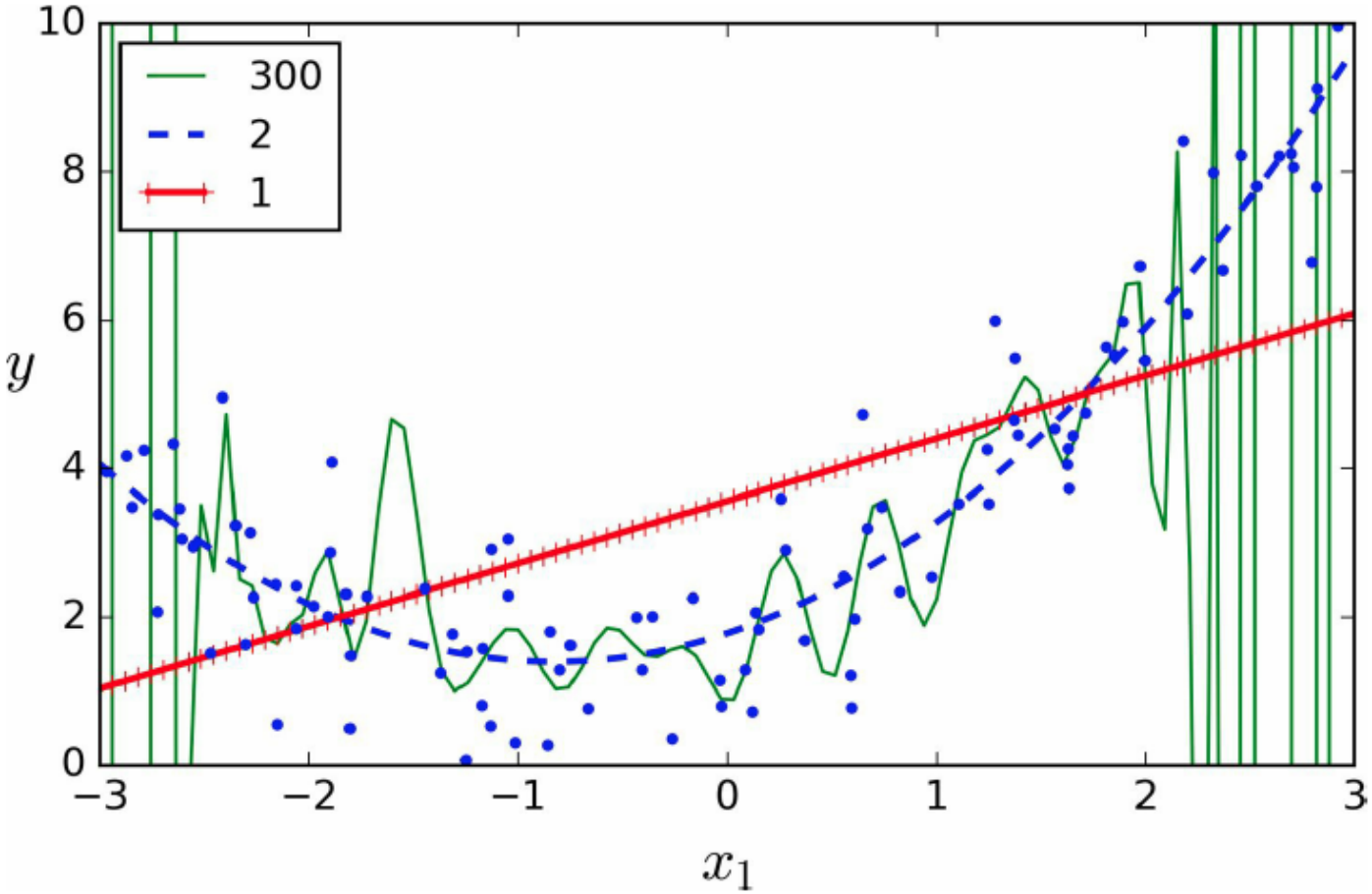


Figure 4-14. High-degree Polynomial Regression

Learning Curves

In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance.

- If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting.
- If it performs poorly on both, then it is underfitting.

This is one way to tell when a model is too complex or too simple.

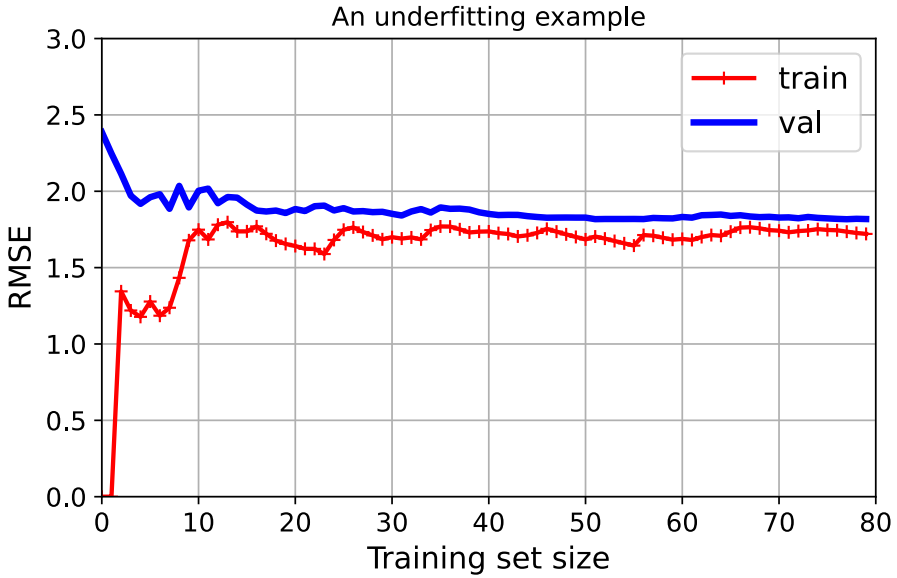
Another way is to look at the *learning curves*.

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.

See the code in the text book for `plot_learning_curves(model, X, y)`.

Plot of learning curves: plain Linear Regression model



(Simulated data is:

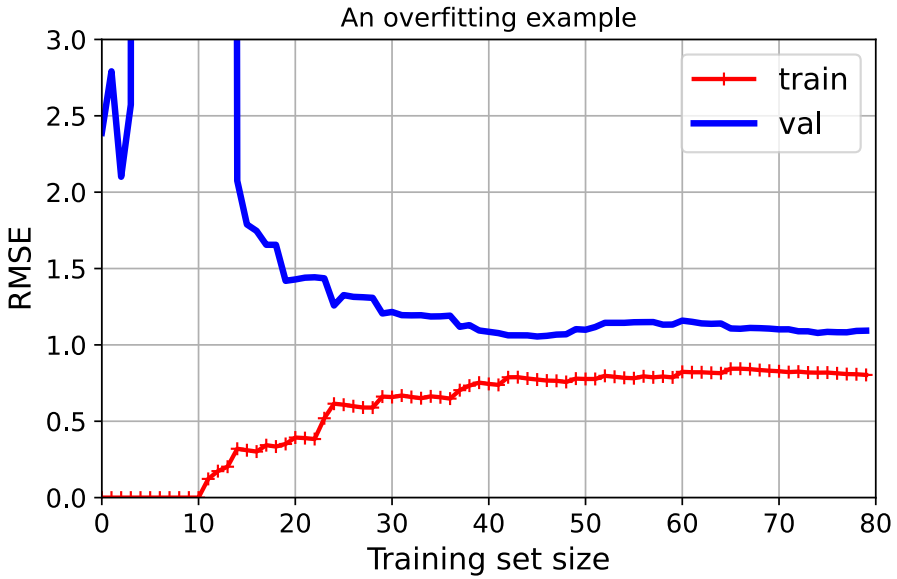
y = 0.5 * X**2 + X + 2 + noise

Try to fit a linear model)

Both curves reach a plateau; they are close and fairly high[1]. A clear under-fitting model.

[1] note that the vertical axis is RMSE so the errors for both curves are fairly high

Plot of learning curves: 10th Polynomial Regression model



(Simulated data is:

y = 0.5 * X**2 + X + 2 + noise

Try to fit a 10th order polynomial model)

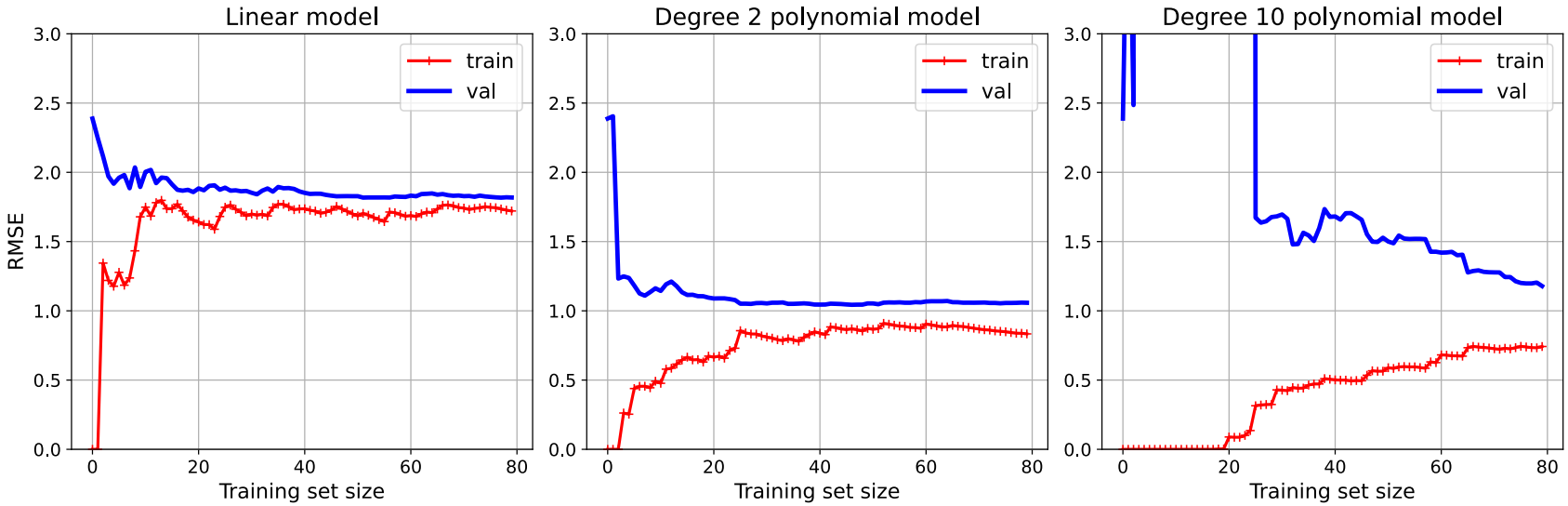
Learning curves for a 10^{th} degree polynomial:

- the error on the training data is much lower than the 1^{st} degree polynomial in the previous slide
- there is a gap between the two curves

Plots of learning curves

how to differentiate better model?

how this is good? the side has error in labels



The simulated data is: y = 0.5 * X**2 + X + 2 + noise

Logistic Regression

Some regression algorithms can be used for classification as well (and vice versa).

Logistic Regression (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labelled "1"), or else it predicts that it does not (i.e., it belongs to the negative class, labelled "0"). This makes it a *binary classifier*.

How does a logistic model make a prediction?

Just like a *Linear Regression* model, a *Logistic Regression* model computes a weighted sum of the input features (plus a bias term), The probability $\hat{p}$ estimated by the Logistic Regression model is given by:

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x})$$

where the logistic function $\sigma(t)$ is defined as follows:

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

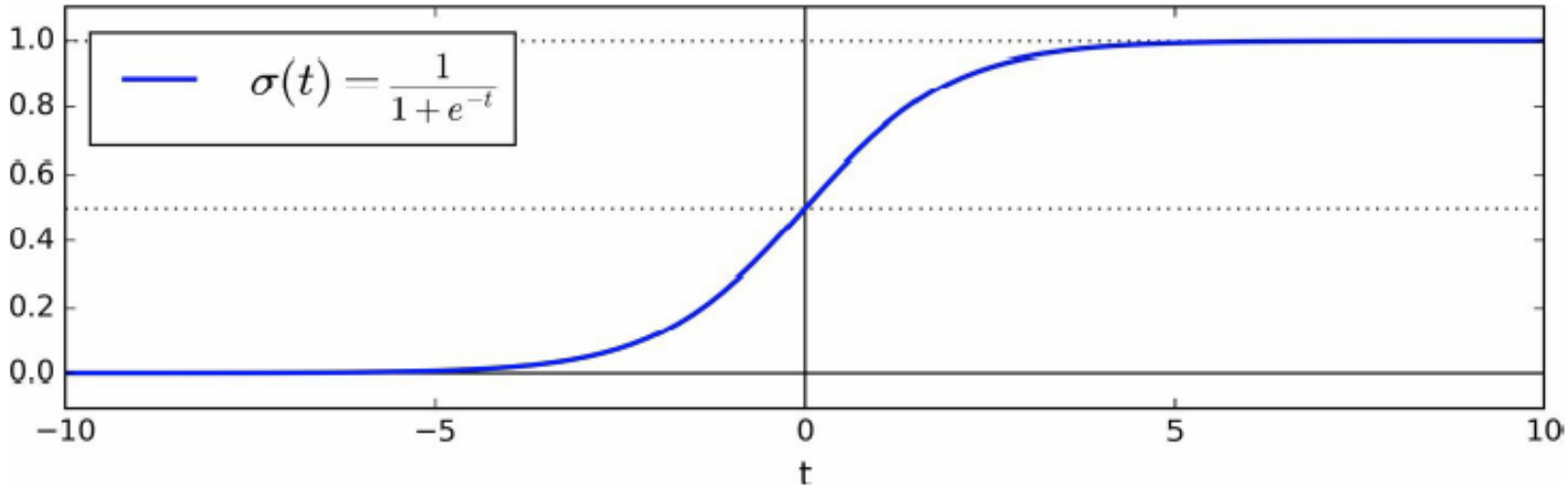


Figure 4-21. Logistic function

How does a logistic model make a prediction?

The probability $\hat{p}$ estimated by the Logistic Regression model is given by:

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^\top \mathbf{x})}$$

The Logistic Regression model prediction is:

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

A Logistic Regression model predicts 1 if $\boldsymbol{\theta}^\top \mathbf{x}$ is positive, and 0 if it is negative.

Training and Cost Function

The *Logistic Regression cost function* (*log loss*) is:

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

C(θ) ε ? → how did you determine that ?

- Want that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances.
- Can do this using a simple log function of the probability.
- Average over all instances.
- No known closed-form equation to compute the value of $\boldsymbol{\theta}$ that minimizes this cost function ← learn more ? ← how do we know that ?
- But cost function is convex, so GD is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).

Iris Example: three species

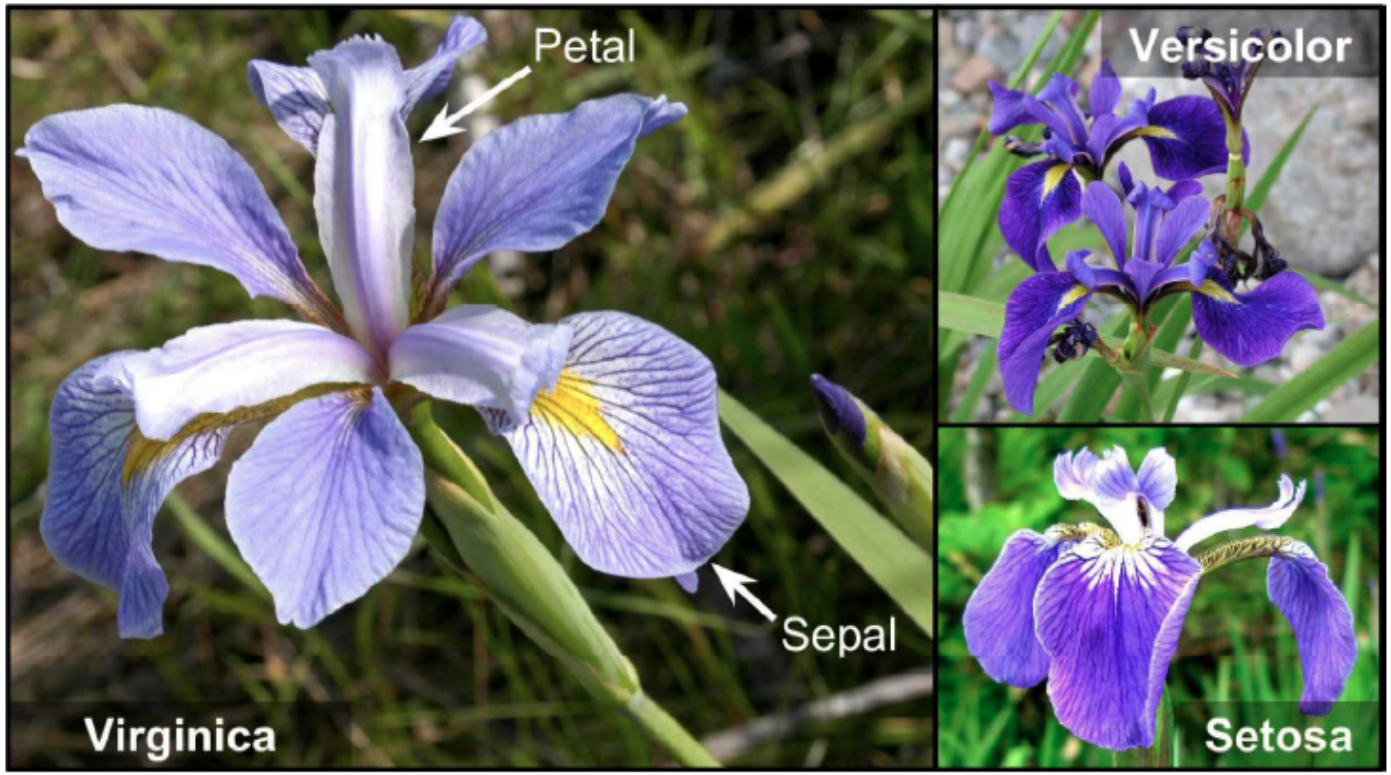


Figure 4-22. Flowers of three iris plant species[16]

Some flower properties vary with the species.
A 3-class classification problem, with 4 attributes (all in cm): *sepal length*, *sepal width*, *petal length*, and *petal width*.

Just using petal length feature

Suppose that we want to do a binary classification: to classify *Iris-Virginica* type versus *not Iris-Virginica* type, using just the *petal width* feature. After training a Logistic Regression classifier, we get a *decision boundary* at about 1.6cm.

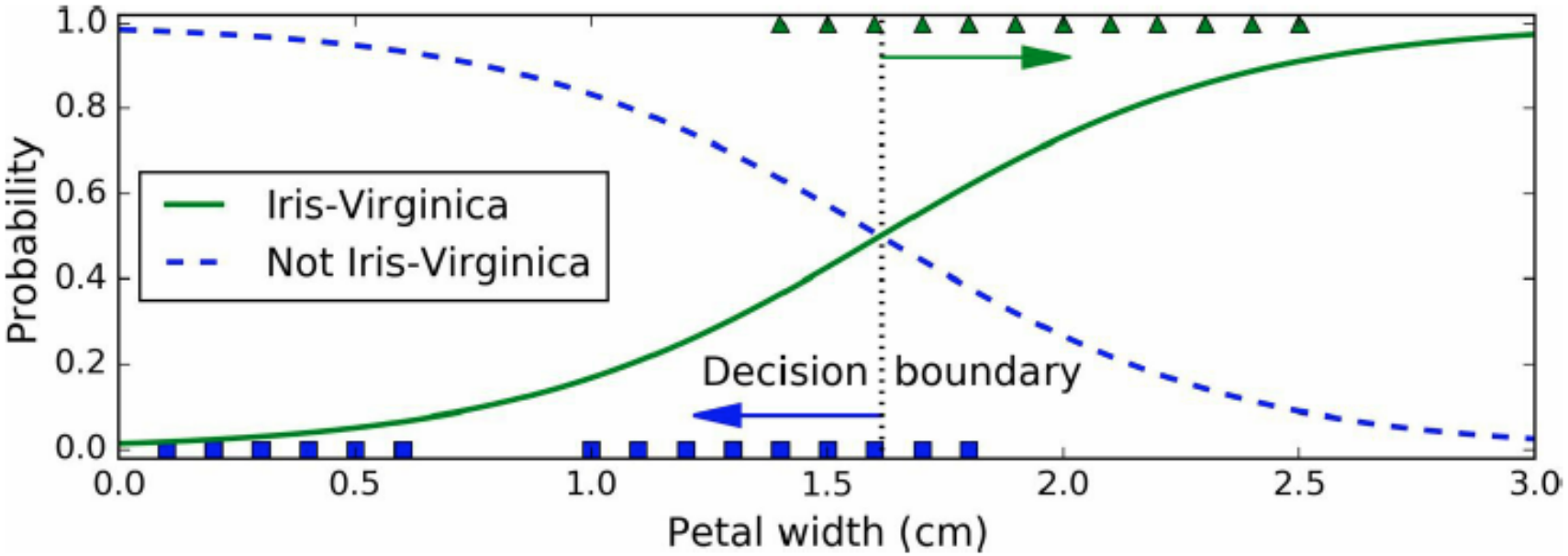


Figure 4-23. Estimated probabilities and decision boundary

green triangles: Iris-Virginica; blue squares: other iris species

With two features...

Again, we use the Logistic Regression classifier but on two features: *petal length* and *petal width*.
... we also get a linear boundary.

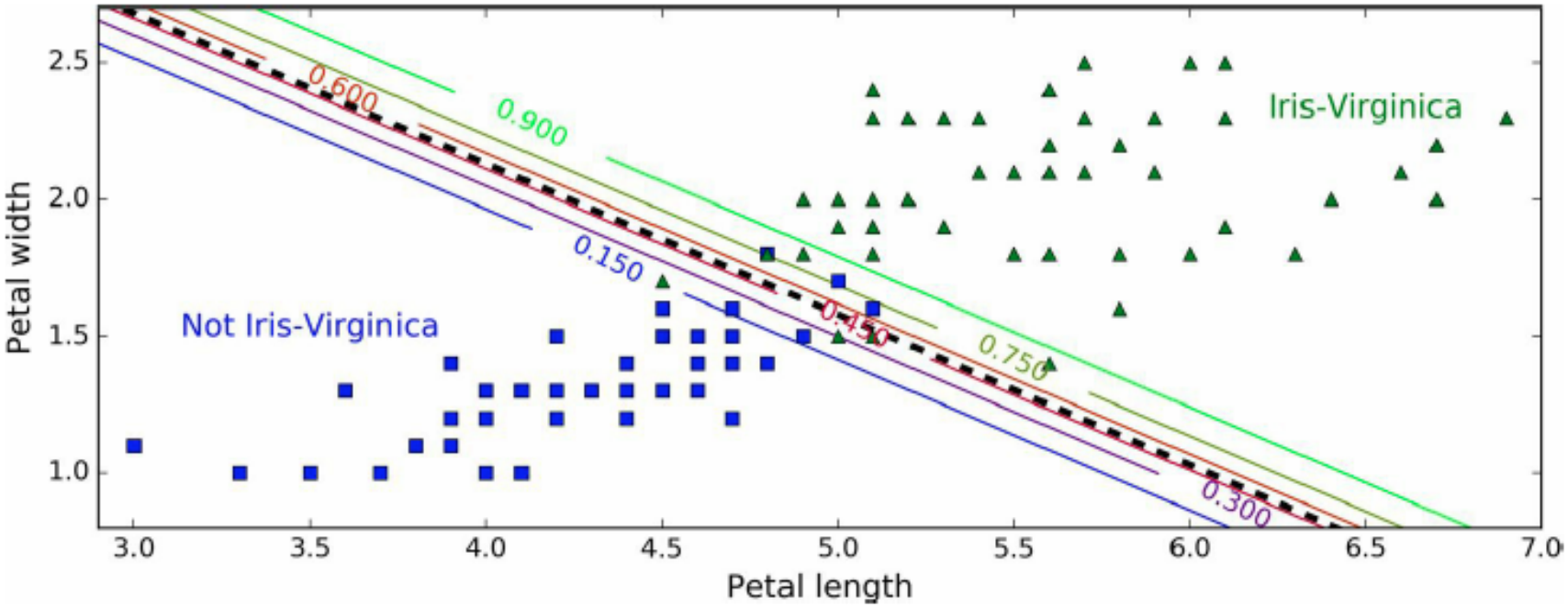


Figure 4-24. Linear decision boundary

The black dashed line represents the model's decision boundary (50% probability)

For next week

Work through your first assignment and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Assignment 1 (10%) requires a submission to *LMS* - check the unit calendar.

Read up to Chapter 4 Training Models.

And that's all for the third lecture.

Have a good week.