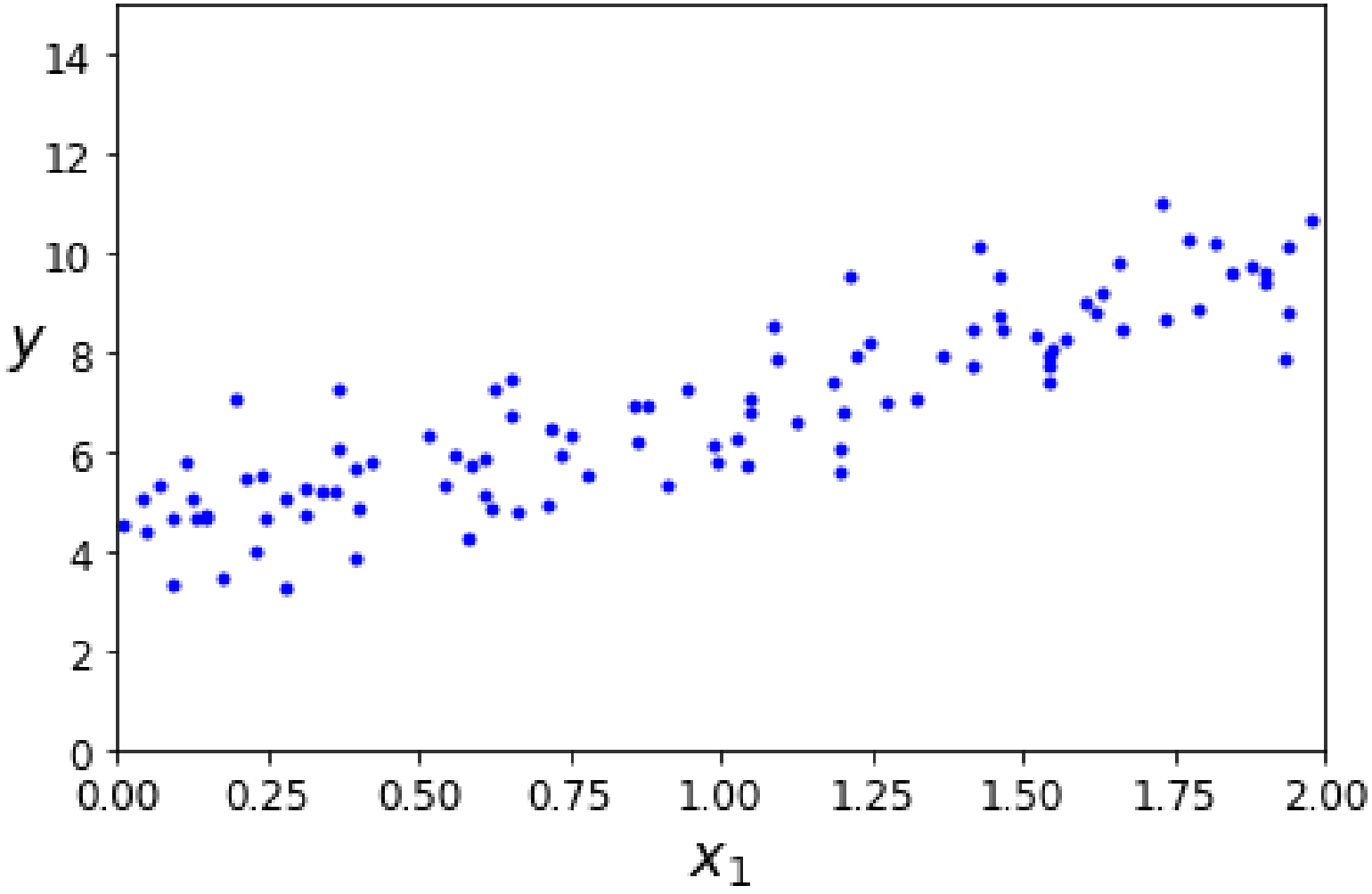
CSY2082 Introduction to Artificial Intelligence

Regression 1

Regression

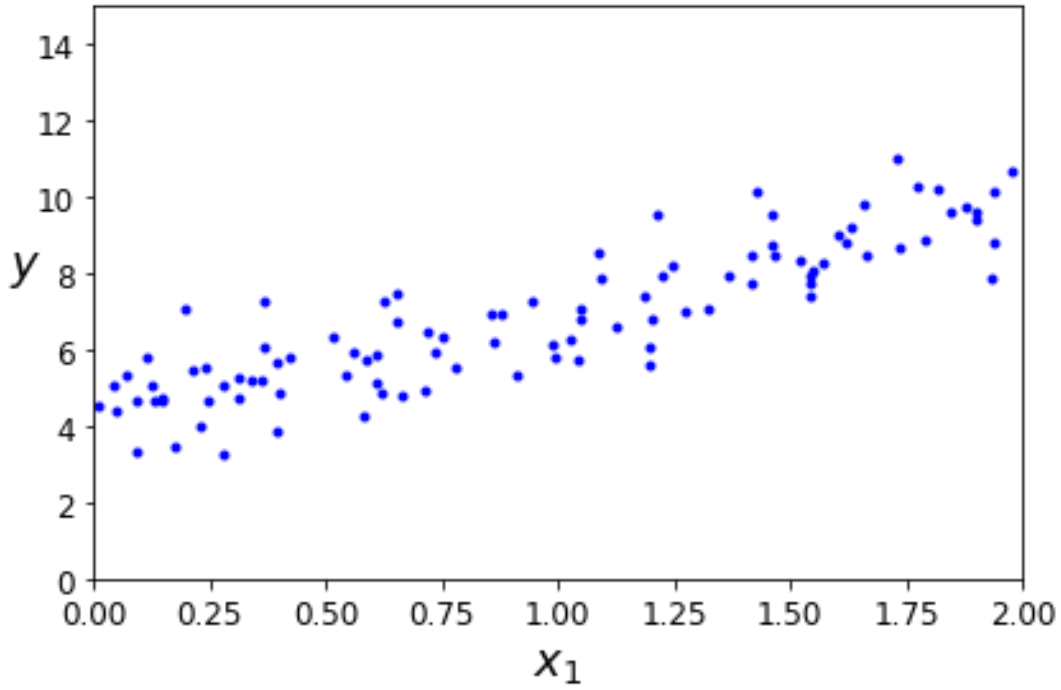
- Relationship between one dependent variable and a series of other variables (independent variables)



Regression

- Create a dataset

```
import numpy as np

np.random.seed(42)  # to make this code example reproducible
m = 100  # number of instances
X = 2 * np.random.rand(m, 1)  # column vector
y = 4 + 3 * X + np.random.randn(m, 1)  # column vector
```



numpy.random.rand

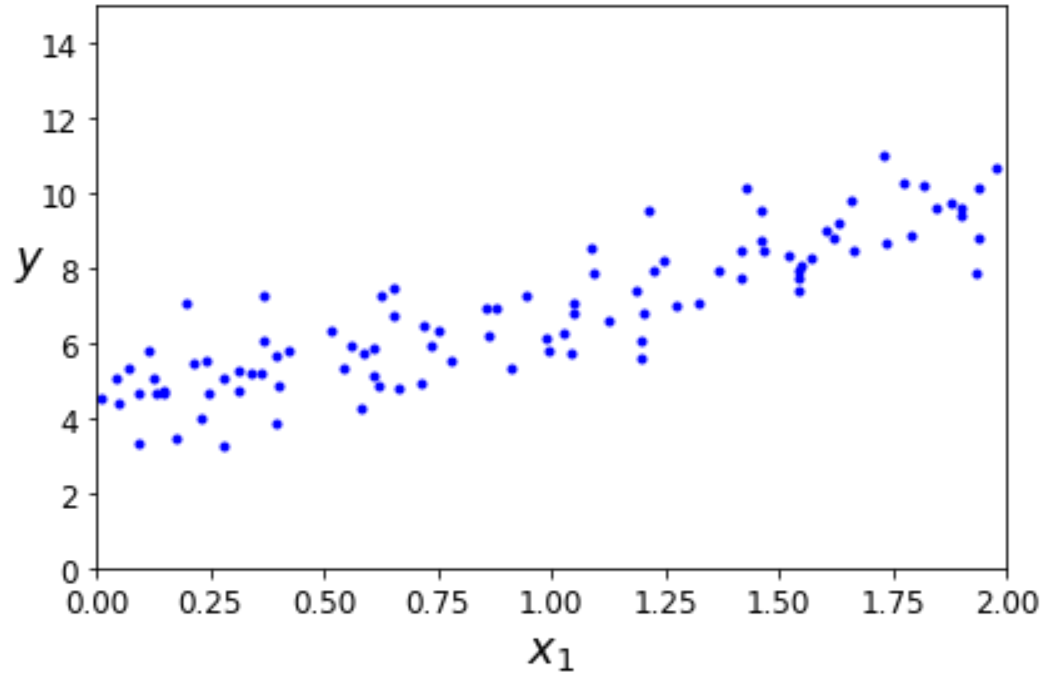
random.rand(*d0*, *d1*, ..., *dn*)

Random values in a given shape.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

Regression

- Two very different ways to train it:
  - Using a direct "**closed-form**" equation that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimise the cost function over the training set). More details in the reference book.
  - Using an **iterative optimization approach**, called Gradient Descent (GD), that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method.



Regression

- Linear Regression
- Polynomial Regression
- Logistic Regression
- Softmax Regression

Linear Regression

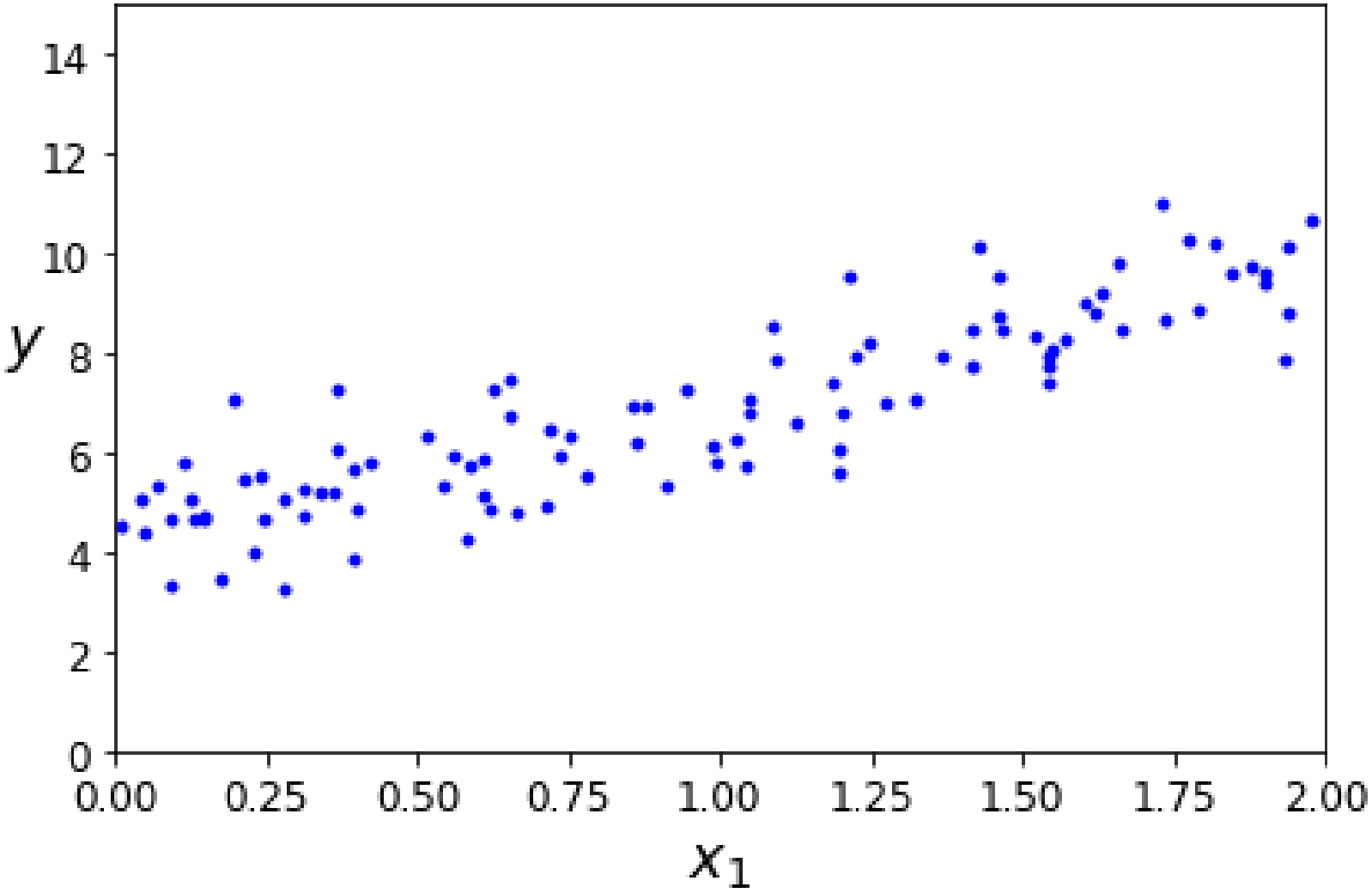
- a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (also called the intercept term)

Equation 4-1. Linear Regression model prediction

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

Equation 4-3. MSE cost function for a Linear Regression model

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



Using sklearn's LinearRegression (when dataset is small)

```
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]]))
```

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

```
y_predict=lin_reg.predict(X_new)
```

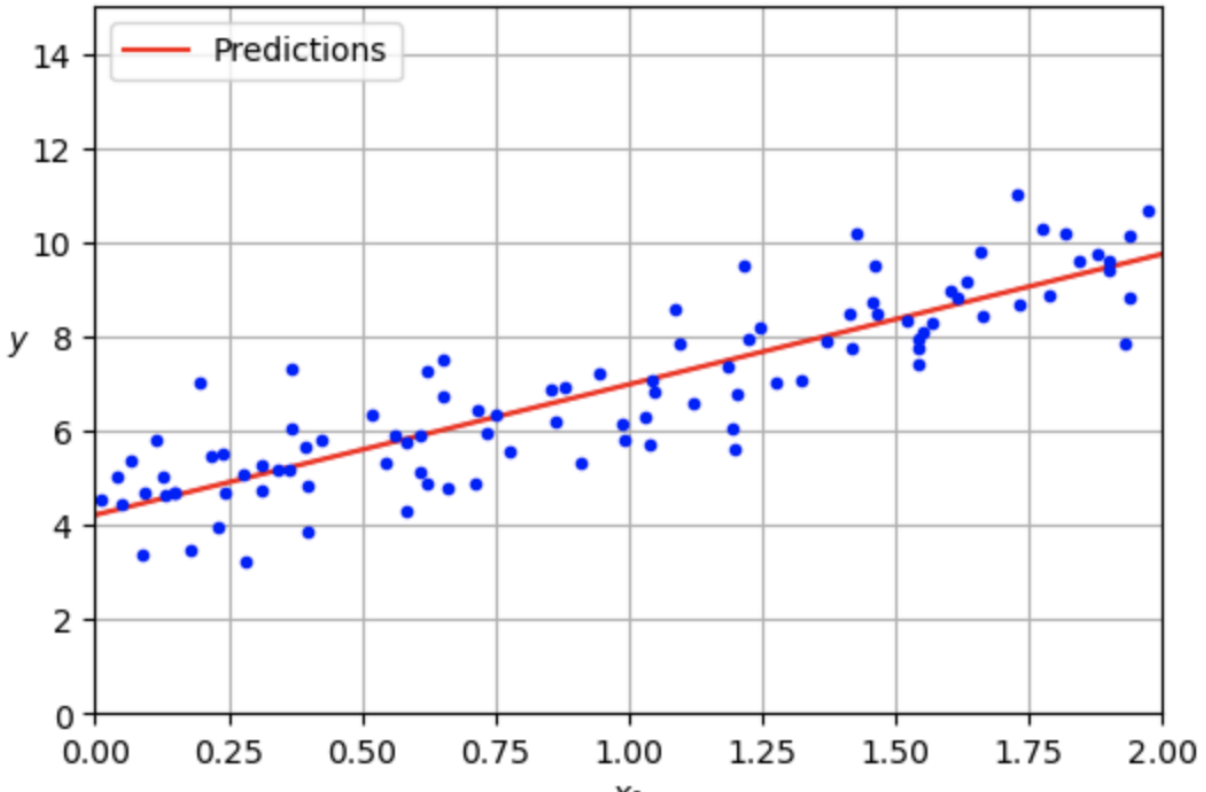
```
y_predict
```

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

The `LinearRegression` class is based on "least squares parameter approximation":

https://www.youtube.com/watch?v=JvS2triCgOY

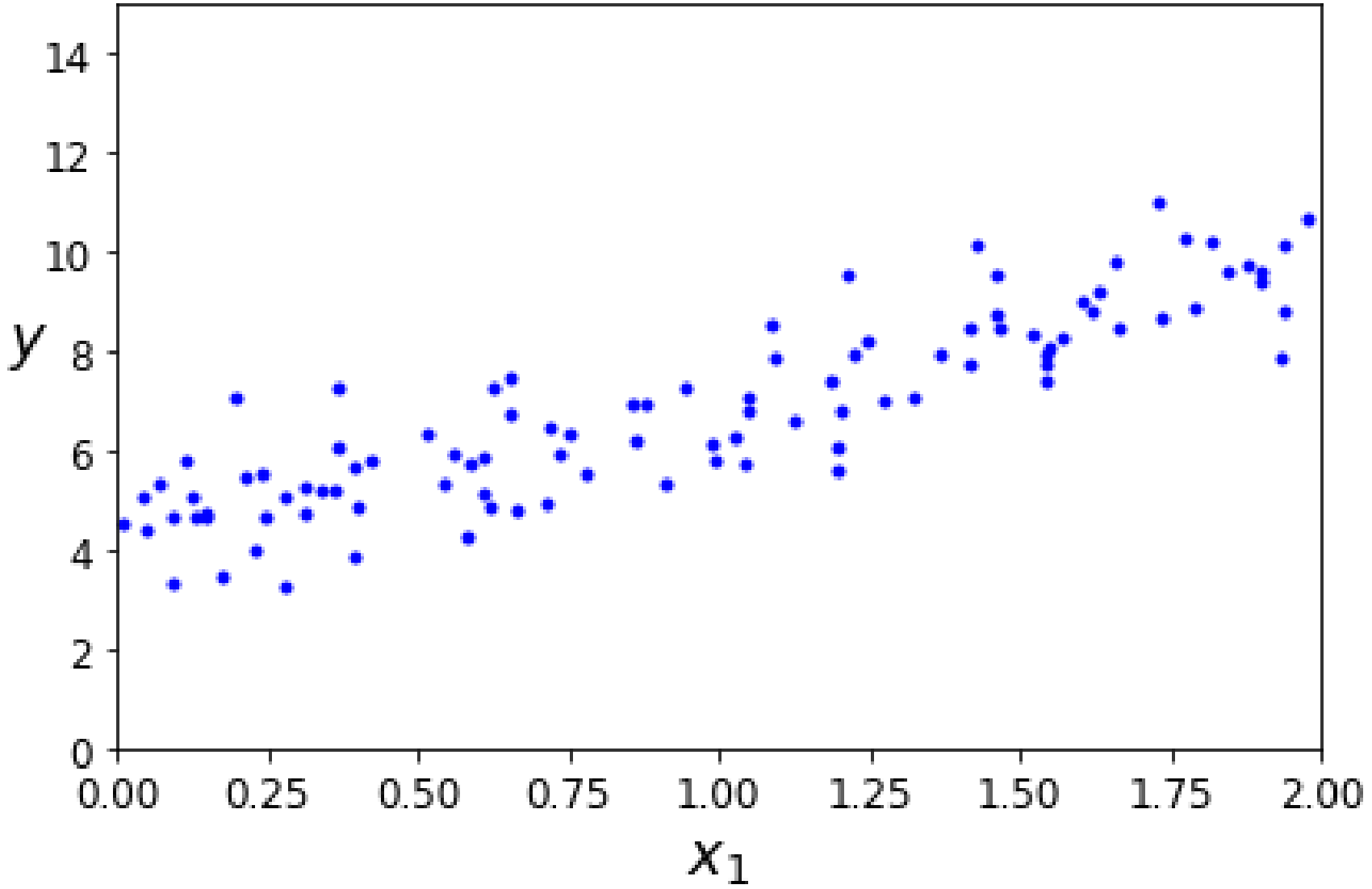
```python
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))  # extra code – not needed, just formatting
plt.plot(X_new, y_predict, "r-", label="Predictions")
plt.plot(X, y, "b.")

# extra code – beautifies and saves Figure 4–2
plt.xlabel("$x_1$")
plt.ylabel("$y$", rotation=0)
plt.axis([0, 2, 0, 15])
plt.grid()
plt.legend(loc="upper left")
#save_fig("linear_model_predictions_plot")

plt.show()
```



Gradient Descent

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

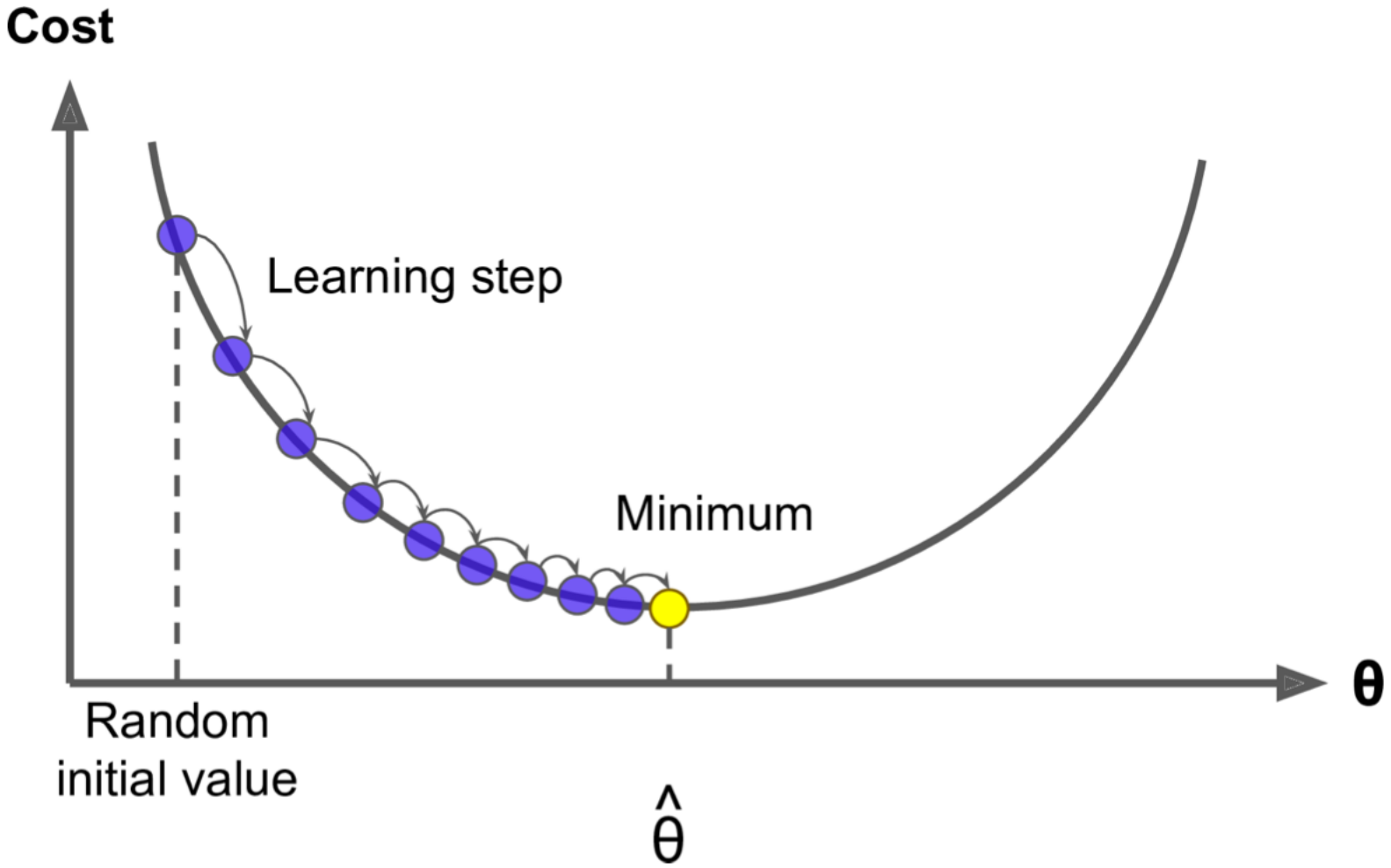
$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)} \right)^2$$



Gradient Descent

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

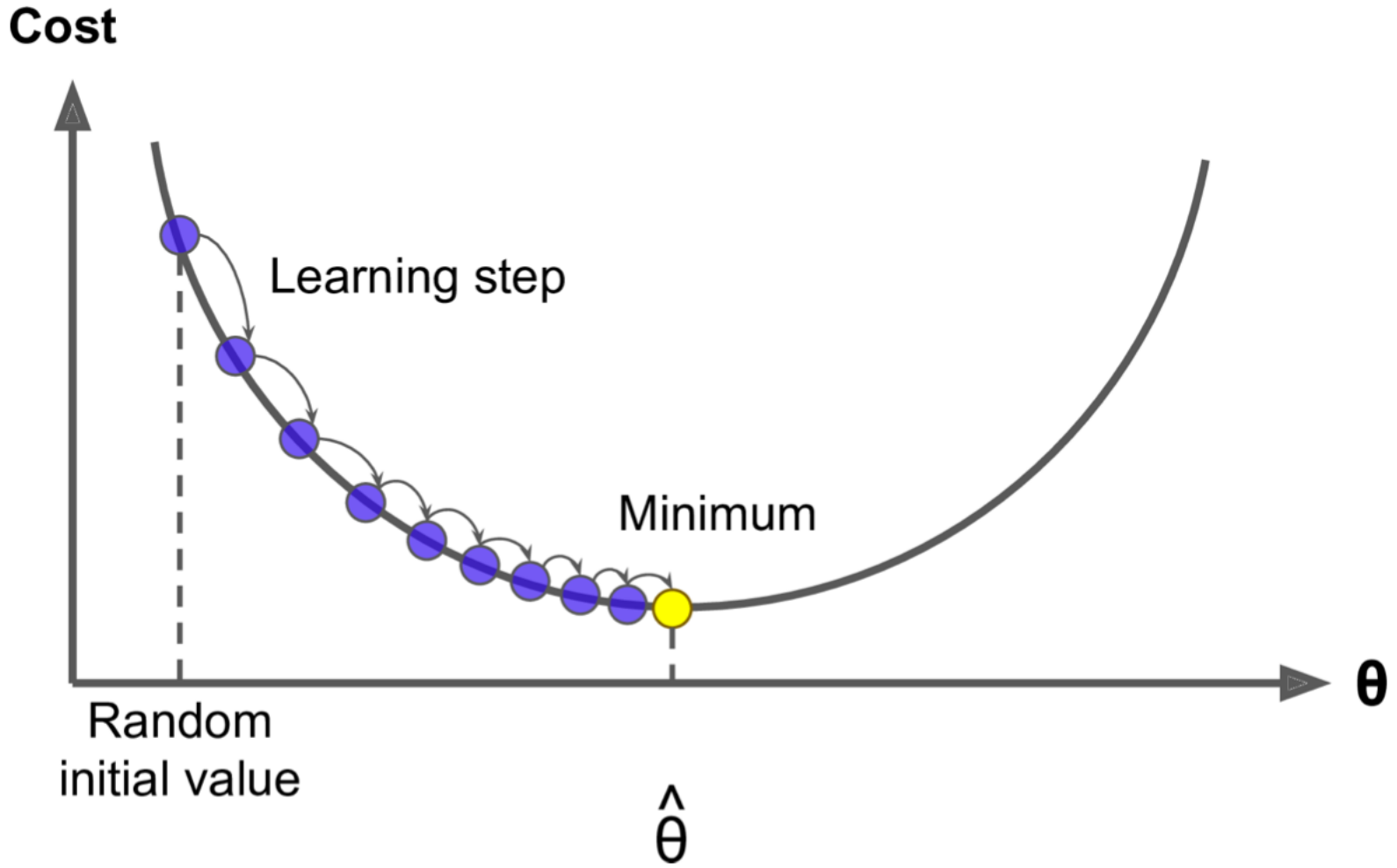
- Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to **tweak** parameters iteratively in order to **minimise a cost function**.



Gradient Descent

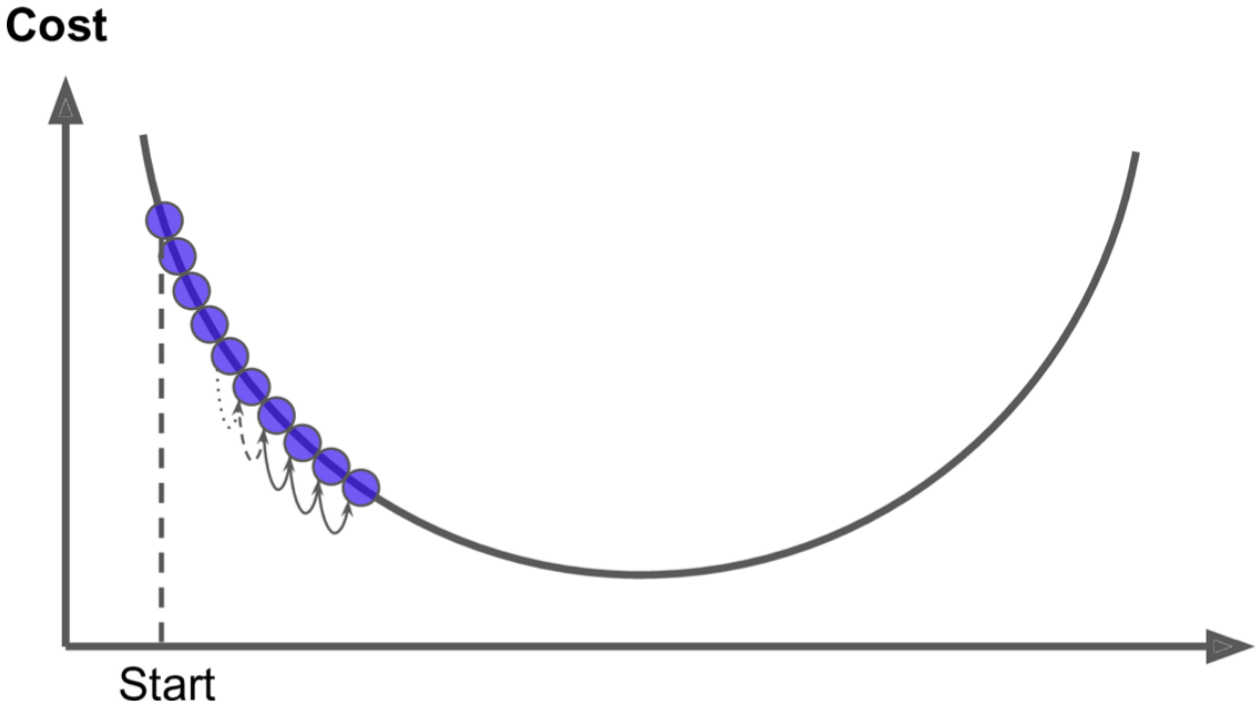
$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

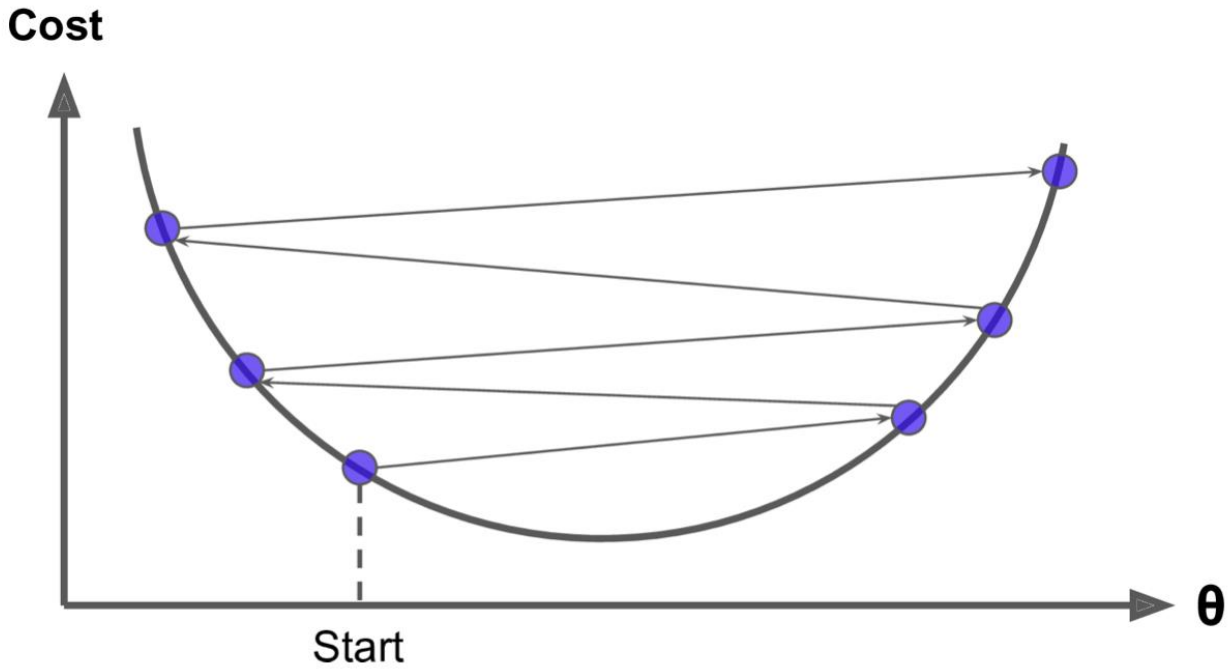
- Concretely, you start by filling θ with random values (this is called random initialization), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum



Gradient Descent

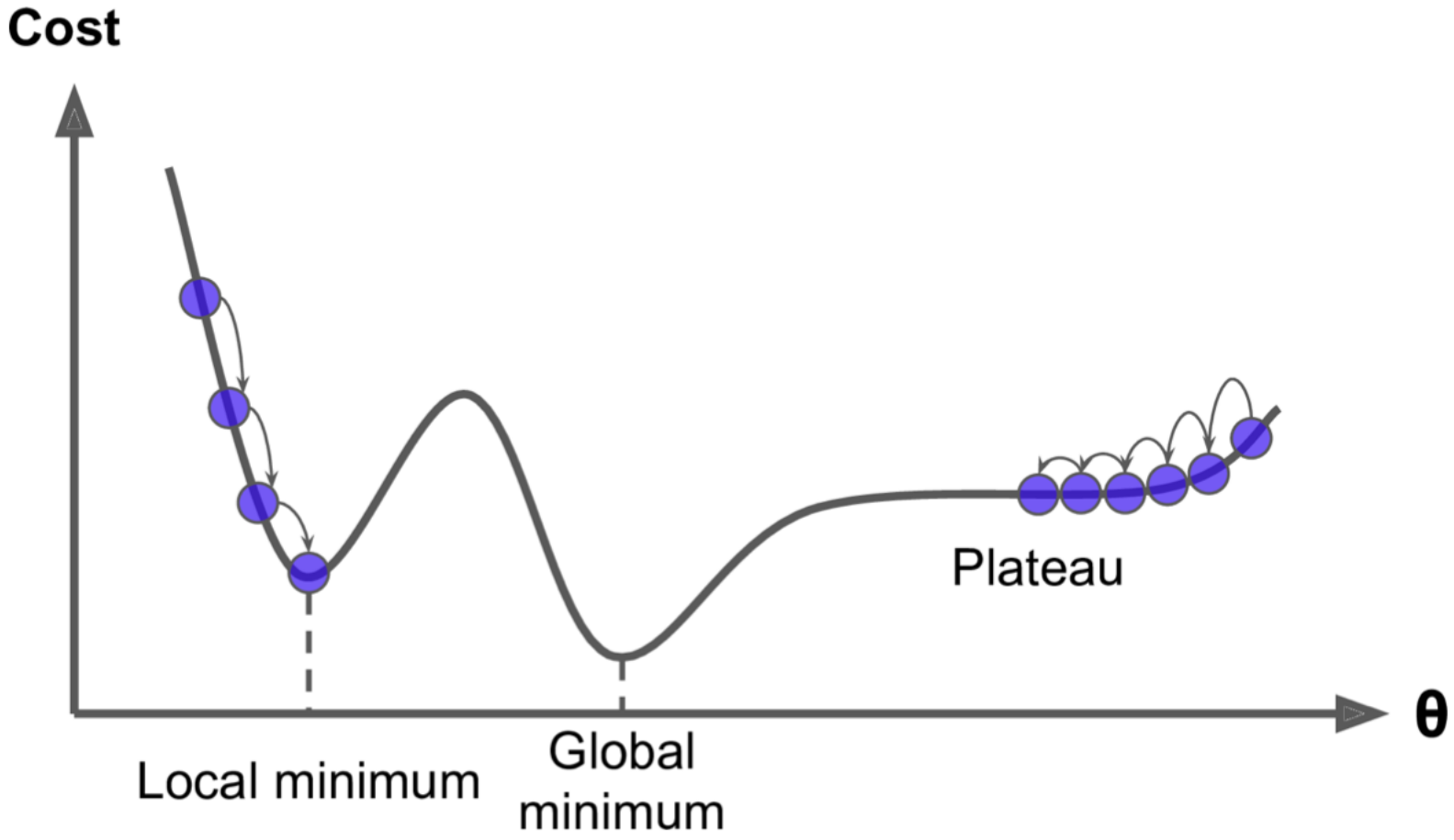
- An important parameter in Gradient Descent is the size of the steps, determined by the learning rate hyperparameter.





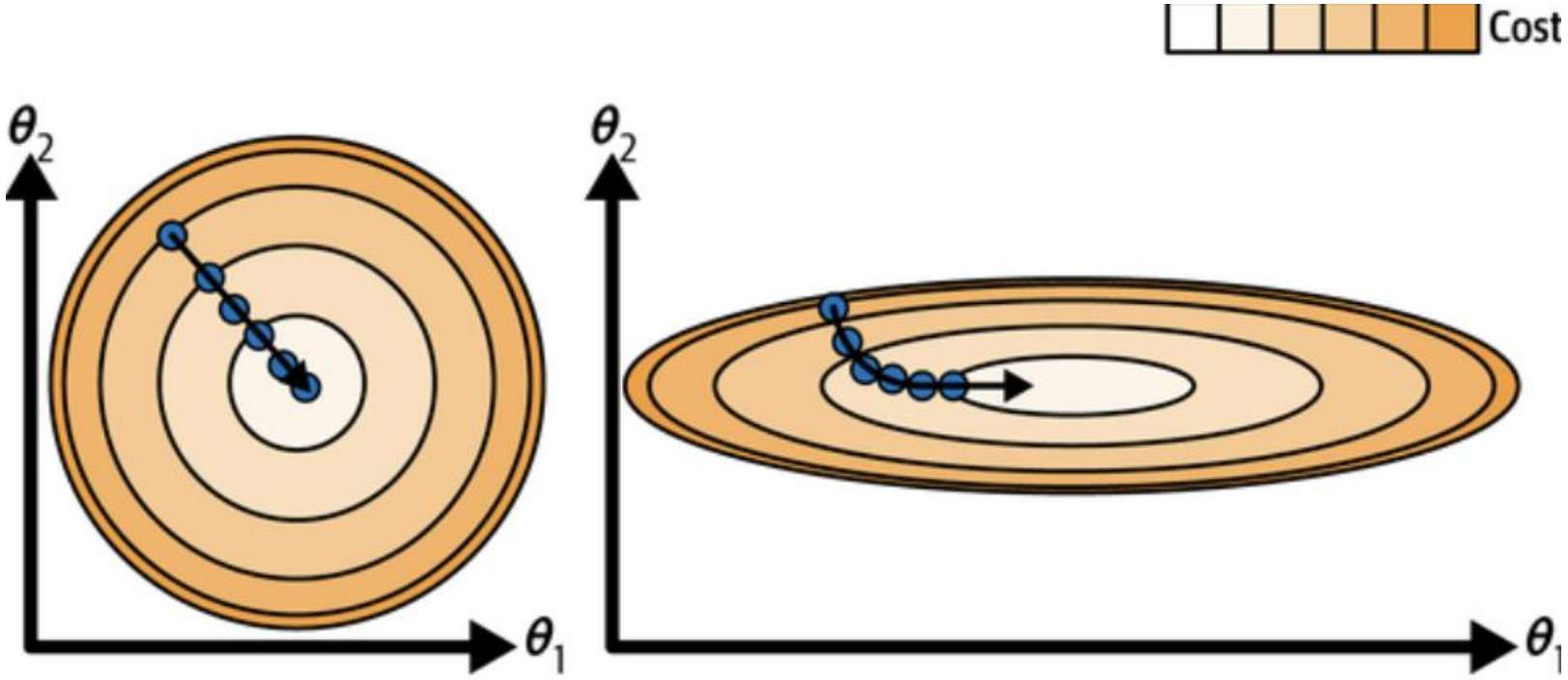
Gradient Descent

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



Feature scaling

- “When using gradient descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn’s StandardScaler class), or else it will take much longer to converge.”



Batch Gradient Descent

- To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θj. multiply the gradient vector by η to determine the size of the downhill step.
- "Batch" means "use all training data at every step of the calculation".

Equation 4-6. Gradient vector of the cost function

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

Equation 4-7. Gradient Descent step

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

Batch Gradient Descent (manual calculation)

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model
parameters

for epoch in range(n_epochs):
      gradients = 2 / m * X_b.T @ (X_b @ theta - y)
      theta = theta - eta * gradients
      print(epoch, gradients, theta)
```

Batch Gradient Descent

```
theta = np.random.randn(2, 1) # randomly initialized model parameters
```

```
[[0.49671415] [-0.1382643]]
```

```
print(epoch, gradients, theta)
```

```
0 [[-6.09276572] [-7.60024511]] [[1.10599072] [0.62176021]]

1 [[-3.44481702] [-4.57750006]] [[1.45047243] [1.07951022]]

2 [[-1.89495266] [-2.79922136]] [[1.63996769] [1.35943235]]

…

5 [[-0.15667432] [-0.76622837]] [[1.80080402] [1.72450491]]

6 [[ 0.01876687] [-0.54754378]] [[1.79892734] [1.77925929]]

…
```

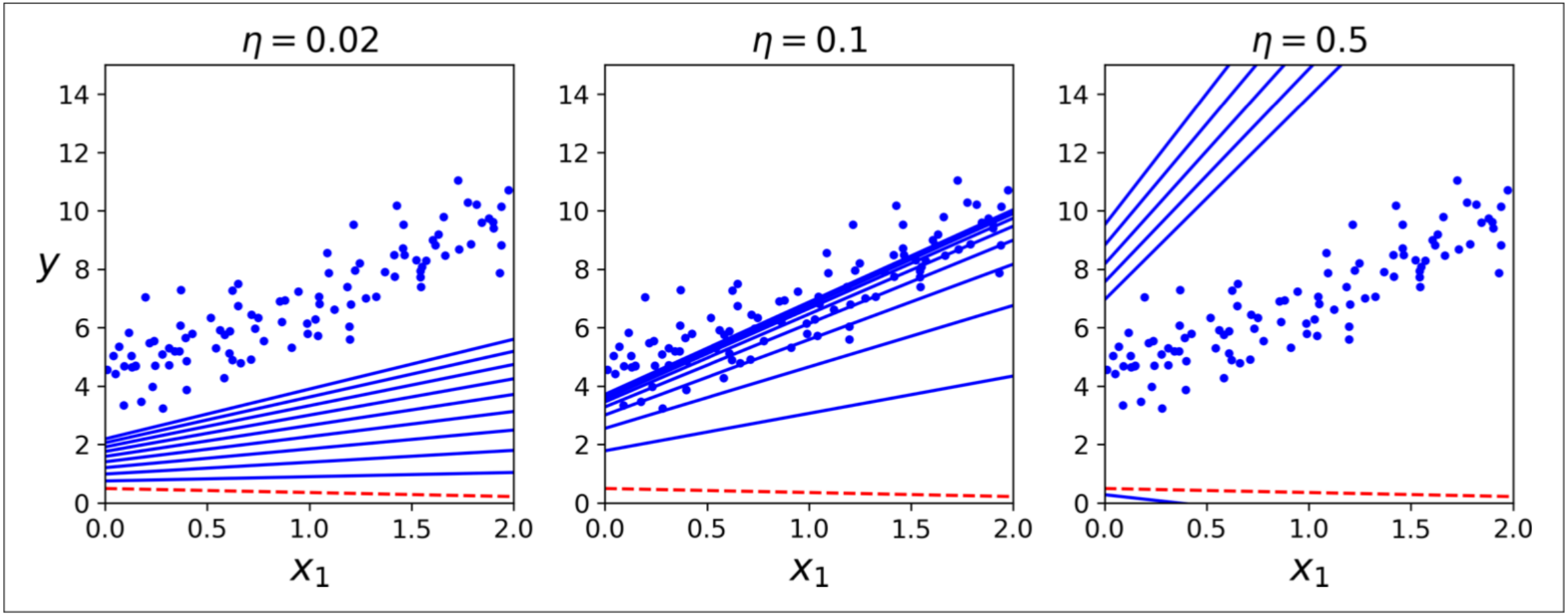


Figure 4-8. Gradient Descent with various learning rates

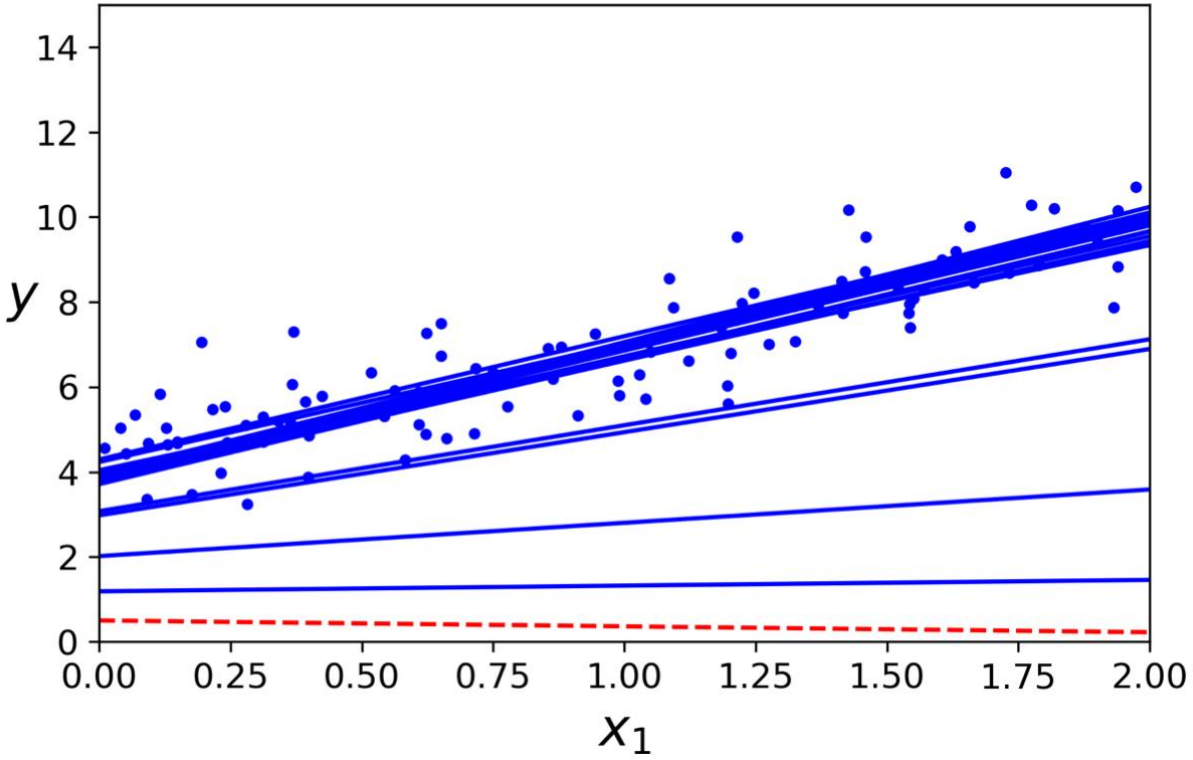
Weekly Practical 1

- Edit the “Batch Gradient Descent” source code with adaptive epoch configuration, i.e., continue the learning process until a good solution is found.

- Can you adjust the code to detect a learning rate that is set too high?

Stochastic Gradient Descent

- The main problem with Gradient Descent is the fact that it uses the **whole training set** to compute the gradients at every step, which makes it **very slow** when the training set is large.
- At the opposite extreme, **Stochastic Gradient Descent** just picks a **random instance** in the training set at every step and computes the gradients based only on that single instance. Obviously, this makes the algorithm much faster since it has very little data to manipulate at every iteration.



Stochastic Gradient Descent

- On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much **less regular** than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down.
- One solution to this dilemma is to **gradually reduce the learning rate**. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.

Stochastic Gradient Descent

```
for epoch in range(n_epochs):
        for iteration 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 @ (xi @ theta - yi)
            eta = learning_schedule(epoch * m + iteration)
            theta = theta - eta * gradients
```

Stochastic Gradient Descent

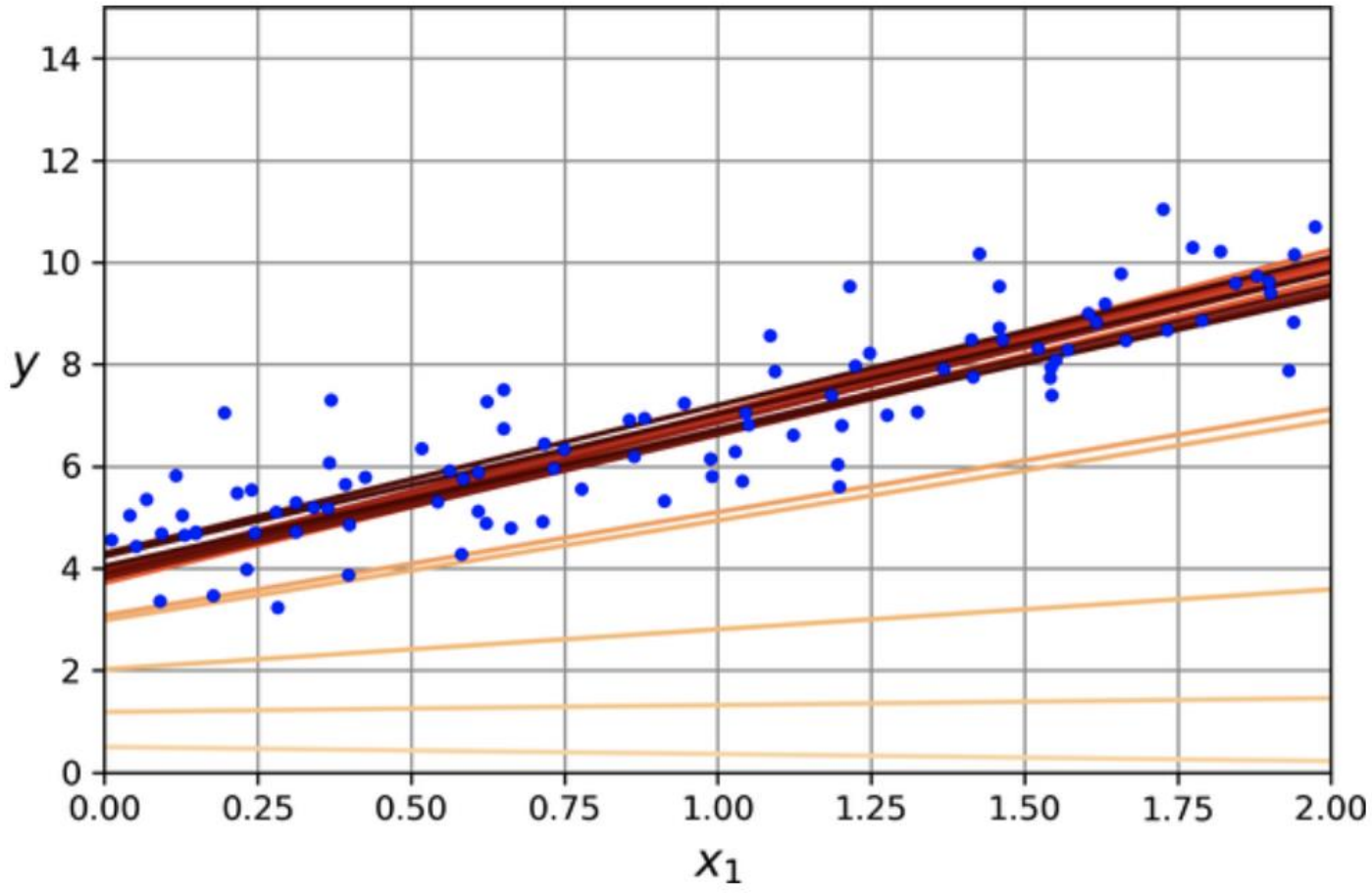


Figure 4-10. The first 20 steps of stochastic gradient descent

Stochastic Gradient Descent

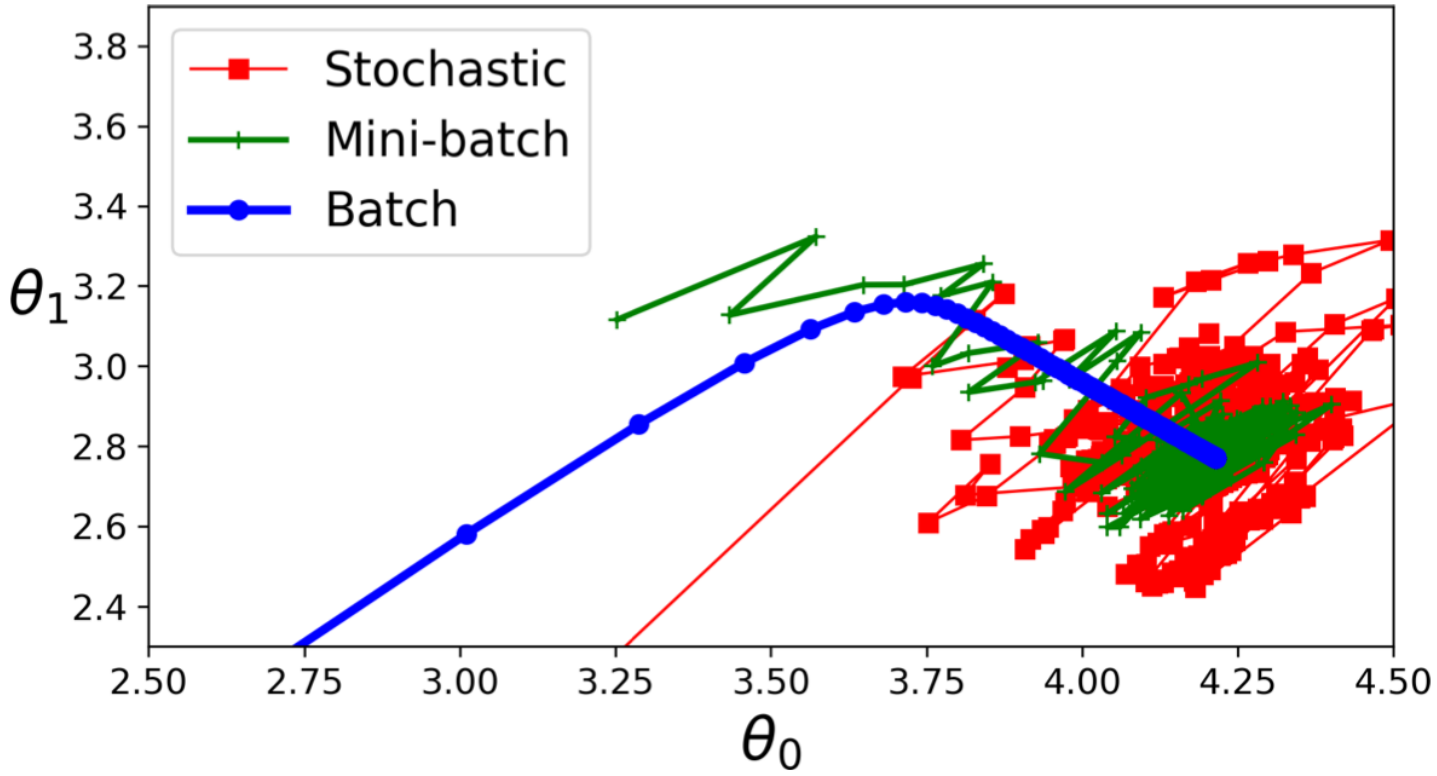
using SGDGegressor

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None,
eta0=0.01,n_iter_no_change=100, random_state=42)

sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets
```

- To perform Linear Regression using SGD with Scikit-Learn, you can use the *SGDRegressor* class, which defaults to optimising the squared error cost function.
- The above code runs for maximum 1,000 epochs (max_iter) or until the loss drops by less than 10–5 (tol) during 100 epochs (n_iter_no_change). It starts with a learning rate of 0.01 (eta0), using the default learning schedule (different from the one we used). Lastly, it does not use any regularisation (penalty=None).

Mini-batch Gradient Descent

- 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**.
- The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.



Mini-batch Gradient Descent

Or using:
random.sample() and
sgd_reg.partial_fit()

```
for epoch in range(n_epochs):
    shuffled_indices = np.random.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for iteration in range(0, n_batches_per_epoch):
        idx = iteration * minibatch_size
        xi = X_b_shuffled[idx : idx + minibatch_size]
        yi = y_shuffled[idx : idx + minibatch_size]
        gradients = 2 / minibatch_size * xi.T @ (xi @ theta - yi)
        eta = learning_schedule(iteration)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
```

Mini-batch SGD using SGDRegressor and partial_fit (one step fitting)

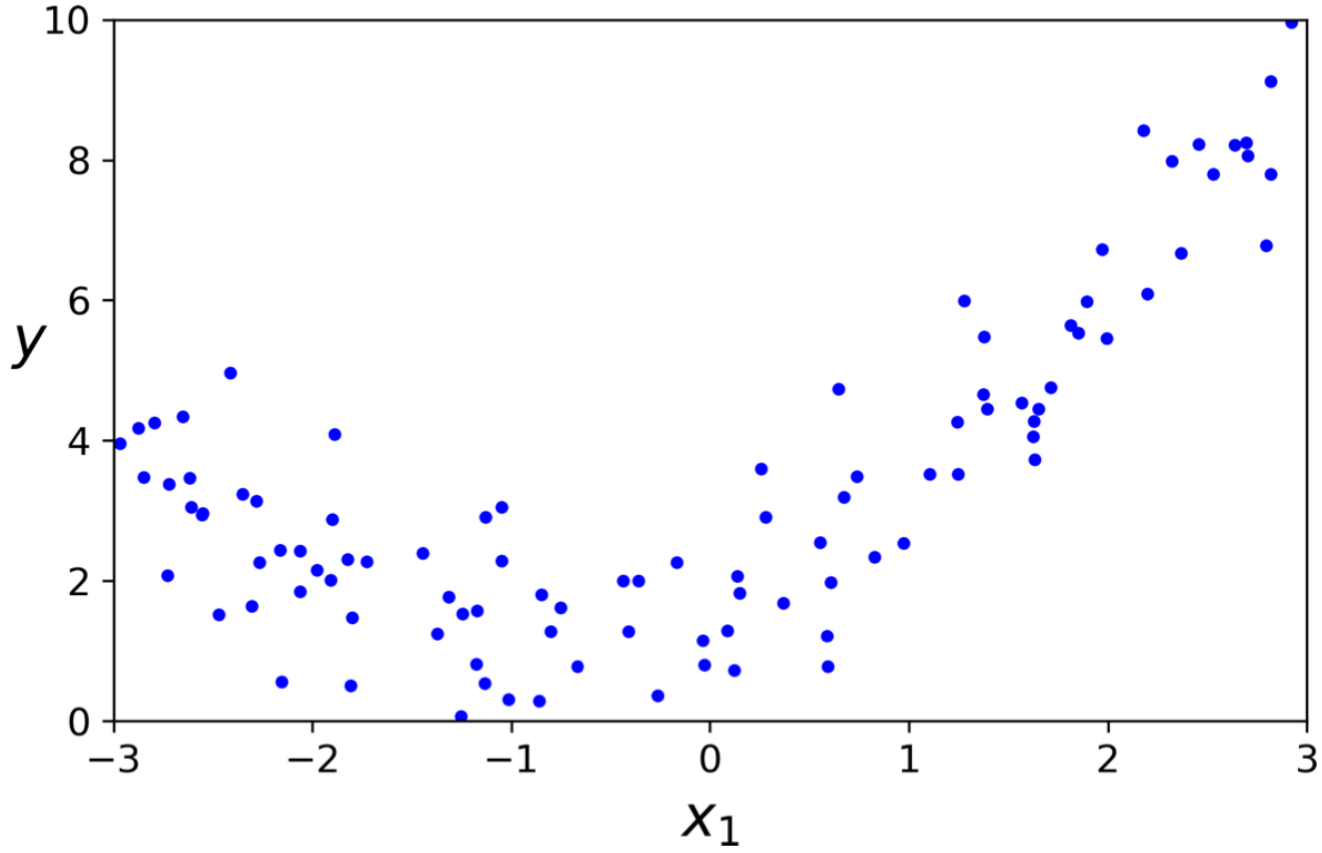
```
import random

minisgd = SGDRegressor(eta0=0.01)

batch_size = 30

for i in range(1000):

    idx = random.sample(range(X.shape[0]),batch_size)
    minisgd.partial_fit(X[idx], y.ravel()[idx])

minisgd.intercept_, minisgd.coef_
```

```
(array([4.19209311]), array([2.77188334]))
```

Polynomial Regression

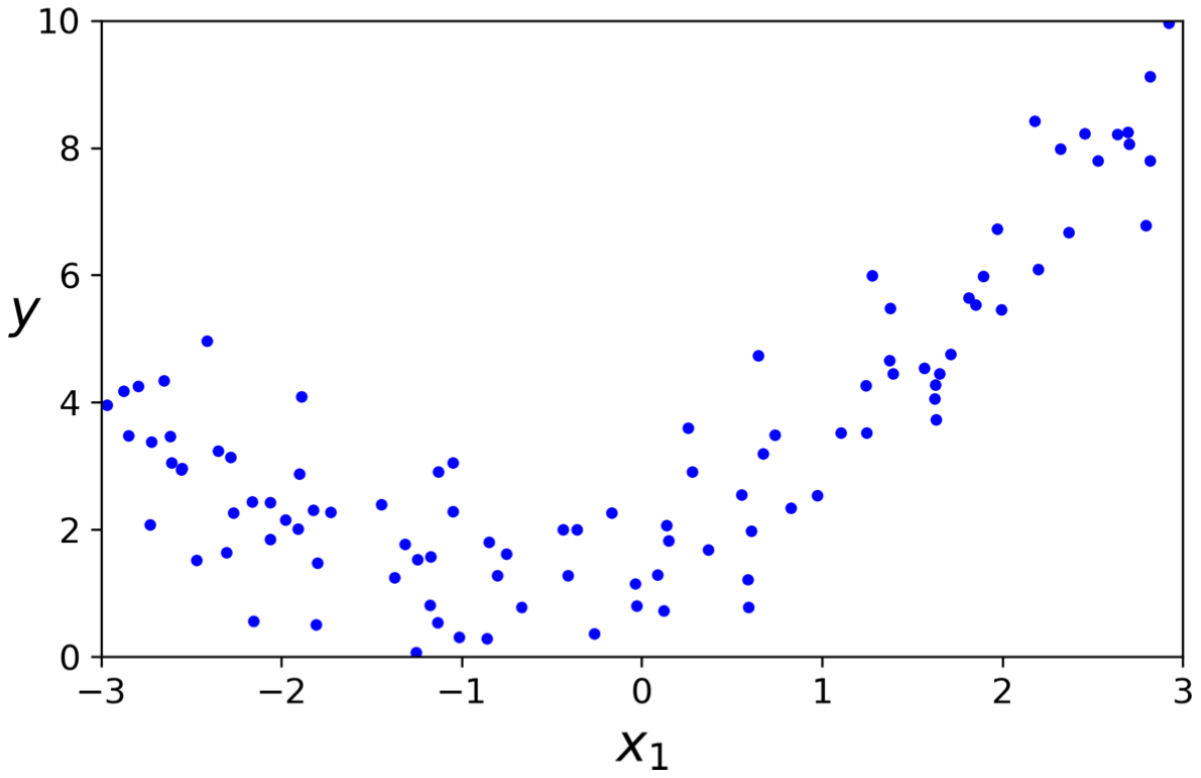
Polynomial Regression

- Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's *PolynomialFeatures* class to transform our training data, adding the square (2nd-degree polynomial) of each feature in the training set as new features.



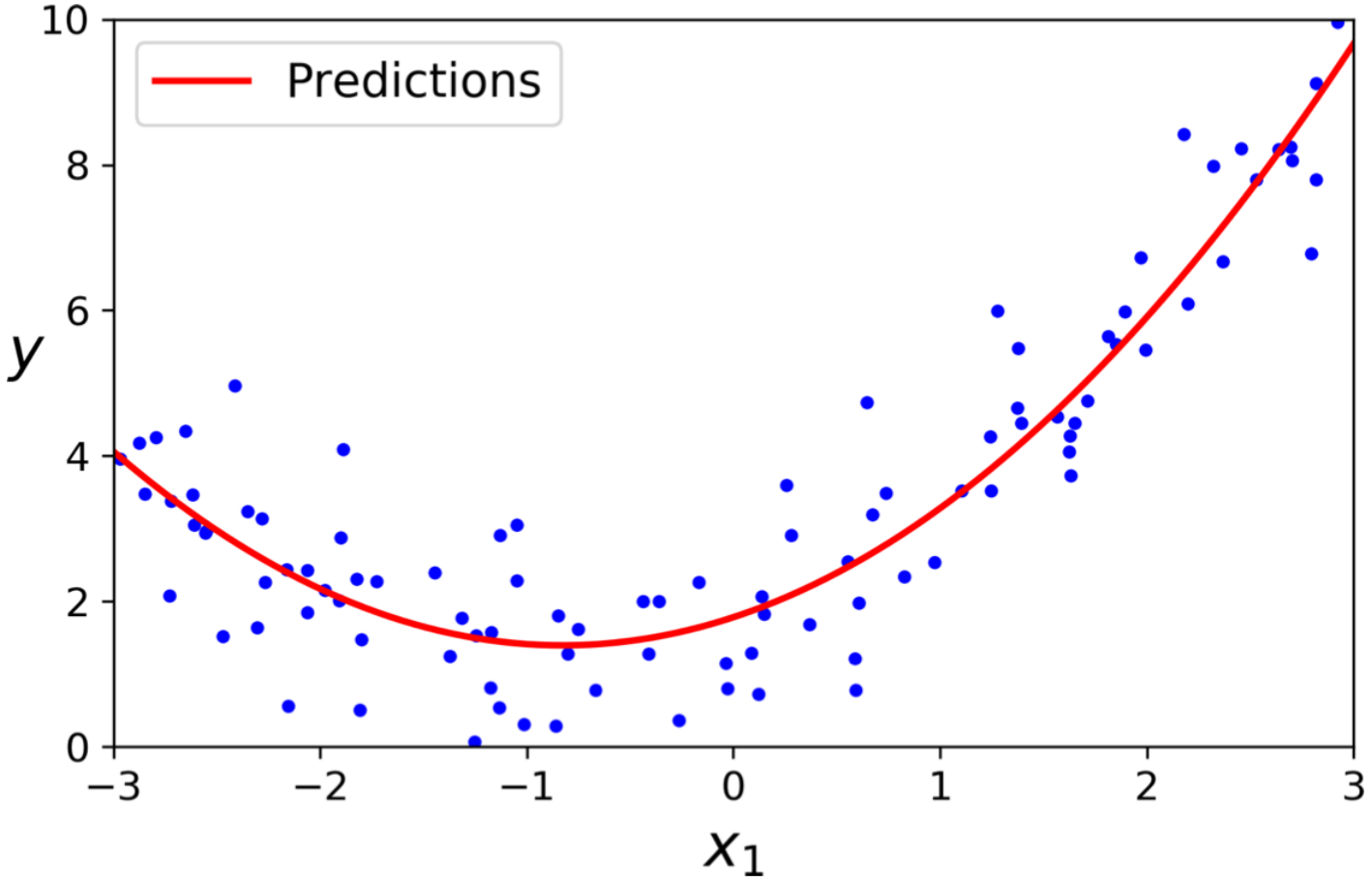
Polynomial Regression

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```



Polynomial Regression

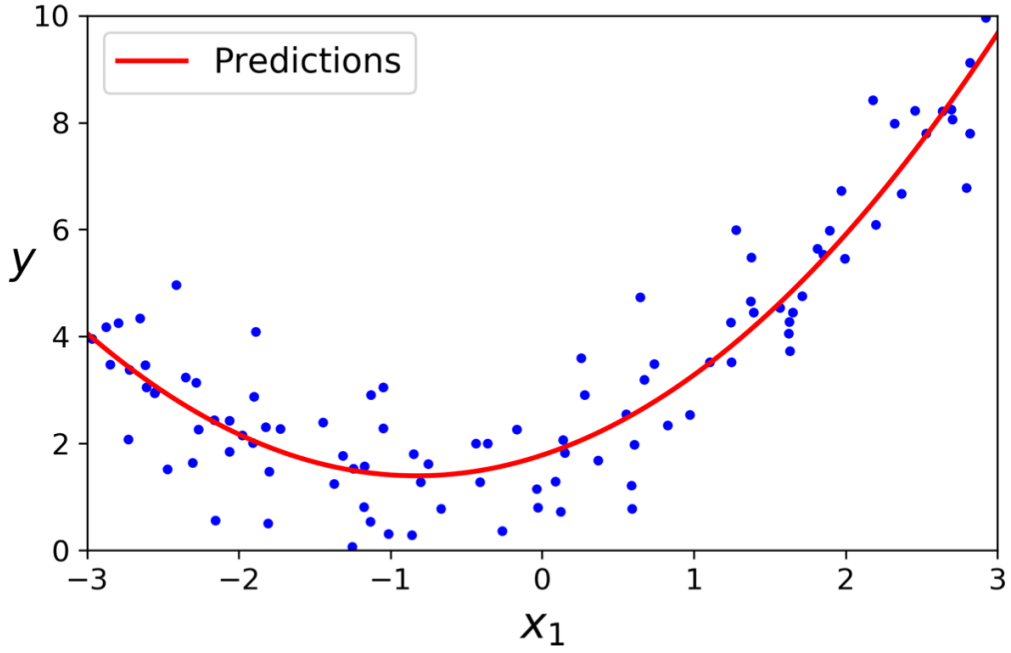
```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

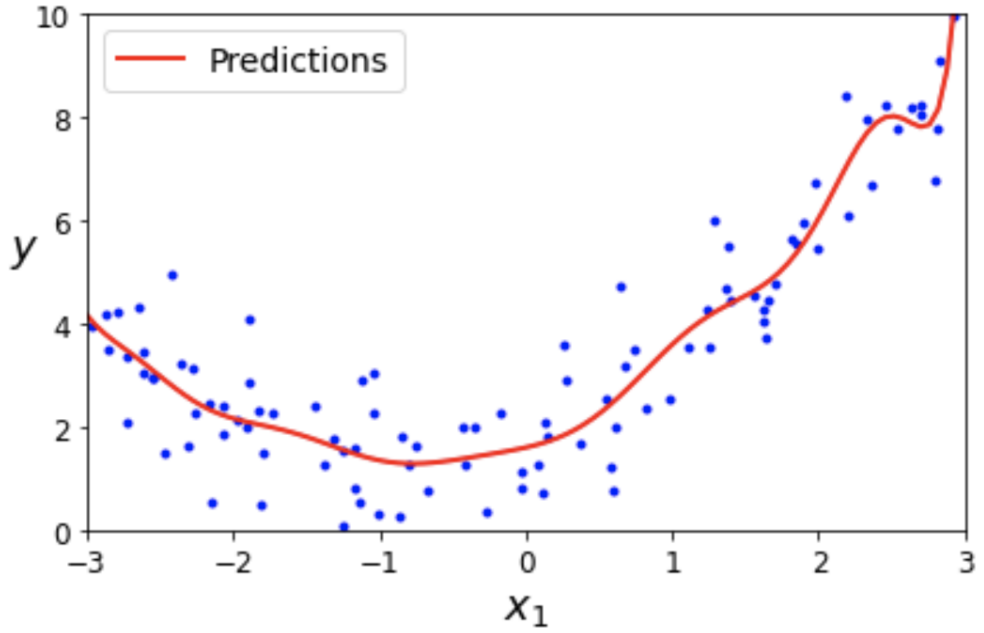


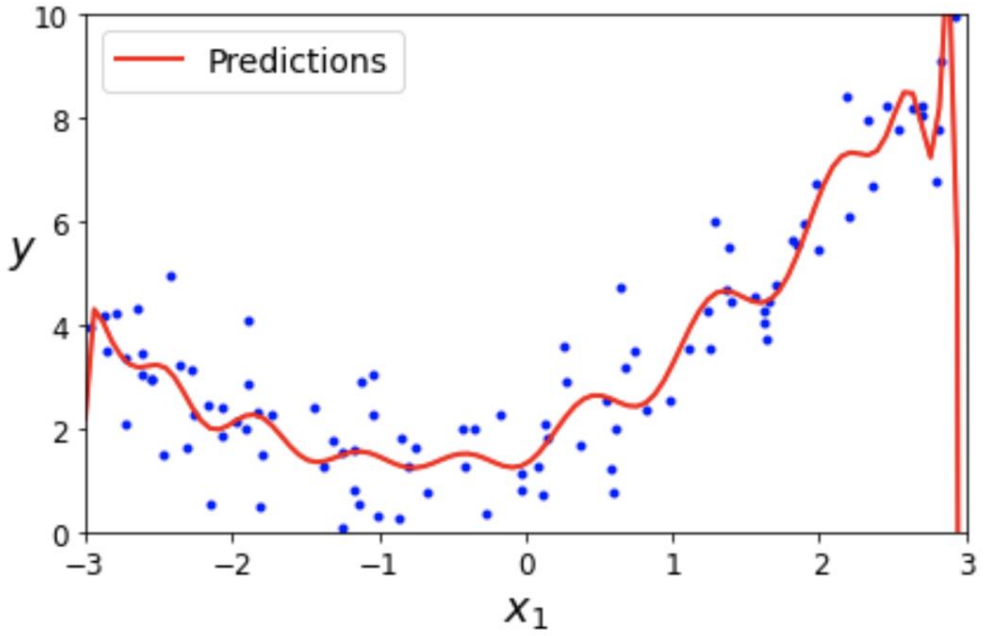
$$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$$

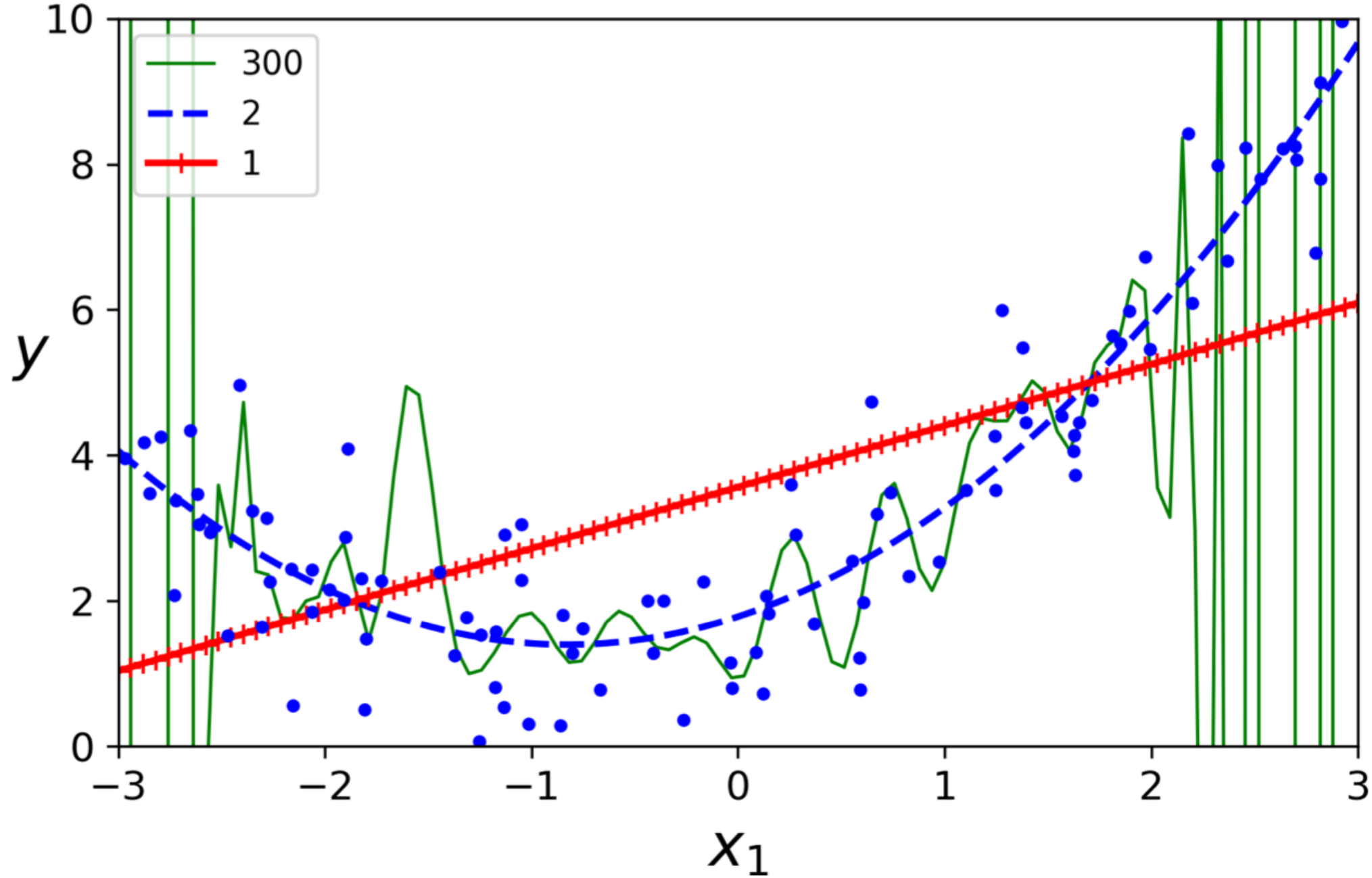
Polynomial Regression

- If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression.








300
2
1
y
x_1
10
8
6
4
2
0
−3
−2
−1
0
1
2
3

Weekly Practical

- Download dataset file "`Real estate dataset.csv`" from Nile
- Model the data to find the price based on the input features.

Further reading - Regularization

- a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data.
- A regularization term is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible

Equation 4-8. Ridge Regression cost function

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

Equation 4-10. Lasso Regression cost function

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