Machine Learning

Lecture 3: Linear Regression

Instructor: Dr. Farhad Pourkamali Anaraki

Introduction

- Most of the topics discussed in this lecture note will be essential in understanding, building, and training neural networks

1. We focus on training the linear regression model
    - Closed-form equation that directly computes the model parameters that best fit to the training set
    - Iterative optimization technique called Gradient Descent (GD) that gradually updates the model parameters
2. We then look at the Polynomial Regression to learn from non-linear datasets
3. Finally, we look at several regularization techniques that can reduce the risk of overfitting

Case study: univariate linear regression

- A list of training instances $\mathscr{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$
- Model selection

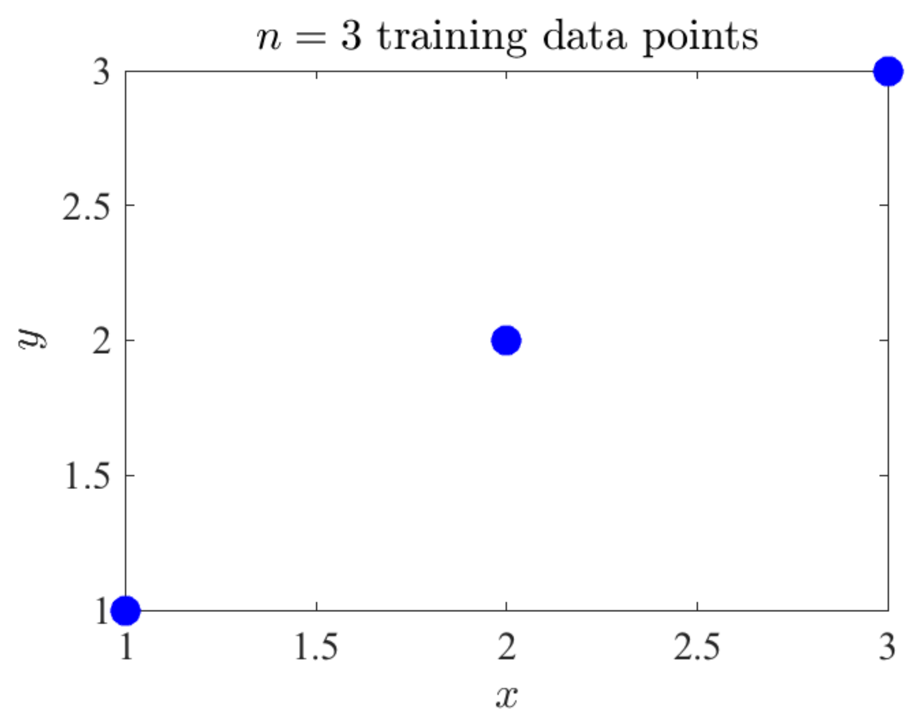
$$h(x) = \theta_0 + \theta_1 x$$

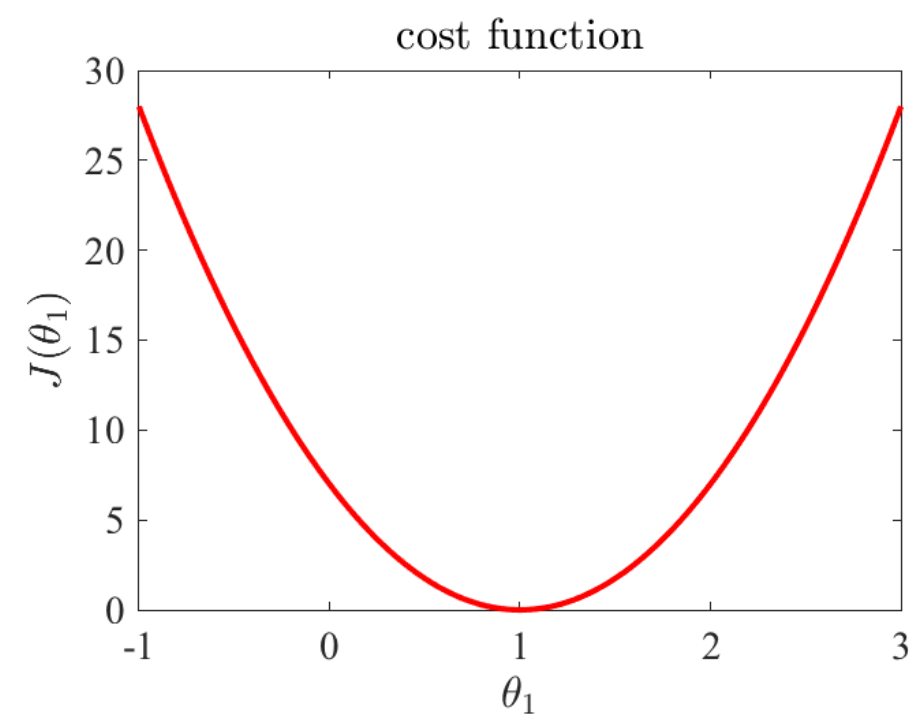
- Parameters: θ_0, θ_1
- Criterion: Choose these two parameters such that $h(x^{(i)})$ is close to $y^{(i)}$ for our training dataset
- Cost function or objective function:

$$J(\theta_0, \theta_1) = \frac{1}{2} \sum_{i=1}^{n} \left(h(x^{(i)}) - y^{(i)} \right)^2$$

How to choose model parameters?

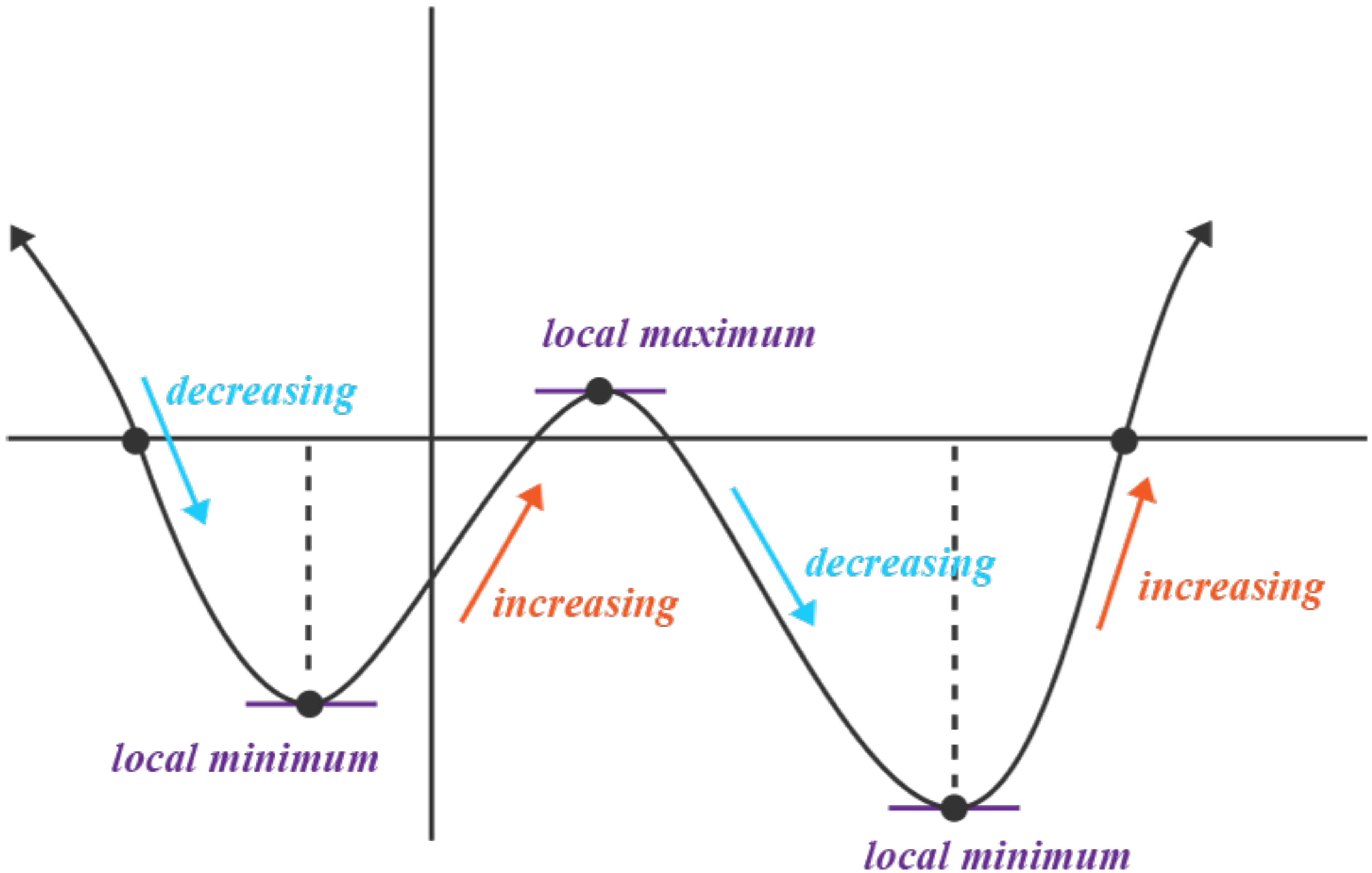
- Let's assume $\theta_0 = 0$, which means that $h(x) = \theta_1 x$
- Goal: Find the value of θ_1 that leads to the minimum value of $J(\theta_1)$





- Therefore, mathematical optimization is a central part of machine learning

First derivative test



Gradient

- Gradient captures all the partial derivatives of a multi-variable function

- Example:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad f(\mathbf{x}) = \mathbf{x}^T\mathbf{x} = x_1^2 + x_2^2$$

$$\frac{\partial f}{\partial x_1} = 2x_1$$

$$\frac{\partial f}{\partial x_2} = 2x_2$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix} = 2\mathbf{x}$$

Back to our previous example

- Recall the univariate linear regression problem

$$J(\theta_0, \theta_1) = \frac{1}{2}\sum_{i=1}^{n}\left(h(x^{(i)}) - y^{(i)}\right)^2, \, h(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$$

- Find partial derivatives

Back to our previous example

- Recall the univariate linear regression problem

$$J(\theta_0, \theta_1) = \frac{1}{2}\sum_{i=1}^{n}\left(h(x^{(i)}) - y^{(i)}\right)^2,\ h(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$$

- Find partial derivatives

$$\frac{\partial J}{\partial \theta_0} = \sum_{i=1}^{n}\left(h(x^{(i)}) - y^{(i)}\right)$$

$$\frac{\partial J}{\partial \theta_1} = \sum_{i=1}^{n}\left(h(x^{(i)}) - y^{(i)}\right)x^{(i)}$$

Multivariate Linear Regression

- Assumption: parametric model where $h(\mathbf{x})$ is a linear function of $\mathbf{x}$

$$h(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d = \theta_0 + \sum_{i=1}^{d} \theta_i x_i$$

- Tick: let $x_0 = 1$

$$h(\mathbf{x}) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d = \sum_{i=0}^{d} \theta_i x_i = \langle \mathbf{x}, \boldsymbol{\theta} \rangle$$

Compact representation

- Given the training set $(\mathbf{x}^{(i)}, y^{(i)}), i = 1,\ldots,n,$ use dot products

$$\begin{bmatrix} h(\mathbf{x}^{(1)}) \\ \vdots \\ h(\mathbf{x}^{(n)}) \end{bmatrix} = \begin{bmatrix} \langle \mathbf{x}^{(1)}, \boldsymbol{\theta} \rangle \\ \vdots \\ \langle \mathbf{x}^{(n)}, \boldsymbol{\theta} \rangle \end{bmatrix} = \begin{bmatrix} \mathbf{x}^{(1)} \\ \cdots \\ \mathbf{x}^{(n)} \end{bmatrix} \boldsymbol{\theta} = \mathbf{X}\boldsymbol{\theta}$$

- Thus, we can rewrite the cost function

$$J(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2$$

A closed form solution

- The partial derivatives are all zero at the minimum value

$$\frac{1}{2}(2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - 2\mathbf{X}^T\mathbf{y}) = \mathbf{0}$$

$$\downarrow$$

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

$$\downarrow$$

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

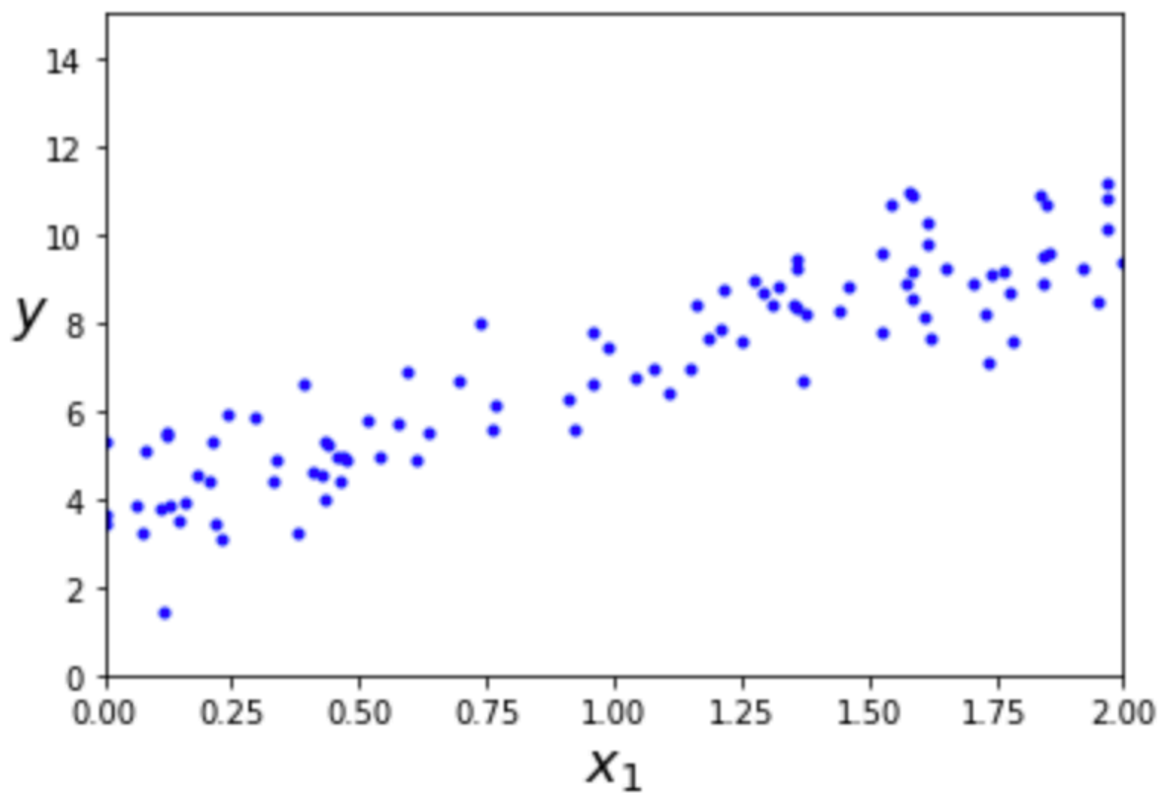
- This expression is known as the normal equation solution of the least squares problem

Linear regression using the Normal equation

- Let's generate some linear-looking data

```
import numpy as np
import matplotlib.pyplot as plt

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

```
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()
```



Linear regression using the Normal equation

- Training

```
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)
```

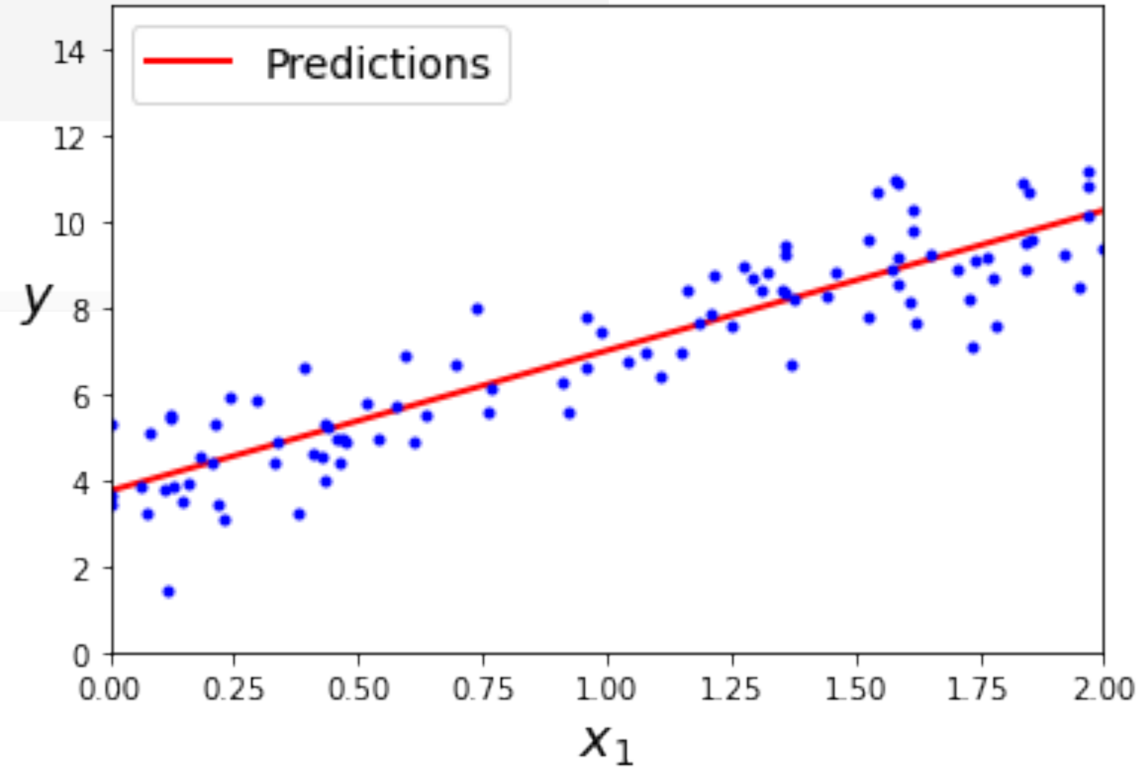
```
theta_best
```

```
array([[3.73953954],
       [3.25407465]])
```

- Let's make predictions

```
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([[ 3.73953954],
       [10.24768885]])
```



Linear regression using Scikit-Learn

```
from sklearn.linear_model import LinearRegression

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

```
(array([3.73953954]), array([[3.25407465]]))
```

- How does LinearRegression from sklearn work?

```
theta_best_svd, _, _, _ = np.linalg.lstsq(X_b, y, rcond=1e-6)
theta_best_svd
```

```
array([[3.73953954],
       [3.25407465]])
```

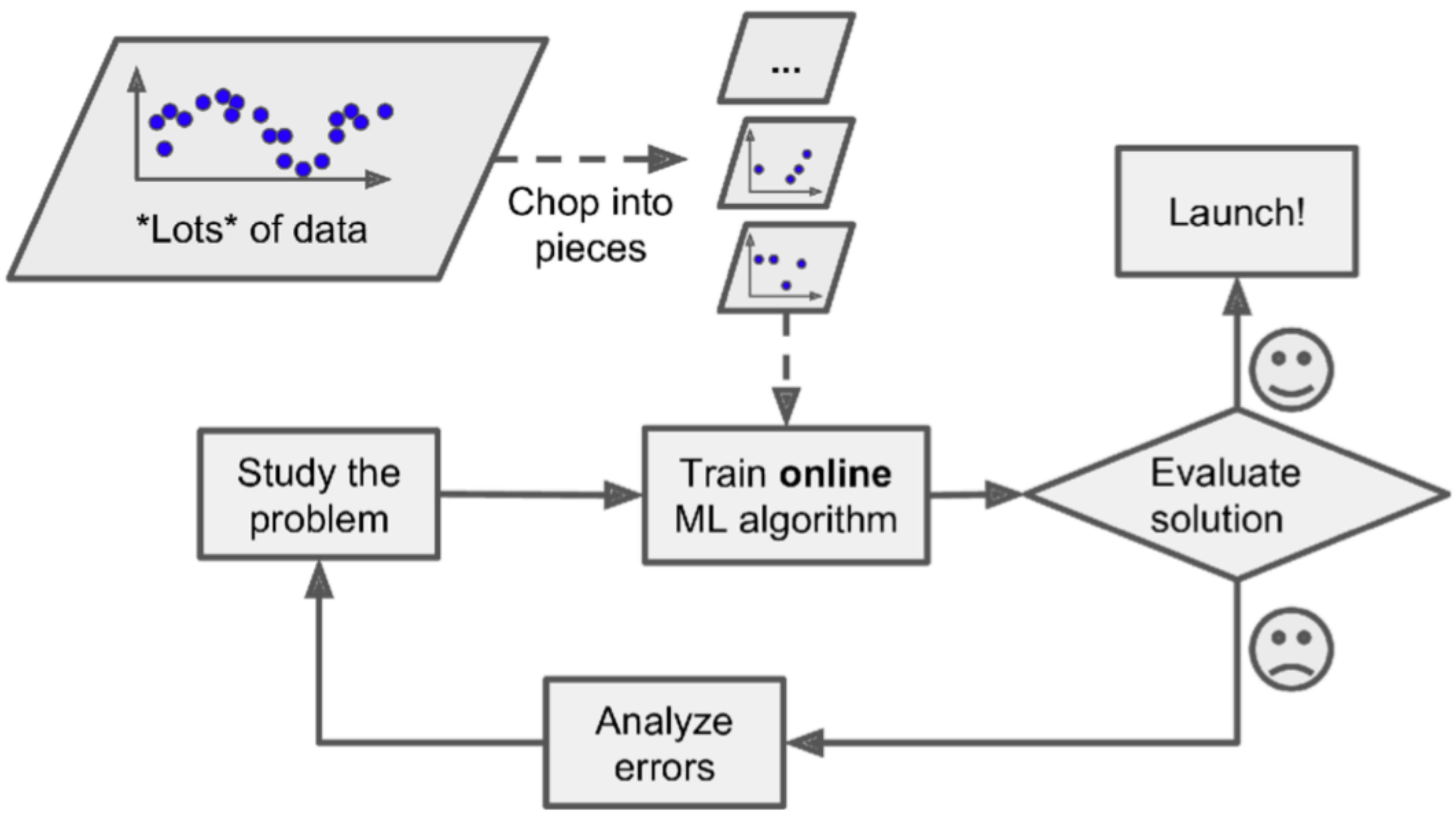
numpy.linalg.lstsq

numpy.linalg.**lstsq**(*a*, *b*, *rcond='warn'*)

Return the least-squares solution to a linear matrix equation.

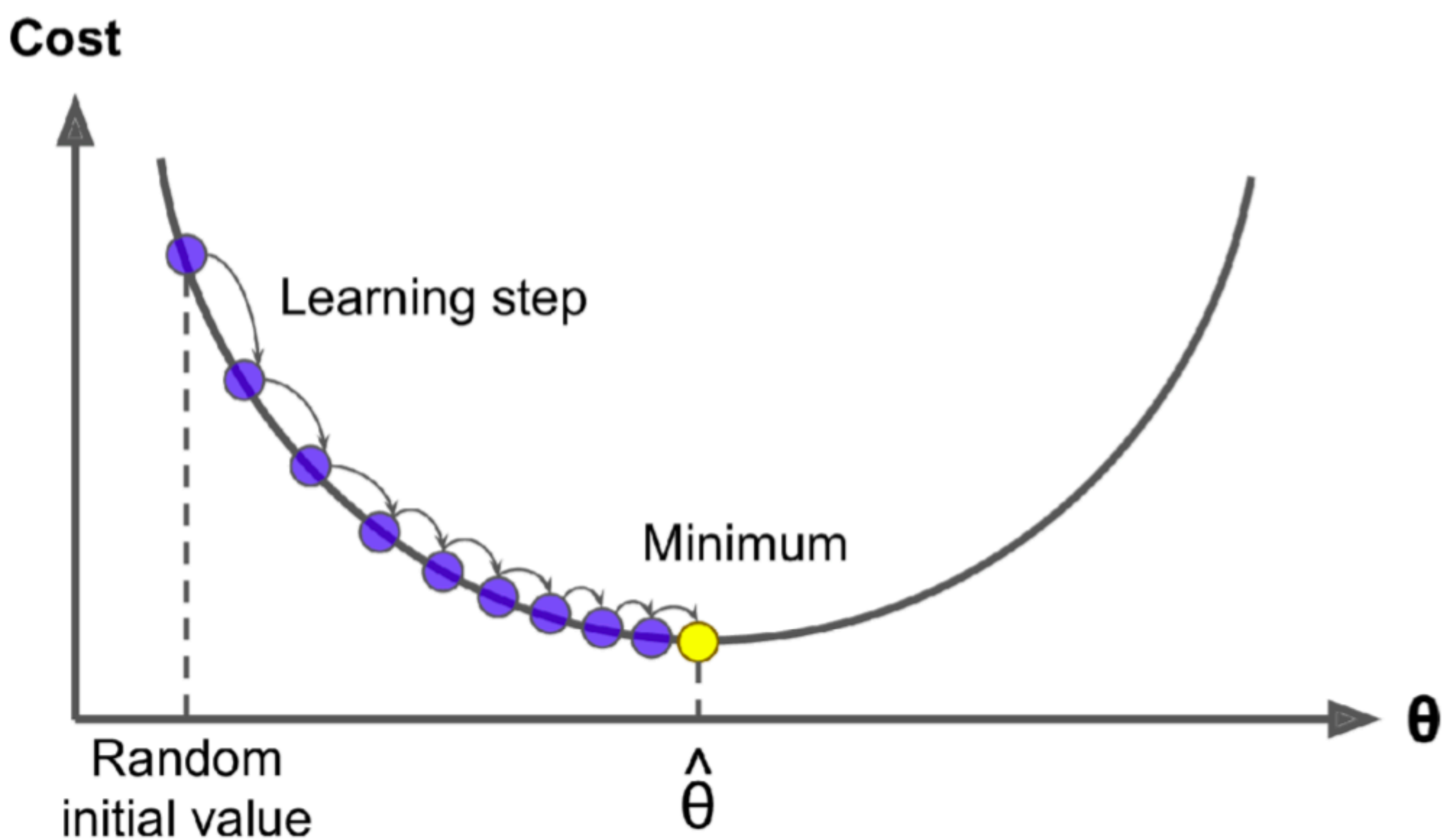
Computational complexity

- When we have d features, we should compute an inverse of $d \times d$ matrix
  - Double the number of features, you multiply the computation time by $2^3 = 8$
- Too many training instances n to fit in memory
- Thus, we look at a different way to train a linear regression model



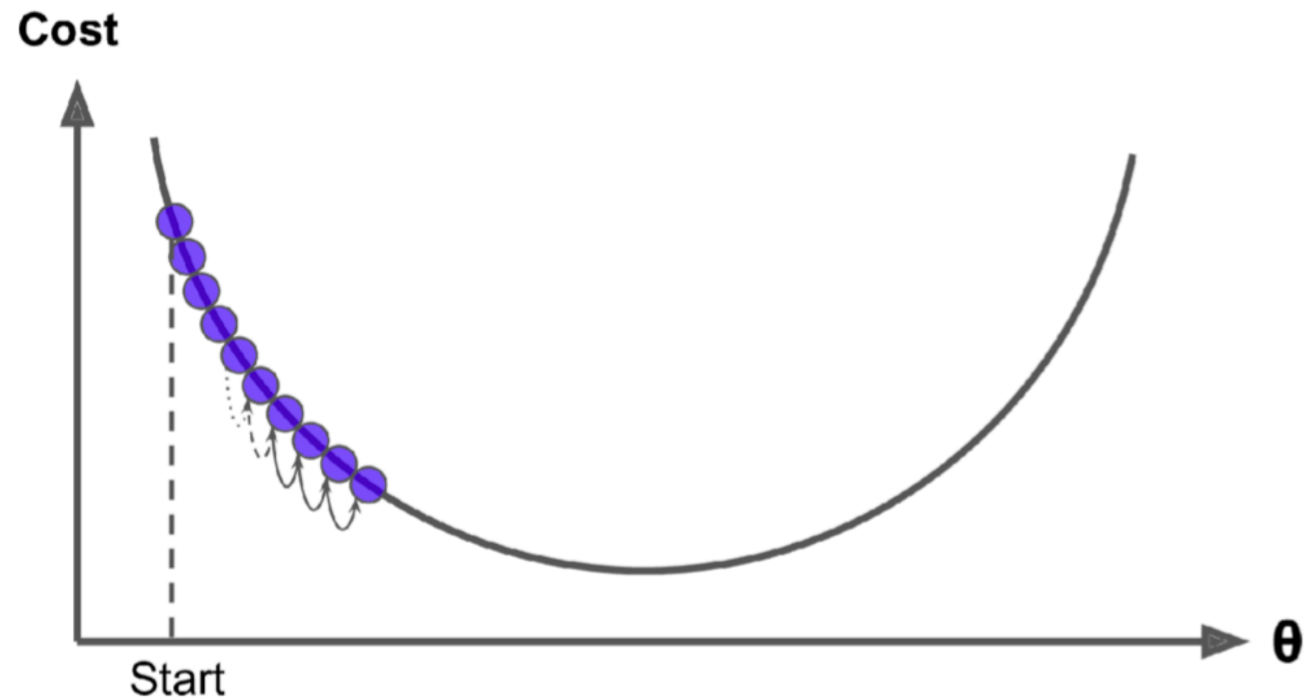
Gradient descent

- Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems
- Start by filling $\boldsymbol{\theta}$ with random values
- Take steps to decrease the cost function until convergence

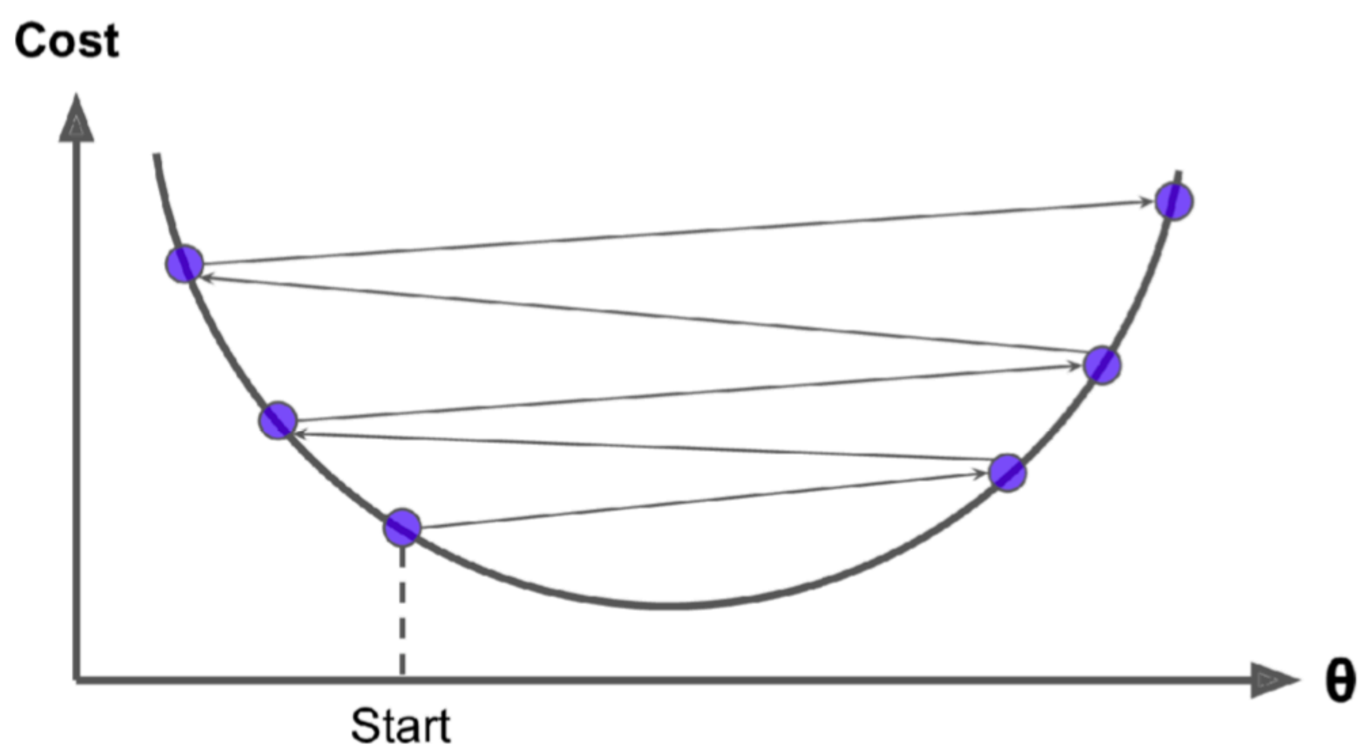


Hyperparameter

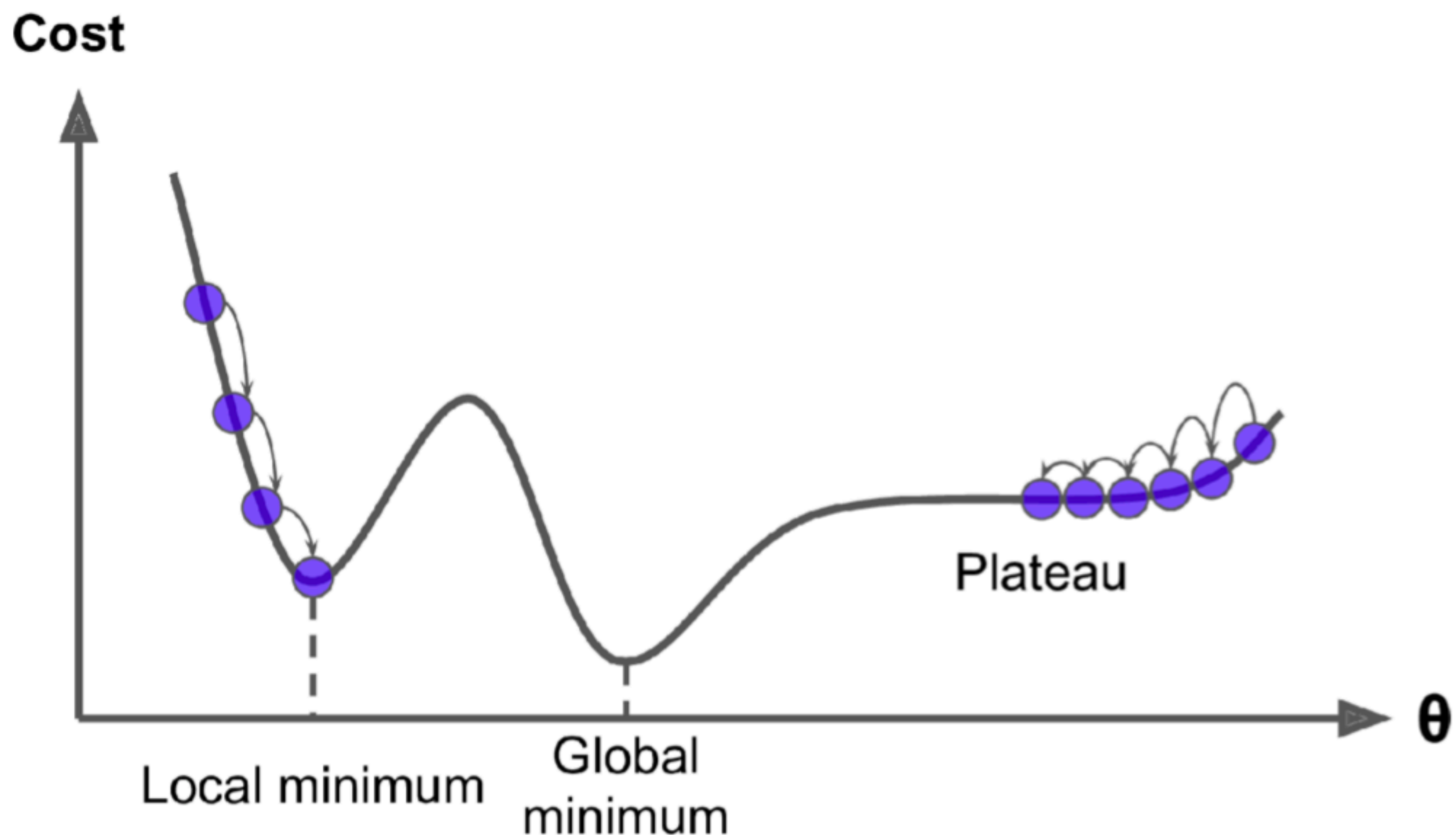
- If the learning rate is too small: many iterations and take a long time



- If the learning rate is too large: fail to find a good solution



Challenges of using gradient descent



Gradient descent implementation

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla J(\boldsymbol{\theta})$$

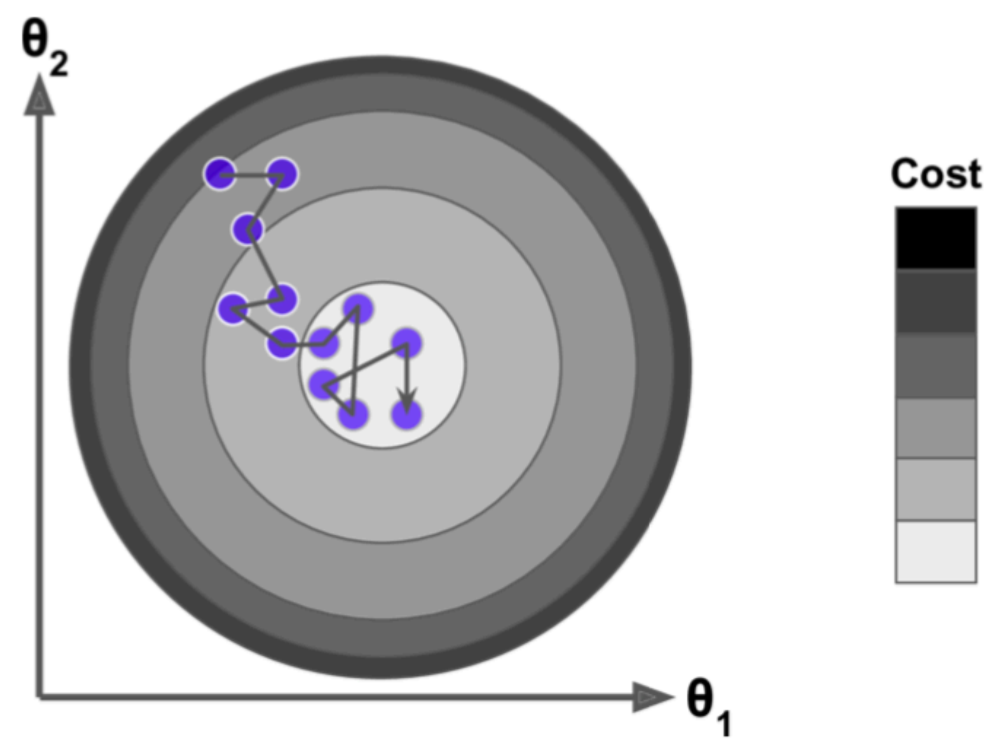
```
eta = 0.001  # learning rate
n_iterations = 1000

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

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

```
[[3.73953954]
 [3.25407466]]
```

This is known as Batch Gradient Descent

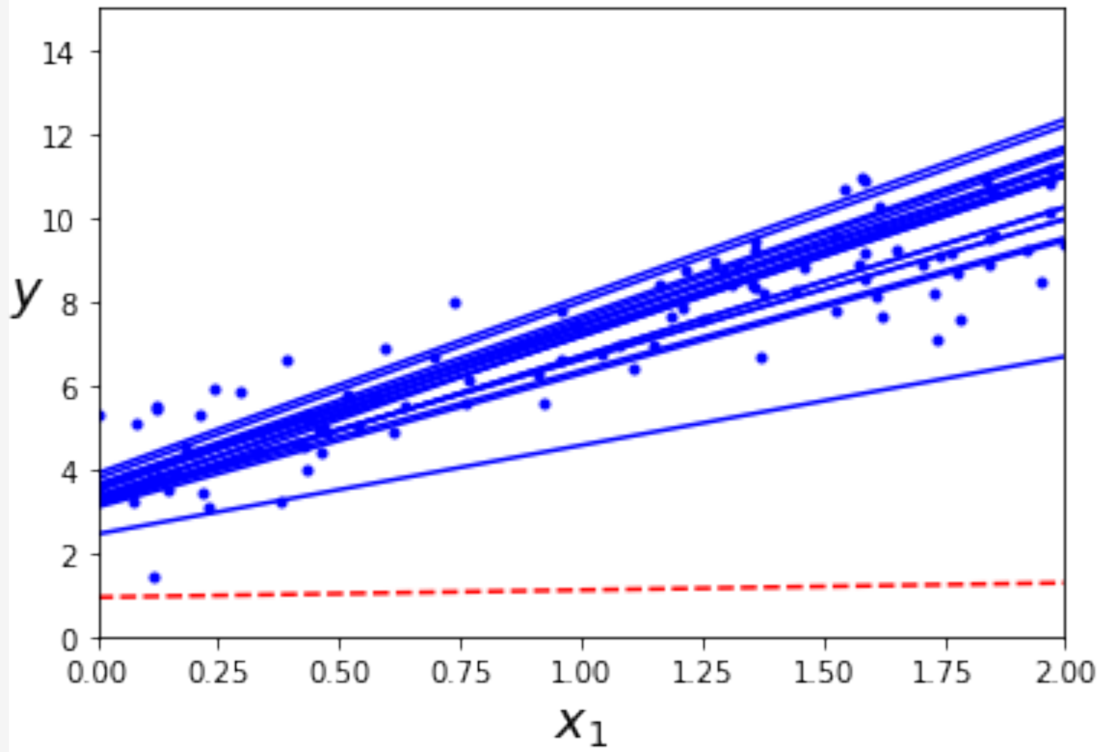
Stochastic gradient descent (SGD)

- Stochastic gradient descent picks a random instance in the training set at every step and computes the gradient based on that single instance.
    - Each training step is much faster but also much more stochastic (wiggly)
    - Randomness can be good to escape from local minima



Implementation of stochastic gradient descent

```
theta_path_sgd = []
n = len(X_b) # number of samples

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(n):
        if epoch == 0 and i < 20:
            y_predict = X_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        random_index = np.random.randint(n)
        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 * n + i)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)

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()
```




```
theta
```

```
array([[3.70853532],
       [3.2241968 ]])
```

Linear regression using SGD

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1, random_state=42)
sgd_reg.fit(X, y.ravel())
```

```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=1000,
             n_iter_no_change=5, penalty=None, power_t=0.25, random_state=42,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

```
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([3.75752755]), array([3.26908865]))
```

Mini-batch gradient descent

- Computes the gradients on small random sets of instances called mini-batches

```
n_iterations = 50
minibatch_size = 20
theta_path_mgd = []
theta = np.random.randn(2,1)  # random initialization

t0, t1 = 5, 50
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(n)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, n, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
```

```
list(range(0, n, minibatch_size))
```

```
[0, 20, 40, 60, 80]
```

Comparison of linear regression implementations

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

Textbook notation: m (number of samples) and n (number of features)

Polynomial regression

- A polynomial of degree 1 gives us the linear regression model

$$h(x) = \theta_0 + \theta_1 x$$

- Let us introduce x^2 as another feature

$$h(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$

- We can add more powers of x as new features

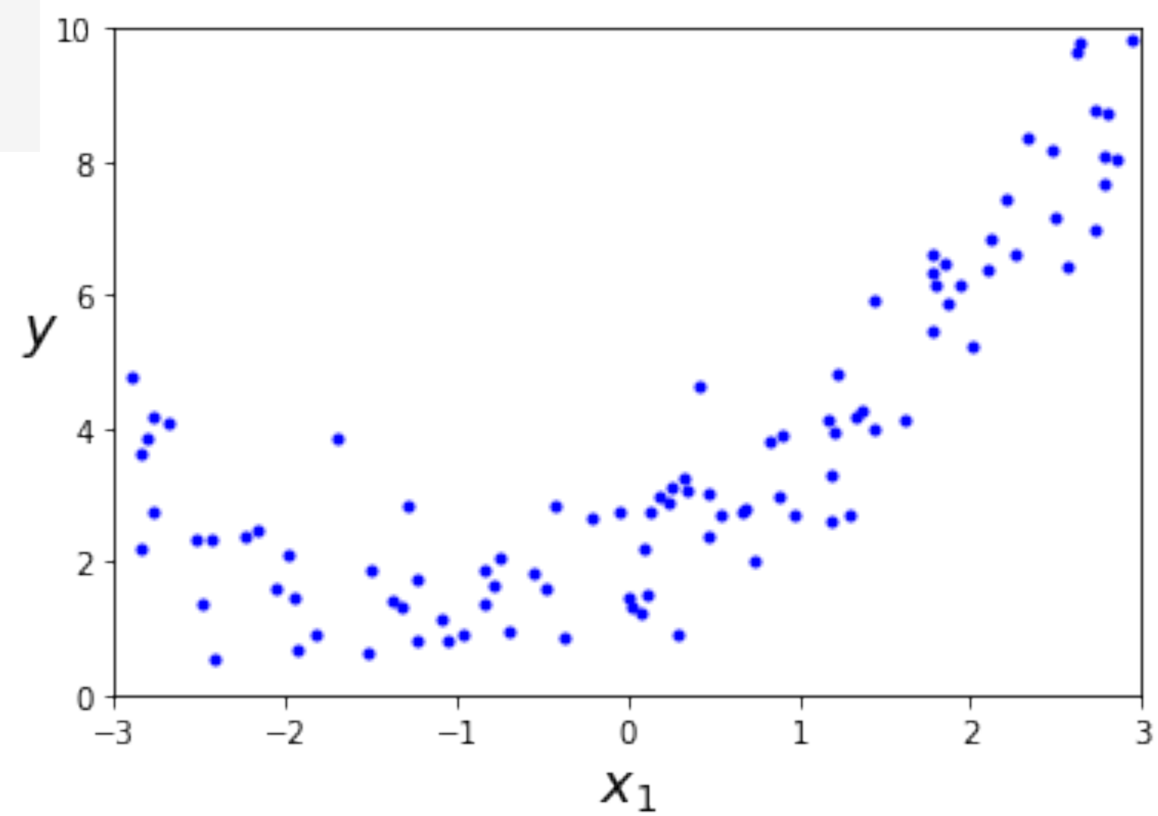
$$h(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

- The output is a linear function of the parameters

Polynomial regression

```
import numpy as np
import numpy.random as rnd

np.random.seed(42)
```

```
n = 100
X = 6 * np.random.rand(n, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(n, 1)
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.show()
```



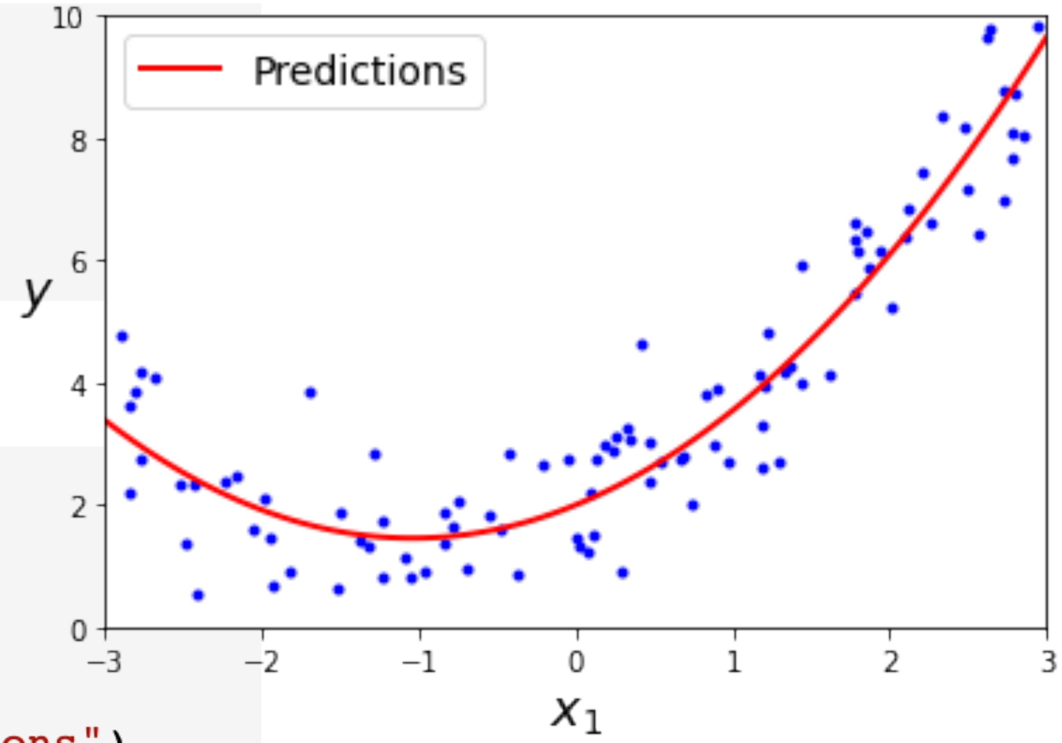
Polynomial regression

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

```
[1.76886782] [1.76886782 3.12889337]
```

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

```
(array([1.99958228]), array([[1.04630034, 0.5015459 ]]))
```

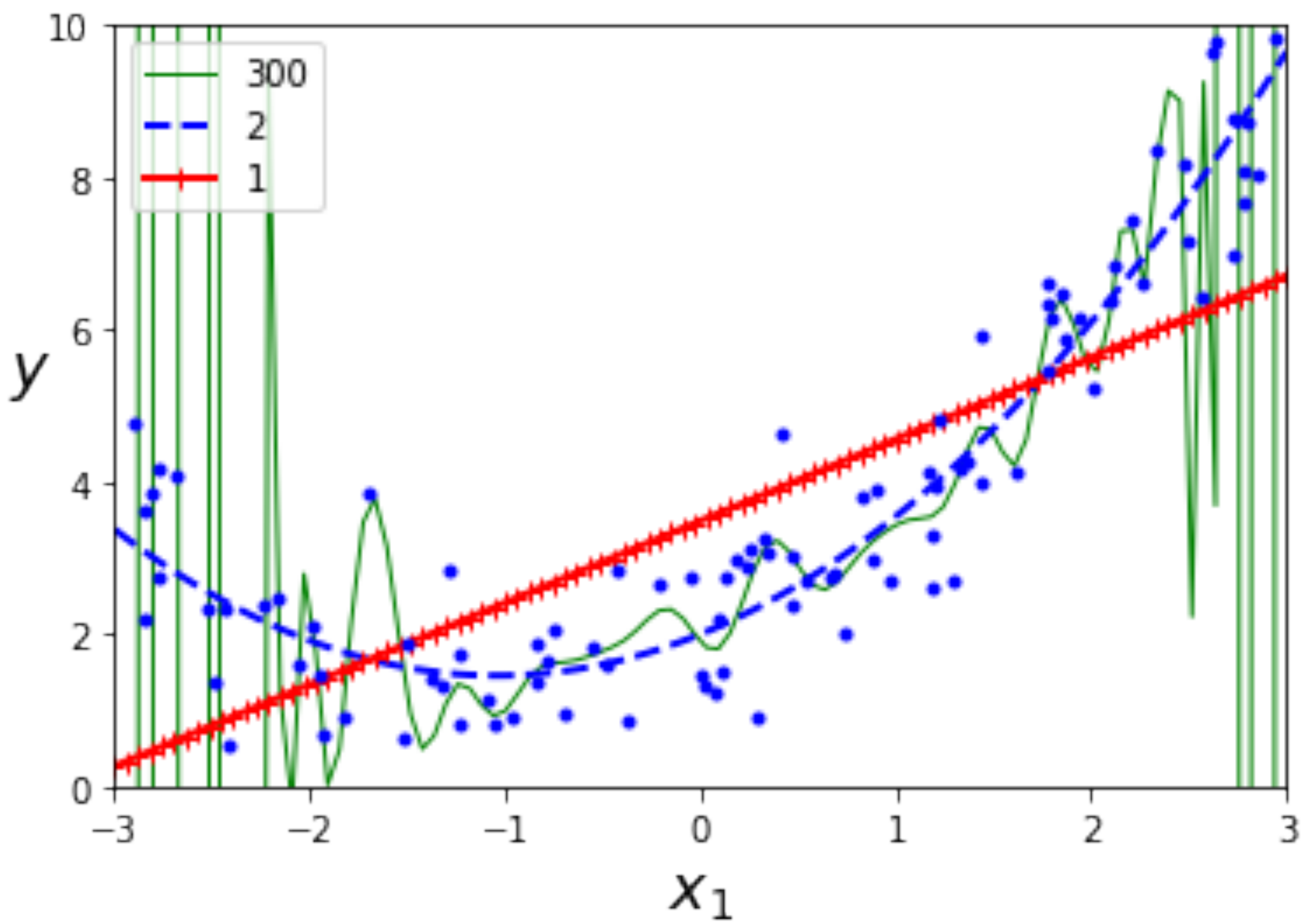



```
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
plt.show()
```

Selecting higher-degree polynomials

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.show()
```

Selecting higher-degree polynomials



- Observations:
  - 300-degree polynomial model wiggles around to get as close as possible to the training instances
  - Overfitting: performs well on training data but generalizes poorly

Learning curve

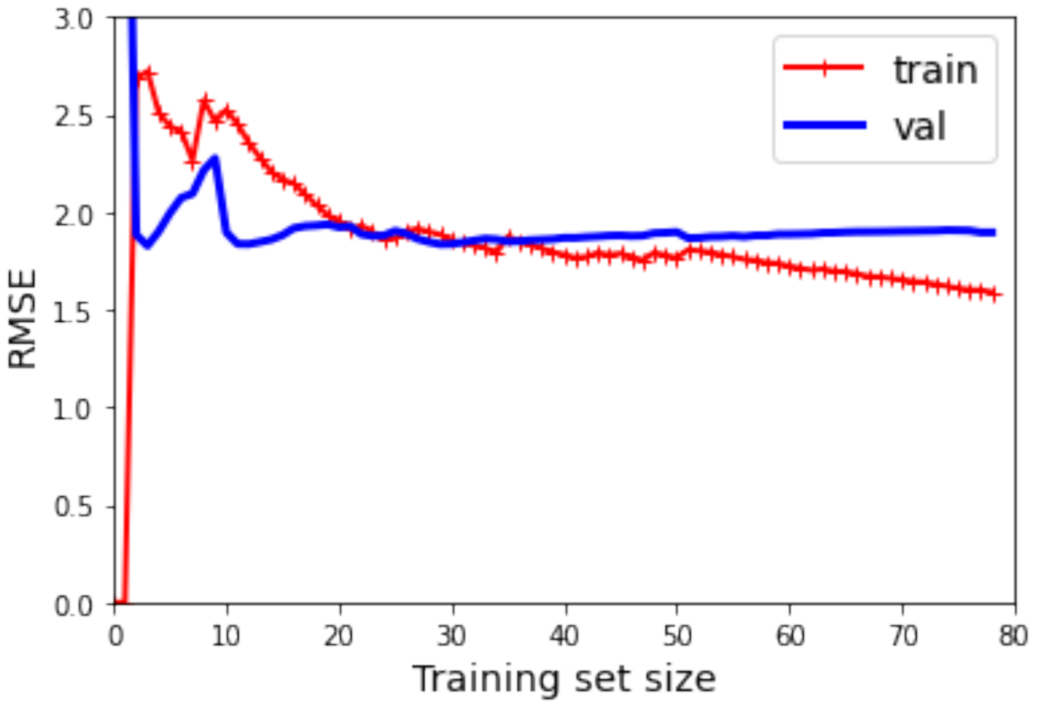
- Plots of the model's performance on the training set and the validation set as a function of the training set size

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=10)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)
```

Learning curve

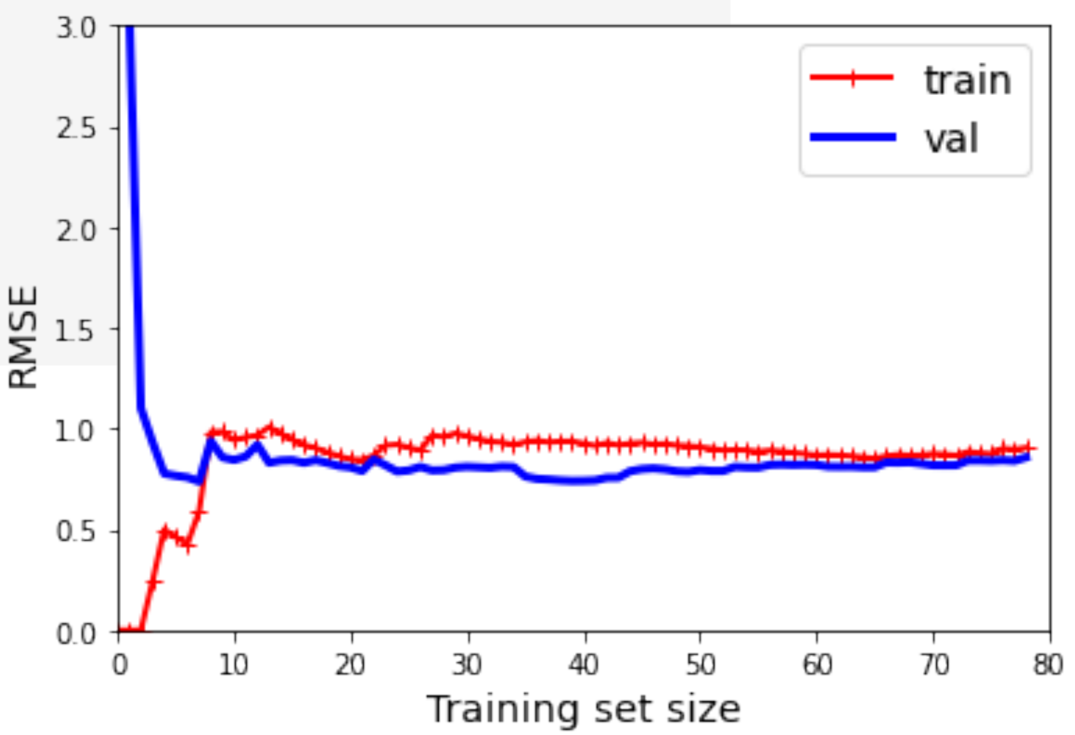
- Linear regression

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])
plt.show()
```



- 2-degree polynomial regression

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
plt.show()
```



Bias/variance tradeoff

- Generalization error can be expressed as the sum of three different errors:

  1. Bias: This is due to wrong assumptions, such as assuming the data is linear when it is actually quadratic

  2. Variance: This is due to the model's excessive sensitivity to small variations in the training data (i.e., degrees of freedom)

  3. Irreducible error: This is due to the noisiness of the data

Regularized linear models

- One way to reduce overfitting is to regularize linear models
- For linear models, regularization is typically achieved by containing the weights of the model

$$h(\mathbf{x}) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d = \sum_{i=0}^{d} \theta_i x_i = \langle \mathbf{x}, \boldsymbol{\theta} \rangle$$

  - Keep the model weights as small as possible
- Regularization is used only during the training process
- We will look at
  - Ridge regression
  - Lasso regression

Ridge regression

- We minimize the following loss function

$$\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 + \alpha\|\boldsymbol{\theta}\|_2^2$$

  - $\alpha > 0$ is hyperparameter

- This problem has closed-form solution

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X} + \alpha\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

  - Or, we can use gradient descent

Ridge regression

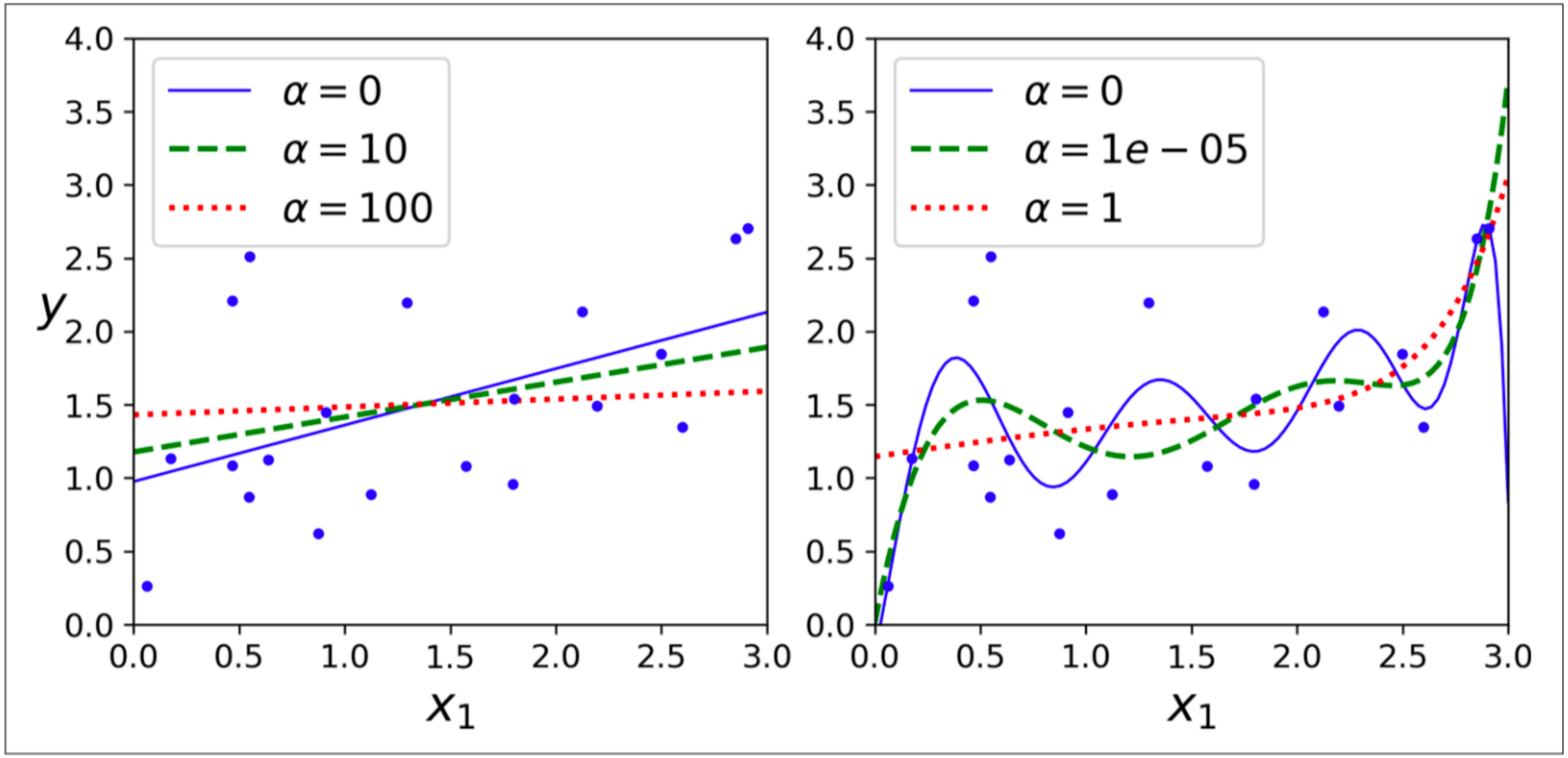


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

Lasso

- We minimize the following loss function

$$\frac{1}{2n}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2 + \alpha \sum_i |\theta_i|$$

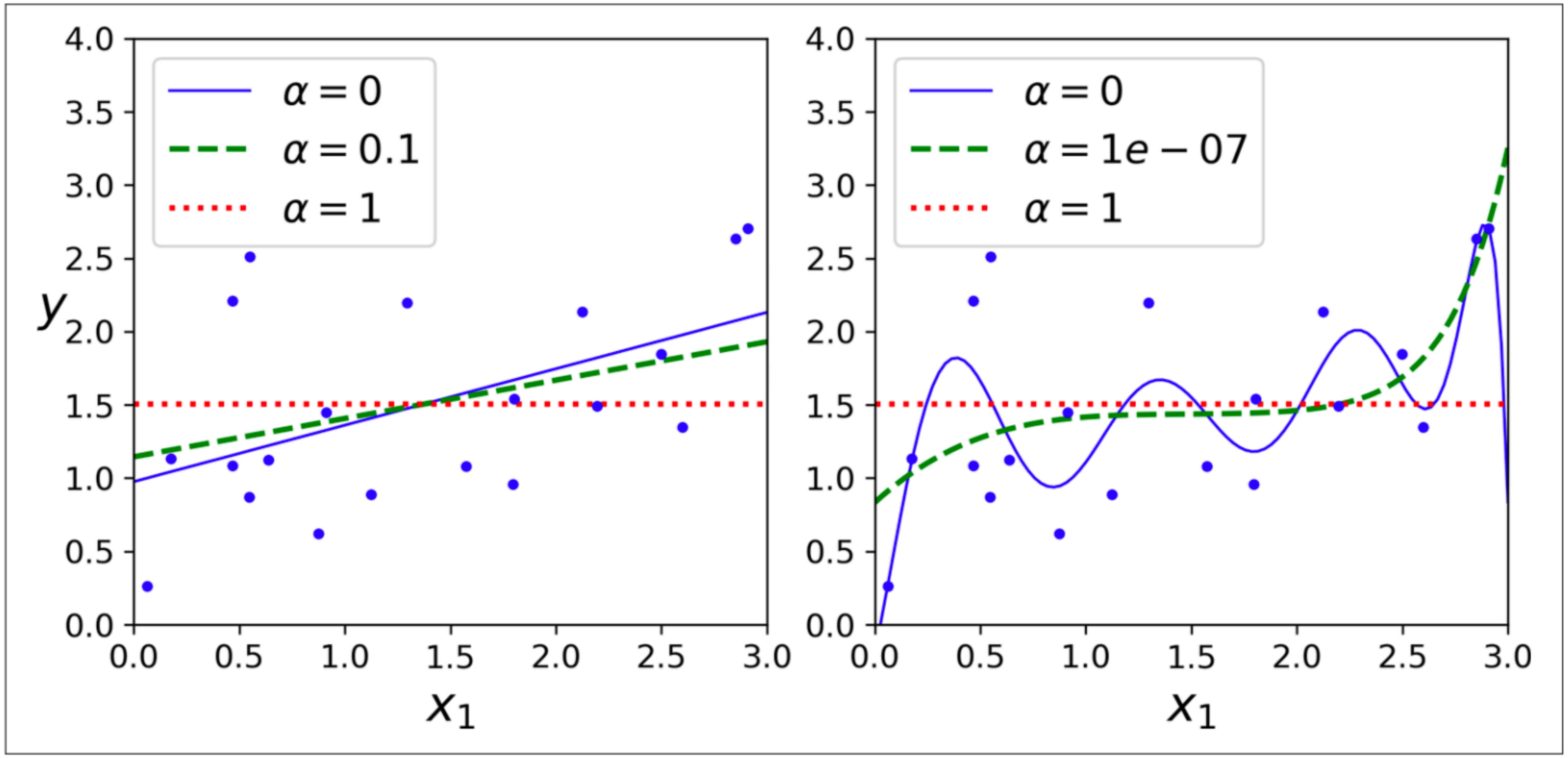


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization