

## 实验五：多项式回归与正则化

### 1. 线性回归

#### 1.1 最小二乘法

#### 1.2 梯度下降

##### 1.2.1 批量梯度下降

##### 1.2.2 随机梯度下降

##### 1.2.3 MiniBatch梯度下降

##### 1.2.4 3种策略的对比实验

### 2. 多项式回归

#### 2.1 欠拟合与过拟合

#### 2.2 样本数量对结果的影响

#### 2.3 多项式回归的过拟合风险

#### 2.4 正则化

##### 2.4.1 岭回归

##### 2.4.2 Lasso回归

### 3. 动手实践

## 实验五：多项式回归与正则化

本实验旨在让同学们回顾上次实验线性回归的知识，并了解多项式回归与正则化的相关知识以及相关使用。

## 1. 线性回归


### 1.1 最小二乘法

导入相关库

```
import numpy as np
import os
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
import warnings
warnings.filterwarnings('ignore')
np.random.seed(42)
```

回归方程：

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

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


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

### 1. 线性回归模型:

- $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$
- 这是多变量线性回归模型的预测方程，其中:
  - $\hat{y}$  是预测值。
  - $\theta_0, \theta_1, \dots, \theta_n$  是模型的参数 ( $\theta_0$  是截距项,  $\theta_1$  到  $\theta_n$  是特征的权重)。
  - $x_1, x_2, \dots, x_n$  是特征变量。

### 2. 均方误差 (MSE) :

- $\text{MSE}(\mathbf{X}, h_{\theta}) = (1/m) \sum (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$
- 这是均方误差公式，用于衡量线性回归模型的性能，它计算模型预测和实际值之间差异的平方的平均值。
- 其中:
  - $m$  是样本数量。
  - $\theta^T \cdot \mathbf{x}^{(i)}$  是第  $i$  个样本的预测值。
  - $y^{(i)}$  是第  $i$  个样本的实际值。

### 3. 正规方程:

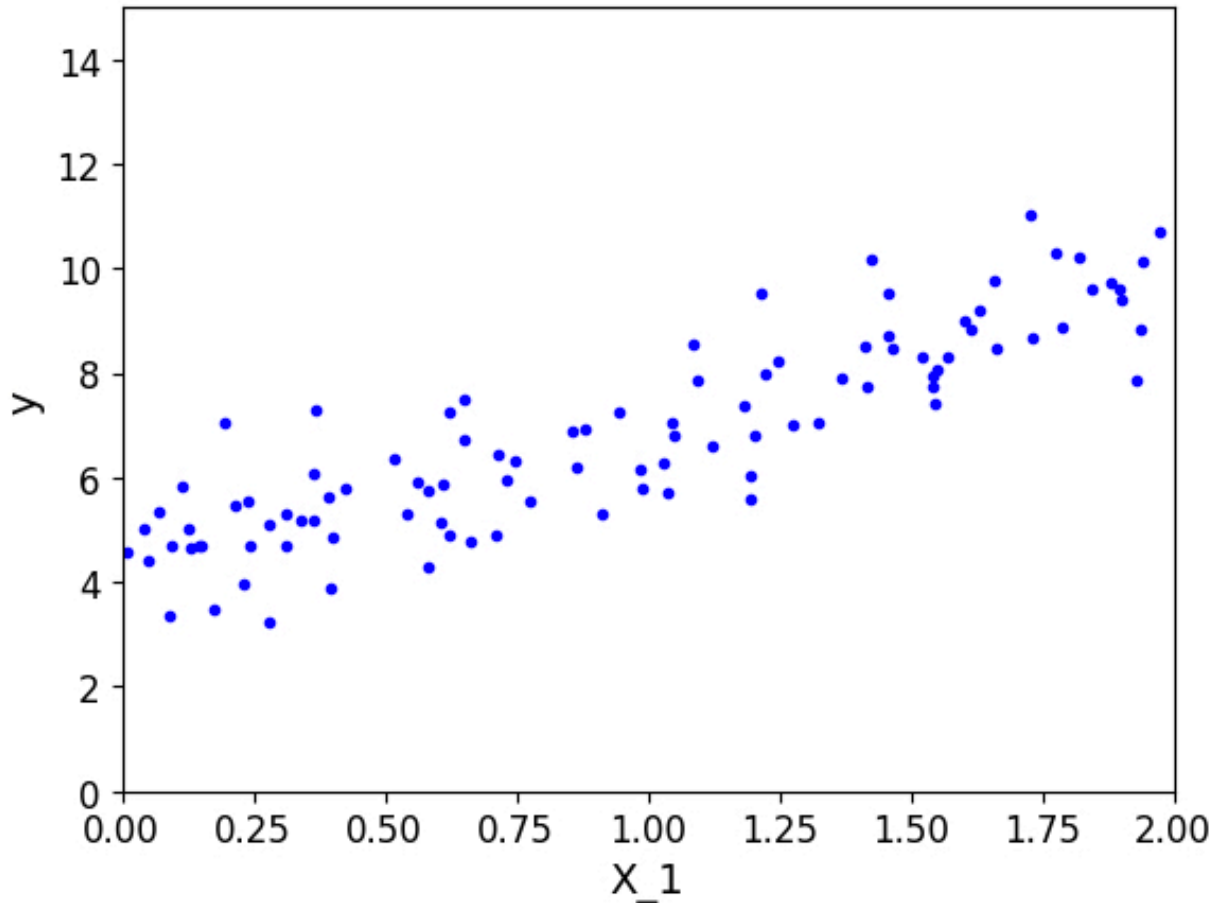
- $\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$
- 这是一个闭合形式的解，用于直接计算出线性回归模型的最优参数  $\hat{\theta}$ 。
- $\hat{\theta}$  是使成本函数（通常是均方误差）最小化的参数向量。
- $\mathbf{X}$  是设计矩阵，其中每一行代表一个训练样本，每一列代表一个特征（第一列通常是全1，对应于截距项  $\theta_0$ ）。
- $\mathbf{X}^T$  是  $\mathbf{X}$  的转置。
- $(\mathbf{X}^T \cdot \mathbf{X})^{-1}$  是  $\mathbf{X}^T \cdot \mathbf{X}$  矩阵的逆矩阵。
- $\mathbf{y}$  是目标值的向量。

正规方程是一种求解线性回归的方法(最小二乘法)，**不需要迭代优化**，直接通过数学公式求解参数。但是当特征数量很大或者  $\mathbf{X}^T \cdot \mathbf{X}$  **不可逆**（或接近不可逆，也称为奇异或退化）时，计算这个逆矩阵可能是计算上不可行的，此时通常会用梯度下降或其他优化算法来求解  $\theta$ 。

生成一个简单的线性回归数据集

```
import numpy as np
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')
plt.ylabel('y')
plt.axis([0,2,0,15])
plt.show()
```



```
X_b = np.c_[np.ones((100,1)),X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

计算线性回归模型的参数，使用的是前面提到的正规方程方法。

1. `X_b = np.c_[np.ones((100,1)),X]` :

- 这行代码在原始的 `x` 数组前面添加了一列1。这一列1对应于线性回归方程中的截距项 $\theta_0$ 。
- `np.ones((100,1))` 创建了一个100行1列的数组，其中的所有元素都是1。
- `np.c_` 是NumPy中的一个功能，它用于沿着第二个轴（即列）连接两个或多个数组。所以 `np.c_[np.ones((100,1)),X]` 的结果是一个100行2列的数组，第一列是全1（对应截距 $\theta_0$ ），第二列是原始的 `x` 数据。

2. `theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)` :

- 这行代码利用正规方程计算最佳参数 $\theta$ 的值。
- `X_b.T` 是 `X_b` 的转置。
- `X_b.T.dot(X_b)` 是 `X_b` 的转置和 `X_b` 本身的矩阵乘积。这一步计算了设计矩阵的协方差矩阵。

- `np.linalg.inv(...)` 计算括号内矩阵的逆矩阵。
- `dot(X_b.T)` 再次与 `x_b` 的转置进行矩阵乘积。
- 最后, `.dot(y)` 与目标值 `y` 进行矩阵乘积, 得到 $\theta$ 的最佳估计值。

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

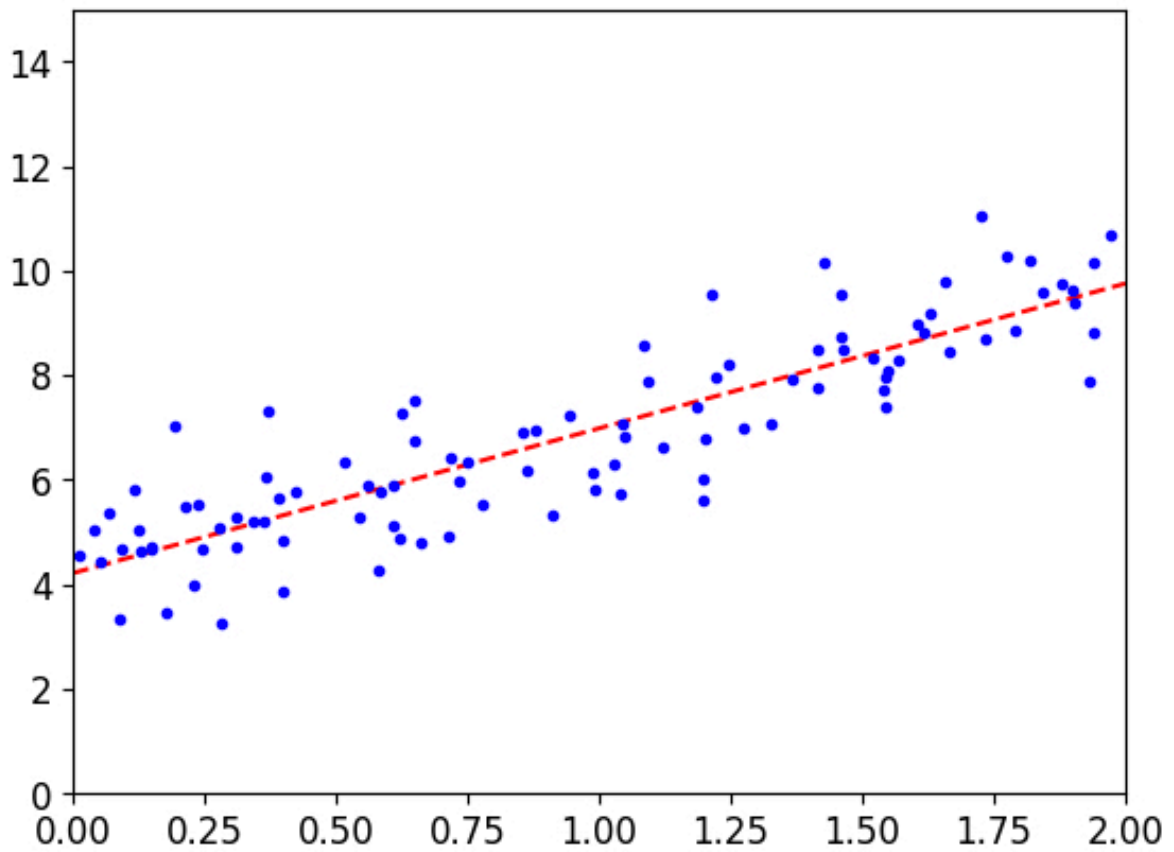
用来做预测, 基于之前计算出的 `theta_best`, 也就是通过正规方程得到的最优线性回归模型参数。

```
X_new = np.array([[0],[2]])
X_new_b = np.c_[np.ones((2,1)),X_new]
y_predict = X_new_b.dot(theta_best)
y_predict
# array([[4.21509616],
#        [9.75532293]])
```

1. `X_new = np.array([[0],[2]])`:
  - 这行代码创建了一个新的NumPy数组 `x_new`, 包含两个元素0和2, 这两个值是我们想要进行预测的 `x` 值。这个数组的形状是(2, 1), 即两行一列, 分别代表两个不同的测试点。
2. `X_new_b = np.c_[np.ones((2,1)),X_new]`:
  - 这行代码给 `x_new` 数组增加了一列, 这一列的值全部为1, 对应于线性回归模型中的截距项 $\theta_0$ 。
  - 结果是 `x_new_b`, 它是一个2x2的数组, 第一列全为1 (对应截距项 $\theta_0$ ), 第二列是 `x_new` 的值。
3. `y_predict = X_new_b.dot(theta_best)`:
  - 这行代码通过矩阵乘法计算预测值 `y_predict`。它将 `x_new_b` 和 `theta_best` 相乘。因为 `theta_best` 包含了从训练数据中学到的模型参数, 所以这个操作实质上是在对新的 `x` 值使用我们的模型做出预测。

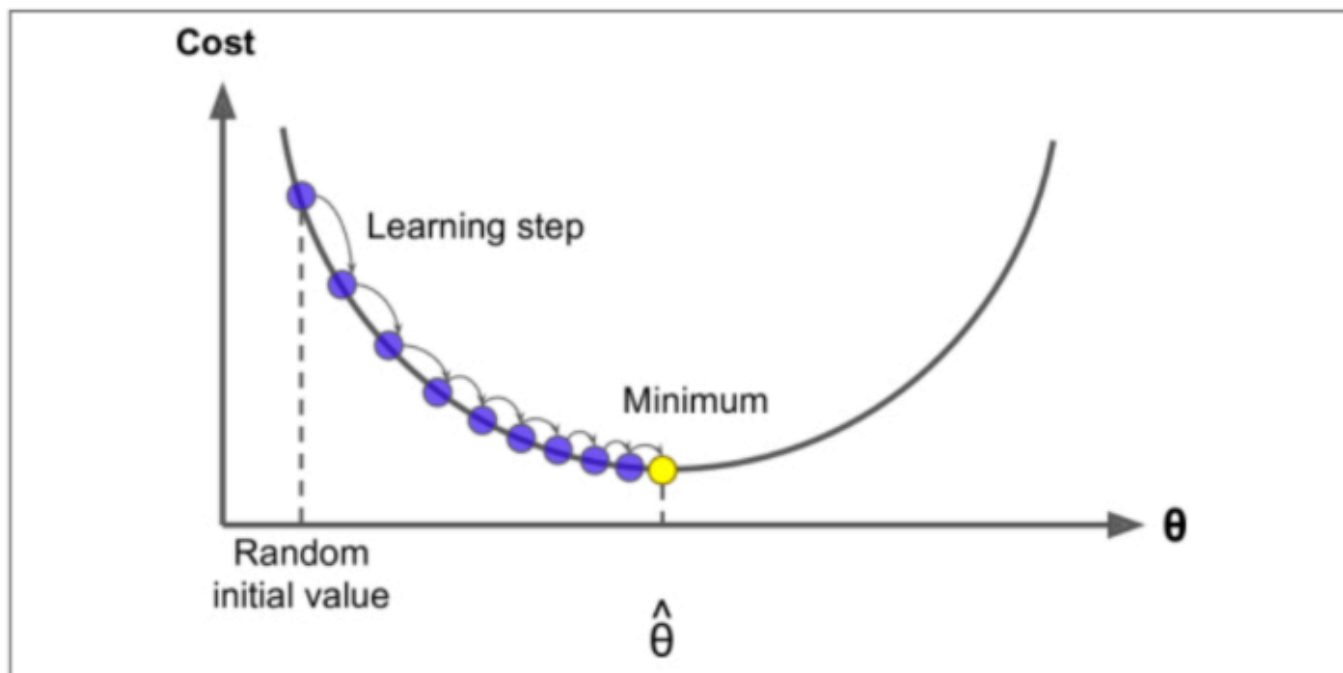
绘图

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



## 1.2 梯度下降

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X,y)
print (lin_reg.coef_) # [[2.77011339]]
print (lin_reg.intercept_) # [4.21509616]
```

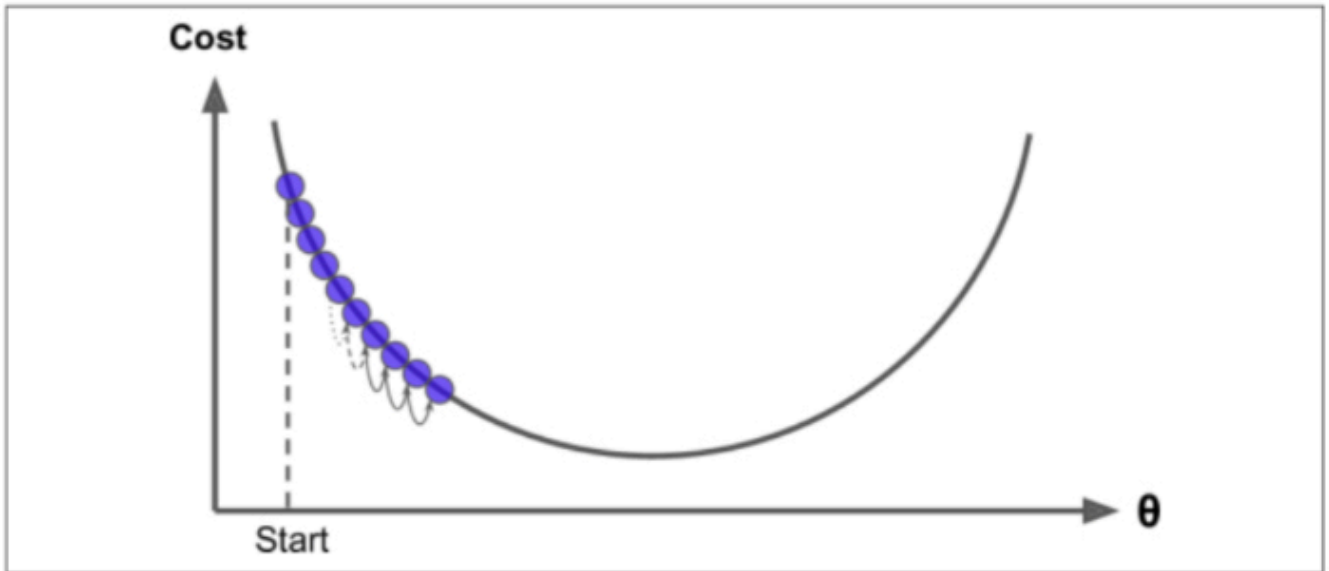


这个图展示了梯度下降算法在寻找最小成本函数值的过程。梯度下降是一个优化算法，用于最小化一个函数，常用于机器学习的参数优化。图中的要素解释如下：

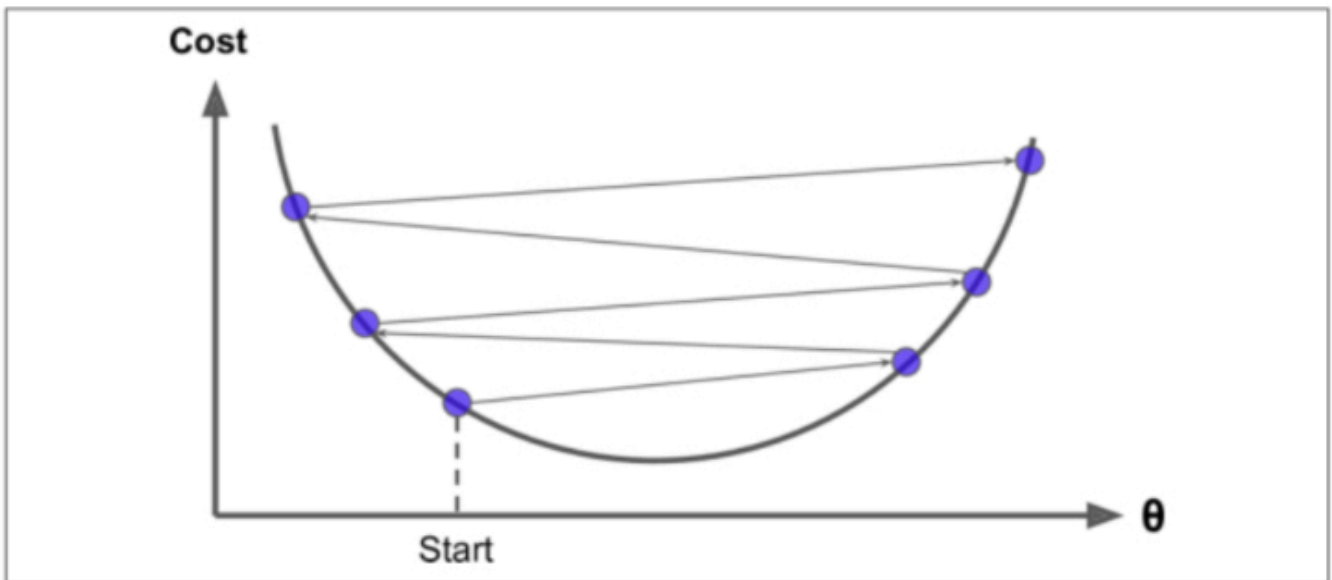
- **成本 (Cost)**：这是垂直轴，表示成本函数（或损失函数）的值。在机器学习中，成本函数通常用于衡量模型预测值与实际值之间的差异。目标是找到一组参数 $\theta$ ，使得这个成本最小。
- **参数 ( $\theta$ )**：这是水平轴，代表模型的参数。虽然实际的模型可能有多个参数，但为了可视化，这里只显示了一个参数。
- **随机初始值**：这个点表示算法开始时参数 $\theta$ 的初始值。梯度下降通常从随机选择的初始值开始迭代过程。
- **学习步长 (Learning step)**：这些蓝色的点表示梯度下降算法的每一步迭代。在每一步，算法计算成本函数关于参数 $\theta$ 的梯度（导数），然后在梯度的反方向上更新 $\theta$ 值，因为这是成本降低最快的方向。学习步长的大小通常由学习率控制，学习率太大可能会越过最低点，太小则收敛速度慢。
- **最小值**：这个黄色点表示成本函数的最小值，这也是梯度下降算法寻找的目标。理论上，梯度下降应该在这一点停止迭代，因为在这一点，成本函数已经不能再减小了。
- **最优参数 ( $\hat{\theta}$ )**：这个地方在水平轴上表示梯度下降算法最终找到的最优参数值。如果一切顺利，这会是使成本函数最小化的参数值。

图形中的曲线通常被称为成本函数的等高线，梯度下降算法的目的是找到这个曲线的最低点。在多维空间中，这个过程涉及多个参数，梯度下降将在高维曲面上寻找最低点。

- 步长太小，训练太慢：

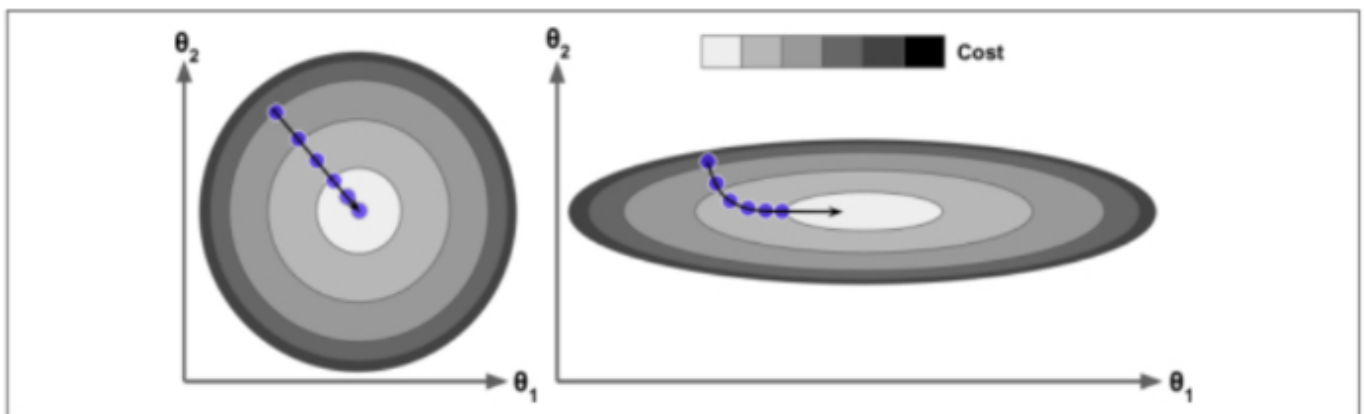


- 步长太大，可能会导致参数更新过头，从而错过最低点。这在图中表现为参数值在最小值两侧反复跳动，无法稳定下来，最终可能不会收敛到最小值。



总结：学习率应当尽可能小，随着迭代的进行应当越来越小。

- 标准化



这个图展示了在应用梯度下降算法时特征标准化（也称为特征缩放）的重要性。特征标准化通常涉及将所有特征调整到相同的尺度，例如使它们具有相同的均值和方差。图中展示了两个不同的情况：

- **左图：**表示进行了特征标准化后的成本函数等高线。当两个参数（ $\theta_1$ 和 $\theta_2$ ）的尺度相似时，成本函数形成的等高线接近圆形。这时，梯度下降算法可以更直接地朝着成本最小点（即中心的最低点）移动，因此通常能够更快速地收敛。
- **右图：**表示没有进行特征标准化的成本函数等高线。在这种情况下，由于参数尺度的不同，等高线呈现拉长的椭圆形状。这会使梯度下降路径曲折，因为梯度在不同参数维度上的变化量差异很大，从而可能导致梯度下降在达到最低点之前需要更多的迭代，甚至可能在椭圆的窄方向上震荡。

### 1.2.1 批量梯度下降

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)} \quad \nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

批量梯度下降算法在Python中的实现，用于最小化一个线性回归模型的成本函数。

```
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
```

这里，我们假设已经有了一个包含截距项的输入特征矩阵 `x_b`，以及对应的目标变量 `y`。代码的各部分功能如下：

1. `eta = 0.1`：设置学习率 `eta`（ $\eta$ ），这是梯度下降中每一步更新参数时使用的乘数。学习率控制了步长的大小。
2. `n_iterations = 1000`：设置迭代次数为1000，这意味着梯度下降算法将进行1000次参数更新。
3. `m = 100`：假设我们有100个训练样本，这在计算平均梯度时会用到。
4. `theta = np.random.randn(2,1)`：从正态分布中随机初始化参数 $\theta$ 。在这里， $\theta$ 是一个2x1的向量，因为我们假设有一个参数和一个截距项。
5. 接下来的for循环进行梯度下降算法的迭代：
  - a. `gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)`：计算成本函数的梯度。这是通过对成本函数关于参数 $\theta$ 的偏导数的估计，这里用的是成本函数对 $\theta$ 的梯度向量。`X_b.T.dot(X_b.dot(theta) - y)`部分计算了预测值和实际值之间的误差，然后乘以输入特征矩阵的转置（`X_b.T`），最后乘以2/m得到平均梯度。



b. `theta = theta - eta * gradients`: 更新 $\theta$ 。这里，当前的 $\theta$ 减去学习率乘以梯度。这个操作是梯度下降算法的核心，它“下降”在成本函数梯度的方向上，目的是减少成本。

执行这段代码后，`theta` 将包含线性回归模型参数的估计值，这些参数应该会最小化模型在给定数据上的均方误差。在1000次迭代后，算法收敛，得到的参数 $\theta$ 将用于新数据的预测。

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

```
X_new_b.dot(theta)
# array([[4.21509616],
#        [9.75532293]])
```

定义了一个函数 `plot_gradient_descent`，用于执行梯度下降并可视化每一步迭代后的线性回归模型。

```
theta_path_bgd = []
def plot_gradient_descent(theta, eta, theta_path = None):
    m = len(X_b)
    plt.plot(X, y, 'b.')
    n_iterations = 1000
    for iteration in range(n_iterations):
        y_predict = X_new_b.dot(theta)
        plt.plot(X_new, y_predict, 'b-')
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta = theta - eta * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel('X_1')
    plt.axis([0, 2, 0, 15])
    plt.title('eta = {}'.format(eta))
```

1. `theta_path_bgd = []`: 创建一个空列表，用于存储梯度下降过程中的每一个 $\theta$ 值。
2. `def plot_gradient_descent(theta, eta, theta_path=None):`: 定义了函数 `plot_gradient_descent`。这个函数接受三个参数: `theta` (参数 $\theta$ 的初始值)，`eta` (学习率)，以及可选的 `theta_path` (存储 $\theta$ 值的路径)。
3. `m = len(X_b)`: 获取特征矩阵 `X_b` 中的样本数量 `m`。
4. `plt.plot(X, y, 'b.')`: 使用蓝点 ('b.') 在图中绘制训练数据集 (特征 `X` 与目标值 `y`)。
5. `n_iterations = 1000`: 设置梯度下降的迭代次数为1000。
6. `for iteration in range(n_iterations):`: 开始一个循环，进行1000次迭代。
7. `y_predict = X_new_b.dot(theta)`: 计算当前 $\theta$ 值下的预测值 `y_predict`。

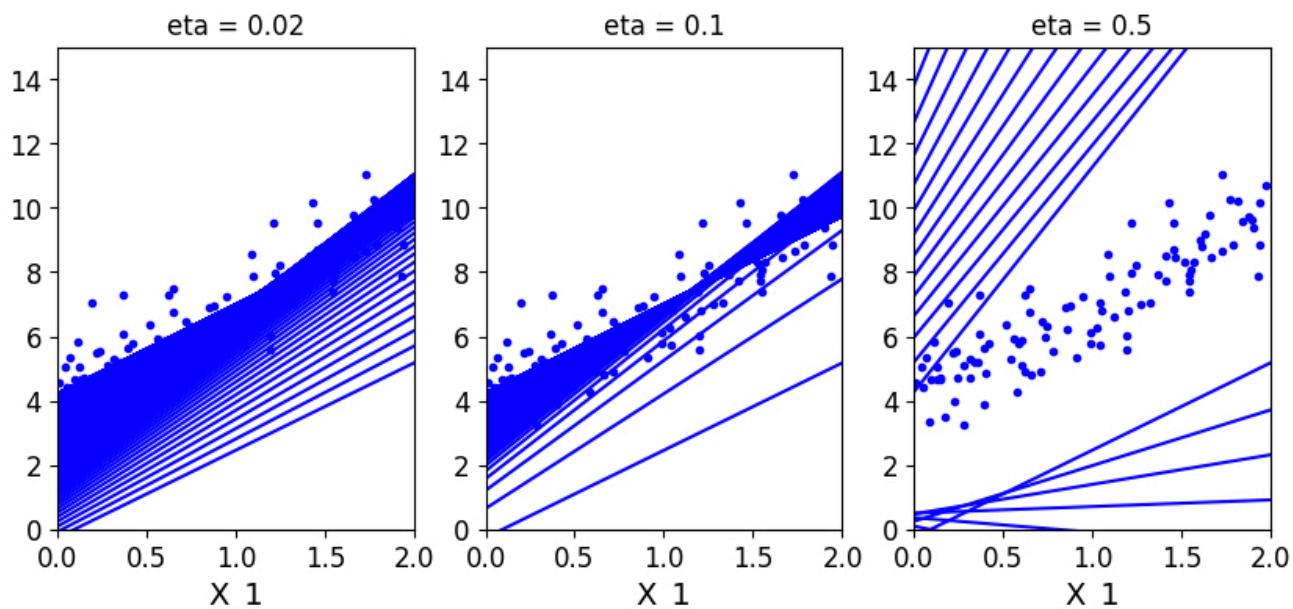
8. `plt.plot(X_new, y_predict, 'b-')`: 使用蓝线 ('b-') 绘制当前迭代步的模型预测线。
9. `gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)`: 计算成本函数的梯度。
10. `theta = theta - eta * gradients`: 按学习率 `eta` 更新  $\theta$  值。
11. `if theta_path is not None:`: 如果提供了 `theta_path` 参数, 将当前的  $\theta$  添加到路径列表中。
12. `plt.xlabel('X_1')`: 设置x轴的标签为" $X_1$ "。
13. `plt.axis([0,2,0,15])`: 设置图表的x轴和y轴的范围。
14. `plt.title('eta = {}'.format(eta))`: 设置图表的标题, 显示当前的学习率 `eta` 的值。

此函数的主要目的是为了演示在不同的学习率下梯度下降算法如何工作, 并直观展示每次迭代中线性回归模型如何逐渐逼近最优解。每次迭代后, 都会绘制出模型的预测线, 以便我们可以看到模型是如何随着迭代而更新的。如果 `theta_path` 参数被提供, 那么每个  $\theta$  值都会被记录下来, 这可以用于之后分析梯度下降的路径。

绘图。

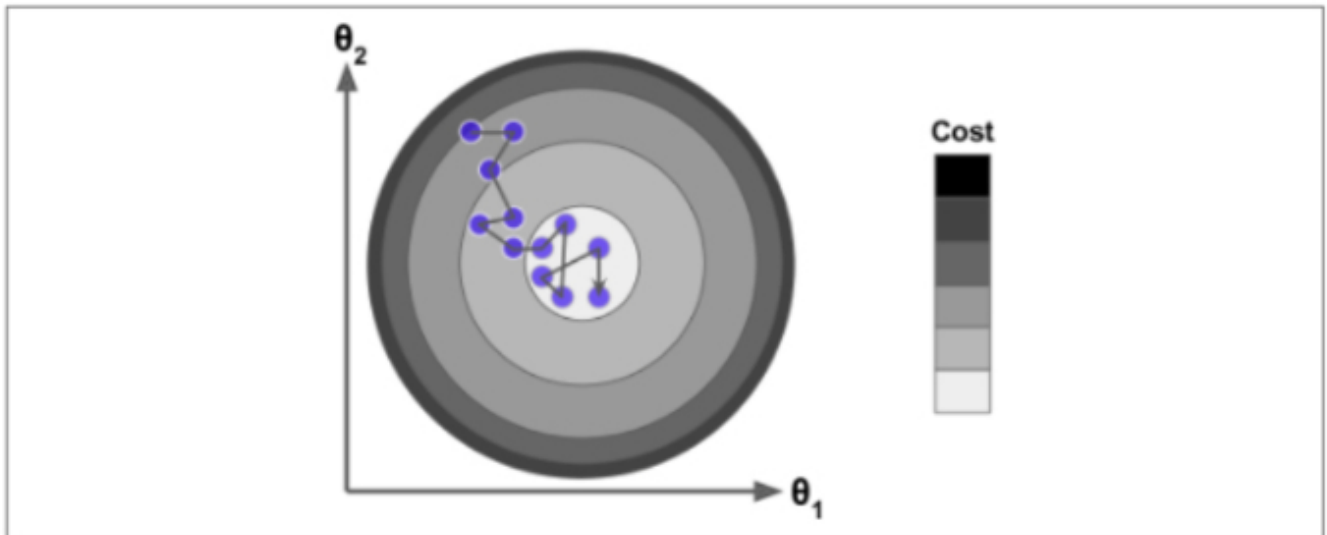
```
theta = np.random.randn(2,1)

plt.figure(figsize=(10,4))
plt.subplot(131)
plot_gradient_descent(theta,eta = 0.02)
plt.subplot(132)
plot_gradient_descent(theta,eta = 0.1,theta_path=theta_path_bgd)
plt.subplot(133)
plot_gradient_descent(theta,eta = 0.5)
plt.show()
```



每个子图都将显示使用特定学习率执行梯度下降时模型预测的演变。通过比较三个子图, 我们可以观察到学习率如何影响梯度下降的收敛速度和稳定性。一个很小的学习率 (如0.02) 可能会导致收敛速度很慢, 而一个较大的学习率 (如0.5) 可能导致算法发散或在最小值附近震荡, 而一个适中的学习率 (如0.1) 可能会相对较快并稳定地收敛到最小值。

## 1.2.2 随机梯度下降



这段代码实现了随机梯度下降（Stochastic Gradient Descent, SGD）算法，用于线性回归模型。SGD是梯度下降的一种变体，每次迭代只使用一个数据点来更新参数 $\theta$ ，而不是基于整个数据集。这可以显著加快算法的运行速度，并且可以用于在线学习。

```
theta_path_sgd=[]
m = len(X_b)
np.random.seed(42)
n_epochs = 50

t0 = 5
t1 = 50

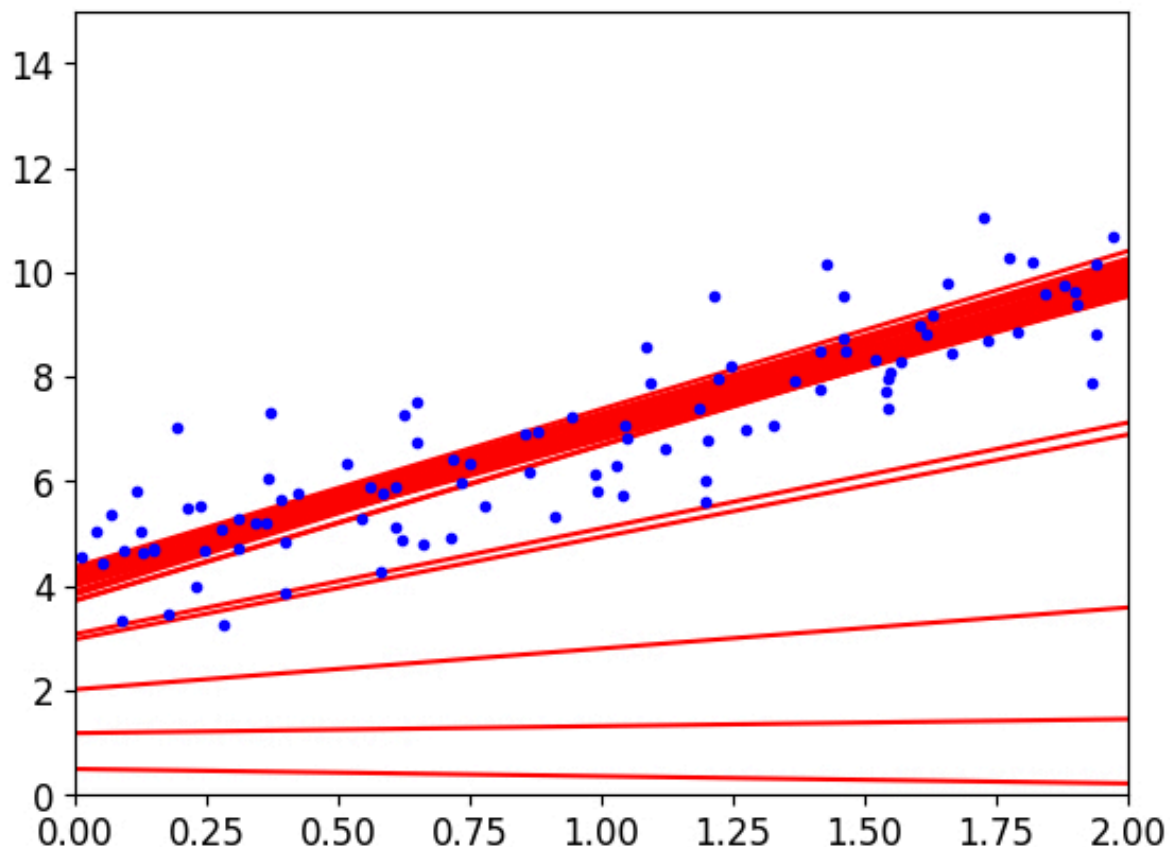
def learning_schedule(t):
    return t0/(t1+t)

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

for epoch in range(n_epochs):
    for i in range(m):
        if epoch < 10 and i<10:
            y_predict = X_new_b.dot(theta)
            plt.plot(X_new,y_predict, 'r-')
            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
            theta_path_sgd.append(theta)

plt.plot(X,y, 'b. ')
plt.axis([0,2,0,15])
```

```
plt.show()
```



1. `theta_path_sgd = []`: 初始化一个空列表, 用来存储每次迭代后的参数 $\theta$ 。
2. `m = len(X_b)`: 获取特征矩阵 `X_b` 中的样本数量 `m`。
3. `np.random.seed(42)`: 设置随机数生成器的种子, 以保证结果可重现。
4. `n_epochs = 50`: 设置遍历整个数据集的次数, 称为epochs。每个epoch包含 `m` 次迭代, 其中 `m` 是样本数量。
5. `t0, t1 = 5, 50`: 这是学习调度函数的超参数, 用于逐渐减少学习率。
6. `def learning_schedule(t)`: 定义了一个学习调度函数, 它根据给定的迭代次数 `t` 计算学习率。随着 `t` 的增加, 返回的学习率逐渐减小。
7. `theta = np.random.randn(2,1)`: 从标准正态分布中随机初始化参数 $\theta$ 。
8. 双层循环 `for epoch in range(n_epochs):` 和 `for i in range(m):`: 外层循环遍历每一个epoch, 内层循环遍历每一个样本。
9. 条件判断 `if epoch < 10 and i < 10:`: 在前10个epochs中, 仅在每个epoch的前10次迭代中, 绘制模型预测的线。
10. `random_index = np.random.randint(m)`: 随机选择一个样本的索引。
11. `xi = X_b[random_index:random_index+1]` 和 `yi = y[random_index:random_index+1]`: 选择随机索引对应的样本和标签。
12. `gradients = 2 * xi.T.dot(xi.dot(theta) - yi)`: 计算随机选择的样本的梯度。
13. `eta = learning_schedule(epoch * m + i)`: 计算当前迭代的学习率。
14. `theta = theta - eta * gradients`: 更新参数 $\theta$ 。

15. `theta_path_sgd.append(theta)`：将更新后的 $\theta$ 添加到路径列表中。
16. `plt.plot(x, y, 'b.')`：在图上以蓝点表示数据点。
17. `plt.axis([0,2,0,15])`：设置x轴和y轴的范围。
18. `plt.show()`：显示图形。

通过随机选择样本来更新参数 $\theta$ ，随机梯度下降算法通常能更快地收敛到最小值附近，尽管它可能在最小值附近波动，而不是平稳地收敛。每次迭代仅使用一个数据点，使得SGD特别适用于处理大规模数据集。

### 1.2.3 MiniBatch梯度下降

实现小批量梯度下降（Minibatch Gradient Descent）算法。小批量梯度下降是介于随机梯度下降（SGD）和批量梯度下降（BGD）之间的方法，它每次使用一个小批量（minibatch）的样本来更新参数 $\theta$ ，而不是单个样本（像SGD）或全部样本（像BGD）。

```
theta_path_mgd=[]
n_epochs = 50
minibatch = 16
theta = np.random.randn(2,1)
t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)
np.random.seed(42)
t = 0
for epoch in range(n_epochs):
    ### 自行完成
```

```
theta
# array([[4.25490685],
#        [2.80388784]])
```

1. `theta_path_mgd = []`：初始化一个空列表，用来存储每次迭代后的参数 $\theta$ 。
2. `n_epochs = 50`：设置遍历整个数据集的次数，称为“epoch”。
3. `minibatch = 16`：设置每个小批量包含的样本数为16。
4. `theta = np.random.randn(2,1)`：从标准正态分布中随机初始化参数 $\theta$ 。
5. `t0, t1 = 200, 1000`：设置学习调度函数的超参数。
6. `def learning_schedule(t)`：定义学习调度函数，用于随着迭代次数 $t$ 的增加而降低学习率。
7. `np.random.seed(42)`：设置随机数生成器的种子，以保证结果可重现。
8. `t = 0`：初始化时间步 $t$ ，用于学习调度函数。
9. `for epoch in range(n_epochs):`：外层循环遍历每个epoch。

。 。 。

通过这种方式，小批量梯度下降结合了SGD的快速迭代和BGD稳定收敛的优点。小批量梯度下降通过在单个epoch内的多个小批量上进行迭代来减少参数更新的方差，并有助于更快地收敛到最小值。此外，小批量处理也更适合现代计算机和GPU的矩阵运算优化。

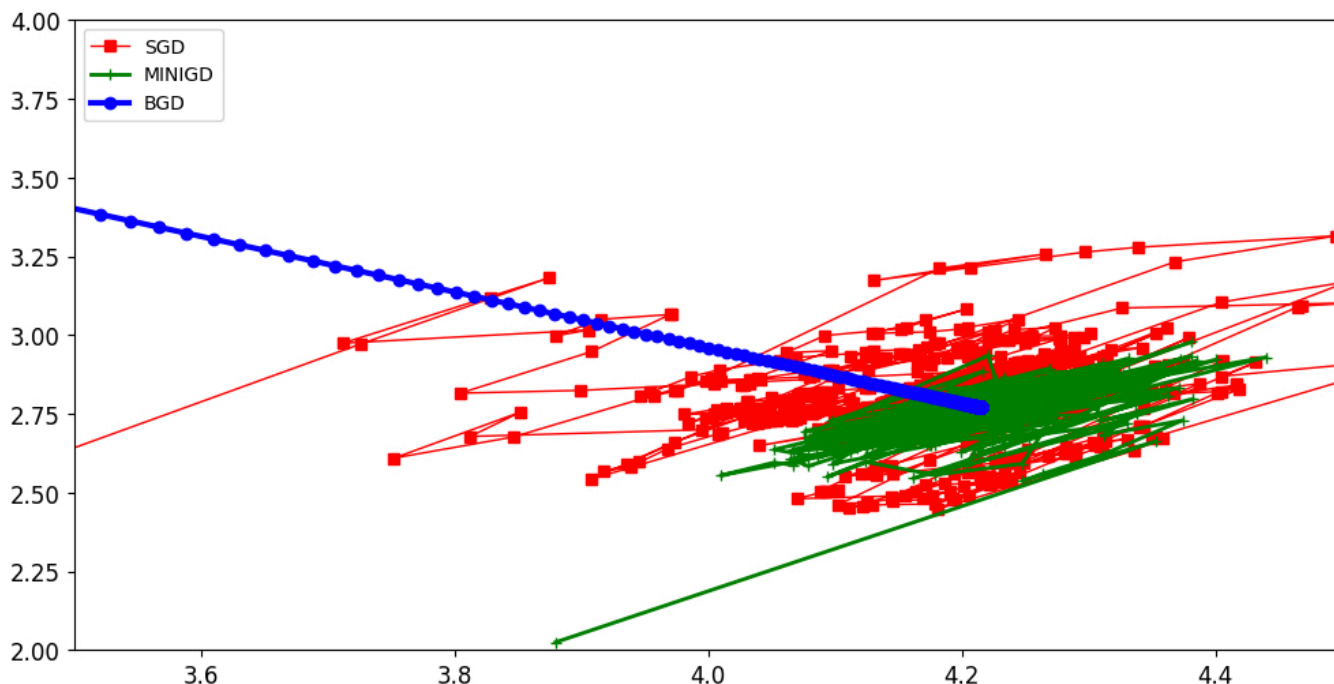
## 1.2.4 3种策略的对比实验

将三个不同梯度下降方法生成的参数路径（theta路径）列表转换成NumPy数组。这通常是为了便于后续的分析 and 可视化处理。

```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

使用Matplotlib库来可视化不同梯度下降算法的参数 $\theta$ 路径。

```
plt.figure(figsize=(12,6))
plt.plot(theta_path_sgd[:,0],theta_path_sgd[:,1], 'r-s',linewidth=1,label='SGD')
plt.plot(theta_path_mgd[:,0],theta_path_mgd[:,1], 'g-+',linewidth=2,label='MINIGD')
plt.plot(theta_path_bgd[:,0],theta_path_bgd[:,1], 'b-o',linewidth=3,label='BGD')
plt.legend(loc='upper left')
plt.axis([3.5,4.5,2.0,4.0])
plt.show()
```



自行分析一下，batch数量应该如何选择？

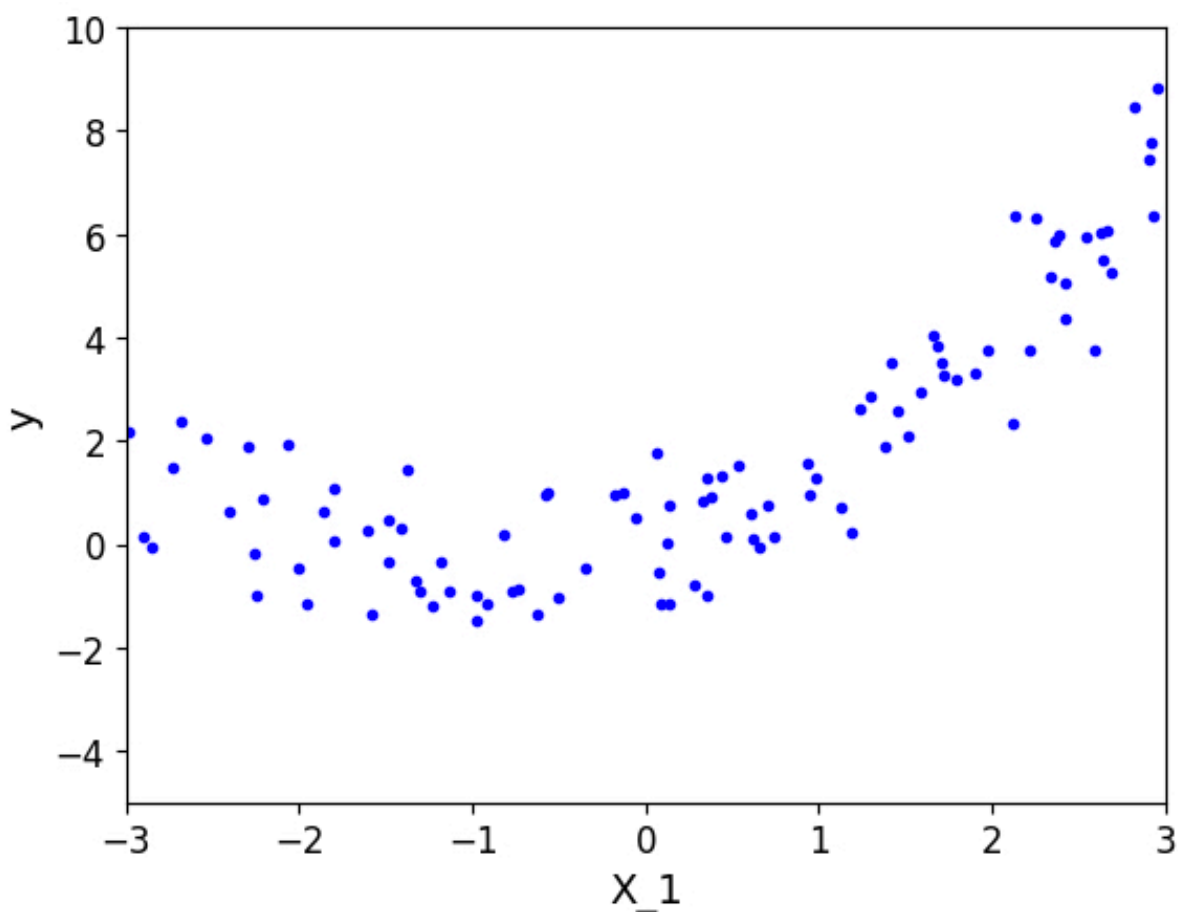
## 2. 多项式回归

生成了一个多项式回归的数据集。多项式回归是线性回归的一个扩展，它能够模拟数据之间非线性的关系。这里的数据集是根据一个二次方程生成的，包含了随机噪声。具体步骤和组成部分如下：

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

绘图

```
plt.plot(X,y,'b.')
plt.xlabel('X_1')
plt.ylabel('y')
plt.axis([-3,3,-5,10])
plt.show()
```



```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree = 2,include_bias = False)
X_poly = poly_features.fit_transform(X)
X[0] # array([2.38942838])
X_poly[0] # array([2.38942838, 5.709368  ])
```



这段代码演示了如何使用 `scikit-learn` 的 `LinearRegression` 模型来拟合通过 `PolynomialFeatures` 预处理的多项式数据，然后打印出模型的系数和截距。

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

1. `from sklearn.linear_model import LinearRegression`: 从 `sklearn.linear_model` 模块导入 `LinearRegression` 类。这是 `scikit-learn` 提供的线性回归模型，可以用来拟合数据集并预测结果。
2. `lin_reg = LinearRegression()`: 创建 `LinearRegression` 类的一个实例 `lin_reg`。这一步实际上初始化了一个线性回归模型，准备用来拟合数据。
3. `lin_reg.fit(X_poly, y)`: 使用 `fit` 方法将线性回归模型拟合到预处理后的多项式特征数据 `x_poly` 和目标变量 `y` 上。在这一步，模型会学习 `x_poly` 和 `y` 之间的关系，即找到最佳的参数（系数和截距），使得模型预测值与实际值之间的均方误差最小。
4. `print(lin_reg.coef_)`: 打印线性回归模型的系数。对于多项式回归来说，这些系数对应于原始特征及其多项式扩展的权重。在这个例子中，由于我们使用的是二次多项式特征，所以系数将包括线性项和二次项的系数。
5. `print(lin_reg.intercept_)`: 打印线性回归模型的截距。截距是指当所有特征值都为0时，模型预测的目标变量的值。

通过这些步骤，我们可以看到即使是使用线性模型，通过引入多项式特征也能够模拟非线性关系。这种方法扩展了线性模型的应用范围，允许它们捕捉更复杂的数据模式。

使用训练好的多项式回归模型来进行预测，并将预测结果与原始数据一起可视化。

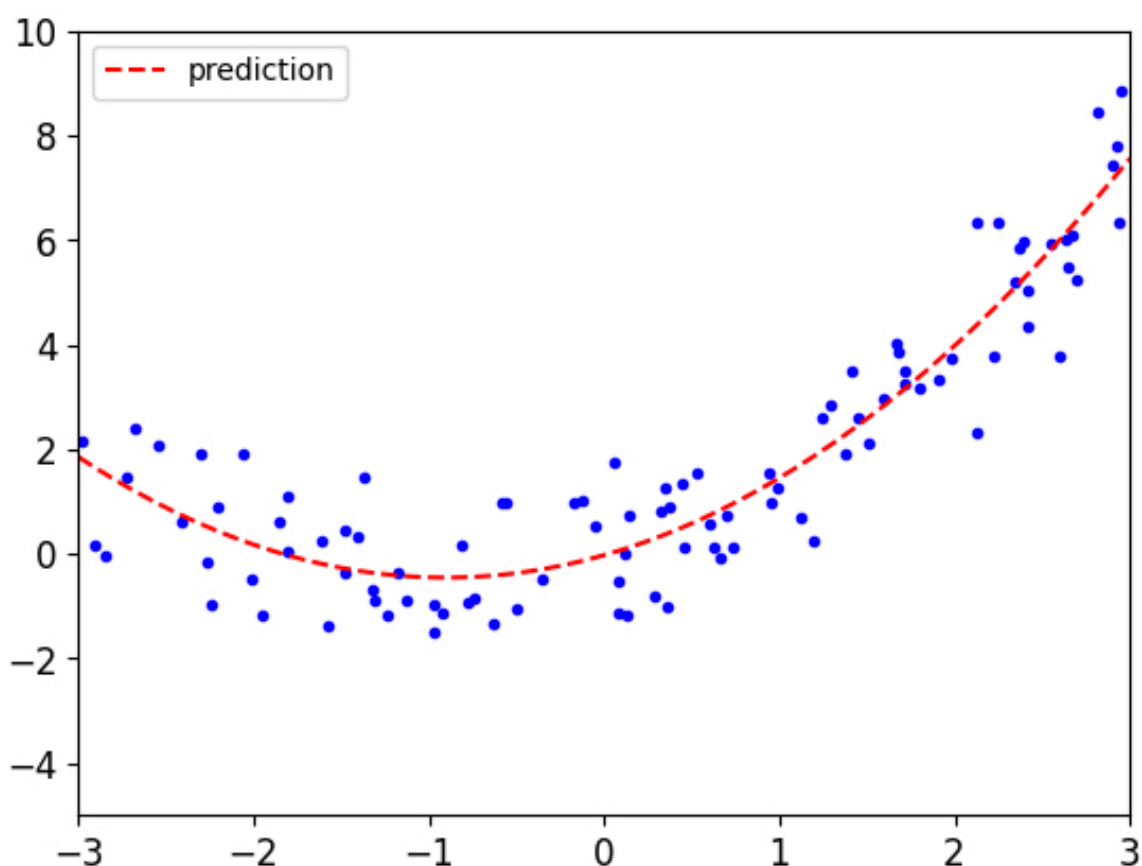
```
X_new = np.linspace(-3,3,100).reshape(100,1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X,y,'b.')
plt.plot(X_new,y_new,'r--',label='prediction')
plt.axis([-3,3,-5,10])
plt.legend()
plt.show()
```

1. `X_new = np.linspace(-3, 3, 100).reshape(100, 1)`: 生成了一个新的特征数组 `x_new`，它包含了从-3到3等间隔的100个点。这个数组被重塑成一个形状为(100, 1)的二维数组，以便我们可以使用它进行多项式变换和模型预测。
2. `X_new_poly = poly_features.transform(X_new)`: 使用之前创建的 `PolynomialFeatures` 实例 `poly_features` 来对 `X_new` 进行多项式变换，产生 `X_new_poly`。这样做是为了确保我们对新数据使用的是与训练数据相同的多项式扩展。
3. `y_new = lin_reg.predict(X_new_poly)`: 使用训练好的线性回归模型 `lin_reg` 来预测转换后的新数据 `X_new_poly` 的目标值，生成预测值 `y_new`。



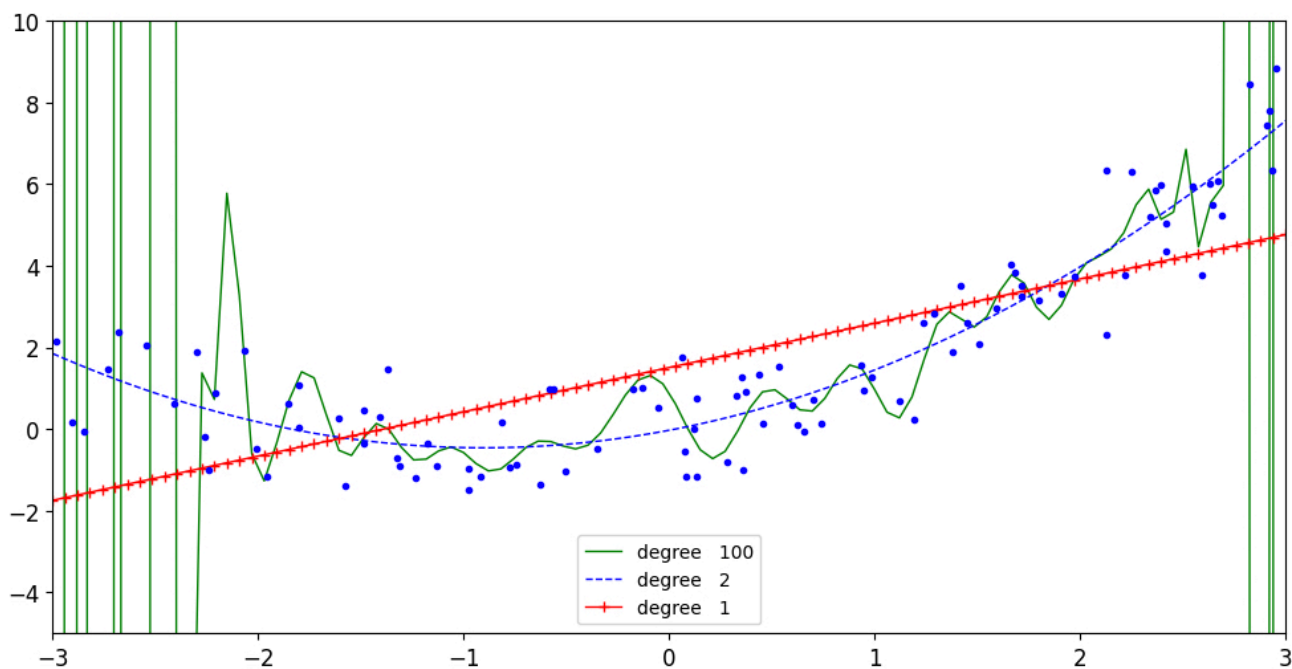
4. `plt.plot(X, y, 'b.')`: 使用蓝色点 ('b.') 在图上绘制原始训练数据点。这有助于我们比较模型预测和实际数据。
5. `plt.plot(X_new, y_new, 'r--', label='prediction')`: 使用红色虚线 ('r--') 在图上绘制预测线，标签设置为"prediction"。这条线显示了模型如何在整个数据范围内拟合原始数据。
6. `plt.axis([-3, 3, -5, 10])`: 设置x轴和y轴的显示范围，确保图形覆盖所有相关的数据点和预测。
7. `plt.legend()`: 添加图例，以便区分图中的不同元素。
8. `plt.show()`: 显示图形。这个命令绘制出了包含原始数据点和预测模型的图形，从而可以直观地评估模型的拟合效果。

通过这些步骤，这段代码有效地演示了多项式回归模型如何在考虑到数据非线性关系的情况下进行预测，并通过可视化方式展示了模型的预测效果。



## 2.1 欠拟合与过拟合

自行设置多项式度数为1, 2, 100, 观察多项式回归模型的结果并进行结果分析。如下:



## 2.2 样本数量对结果的影响

定义了一个函数 `plot_learning_curves`，它用于绘制模型的学习曲线，展示了随着训练集大小的增加，训练误差和验证误差是如何变化的。这些曲线可以帮助分析模型的性能，尤其是其对新数据的泛化能力。

```
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=100)
    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[:m]))
        val_errors.append(mean_squared_error(y_val,y_val_predict))
    plt.plot(np.sqrt(train_errors),'r-+',linewidth = 2,label = 'train_error')
    plt.plot(np.sqrt(val_errors),'b-',linewidth = 3,label = 'val_error')
    plt.xlabel('Training set size')
    plt.ylabel('RMSE')
    plt.legend()
```

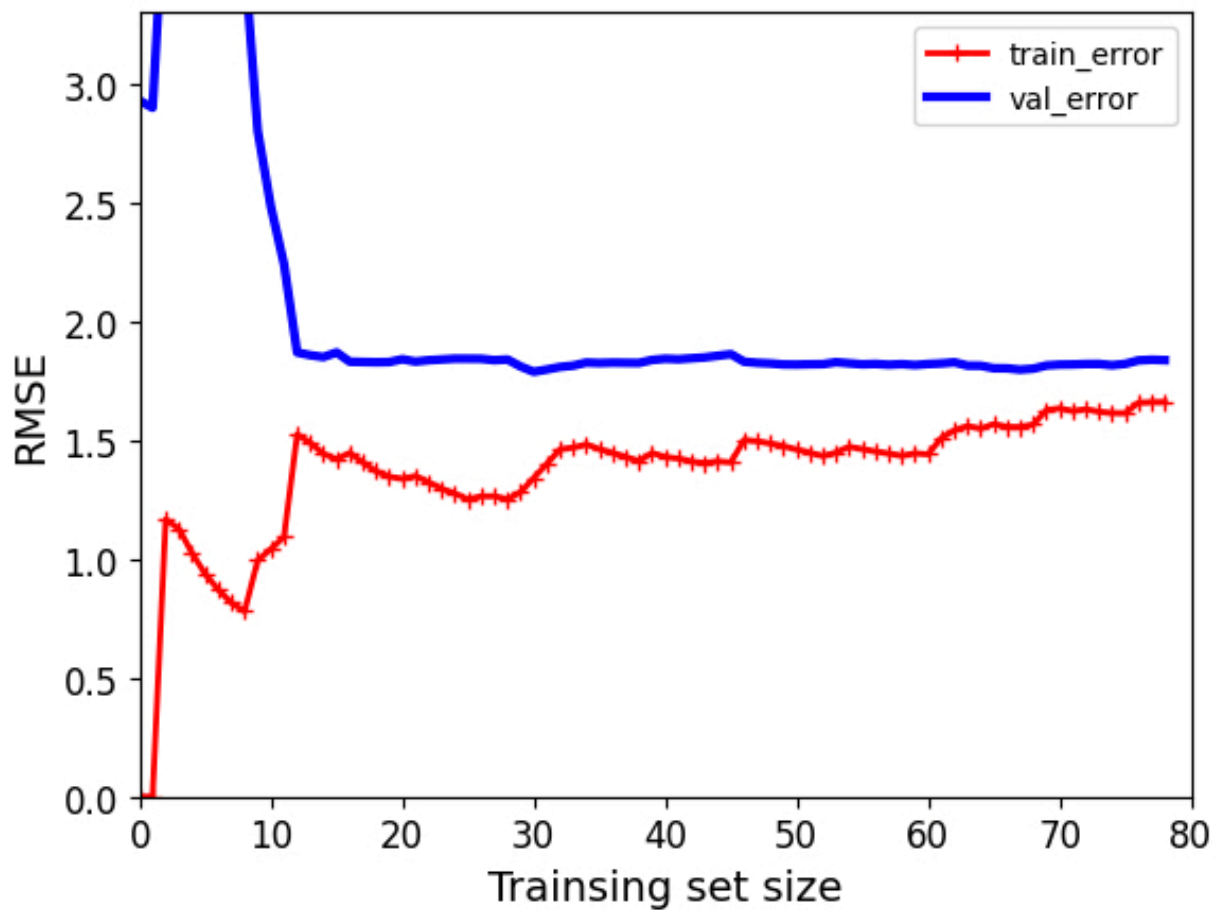
1. `X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=100)`: 使用 `train_test_split` 将数据分为训练集和验证集，其中20%的数据用作验证集。
2. `train_errors, val_errors = [], []`: 初始化两个空列表来存储训练误差和验证误差。

3. `for m in range(1, len(X_train)):`：这个循环逐步增加训练集的大小，从1个样本开始，直到使用整个训练集。
4. `model.fit(X_train[:m], y_train[:m])`：在当前训练集的大小下训练模型。
5. `y_train_predict = model.predict(X_train[:m])`：使用模型对训练集进行预测。
6. `y_val_predict = model.predict(X_val)`：使用模型对验证集进行预测。
7. `train_errors.append(mean_squared_error(y_train[:m], y_train_predict))`：计算当前训练集大小下的训练误差，并将其添加到 `train_errors` 列表中。
8. `val_errors.append(mean_squared_error(y_val, y_val_predict))`：计算验证集的误差，并将其添加到 `val_errors` 列表中。
9. `plt.plot(np.sqrt(train_errors), 'r-+', linewidth=2, label='train_error')`：绘制训练误差曲线，这里使用均方根误差（RMSE），因为它更适合衡量和解释。
10. `plt.plot(np.sqrt(val_errors), 'b-', linewidth=3, label='val_error')`：绘制验证误差曲线。
11. `plt.xlabel('Training set size')` 和 `plt.ylabel('RMSE')`：分别为x轴和y轴设置标签。
12. `plt.legend()`：显示图例。

通过这个函数，我们可以看到，随着训练样本数量的增加，**训练误差**可能会增加，因为模型需要拟合更多的数据点。同时，**验证误差**通常会降低，因为模型能够从更多数据中学习，并提高其对未知数据的预测能力。

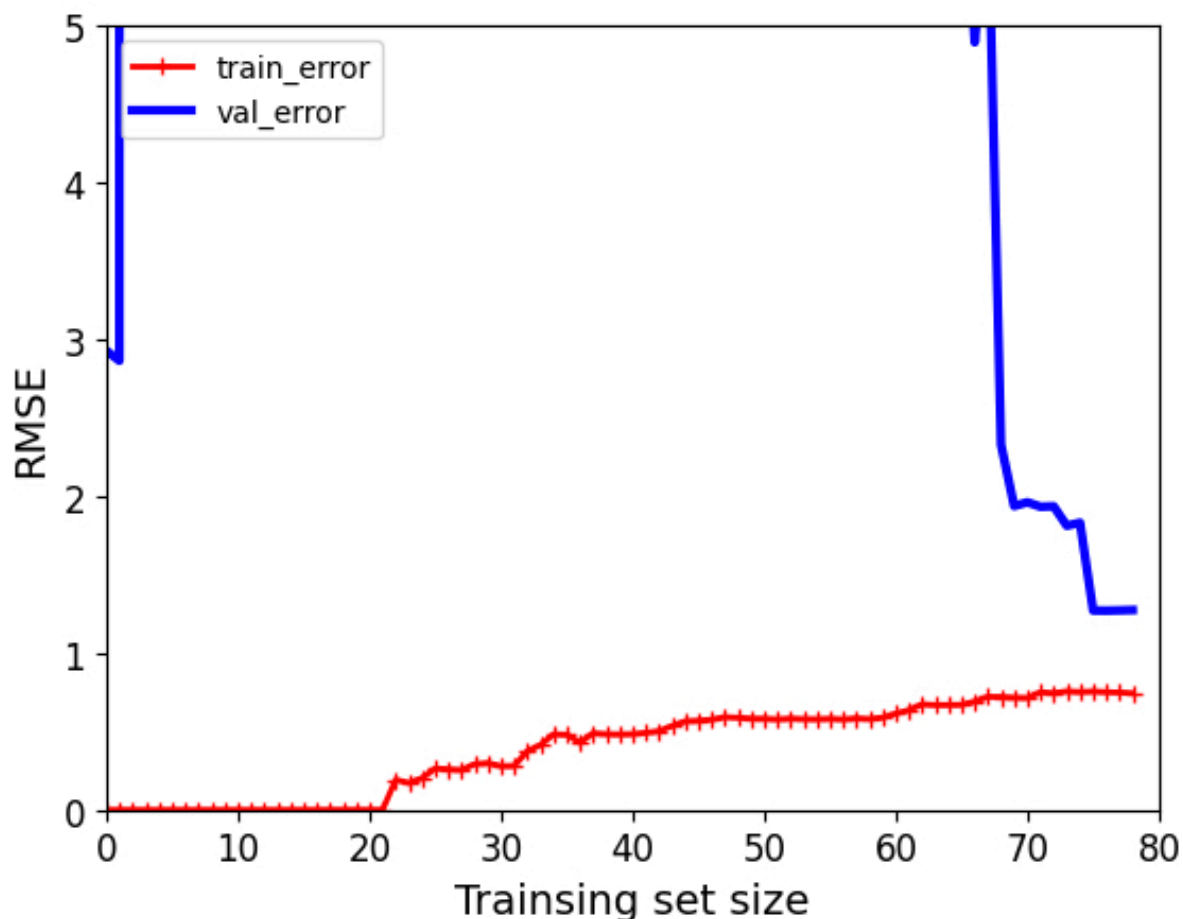
学习曲线是机器学习中一个重要的工具，用于判断模型是否存在欠拟合或过拟合。如果两条曲线最终收敛，并且误差相对较低，那么模型就有很好的泛化能力；如果训练误差与验证误差之间存在很大的差距，那么模型可能存在过拟合；如果两者都很高，则可能是欠拟合。

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg,X,y)
plt.axis([0,80,0,3.3])
plt.show()
```



## 2.3 多项式回归的过拟合风险

```
polynomial_reg = Pipeline([('poly_features',PolynomialFeatures(degree = 25,include_bias
= False)), ('lin_reg',LinearRegression())])
plot_learning_curves(polynomial_reg,X,y)
plt.axis([0,80,0,5])
plt.show()
```



从训练误差（红色）、验证误差（蓝色）、误差差异角度分析一下这幅图，并总结一下过拟合的标志是什么？

## 2.4 正则化

对权重参数进行惩罚，让权重参数尽可能平滑一些，介绍两种不同的方法来进行正则化惩罚

### 2.4.1 岭回归

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

式子表示的是正则化线性回归（也称为岭回归或L2正则化回归）的成本函数。

成本函数 $J(\theta)$ 由两部分组成：

1. **MSE( $\theta$ )**：这是平均平方误差，用于衡量模型预测值与实际值之间的差异。它是回归任务中常用的成本函数，定义为所有样本误差的平方和的平均值。
2. **正则化项**：  $\alpha/2 * \sum(\theta_i)^2$ ，它是模型参数 $\theta$ 的平方和乘以正则化系数 $\alpha$ 的一半。这个正则化项的目的是惩罚模型复杂度，防止过拟合。参数 $\theta$ 的每个元素都被平方并加总起来（从 $i=1$ 到 $n$ ， $n$ 是参数的数量），然后乘以 $\alpha/2$ 。正则化系数 $\alpha$ 控制了正则化项对成本函数的贡献大小。 $\alpha$ 的值越大，正则化的惩罚就越强，模型的参数 $\theta$ 就会被推向更小的值，从而得到一个更简单（可能是更平滑）的模型。这通常可以帮助提高模型在新数据上

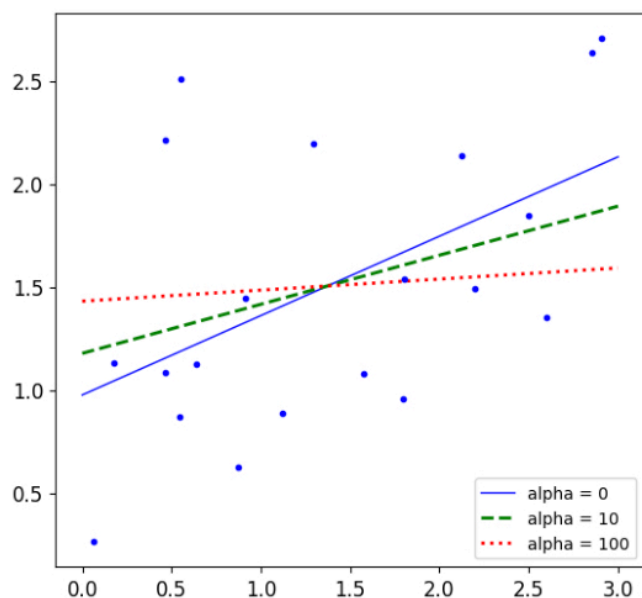
的泛化能力。

整个成本函数 $J(\theta)$ 是优化问题的目标，通过找到最小化 $J(\theta)$ 的参数 $\theta$ 来训练模型。通过这种方式，在拟合数据的同时保持模型的简单性，可以避免模型变得过于复杂，并且对训练数据中的噪声或不重要的特征过度敏感。这样的模型通常对未知数据的预测能力更强。

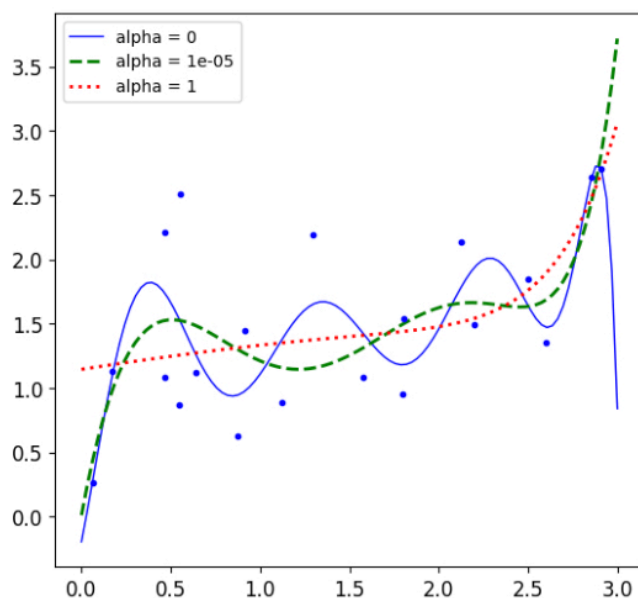
```
from sklearn.linear_model import Ridge # 岭回归
np.random.seed(42)
m = 20
X = 3*np.random.rand(m,1)
y = 0.5 * X +np.random.randn(m,1)/1.5 +1
X_new = np.linspace(0,3,100).reshape(100,1)

def plot_model(model_calss,polynomial,alphas,**model_kargs):
    for alpha,style in zip(alphas,('b-','g--','r:')):
        model = model_calss(alpha,**model_kargs)
        if polynomial:
            model = Pipeline([('poly_features',PolynomialFeatures(degree
=10,include_bias = False)),
                ('StandardScaler',StandardScaler()),
                ('lin_reg',model)])
        model.fit(X,y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new,y_new_regul,style,linewidth = lw,label = 'alpha =
{}'.format(alpha))
        plt.plot(X,y,'b.',linewidth =3)
        plt.legend()

plt.figure(figsize=(14,6))
plt.subplot(121)
plot_model(Ridge,polynomial=False,alphas = (0,10,100))
plt.subplot(122)
plot_model(Ridge,polynomial=True,alphas = (0,10**-5,1))
plt.show()
```



惩罚力度越大，alpha值越大的时候，得到的决策方程越平稳。



这段代码使用了Scikit-Learn的 `Ridge` 回归模型来展示不同正则化强度下模型的表现。同时，它比较了多项式扩展（并伴随标准化处理）与原始特征下的模型效果。这段代码的流程如下：

1. 首先，设置随机种子确保结果的可复现性。
2. 生成一个简单的线性数据集，其中 `x` 是在 `[0, 3]` 区间内均匀分布的随机数，`y` 是 `x` 的线性函数，带有一些正态分布的噪声和一个常数项。
3. `x_new` 被创建为新的测试数据集，用于模型预测。
4. 定义了 `plot_model` 函数，它接受几个参数：
  - `model_class`：将要使用的模型类，这里是 `Ridge` 回归。
  - `polynomial`：一个布尔值，指示是否应用多项式特征扩展。
  - `alphas`：一个包含不同正则化强度参数 `alpha` 的元组。
  - `**model_kargs`：任何额外的关键字参数都将传递给模型构造函数。
5. 在 `plot_model` 函数内部，它根据 `polynomial` 参数的值决定是否将原始特征进行多项式扩展和标准化处理。
6. 对于每一个 `alpha` 值和相应的样式（蓝线、绿虚线、红点线），函数创建了一个模型，训练它，并使用 `x_new` 数据集进行预测。
7. 设置线宽：如果 `alpha` 为 0（即没有正则化），线宽设为 1；如果 `alpha` 大于 0，则线宽设为 2，以便在图上更清楚地区分不同的正则化强度。
8. 使用 `plt.plot` 函数绘制预测曲线，并用 `plt.legend` 添加图例。
9. `plt.figure(figsize=(14,6))` 创建了一个宽 14 英寸、高 6 英寸的图形窗口。
10. `plt.subplot(121)` 和 `plt.subplot(122)` 创建了一个 1 行 2 列的子图布局，并在第一个和第二个子图上分别调用 `plot_model` 函数。
  - 第一个子图（`subplot(121)`）显示的是原始特征（非多项式）下不同正则化强度的模型。
  - 第二个子图（`subplot(122)`）显示的是使用多项式特征扩展（10 次多项式）的情况下不同正则化强度的模型。

11. `plt.show()` 展示了最终的图形。

综上，这段代码的目的是为了展示正则化（特别是岭回归）在没有多项式扩展和有多项式扩展情况下对模型预测效果的影响。正则化强度由alpha参数控制，当alpha为0时，岭回归就退化为普通的线性回归。通过这个示例，可以直观地看到随着alpha增加，模型的复杂性如何受到限制，从而减轻过拟合的风险。同时，也展示了多项式特征扩展如何让线性模型能够捕捉更复杂的数据关系。

## 2.4.2 Lasso回归

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

这个正则化的式子定义了Lasso回归（Least Absolute Shrinkage and Selection Operator Regression）的成本函数。

与岭回归（L2正则化）类似，Lasso回归在成本函数中也包含两项：

1. **MSE(θ)**：这部分是模型的平均平方误差，用来衡量模型预测与实际值之间的差距。这是回归任务中最常见的损失函数。
2. **正则化项**： $\alpha * \sum |\theta_i|$ ，这部分是模型参数的绝对值之和，乘以正则化系数α。这是Lasso回归特有的正则化项，它对模型参数进行L1正则化。正则化系数α控制着正则化的强度，正如文本所述，α的值较大时，它会增加模型参数的惩罚强度。

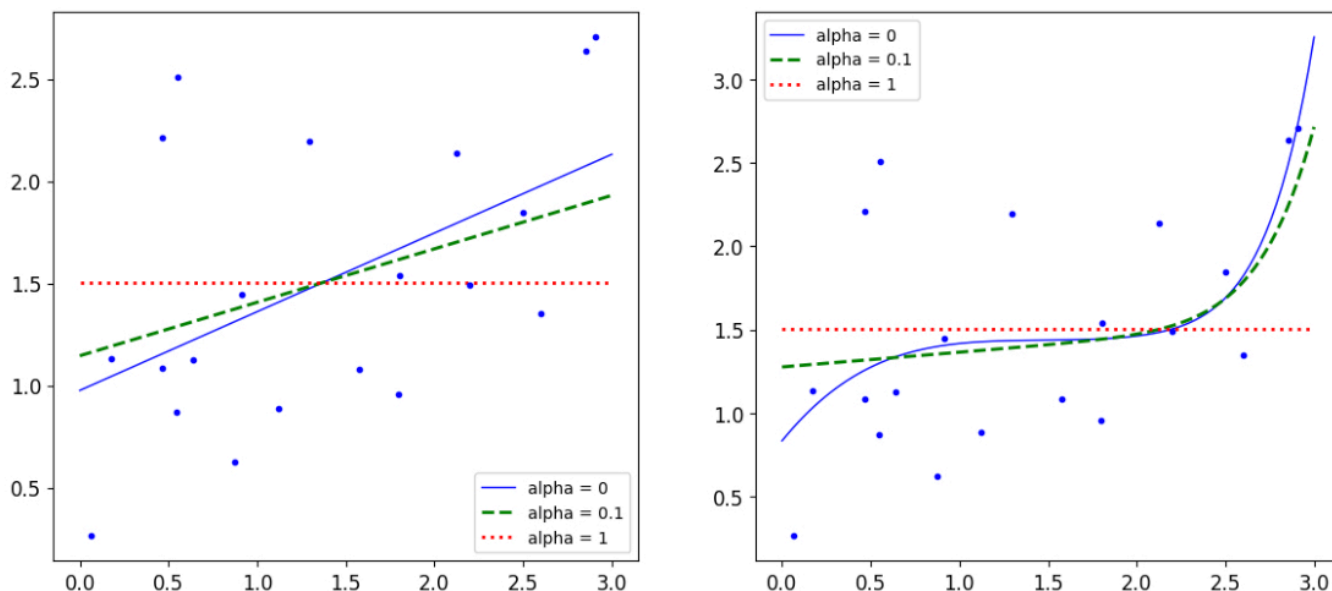
L1正则化的特点是它倾向于将某些参数值完全压缩到零，这导致了模型的稀疏性。在特征选择的上下文中，这意味着Lasso回归可以帮助识别出那些最重要的特征，因为它会自动将不重要的特征的系数设置为零。这使得Lasso回归不仅有助于避免过拟合，还可以用作特征选择的工具，从而提高模型的可解释性。

整个成本函数  $J(\theta)$  在训练过程中被最小化，以训练模型。在选择合适的α值时需要小心，因为一个过大的α值可能会导致模型过于简单（欠拟合），而一个太小的α值则可能让模型的正则化效果不明显（过拟合）。

```
from sklearn.linear_model import Lasso

plt.figure(figsize=(14,6))
plt.subplot(121)
plot_model(Lasso,polynomial=False,alphas = (0,0.1,1))
plt.subplot(122)
plot_model(Lasso,polynomial=True,alphas = (0,10**-1,1))
plt.show()
```





这段代码用来展示Lasso回归模型在不同正则化参数 `alpha` 下的表现。Lasso回归是线性回归的一种，它包含了对系数的L1正则化，这种正则化可以生成一个稀疏模型，即许多系数可以被压缩至零。

具体步骤包括：

1. `from sklearn.linear_model import Lasso`：从 `sklearn.linear_model` 中导入 `Lasso` 类。
2. `plt.figure(figsize=(14,6))`：创建一个14x6英寸的新图形。
3. 在两个子图上分别调用自定义的 `plot_model` 函数。这个函数绘制了给定模型在一系列 `alpha` 值下的学习曲线。
4. `plt.subplot(121)`：这指定了第一个子图的位置，即1行2列网格的第1列。
5. `plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1))`：这个调用用于在第一个子图上绘制没有多项式特征变换的Lasso回归模型的学习曲线。`alphas` 参数设定了不同的正则化强度，其中`alpha`为0相当于普通的线性回归。
6. `plt.subplot(122)`：指定第二个子图的位置。
7. `plot_model(Lasso, polynomial=True, alphas=(0, 10**-1, 1))`：在第二个子图上绘制带有10次多项式特征变换的Lasso回归模型的学习曲线。这里 `alphas` 也指定了不同的正则化强度，但请注意，使用多项式特征变换时，可能需要更小的`alpha`值来平衡模型的复杂性。
8. `plt.show()`：显示图形。这会绘制出所有的 `plot` 命令结果。

通过这段代码，我们可以观察到，正则化强度的增加（即更大的 `alpha` 值）如何影响模型的学习曲线。特别是，我们可以看到正则化对于过拟合（在模型包含多项式特征时尤其重要）的影响，以及对模型泛化能力的潜在提升。这是因为Lasso倾向于减少不重要的特征的权重，从而导致一个更简单、更泛化的模型。

## 3. 动手实践

- 1、自行完成1.2.3小节中MiniBatch梯度下降方法中的部分代码，得到参数 $\theta$ 路径，并能够在1.2.4小节中进行路径对比。自行思考一下1.2.4小节中的问题，batch数量应如何选择？
- 2、自行完成2.1小节中多项式度数为1, 2, 100的设置，绘图观察其结果，并加以分析。

3、分析2.3小节，从训练误差（红色）、验证误差（蓝色）、误差差异角度分析一下图，并总结一下过拟合的标志是什么？