哈尔滨工业大学计算学部

实验报告

课程名称：　机器学习

课程类型：　选修

实验题目：　多项式拟合正弦曲线

学号：**1190201215**
姓名：　冯开来

一、实验目的

掌握最小二乘法求解（无惩罚项的损失函数）、掌握加惩罚项（2 范数）的损失函数优化、梯度下降法、共轭梯度法、理解过拟合、克服过拟合的方法(如加惩罚项、增加样本)

二、实验要求及实验环境

2.1 实验要求

1. 生成数据，加入噪声；
2. 用高阶多项式函数拟合曲线；
3. 用解析解求解两种 loss 的最优解（无正则项和有正则项）
4. 优化方法求解最优解（梯度下降，共轭梯度）；
5. 用你得到的实验数据，解释过拟合。
6. 用不同数据量，不同超参数，不同的多项式阶数，比较实验效果。
7. 语言不限，可以用 matlab，python。求解解析解时可以利用现成的矩阵求逆。梯度下降，共轭梯度要求自己求梯度，迭代优化自己写。不许用现成的平台，例如 pytorch，tensorflow 的自动微分工具。

2.2 实验环境

Windows10; python3.9;PyCharm 2021.2.2

三、设计思想（本程序中的用到的主要算法及数据结构）

在线性回归中，多项式拟合就是用类似泰勒级数，使用多项式来拟合正弦函数$sin(2\pi x)$ 。可以理解为

$$f_i(x_i,w) = w_0x_i^0 + w_1x_i^1 + w_2x_i^2 + \cdots w_mx_i^m$$

当然，用矩阵来表示就是：

$$f_i = \boldsymbol{w}^T\boldsymbol{x_i}$$

其中：

$$\boldsymbol{x_i} = \begin{bmatrix} x_i^0 \\ \vdots \\ x_i^m \end{bmatrix} \qquad \boldsymbol{w} = \begin{bmatrix} w_0 \\ \vdots \\ w_m \end{bmatrix}$$

如果样本个数为 10 的话：

$$\boldsymbol{F}(x_i,w) = \boldsymbol{Xw}$$

其中：

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x_1^T} \\ \vdots \\ \boldsymbol{x_{10}^T} \end{bmatrix} \qquad \boldsymbol{F} = \begin{bmatrix} f_1 \\ \vdots \\ f_{10} \end{bmatrix}$$

这里的 F 是我们预测的样本，那如何判断我们预测的模型（参数 w）是比较好的呢？这就要和真实的样本做比较，假设真实的样本为：

$$\boldsymbol{Y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{10} \end{bmatrix}$$

那么，我们就可以用欧式距离来衡量我们预测的样本和真实样本之间的

误差。可以得到下面的式子：

$$E_{(w)} = \sum_{i=1}^{10} \{\boldsymbol{F}(x_i, \boldsymbol{w}) - y_i\}^2$$

最后，为了得到最好的模型（参数 w），就要找到使 E 最小的 w。

使 E 最小的方法有很多，实验中，用到了求导，最小梯度法，共轭梯度法，以及为了防止过拟合我们会加入正则项，但最终，我们的目标只有一个，就是找到 w，使 E 最小。

3.1 生成真实的样本数据

因为题目中要求拟合正弦函数，样本数量为 10，定义域范围再[0,1]。所以，我选择在[0,1]之间等步长取 10 个数，得到正弦函数的值之后，加入高斯噪声，这十个样本就是我们最开始最真实的样本，即训练样本。

```
# 生成数据
def generate_data(m, size=10, var=0.25, mean = 0, begin=0, end=1):
    x = np.linspace(begin, end, size) #在[begin, end]生成size大小的数组
    Y = np.sin(2 * np.pi * x) #生成样本y. 未加入高斯噪声
    for i in range(size):
        Y[i] += np.random.normal(mean, var) #加入高斯噪声
    Y = Y.T # Y需要转置
    X = np.zeros((m+1, size)) #初始化X矩阵
    plt.plot(x, Y, 'bo') #绘制测试样本
    for i in range(m+1):
        X[i] = x ** i #得到每一行的X
    X = X.T #X需要转置
    # print(x)
    # print(X)
    # print(y)

    return X, Y
```

3.2 求解 w

3.2.1 最小二乘法（无正则项）

$$E(w) = \sum_{i=1}^{10} \{\boldsymbol{F}(x_i, \boldsymbol{w}) - y_i\}^2$$

$$\begin{aligned} E &= (Xw - Y)^2 = (Xw - Y)^T(Xw - Y) \\ &= 2(w^T X^T - Y^T)(Xw - Y) \\ &= 2(w^T X^T Xw - w^T X^T Y - Y^T Xw + X^T Y) \\ &= 2(w^T X^T Xw - 2w^T X^T Y + X^T Y) \end{aligned}$$

令

$$\frac{\partial E}{\partial w} = X^T Xw - X^T Y = 0$$

即

$$\boldsymbol{w} = (X^T X)^{-1} X^T Y$$

```python
#最小二乘法，不带正则项
def loss_1(X, Y):
    w = np.linalg.inv(np.dot(X.T, X)).dot(X.T).dot(Y)
    return w
```

3.2.2 最小二乘法（有正则项）

$$E(w) = \frac{1}{2}\sum_{i=1}^{10}\{\boldsymbol{F}(x_i, \boldsymbol{w}) - y_i\}^2 + \frac{\lambda}{2}||w||^2$$

$$E = (w^T X^T X w - 2w^T X^T Y + X^T Y) + \frac{\lambda}{2} w^T w$$

令

$$\frac{\partial E}{\partial w} = X^T X w - X^T Y + \lambda w = 0$$

即

$$\boldsymbol{w} = (X^T X + \lambda)^{-1} X^T Y$$

```python
#最小二乘法，有正则项
def loss_2(X, Y, lamda):
    w = np.linalg.inv(np.dot(X.T, X) + lamda).dot(X.T).dot(Y)
    return w
```

3.2.3 最小梯度法

首先得到梯度：

$$gradient\_w = \frac{\partial E}{\partial w} = X^T X w - X^T Y + \lambda w$$

设步长（学习率）为α，对 w 梯度下降，不停迭代更新 w 和 gradient_w，直到梯度收敛于一个很小的数 accept_gradient：

$$w = w - \alpha\frac{\partial E}{\partial w}$$

```python
#梯度下降法
def gradient_descent(X, Y, alpha, accept_gradient = 0.01):
    w = np.zeros(m+1).T
    gradient_w = np.dot(X.T, X).dot(w) - np.dot(X.T, Y) + lamda * w
    while np.linalg.norm(gradient_w) >= accept_gradient:
        w = w - alpha * gradient_w
        gradient_w = np.dot(X.T, X).dot(w) - np.dot(X.T, Y) + lamda * w
    return w
```

3.2.4 共轭梯度法

共轭梯度法有点难理解，我也看了非常多的文章才稍稍感觉有带入门。所以报告里我打算类比这梯度下降法来解释共轭梯度法。在梯度下降法中，核心迭代公式是：

$$w = w - \alpha \frac{\partial E}{\partial w}$$

而在共轭梯度法中，我们要做两点改变，一是步长二是方向。因为梯度下降法是沿着一个方向不停下降，而共轭梯度是在众多方向中找到一个最快的下降方法。初始点的下降方向仍然是负梯度方，但后面的迭代方向是该点的负梯度方向和前一次的迭代方向（共轭方向）行程的凸锥中的一个方向（大概就是两个方向的线性组合），这样可以避免梯度下降的“锯齿”现象。

解释一下共轭方向，假设 d_0 和 d_1 关于 A 共轭，那么满足${d_0}^T A d_1 = 0$。同理，n 个方向共轭的话，即两两满足上式。

那么我们同样设步长为α，方向向量为 d，替代原式中的梯度，得到：

$$w = w - \alpha d$$

如何求由 d_k 得到 d_{k+1}？因为 d_k 和 d_{k+1} 既满足共轭，也满足和负梯度方向是线性关系，所以我们可以得到两个等式：

$$d_{k+1} = -gradient\_w + \beta d_k$$

$${d_{k+1}}^T A d_k = 0$$

所以可以解出：

$$\alpha = -\frac{(gradient\_w)^T d_k}{d_k^T A d_k}$$

$$\beta = \frac{{d_k}^T A (gradient\_w)}{d_k^T A d_k}$$

这样带入$w = w - \alpha d$完成迭代，迭代次数也很有意思，因为每一步都是朝着最陡的方向下降，所以迭代次数就是 w 的维数。实事证明，迭代这么点次数，确实收敛。

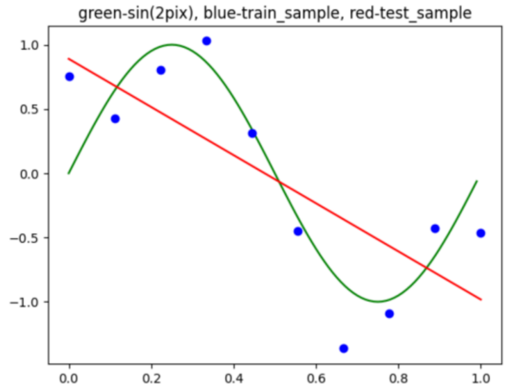
```
#共轭梯度法
def conjugate_gradient(X, Y, m, accept_gradient):
    cnt = 0 #限制迭代次数
    w = np.zeros(m+1).T #初始化需要求解的w
    A = np.dot(X.T, X) #方便计算，为原式二次型的正定矩阵
    gradient_w = np.dot(A, w) - np.dot(X.T, Y) + lamuda * w #计算第一次梯度方向
    d = - gradient_w #第一次迭代方向为负梯度方向
    alpha = np.dot(A, w) - np.dot(X.T, Y) #初始化步长
    while cnt <= m: #控制迭代次数为w的维数
        alpha = - np.dot(gradient_w.T, d) / np.dot(d.T, A).dot(d) #更新步长，使损失函数达到最小的步长
        w = w + alpha * d #更新w矩阵，沿着共轭方向下降
        gradient_w = np.dot(A, w) - np.dot(X.T, Y) + lamuda * w #更新梯度，计算共轭方向和步长需要
        beta = np.dot(d.T, A).dot(gradient_w) / np.dot(d.T, A).dot(d) #计算共轭方向需要的线性关系系数
        d = -gradient_w + np.dot(beta, d) #得到共轭方向
        cnt = cnt + 1
    print(cnt)
    return w
```

四、实验结果与分析

4.1 最小二乘法（不带正则项）

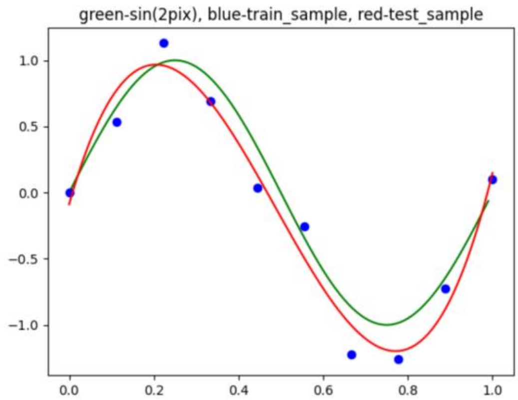
固定训练集大小为 10，在不同多项式阶数下的拟合结果：

1 阶：



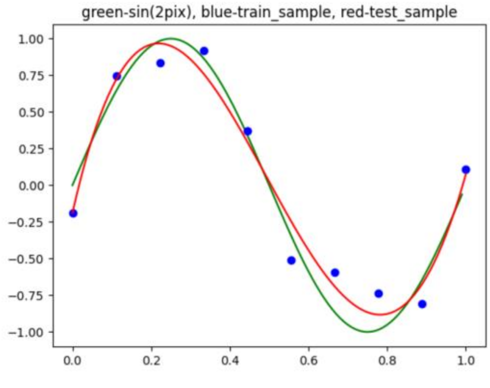
w=[0.5999383 -1.12799647]

3 阶：



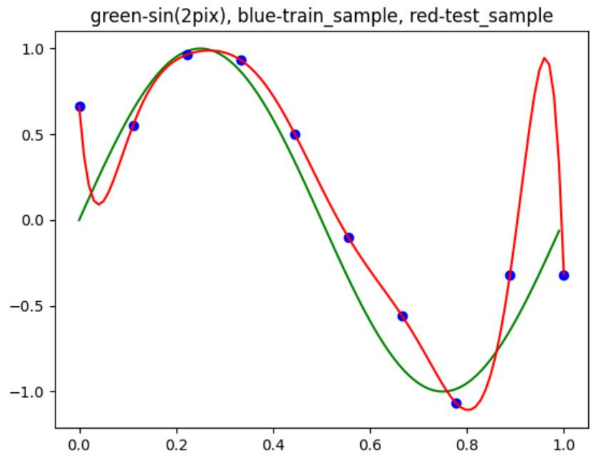
w=[-0.08893756 12.26829444 -36.2780962   24.52429352]

5 阶：



w=[-0.1886 11.8165 -35.7776 27.9358 -4.0469 0.3376]

9 阶：

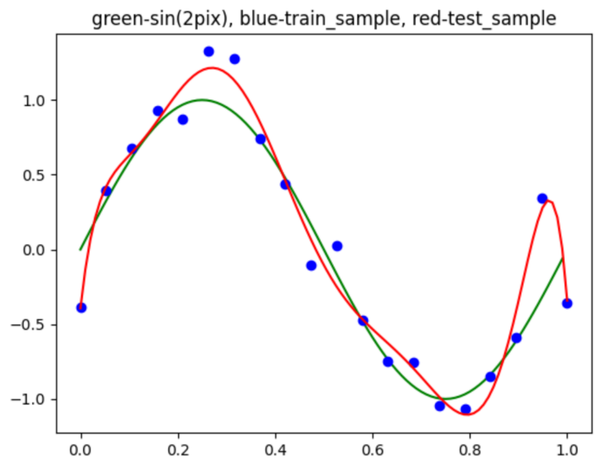


w = [6.64185348e-01 -3.53460497e+01 7.29499487e+02 -6.14024454e+03
2.85778400e+04 -8.00733535e+04 1.37250334e+05 -1.40347000e+05
7.84356376e+04 -1.83983484e+04]

显然，阶数越小，学习能力越差，在一阶的时候是明显的欠拟合状态，当阶数在 3 的时候，拟合效果已经很好了，但是在阶数为 9 的时候，出现了个别值的很大的波动，这就是过拟合的情况，即学习能力太强了。解决过拟合的方法一是可以增加惩罚项，二是可以增加训练样本。

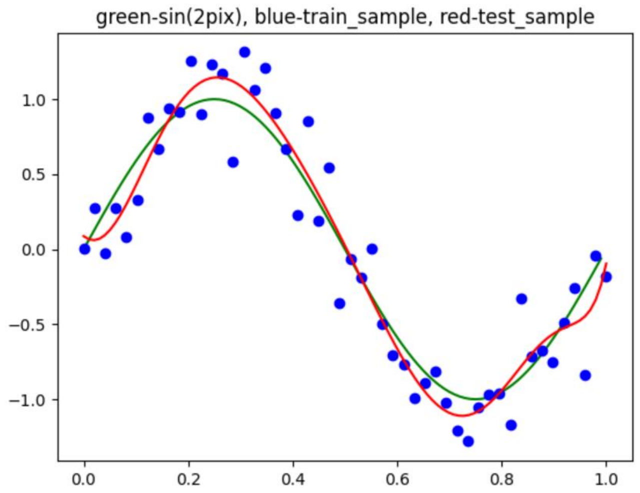
固定阶数大小为 9 阶，在不同训练集下的拟合结果：

20 个训练样本：



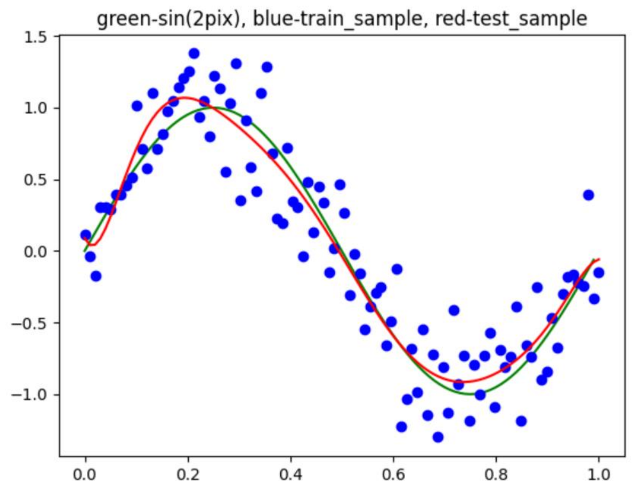
w = [-3.93909987e-01 2.93118250e+01 -3.85260400e+02 2.82611336e+03
-1.07700616e+04 2.15808167e+04 -2.19361059e+04 8.38522428e+03
2.11361082e+03 -1.84359723e+03]

50 个训练样本：



w = [8.67757399e-02 -2.88492763e+00 7.85693391e+01 6.38593634e+00
   -2.61775671e+03 1.19481372e+04 -2.54372297e+04 2.92879297e+04
   -1.75517503e+04 4.28841636e+03]

100 个训练样本：

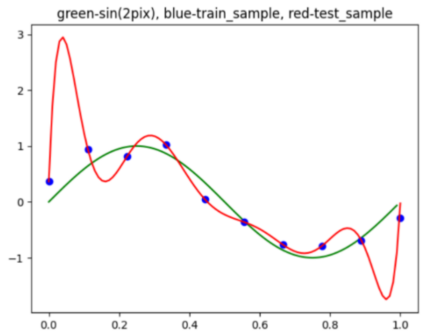


w = [9.03212087e-02 -8.15635721e+00 3.45893692e+02 -3.07311622e+03
   1.33830802e+04 -3.37620306e+04 5.13495798e+04 -4.63630416e+04
   2.28904350e+04 -4.76279333e+03]

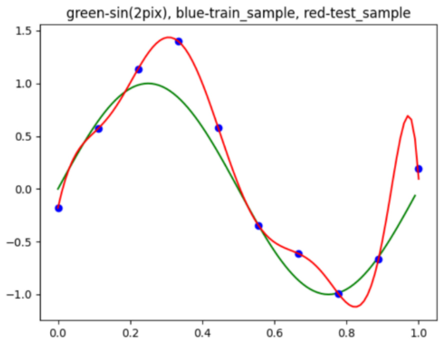
可以看到，即使阶数是 9，在训练样本为 10 的时候出现了明显的过拟合，但在训练样本为 20 的时候，过拟合情况小了很多，当训练样本到 50 的时候，基本没有过拟合了，并且训练的曲线已经开始向正弦函数靠近，当训练样本达到 100 的时候，训练出的曲线几乎已经贴近正弦函数。

4.2 最小二乘法（带正则项）

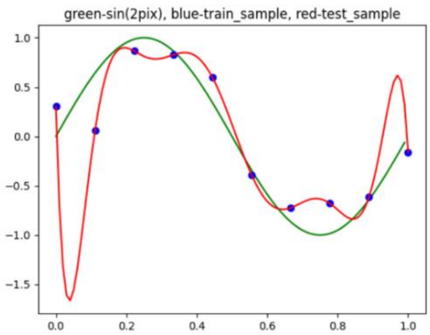
固定训练样本大小为 10，阶数为 9，不断调整 λ 找到最优解：



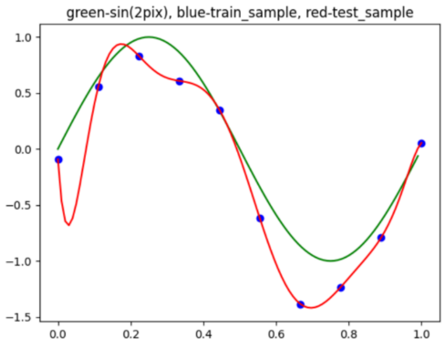
λ = 10



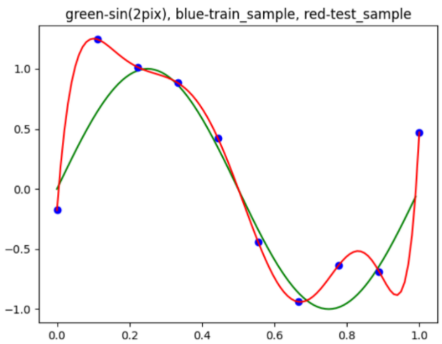
λ = 1



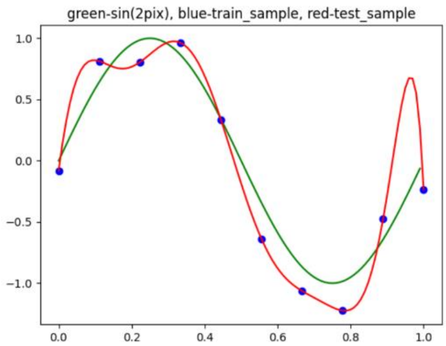
λ = 1e−13



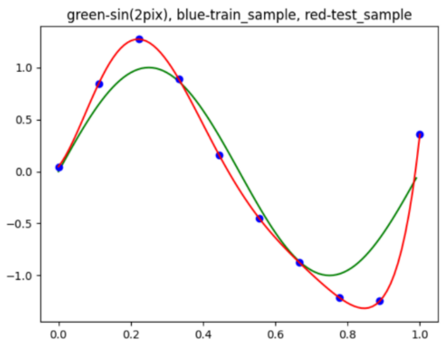
λ = 1e−18



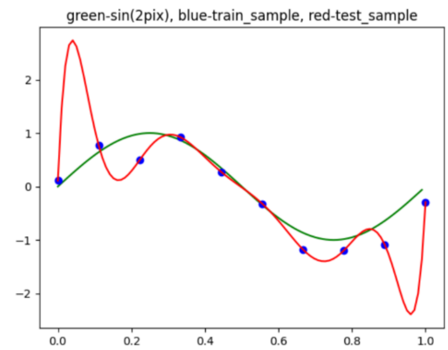
λ =1e-20



λ =1e−26



λ =1e−35



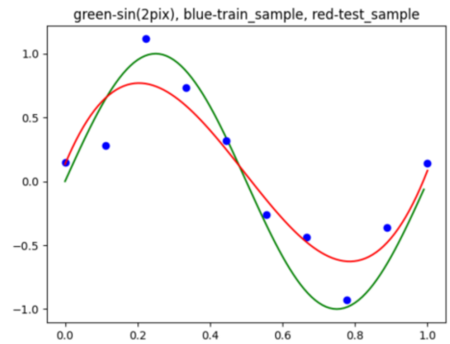
λ =1e−40

可以看到，根据上面不加正则项相比，加入正则项后，拟合情况随着λ的取值改变而改变，在λ>1e-18 的时候，仍然会有过拟合的现象，说明惩罚比重大时没有多大改变，在 1e-18<λ<1e-35 时，明显可以看到过拟合的程度有所减小，甚至在 1e-35 时出现了几乎完美拟合的情况，说明在λ适当的情况下，惩罚项可以有效解决过拟合问题，在λ更小的时候，错误率又开始上升，又可以看到过拟合的情况。但是在实验过程中，仍能看到拟合过程具有很高的随机性。

4.3 梯度下降法

固定训练样本大小为 10，步长为 0.01，改变阶数：

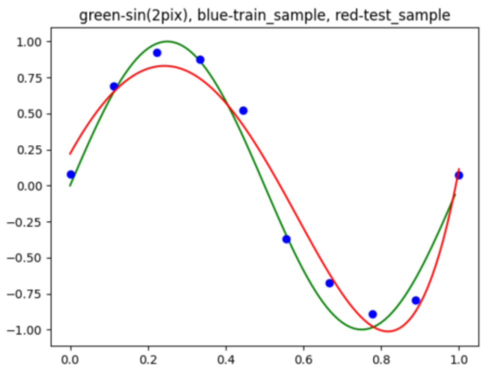
3 阶：



迭代次数：103543

w = [0.07423458   7.19951029 -23.05217788 15.84586233]

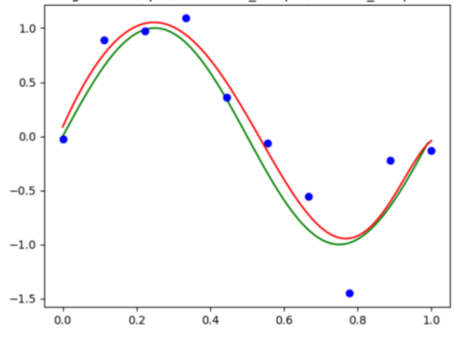
5 阶：



迭代次数：28224

w = [0.22065 4.86346 -8.68037 -4.88889 1.50850 7.09058]

9 阶：



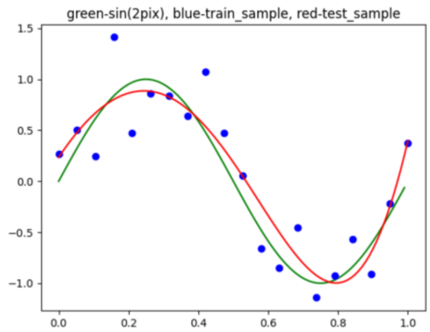
迭代次数：80540

w = [-0.35739164 8.97873881 -16.16099598 -5.01851713 3.76708643 6.86214859 5.95998219 2.87250481 -1.15944382 -5.39965266]

随着阶数的增加，迭代次数有明显的减少，但是在高阶情况下，迭代次数减少没有那么明显。但是高阶过拟合程度没有那么明显。

固定阶数 9 阶，改变训练样本大小：

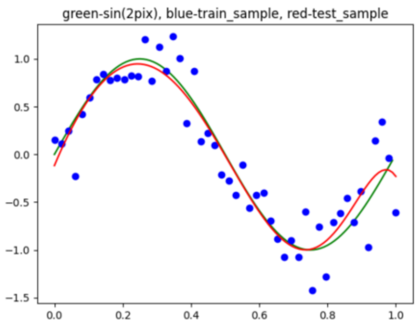
20 训练样本：



迭代次数 49116

w = [0.23394127 5.13551944 -8.98819014 -5.02848854 0.77439854 3.99859885 4.42743 2.86139512 0.13015341 -3.14643747]

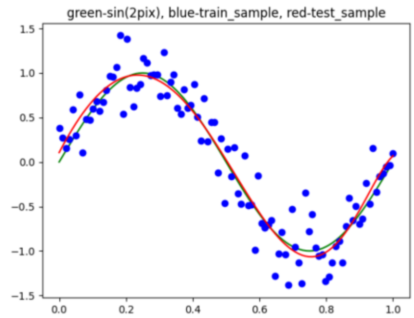
50 训练样本：



迭代次数：41435

w = [-0.11715363 8.5581355 -15.8744001 -7.03987737 3.94511656 9.11289342 8.70721807 4.4667646 -2.0872143 -9.90144905]

100 训练样本：



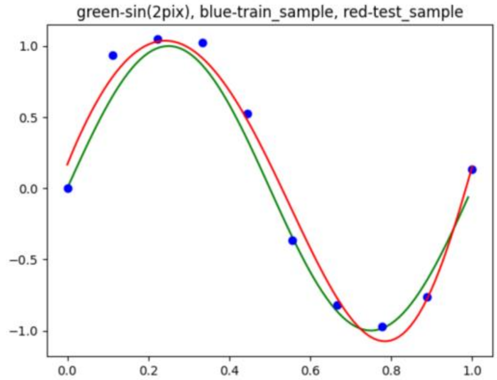
迭代次数：24914

w = [0.10573297   7.25496618 -13.93105818 -6.21446843   3.14578367
   7.45442113   7.09983091   3.69978703 -1.36912497 -7.17642126]

随着训练样本次数的增加，迭代次数有明显的减少。和损失函数一样，训练样本增加后，拟合的曲线越贴近原来的曲线。

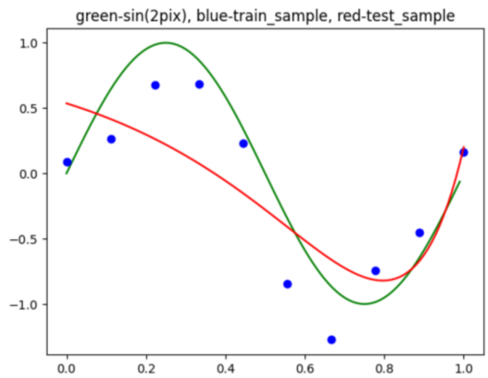
固定训练样本大小 10、阶数为 9，改变可接受梯度的阈值

Accept_gradient = 0.01



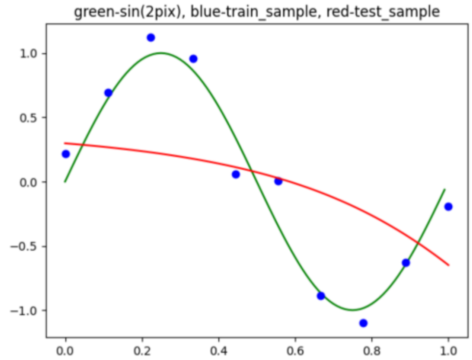
迭代次数：77676

Accept gradient = 0.1



迭代次数：671

Accept gradient = 1



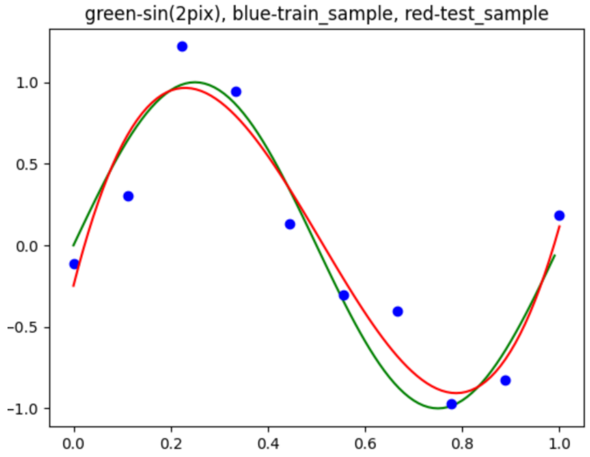
迭代次数：38

明显可以看出当accept_gradient增加的时候，迭代次数减少，随之而来的代价就是拟合情况越糟糕。

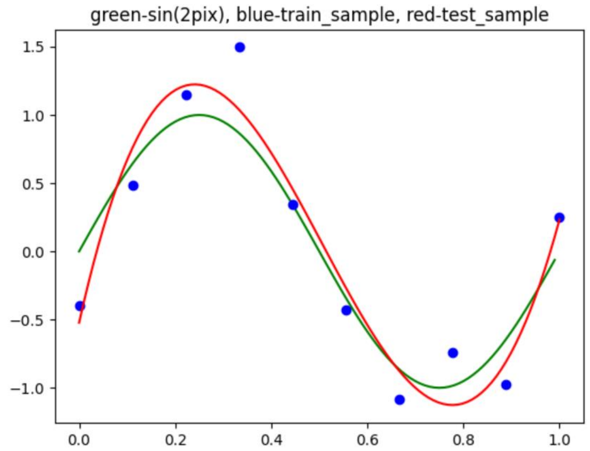
4.4 共轭梯度法

固定训练样本大小 10，改变阶数（迭代次数）

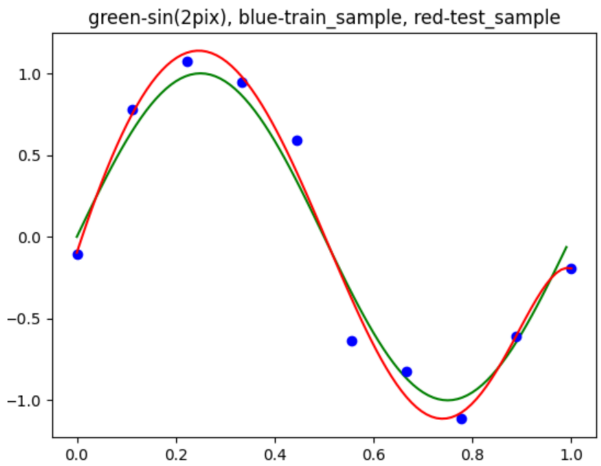
3 阶，迭代次数 4：



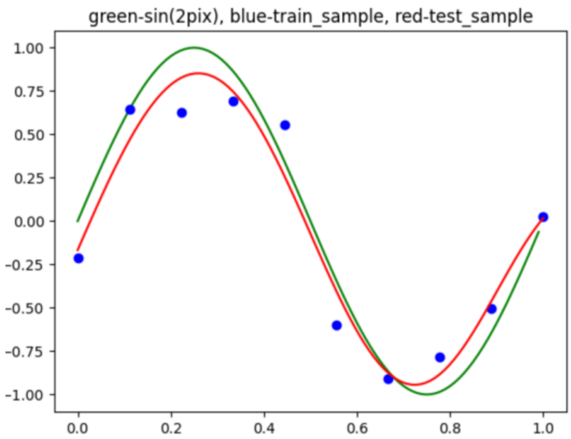
5 阶，迭代次数 6：



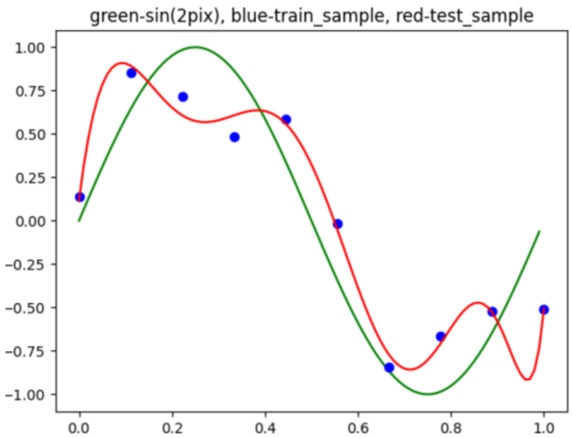
7 阶，迭代次数 8：



9 阶，迭代次数 10：



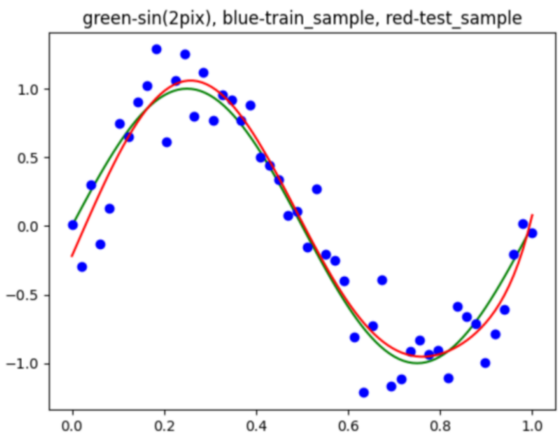
11 阶，迭代次数 12：



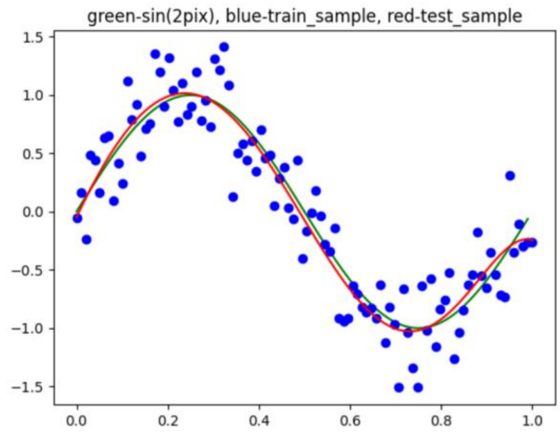
在低阶情况下，都没有出现过拟合，9 阶也没有出现过拟合（最小二乘法会出现），但是，阶数增加，还是会出现过拟合情况的。不过最显著的特点还是迭代次数非常小的情况下，仍能拟合的不错。

固定阶数 9（迭代次数 10），改变训练样本大小：

训练样本大小为 50：



训练样本大小为 100：



还是那句话，训练样本越多，拟合效果越好。

五、结论

1. 上述几种方法，除了最小二乘法是直接使用公式取得之外，另外几种方法都是使用迭代法进行求解。
2. 梯度下降法主要沿着负梯度方向进行更新，只引入了一阶导数。
3. 自己也看了牛顿法，在这里也写点感想，牛顿法则将一阶导数和二阶导数相结合进行参数的迭代更新，但是其二阶导数矩阵涉及到海瑟矩阵求逆（海瑟矩阵又是一个令人头疼的东西，微积分学的东西很多都忘了，还是一个个查的慢慢捡起来），复杂度较高，因此基于牛顿法之上，出现了拟牛顿法，其主要思想是构造一个新的矩阵来近似替代海瑟矩阵的逆，这样可以避免牛顿法中的复杂的海瑟矩阵求逆操作。
4. 本质上来说，牛顿法是二阶收敛，而梯度下降是一阶收敛，所以牛顿法收敛更快，但是每次迭代的时间，牛顿法比梯度下降时间长。牛顿法就是用一个二次曲面去拟合你当前所处位置的局部曲面，而梯度下降法使用一个平面去拟合当前的局部曲面，所有牛顿法的下降路径会更简单，更符合最优路径。
5. 共轭梯度法强就强在不只是二阶收敛，而是基于梯度下降中的负梯度方向之外，引入了共轭向量，可以使收敛更快。
6. 牛顿法是二阶优化方法，拟牛顿法和共轭梯度法一般叫做 1.5 阶优化方法，梯度下降法及其变形则是一阶优化方法。
7. 在一般情况下，阶数 m 越大，其模型学习能力越强，但是 m 过大的时候会导致过拟合情况，而且选用模型（求解参数 w 的方法）的不同，出现过拟合的阶数也不同。这隐式的说明求解方法也有强弱之分，强的模型可以很好的适应高阶。
8. 加入惩罚项确实可以一定程度抑制过拟合，但我还是觉得这存在一定随机性，或者说，在这个正弦函数模型中，惩罚项的作用可能只是停留在理论层面，实际上可能没有多大作用

六、参考文献

无

七、附录：源代码（带注释）

```
'''
多项式拟合
'''
import numpy as np
import matplotlib.pyplot as plt

#  生成数据
def generate_data(m, size=10, var=0.25, mean = 0, begin=0, end=1):
    x = np.linspace(begin, end, size) #在[begin, end]生成 size 大小的数组
    Y = np.sin(2 * np.pi * x) #生成样本 y，未加入高斯噪声
    for i in range(size):
        Y[i] += np.random.normal(mean, var) #加入高斯噪声
    Y = Y.T # Y 需要转置
    X = np.zeros((m+1, size)) #初始化 X 矩阵
    plt.plot(x, Y, 'bo') #绘制测试样本
    for i in range(m+1):
        X[i] = x ** i #得到每一行的 X
    X = X.T #X 需要转置
    # print(x)
    # print(X)
    # print(y)

    return X, Y

#最小二乘法，不带正则项
def loss_1(X, Y):
    w = np.linalg.inv(np.dot(X.T, X)).dot(X.T).dot(Y)
    return w

#最小二乘法，有正则项
def loss_2(X, Y, lamuda):
    w = np.linalg.inv(np.dot(X.T, X) + lamuda).dot(X.T).dot(Y)
    return w

#梯度下降法
def gradient_descent(X, Y, m, alpha, accept_gradient, lamuda):
    cnt = 0 #记录迭代次数
    w = np.zeros(m+1).T #初始化 w 矩阵
```

```
    gradient_w = np.dot(X.T, X).dot(w) - np.dot(X.T, Y) + lamuda * w #计算初始梯度，迭代方向为负梯度方向
    while np.linalg.norm(gradient_w) >= accept_gradient:
        cnt = cnt + 1
        w = w - alpha * gradient_w #?  更新 w 矩阵
        gradient_w = np.dot(X.T, X).dot(w) - np.dot(X.T, Y) + lamuda * w #更新梯度方向
    print(cnt)
    return w

#共轭梯度法
def conjugate_gradient(X, Y, m, accept_gradient):
    cnt = 0 #限制迭代次数
    w = np.zeros(m+1).T #初始化需要求解的 w
    A = np.dot(X.T, X) #方便计算，为原式二次型的正定矩阵
    gradient_w = np.dot(A, w) - np.dot(X.T, Y) + lamuda * w #计算第一次梯度方向
    d = - gradient_w #第一次迭代方向为负梯度方向
    alpha = np.dot(A, w) - np.dot(X.T, Y) #初始化步长
    while cnt <= m: #控制迭代次数为 w 的维数
        alpha = - np.dot(gradient_w.T, d) / np.dot(d.T, A).dot(d) #更新步长，使损失函数达到最小的步长
        w = w + alpha * d #更新 w 矩阵，沿着共轭方向下降
        gradient_w = np.dot(A, w) - np.dot(X.T, Y) + lamuda * w #更新梯度，计算共轭方向和步长需要
        beta = np.dot(d.T, A).dot(gradient_w) / np.dot(d.T, A).dot(d) #计算共轭方向需要的线性关系系数
        d = -gradient_w + np.dot(beta, d) #得到共轭方向
        cnt = cnt + 1
    print(cnt)
    return w




'''
主函数
'''
m = 50 #阶数
lamuda = 0 #惩罚项的 lamda
var = 0.25 #高斯噪声的方差
alpha = 0.01 #梯度下降法的步长
accept_gradient = 0.1 #梯度下降法阈值
train_sample = 100 #训练样本个数
test_sample = 100 #测试样本个数
test_x = np.linspace(0, 1, test_sample)
```

```
test_X = np.zeros((m+1, test_sample))
for i in range(m+1):
    test_X[i] = test_x ** i #生成测试样本 X

x_t = np.arange(0, 1, 0.01)
y_t = np.sin(2 * np.pi * x_t)
plt.title('green-sin(2pix), blue-train_sample, red-test_sample') #绘制 sin（2pix）
plt.plot(x_t, y_t, 'g')

X, Y = generate_data(m, train_sample, var) #生成数据

#不同方法得到 w
w = loss_1(X, Y) #最小二乘法，不带正则项
# w = loss_2(X, Y, lamuda) #最小二乘法，带正则项
# w = gradient_descent(X, Y, m, alpha, accept_gradient, lamuda) #梯度下降法
# w = conjugate_gradient(X, Y, m, accept_gradient)

#展示
print(w)
test_y = np.dot(w.T, test_X)
plt.plot(test_x, test_y, 'r')
plt.show()
```