哈尔滨工业大学计算机科学与技术学院

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：1190201421

姓名：张瑞

**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)

1. **实验要求及实验环境**

**（一）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。

**（二）实验环境**

Windows 10; PyCharm Community Edition 2021.2;Python 3.6

**三、设计思想**

**（一）生成数据**

根据输入的训练集大小在[0,1]上等间隔取正弦函数上的点，再用高斯分布（均值为0，标准差为输入参数）为每个点的纵坐标加上噪声。

**（二）最小二乘法（无惩罚项）**

由泰勒级数可知，足够高阶的多项式可以拟合任意函数。对于正弦函数，完全可以用多项式函数来拟合。已知训练集有个样本点，阶多项式记为

其中多项式系数矩阵**w**为

则用最小二乘法求解析解时的代价函数为

令

则可以将代价函数改写为矩阵形式

展开后可得

拟合时需要使E(**w**)的值最小，则可以对上式求导，令导数等于零，从而求出解析解。

对**w**求导得

令上式为0得

**（三）最小二乘法（带惩罚项）**

若最小二乘法求解析解时无惩罚项，随着阶数的增大， 往往具有较大的2范数，从而使得多项式函数有更强的变化能力，更加贴合训练集中的样本点。但正因为它过于贴合样本点了，将一些不属于训练集的特征（如噪声影响）都学习到了，反而使拟合效果变差，这种现象的本质就是过拟合。于是，我们考虑增加惩罚项，使的2范数没有那么大。

现修改代价函数为

写成矩阵形式为

对**w**求导得

令上式为0得

其中**I**为单位矩阵。

超参数反映了惩罚项的重要程度，可以将代价函数看做以和为变量的函数，手动给定一个的合适范围，在该范围内取值，分别求出对应的，再计算对应的，最终选取最小值对应的，即

**（四）梯度下降法**

在多元函数中，给定点处的梯度是一个向量，其方向指出了函数在该处上升最快的方向。由前面的推导可以发现，代价函数其实是一个关于的多元函数，若给定一个初始点，只要不断沿着当前点的梯度反方向走合适距离，多次迭代后便可逐渐靠近使代价函数值最小的点。

代价函数的梯度如下

迭代关系式如下

其中为下一个点，为当前点，为步长。注意的选取十分重要：过小的会使迭代次数过多，寻找的速度极慢；过大的则可能使得沿梯度反方向移动时越过，无法靠近。

和之前寻找超参数的方法类似，可以手动给定一个的合适范围，在该范围内通过实验找出能使收敛到的尽量大的作为超参数，最终收敛结果对应的即为所求最优解。

**（五）共轭梯度法**

梯度下降法每次都向当前点梯度的反方向移动，但这并不能保证每次在每个维度上都是在靠近最优解，也就会因为在相同方向上的反复移动而增加迭代次数。如果在解空间的每一个维度分别去求解最优解，那么在寻找最优解的过程中绝不重复曾经走过的方向，在维空间最多走步即可。

前面已推出

其中，，**A**对称且正定。

则问题转化为求解。

迭代的过程如下：

直到足够小：

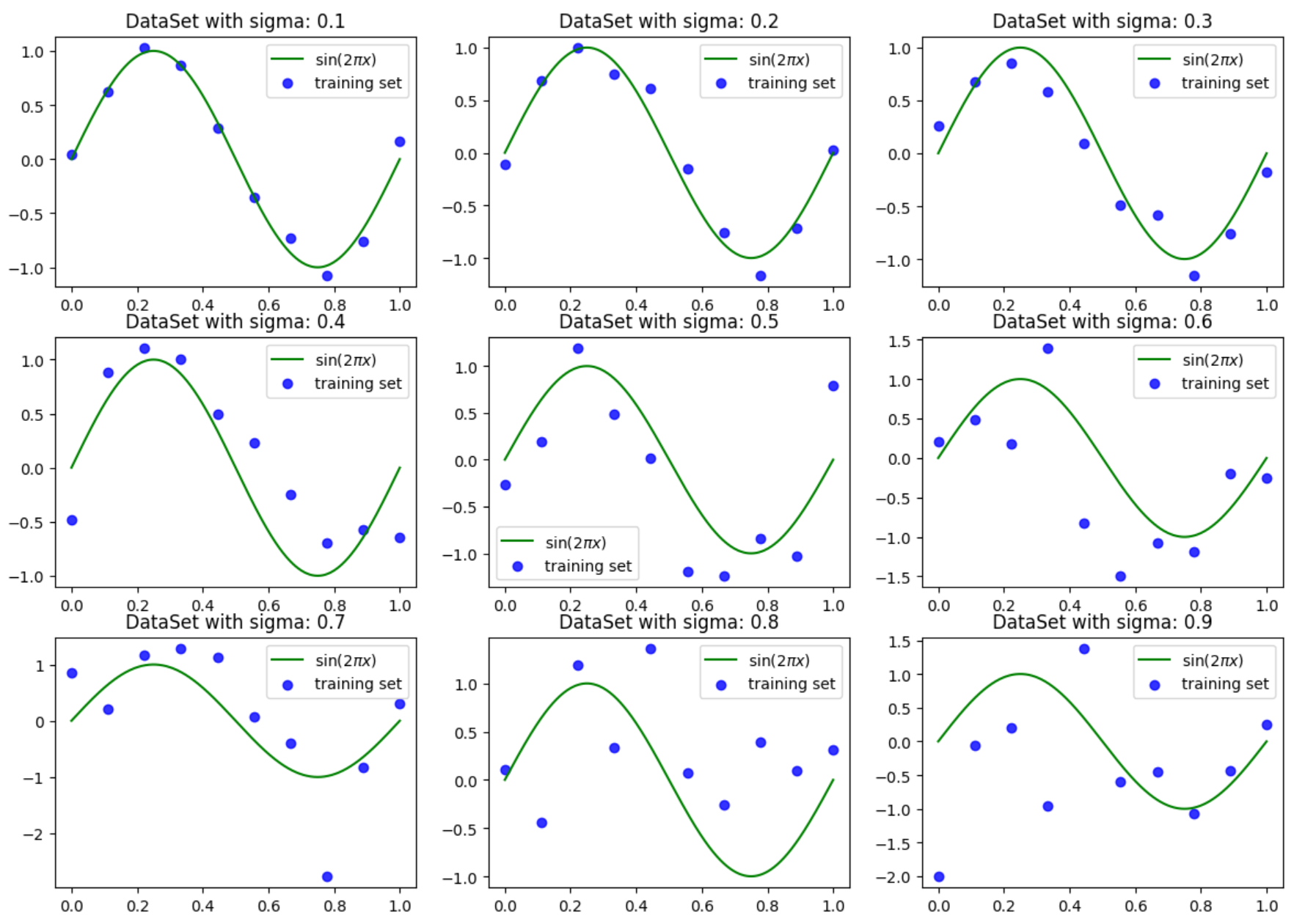
如果：

否则：

1. **实验结果与分析**

**（一）生成数据**

选取训练集大小为，再依次将高斯分布的标准差取为生成数据集，结果如下

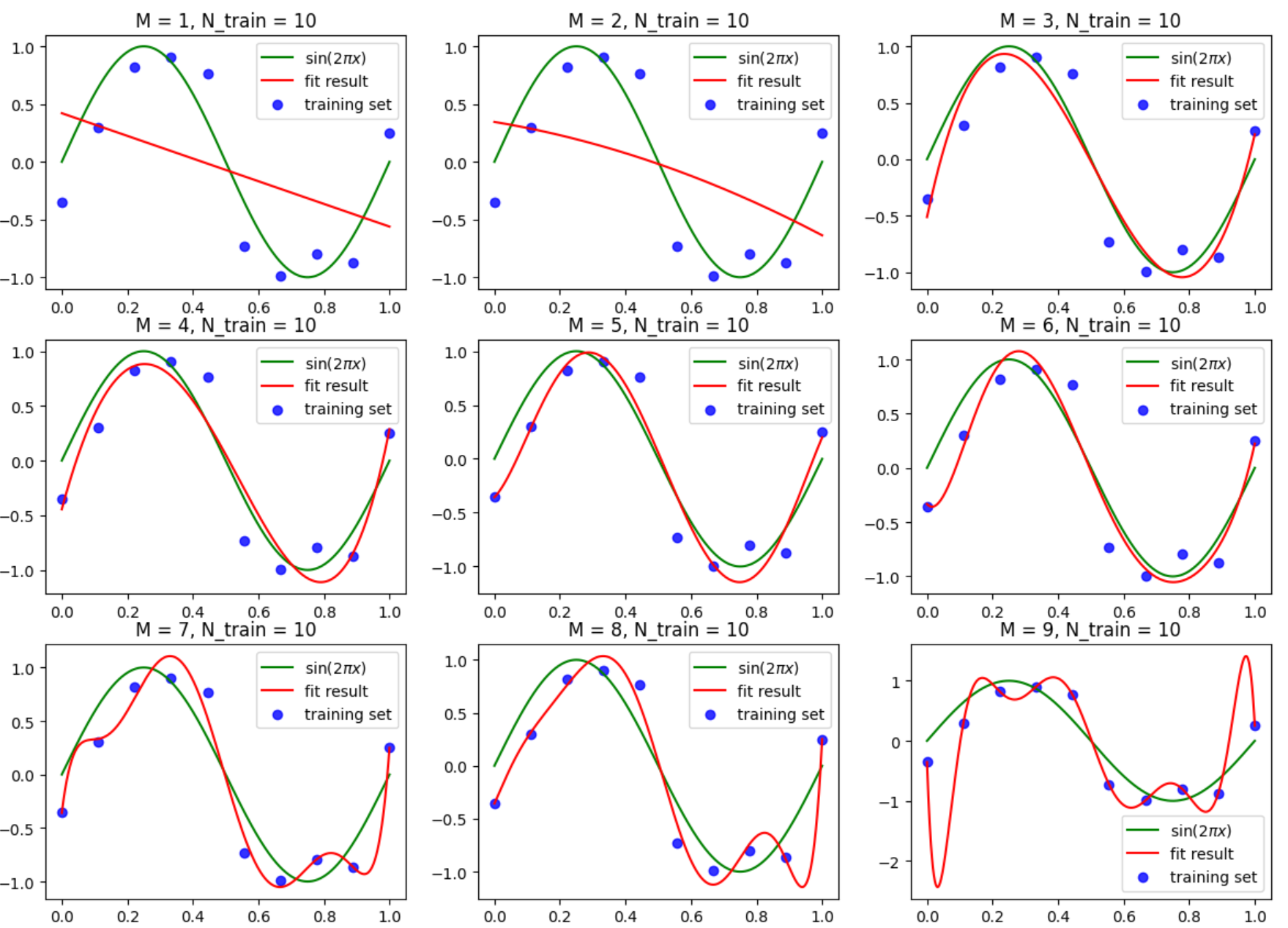


可以发现，随着标准差的增大，生成的数据偏离正弦函数越远。在本次实验中，后续数据的噪声标准差默认为。

**（二）最小二乘法（无惩罚项）**

**1.训练集大小相同，多项式阶数不同**

选取训练集大小为，再依次将多项式阶数取为进行拟合，结果如下



可以发现：

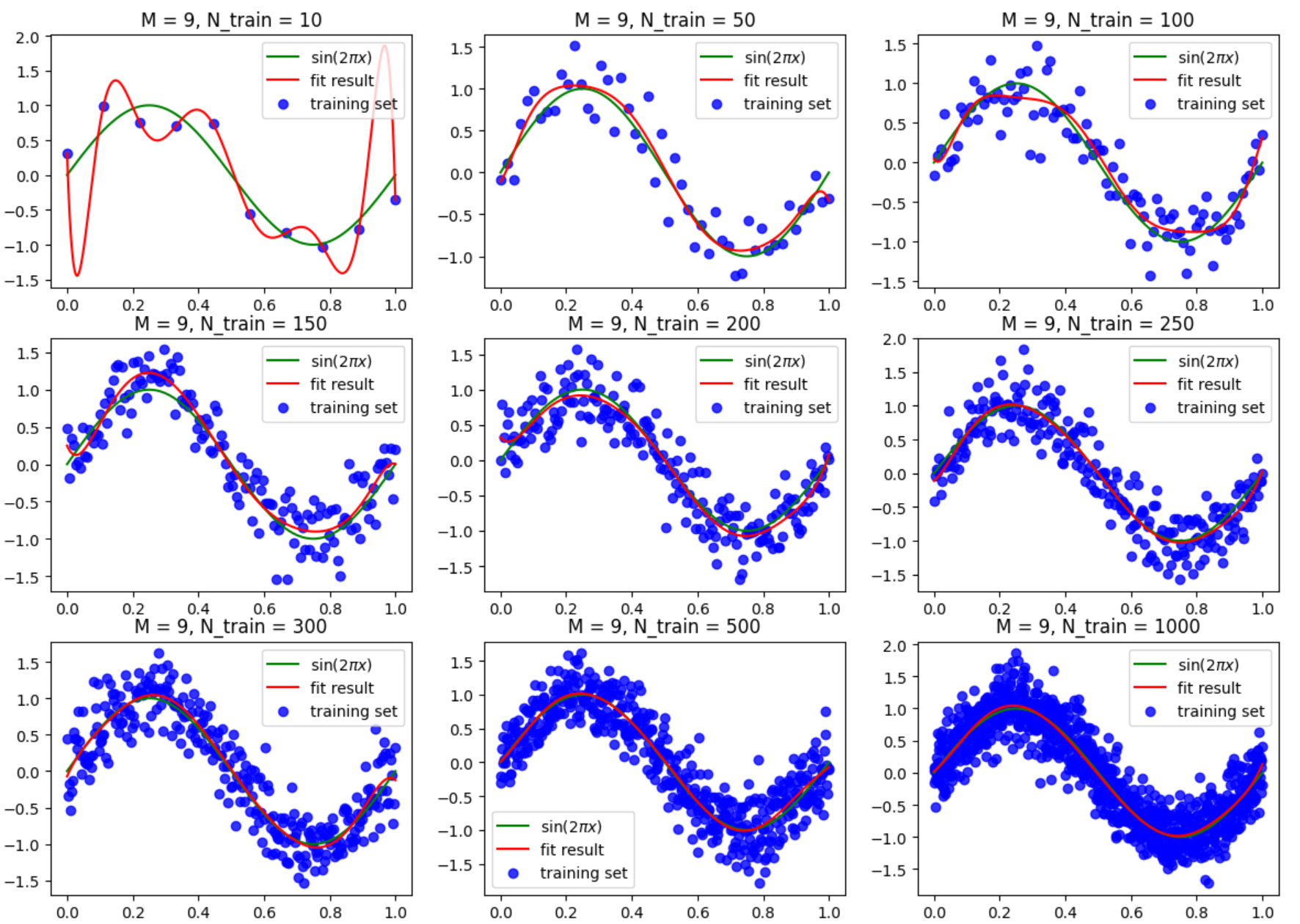
当多项式阶数较小（取）时，拟合效果较差。此时的模型学习能力不足，无法学习到数据集中的“一般规律”，导致泛化能力弱，被称为“欠拟合”。

当多项式阶数适中（取）时，拟合效果不错。此时的模型学习能力很强，能准确学习到数据集中的“一般规律”，泛化能力强。

当多项式阶数较大（取）时，拟合效果又变差。此时的模型学习能力过强，除了数据集中的“一般规律”，单个样本的自身特点都能被捕捉到，导致曲线虽然能相当好地贴近训练集中数据，但泛化能力弱，在测试集上性能较差，被称为“过拟合”。

**2. 多项式阶数相同，训练集大小不同**

选取多项式阶数为，再依次将训练集大小取为进行拟合，结果如下



可以发现，一开始存在着过拟合现象，但随着训练集大小的增大，拟合效果越来越好，越来越贴近真实的正弦曲线。这说明增加训练集大小有助于克服过拟合现象。

**（三）最小二乘法（带惩罚项）**

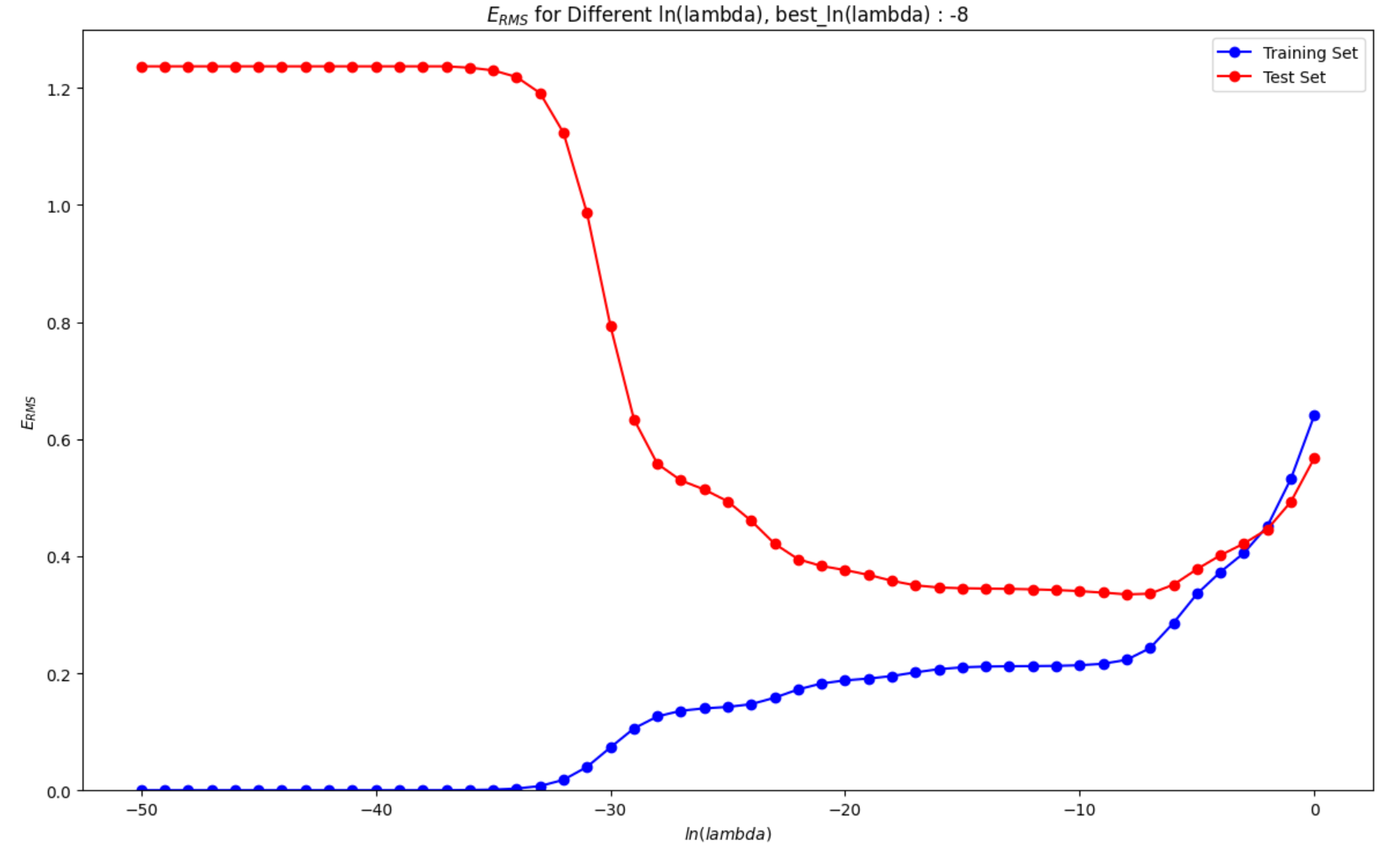
**1.训练集大小和多项式阶数相同，超参数不同**

先引入对拟合优度的评价函数

又因为已推导出

则可以用下式来代替

选取训练集大小为，多项式阶数为，手动给定一个的合适范围，在该范围内取值，分别求出对应的，结果如下



可以发现：

当取时，测试集上较大且几乎不变，训练集上也几乎不变。此时过小，惩罚项比重小，模型退化为原模型。

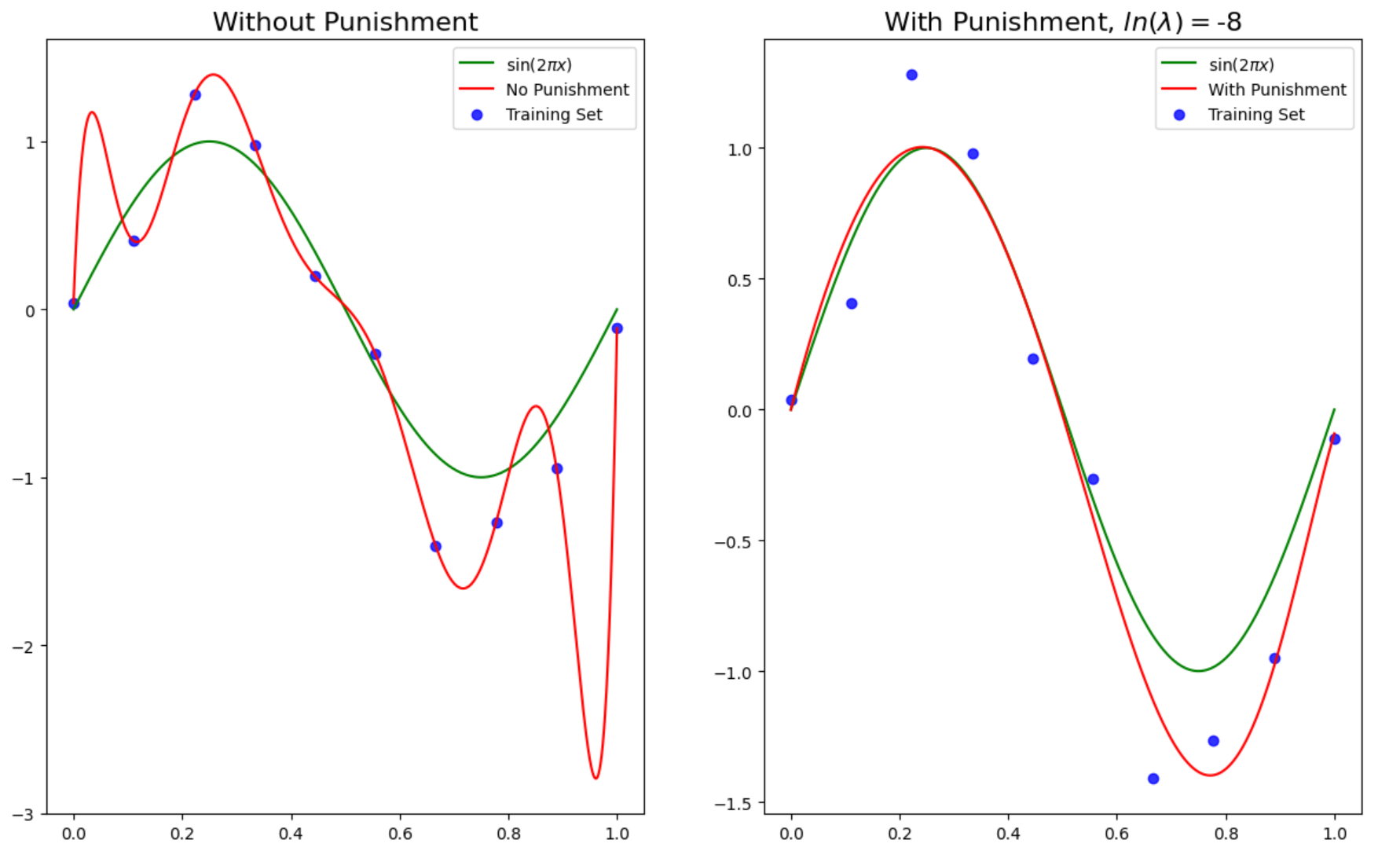
当取时，测试集上开始下降，训练集上有一定上升。此时较合适，惩罚项比重适当，模型复杂度与问题匹配。

当取时，测试集上又开始上升，训练集上大幅上升。此时过大，惩罚项比重大，模型复杂度被降低。

需要注意的是，因为训练集数据的不确定性，每次得到的最佳都不一样，可以通过多次实验选择被判定为最佳次数最多的作为最终结果。通过次实验，得到被判定为最佳次数最多的三个所对应的，结果如下



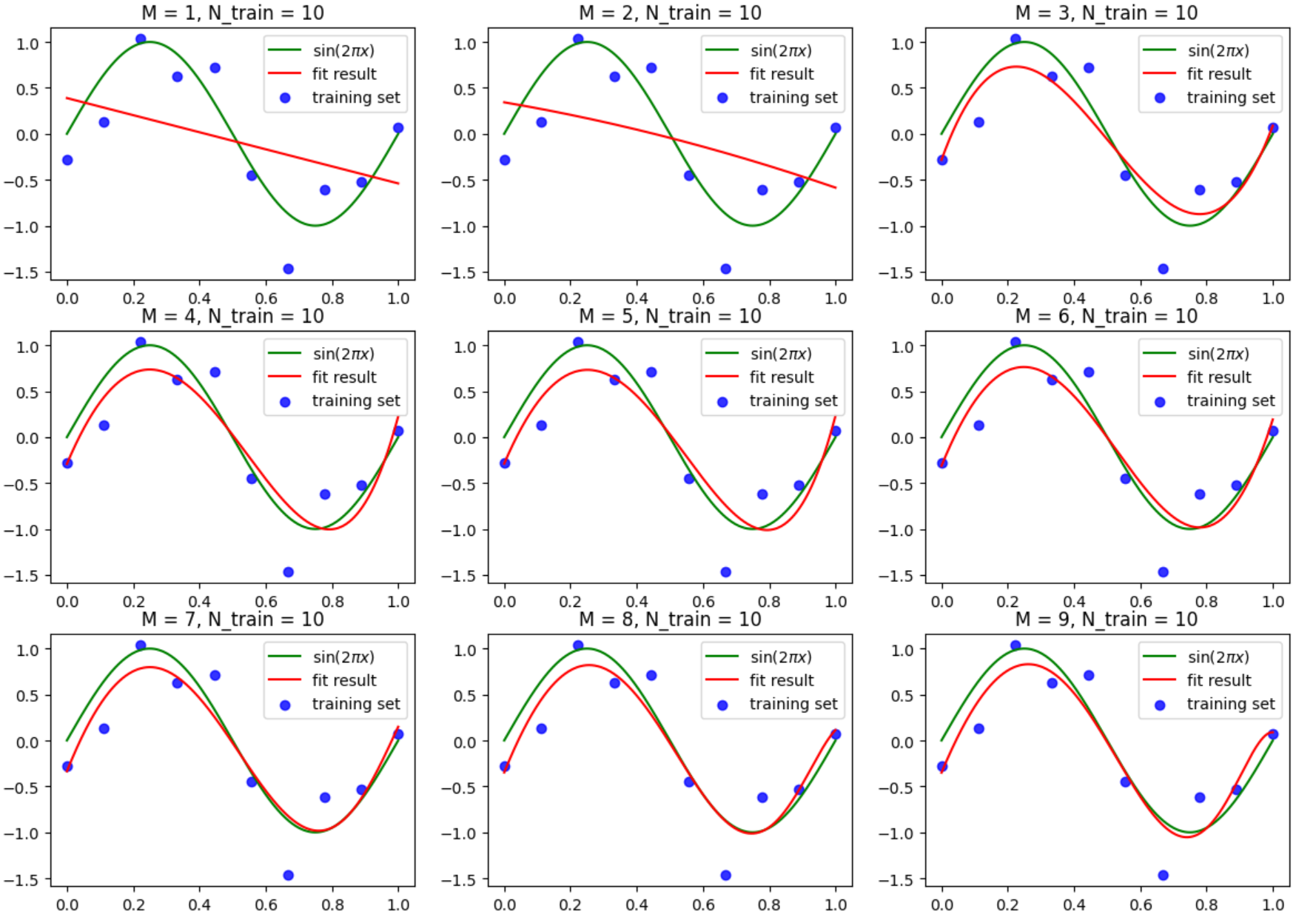
于是，得到最佳超参数，接下来将训练集大小为，多项式阶数为，超参数为情况下的无惩罚项和带惩罚项的拟合结果进行对比，结果如下



可以发现，带惩罚项的模型泛化能力强于无惩罚项的模型，说明增加比重恰当的惩罚项也有助于克服过拟合现象。无特别说明，后续超参数默认为。

**2.训练集大小和超参数相同，多项式阶数不同**

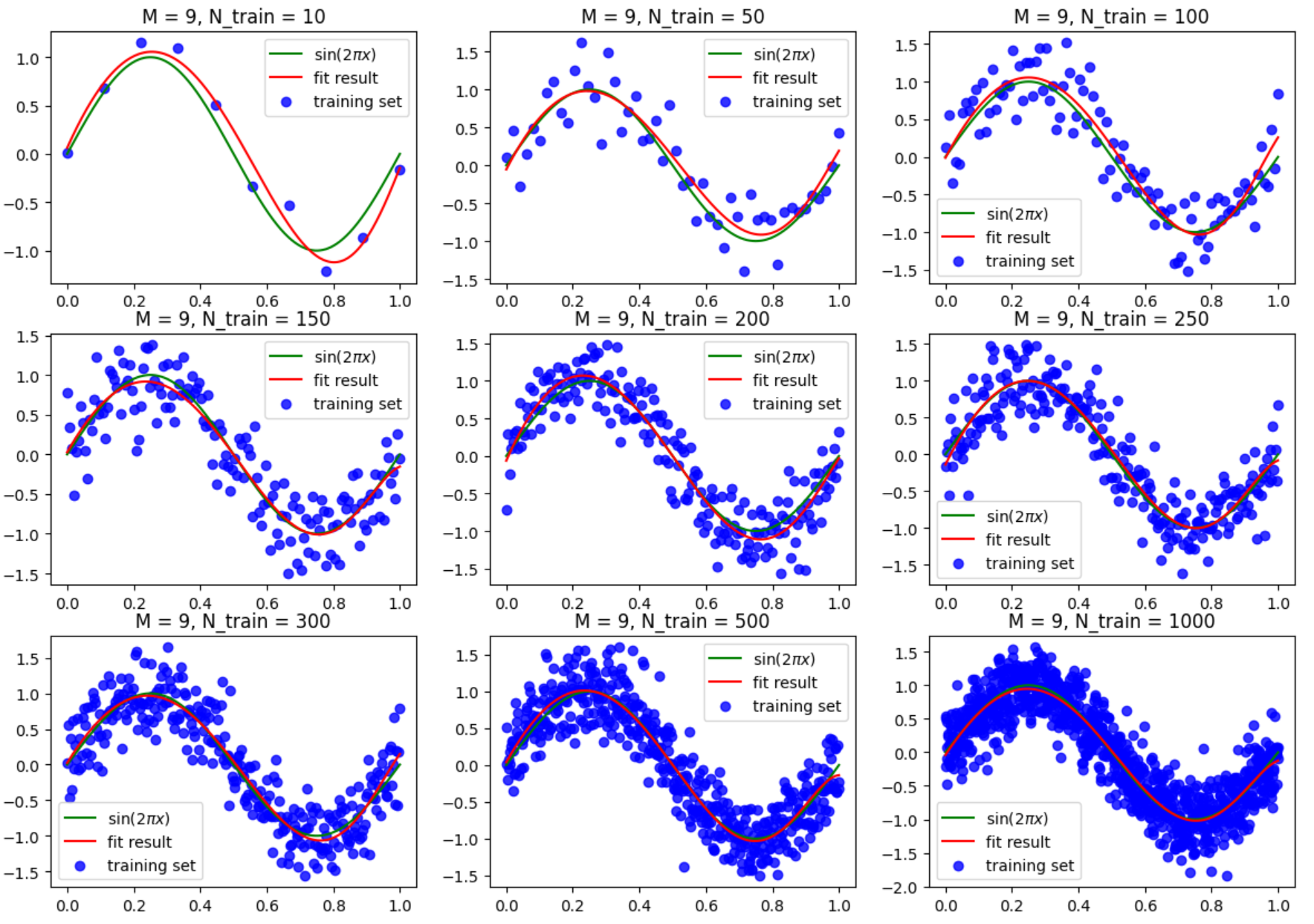
选取训练集大小为，超参数为，再依次将多项式阶数取为进行拟合，结果如下



可以发现，由于惩罚项的加入，即使多项式阶数增大也不会出现过拟合现象，再次证明增加比重恰当的惩罚项也有助于克服过拟合现象。

**3. 多项式阶数和超参数相同，训练集大小不同**

选取多项式阶数为，超参数为，再依次将训练集大小取为进行拟合，结果如下



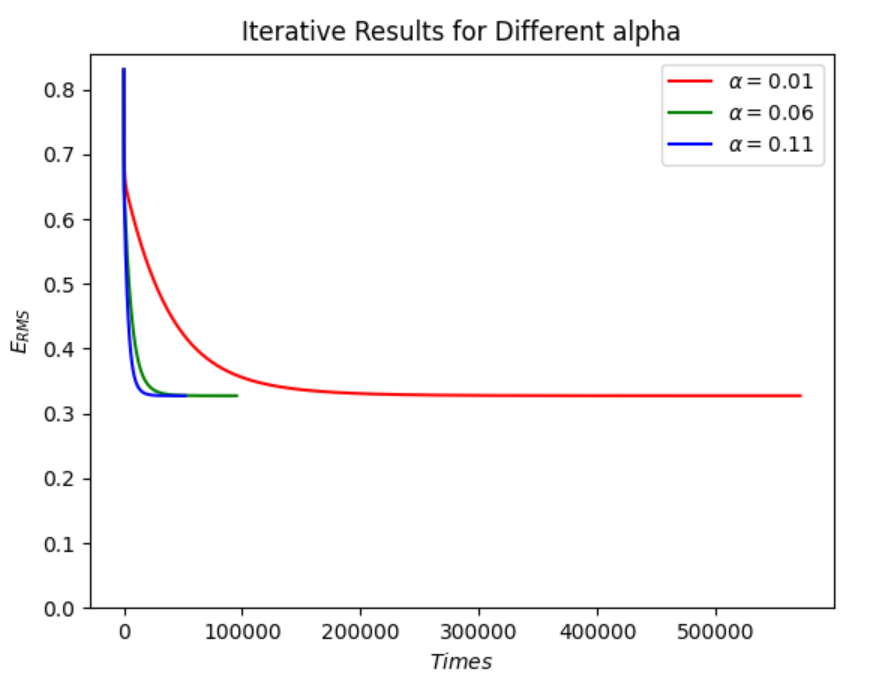
可以发现，由于惩罚项的加入，一开始就不存在过拟合现象了，随着训练集大小的增大，拟合效果依然越来越好，越来越贴近真实的正弦曲线。

**（四）梯度下降法**

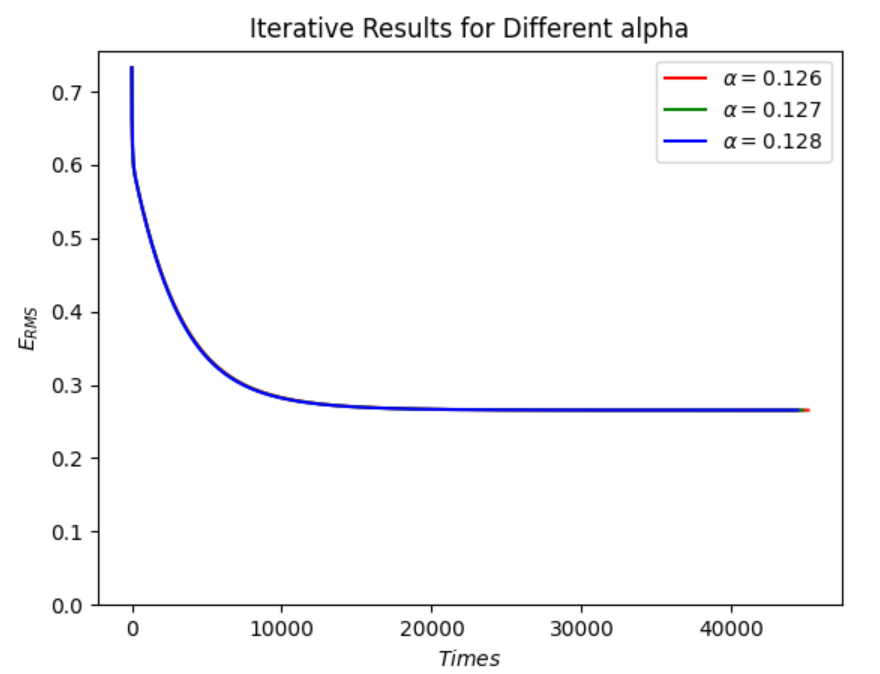
该方法的求解精度被设定为。

**1.训练集大小和多项式阶数相同，超参数不同**

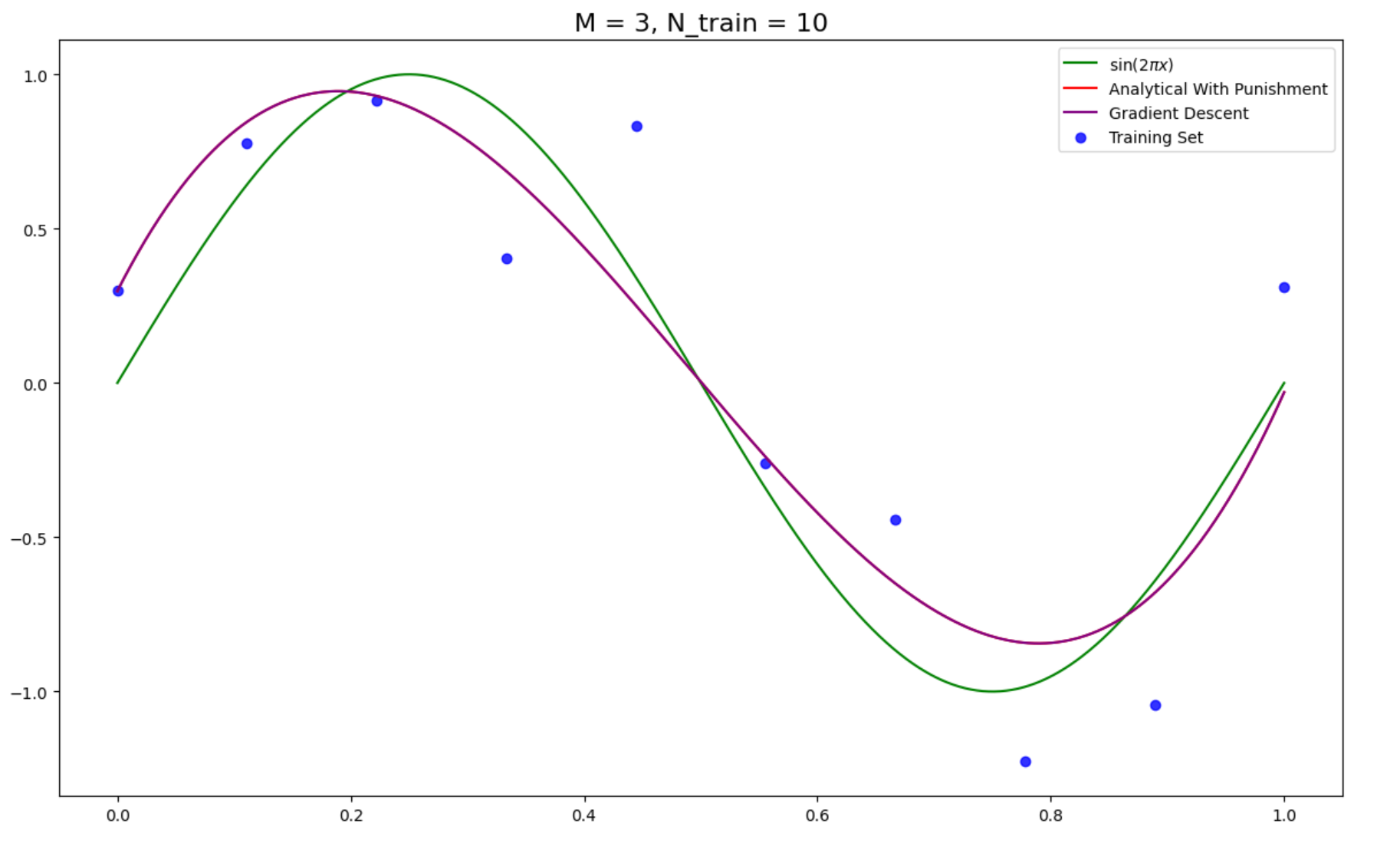
选取训练集大小为，多项式阶数为，手动给定三个超参数的合适值，分别求出对应的，结果如下



要找到使收敛到的尽量大的，需要继续手动调参，直至



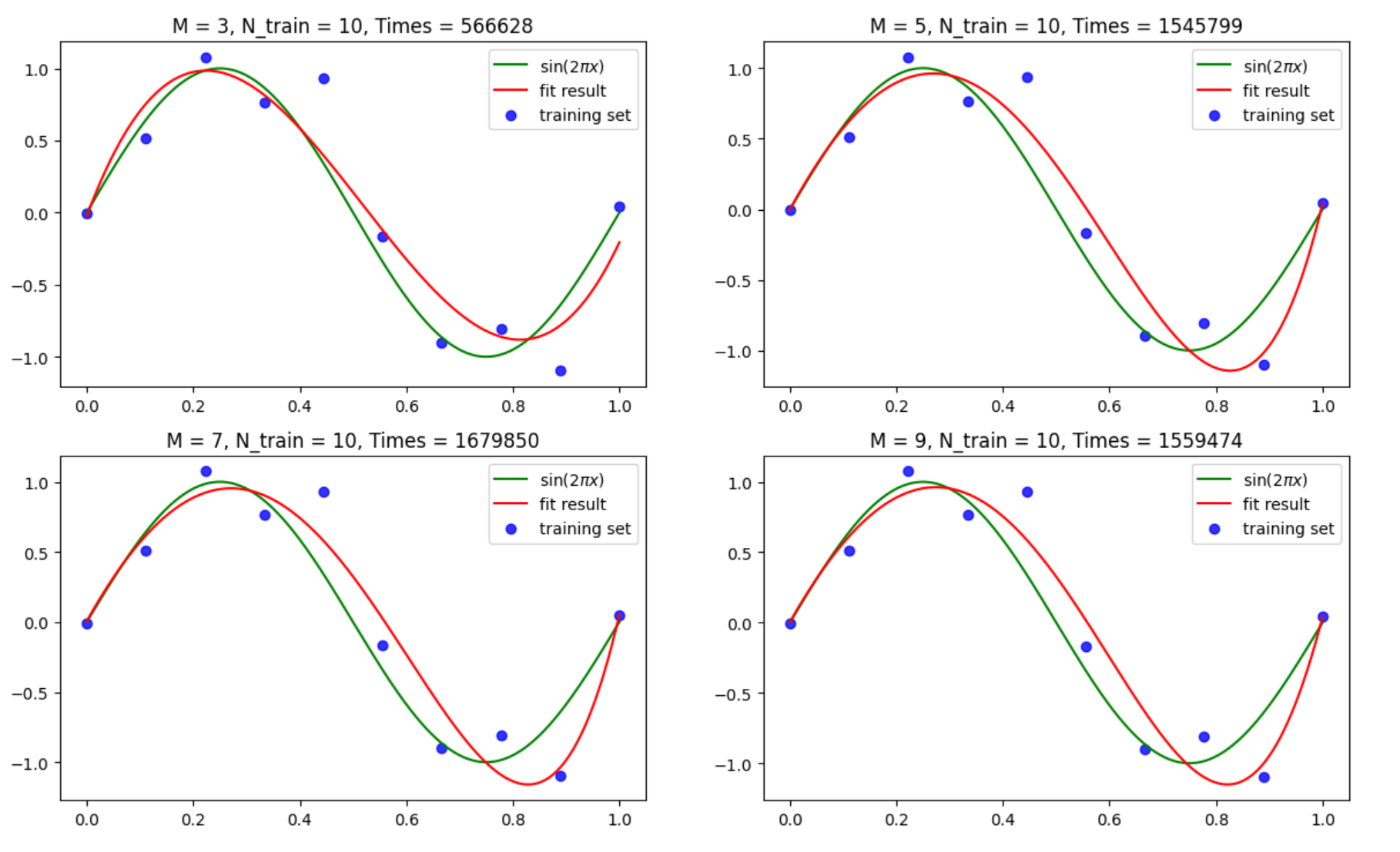
此时的三个超参数对应的图像已极其接近且再增大就无法收敛，于是得到最佳超参数。接下来将训练集大小为，多项式阶数为，超参数为，超参数为情况下的带惩罚项最小二乘法和梯度下降法的拟合结果进行对比，结果如下



可以发现，带惩罚项最小二乘法和梯度下降的拟合结果几乎完全重合，说明梯度下降法得到的结果已经相当接近解析解。

**2. 训练集大小和超参数相同，多项式阶数不同**

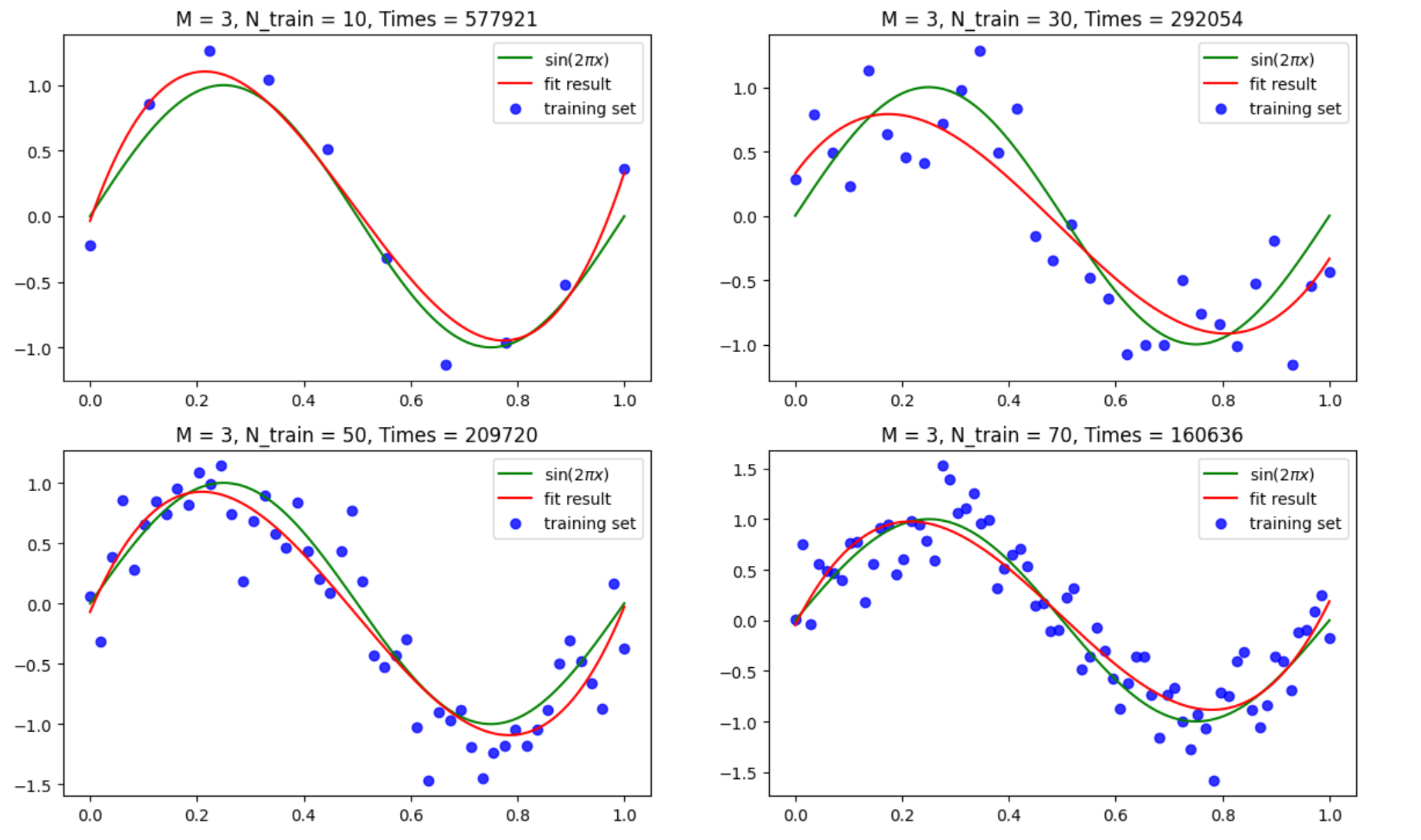
选取训练集大小为，超参数为，再依次将多项式阶数取为进行拟合，结果如下



可以发现，由于梯度下降法里有惩罚项的加入，同带惩罚项的最小二乘法一样，即使多项式阶数增大也不会出现过拟合现象，再次证明增加比重恰当的惩罚项也有助于克服过拟合现象。还需要注意的是，迭代次数和多项式阶数并无明显关系。

**3. 多项式阶数和超参数相同，训练集大小不同**

选取多项式阶数为，超参数为，再依次将训练集大小取为进行拟合，结果如下



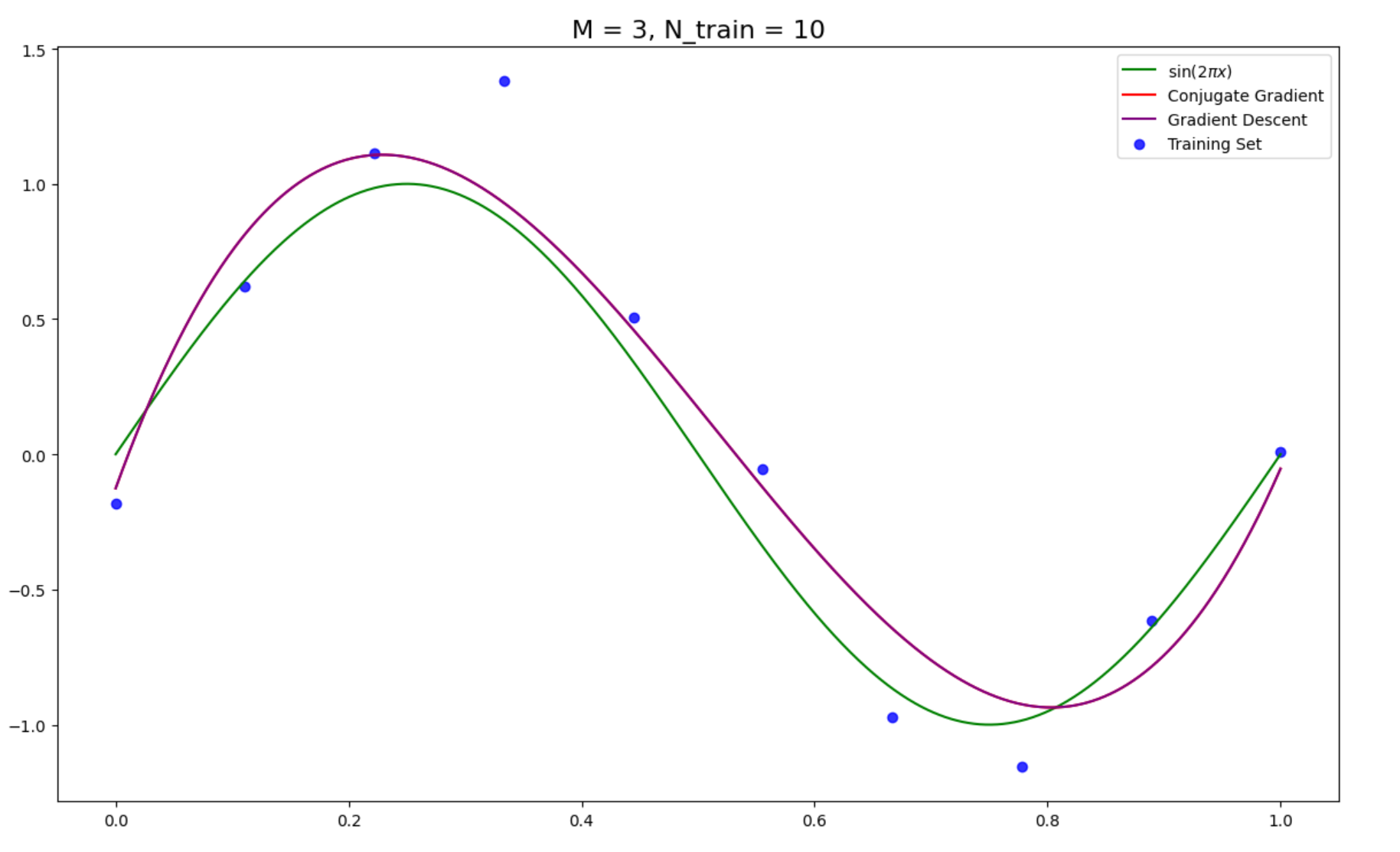
可以发现，随着训练集大小的增大，拟合效果越来越好，越来越贴近真实的正弦曲线。还需要注意的是，随着训练集大小的增大，迭代次数减少。

**（五）共轭梯度法**

该方法的求解精度被设定为。

**1. 共轭梯度法和梯度下降法的对比**

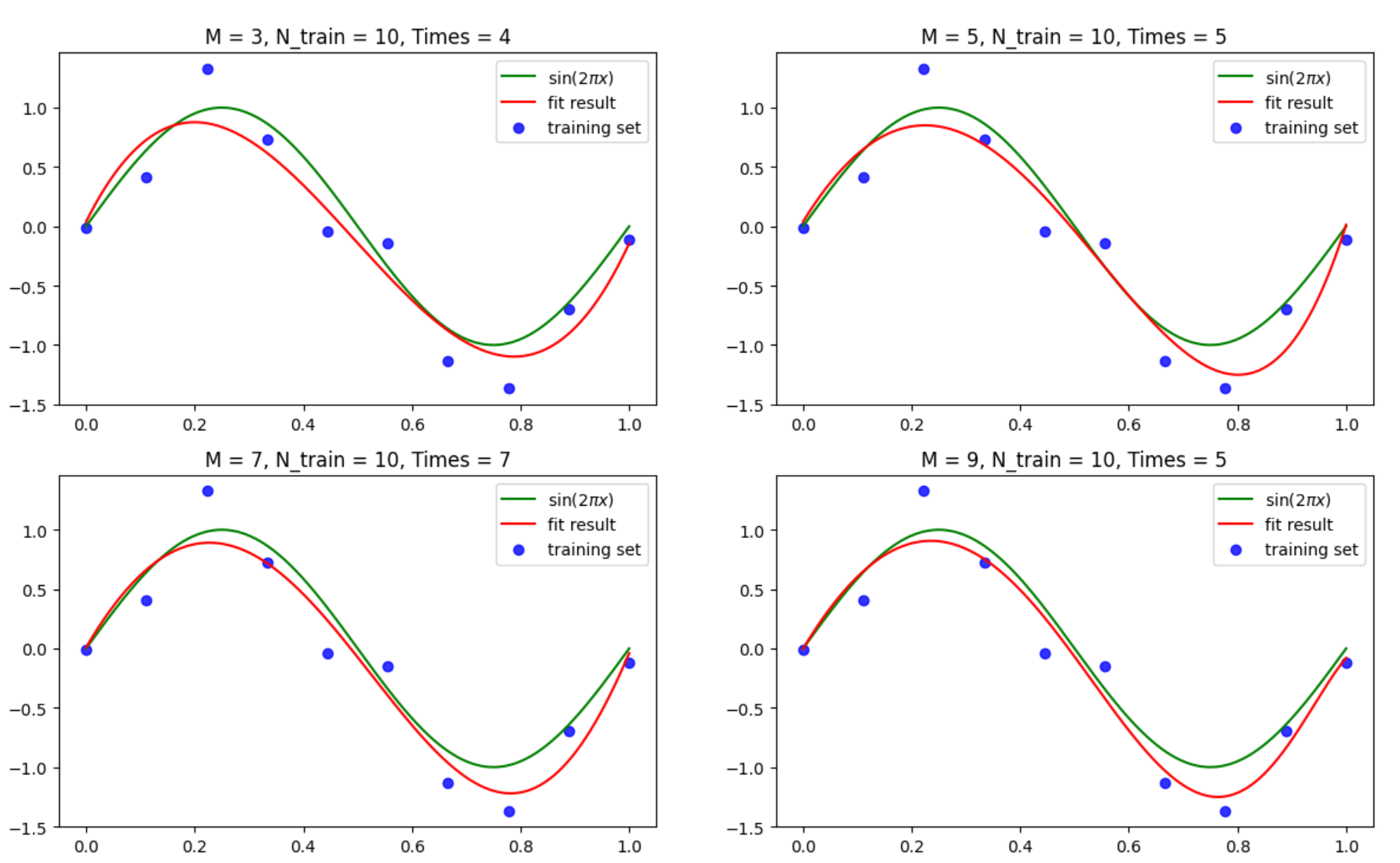
将训练集大小为，多项式阶数为，超参数为情况下的共轭梯度法和梯度下降法的拟合结果进行对比，结果如下



可以发现，共轭梯度法和梯度下降法的拟合结果几乎完全重合，说明共轭梯度法得到的结果和梯度下降法一样，已经相当接近解析解。

**2. 训练集大小相同，多项式阶数不同**

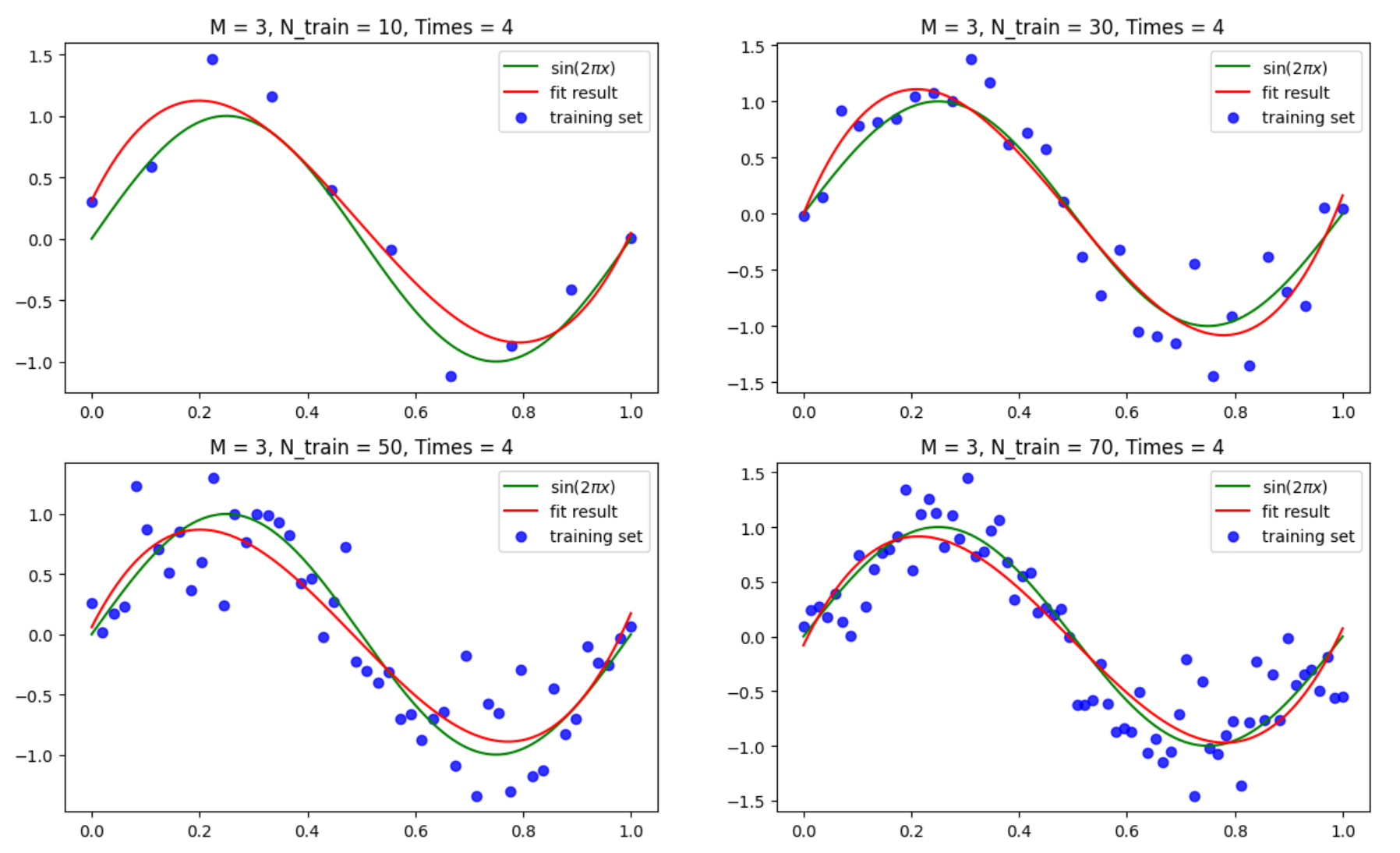
选取训练集大小为，再依次将多项式阶数取为进行拟合，结果如下



可以发现，由于共轭梯度法里有惩罚项的加入，同带惩罚项的最小二乘法一样，即使多项式阶数增大也不会出现过拟合现象，再次证明增加比重恰当的惩罚项也有助于克服过拟合现象。还需要注意的是，共轭梯度法的迭代次数远远小于梯度下降法的迭代次数，且维空间里迭代次数最多达到次。

**3. 多项式阶数相同，训练集大小不同**

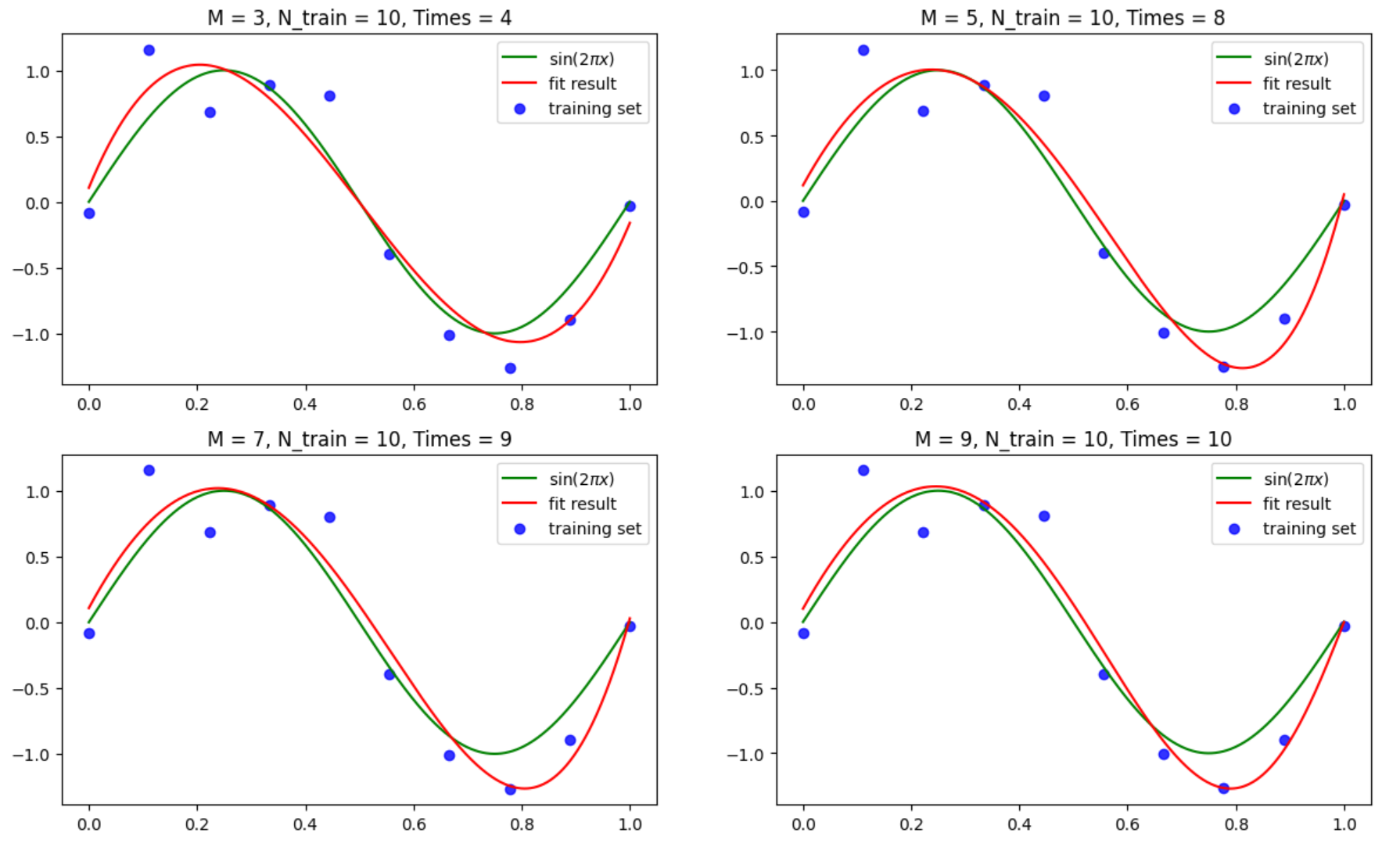
选取多项式阶数为，再依次将训练集大小取为进行拟合，结果如下



可以发现，随着训练集大小的增大，拟合效果越来越好，越来越贴近真实的正弦曲线。还需要注意的是，共轭梯度法的迭代次数远远小于梯度下降法的迭代次数，维空间里迭代次数最多达到次。

**补充：**

共轭梯度法的迭代次数并非一定能满足维空间里不多于次，还受到精度的影响。若将精度设定得过小，会导致迭代次数多于次，但实际上最多在迭代次数为第次的时候，所求解已经相当接近解析解，只是由于不满足设定的高精度而需要继续迭代下去。若将求解精度设定为，选取训练集大小为，再依次将多项式阶数取为进行拟合，迭代次数便会增加，结果如下



从上图可以看见，此时多项式阶数取为和进行拟合时的迭代次数分别超过了和。

1. **结论**

正弦函数可以用多项式进行拟合。

若用无惩罚项的最小二乘法求最优解，当多项式阶数增大时，可能会出现过拟合现象，此时若增加训练集样本数量，可以有效防止过拟合。

也可以在求最优解时使用带惩罚项的最小二乘法，惩罚项的加入也能很好地防止过拟合，即使多项式的阶数很大，模型的泛化能力依然很强。但需要注意把握惩罚项的比重：若惩罚项比重小，模型退化为原模型；若惩罚项比重适当，模型复杂度与问题匹配；若惩罚项比重大，模型复杂度将被降低。

还可以用梯度下降法迭代求解最优解。同样地，此时也要注意把握步长的大小：若步长过小，迭代次数会过多，速度极慢；若步长过大，则可能使得代价函数不收敛，无法找到最优解。

为了解决梯度下降法迭代次数过多的问题，还可以用共轭梯度法求解最优解。理论上来讲，在维空间里求解时，共轭梯度法的迭代次数最多达到次。在相同精度下，共轭梯度法所用迭代次数远远小于梯度下降法。

1. **参考文献**

<https://baike.baidu.com/item/共轭梯度法/7139204?fr=aladdin>

**七、附录：源代码（带注释）**

**1.main.py**

import numpy as np  
  
import myTool as mT  
import analyticalSolution as aS  
import gradientDescent as gD  
import conjugateGradient as cG  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 N\_train = 10 # 训练集的大小  
 N\_exact = 1000 # 从正弦函数上所取样本的数量  
 sigmaRange = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]) # 不同噪声标准差  
 layout = (3, 3) # 画图时的排版格式  
 # 展示不同标准差下，生成的数据情况  
 mT.different\_sigma(sigmaRange, N\_train, N\_exact, layout)  
  
 # 不带惩罚项的解析解(不同阶数、不同数据量)  
 sigma = 0.3 # 噪声的标准差  
 N\_fit = 1000 # 用于拟合的数据集的大小  
 mRange = range(1, 10) # 拟合所用多项式的不同阶数  
 # 展示相同N\_train,不同m时的拟合曲线  
 aS.different\_m(sigma, N\_train, N\_fit, mRange, layout)  
 N\_trainRange = np.array([10, 50, 100, 150, 200, 250, 300, 500, 1000]) # 训练集的不同大小  
 m = 9 # 拟合所用多项式的阶数  
 # 展示相同m,不同N\_train时的拟合曲线  
 aS.different\_n\_train(sigma, N\_trainRange, N\_fit, m, layout)  
  
 # 带惩罚项的解析解(不同超参数、不同阶数、不同数据量)  
 N\_test = 1000 # 测试集的大小  
 ln\_lam\_Range = range(-50, 1, 1) # 超参数的不同取值  
 # 展示不同lambda时，训练集E\_rms与测试集E\_rms的变化情况(m = 9, N\_train = 10, n\_test = 1000)  
 aS.different\_lam\_e\_rms(sigma, N\_train, N\_test, ln\_lam\_Range)  
 # 进行500次实验，寻找拟合最佳的lambda  
 aS.find\_lam(sigma, N\_train, N\_test, ln\_lam\_Range, 500)  
 ln\_lam = -8  
 # 展示m = 9, N\_train = 10, lambda = e^(-8)时，有惩罚项和无惩罚项的拟合曲线  
 aS.with\_and\_no\_punishment(sigma, N\_train, N\_test, ln\_lam)  
 # 展示相同N\_train,相同lambda,不同m时的拟合曲线  
 aS.different\_m\_with\_punishment(sigma, N\_train, N\_fit, mRange, layout, ln\_lam)  
 # 展示相同m,相同lambda,不同N\_train时的拟合曲线  
 aS.different\_n\_train\_with\_punishment(sigma, N\_trainRange, N\_fit, m, layout, ln\_lam)  
  
 # 梯度下降(不同超参数、不同阶数、不同数据量)  
 alphaRange = np.array([0.126, 0.127, 0.128]) # 超参数的不同取值  
 m = 3  
 # 展示不同alpha时，E\_rms随迭代次数的变化情况(m = 3, N\_train = 10)  
 gD.different\_alpha\_e\_rms(sigma, N\_train, m, -8, alphaRange)  
 # 展示m = 3, N\_train = 10, lambda = e^(-8), alpha = 0.128时，梯度下降和有惩罚项的拟合曲线  
 gD.gradient\_descent\_and\_with\_punishment(sigma, N\_train, N\_test, m, ln\_lam, 0.128)  
 mRange = np.array([3, 5, 7, 9])  
 layout = (2, 2)  
 # 展示相同N\_train,相同alpha,不同m时的拟合曲线  
 gD.different\_m(sigma, N\_train, N\_fit, mRange, ln\_lam, 0.01, layout)  
 N\_trainRange = np.array([10, 30, 50, 70])  
 # 展示相同m,相同alpha,不同N\_train时的拟合曲线  
 gD.different\_n\_train(sigma, N\_trainRange, N\_fit, m, ln\_lam, 0.01, layout)  
  
 # 共轭梯度(不同阶数、不同数据量)  
 # 展示m = 3, N\_train = 10, lambda = e^(-8), alpha = 0.128时，梯度下降和共轭梯度的拟合曲线  
 cG.gradient\_descent\_and\_conjugate\_gradient(sigma, N\_train, N\_test, m, ln\_lam, 0.128)  
 # 展示相同N\_train,不同m时的拟合曲线  
 cG.different\_m(sigma, N\_train, N\_fit, mRange, ln\_lam, layout)  
 # 展示相同m,不同N\_train时的拟合曲线  
 cG.different\_n\_train(sigma, N\_trainRange, N\_fit, m, ln\_lam, layout)

**2.myTool.py**

import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
  
  
def generate\_data\_with\_noise(sigma, n):  
 # 生成带（高斯）噪声的数据  
 # sigma为噪声的标准差，默认噪声的均值为0  
 # n为需要生成的数据集的大小  
  
 x = np.linspace(0, 1, n)  
 noise = np.random.normal(0, sigma, n)  
 t = np.sin(2 \* np.pi \* x) + noise  
 return x, t  
  
  
def different\_sigma(sigma\_range, n\_train, n\_exact, layout):  
 # 用不同标准差分别生成带（高斯）噪声的数据  
 # sigma\_range为不同的标准差  
 # n\_train为用来训练的数据集大小  
 # n\_exact为从待拟合函数（正弦函数）上取的数据集大小  
 # layout为画图时排版格式（行数和列数）  
  
 fig, axes = plt.subplots(\*layout)  
 # 对每个sigma分别生成数据并画图  
 for i in range(len(sigma\_range)):  
 sigma = sigma\_range[i]  
 x\_exact = np.linspace(0, 1, n\_exact)  
 x\_train, t\_train = generate\_data\_with\_noise(sigma, n\_train)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_exact, y=np.sin(2 \* np.pi \* x\_exact), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 设置图名  
 title = 'DataSet with sigma: ' + str(sigma)  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()  
  
  
def get\_matrix\_x(x, m):  
 # 用给定行向量生成公式推导中的矩阵X  
 # x为行向量，其中每个元素为待预测点的横坐标  
 # m为拟合所用的多项式的阶数  
 # 例如：  
 # 输入：[x y z], m  
 # 输出：[[1 x x^2 ... x^m]  
 # [1 y y^2 ... y^m]  
 # [1 z z^2 ... z^m]]  
  
 matrix = np.ones((len(x), m + 1))  
 for i in range(0, len(x)):  
 for j in range(1, m + 1):  
 matrix[i][j] = matrix[i][j - 1] \* x[i]  
 return matrix  
  
  
def get\_predictive\_y(x, w, m):  
 # 用推导公式中矩阵X和列向量w预测点的纵坐标列向量Y  
 # x为行向量，其中每个元素为待预测点的横坐标  
 # w为列向量，其中每个元素为拟合所用的多项式的系数  
 # m为拟合所用的多项式的阶数  
  
 matrix\_x = get\_matrix\_x(x, m)  
 return matrix\_x @ w  
  
  
def cal\_e\_rms(y, t, w, lam):  
 # 计算方根均值，进行拟合优度评价  
 # y为预测值  
 # t为真实值  
 # w为列向量，其中每个元素为拟合所用的多项式的系数  
 # lam为公式推导中的超参数  
  
 return np.sqrt(np.mean(np.square(y - t)) + lam \* np.sqrt(w.T @ w) / len(t))

**3.analyticalSolution.py**

from collections import Counter  
import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
  
import myTool as mT  
  
  
class AnalyticalSolution(object):  
  
 def \_\_init\_\_(self, x, t):  
 # 初始化X和T  
 # x为公式推导中的矩阵，t为公式推导中的列向量  
 self.X = x  
 self.T = t  
  
 def no\_punishment(self):  
 # 不带惩罚项的解析解求法  
 return np.linalg.pinv(self.X) @ self.T  
  
 def with\_punishment(self, lam):  
 # 带惩罚项的解析解求法  
 # lam为公式推导中的超参数lambda  
 return np.linalg.pinv(self.X.T @ self.X + lam \* np.identity(len(self.X.T))) @ self.X.T @ self.T  
  
  
def different\_m(sigma, n\_train, n\_fit, m\_range, layout):  
 # 用不同阶数分别拟合正弦函数（无惩罚项）  
 # sigma为数据噪声的标准差  
 # n\_train为用来训练的数据集大小  
 # n\_fit为用来拟合的数据集大小  
 # m\_range为不同的阶数  
 # layout为画图时排版格式（行数和列数）  
  
 fig, axes = plt.subplots(\*layout)  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 对每个order分别进行拟合并画图  
 for i in range(len(m\_range)):  
 m = m\_range[i]  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, m), t\_train.T)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, analytical\_solution.no\_punishment(), m), color="r",  
 label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train)  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()  
  
  
def different\_n\_train(sigma, n\_train\_range, n\_fit, m, layout):  
 # 用不同训练集大小分别训练，以拟合正弦函数（无惩罚项）  
 # sigma为数据噪声的标准差  
 # n\_train\_range为不同的用来训练的数据集大小  
 # n\_fit为用来拟合的数据集大小  
 # m为拟合时的多项式的阶数  
 # layout为画图时排版格式（行数和列数）  
  
 fig, axes = plt.subplots(\*layout)  
 # 对每个n\_train分别进行训练后拟合并画图  
 for i in range(len(n\_train\_range)):  
 n\_train = n\_train\_range[i]  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, m), t\_train.T)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, analytical\_solution.no\_punishment(), m), color="r",  
 label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train)  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()  
  
  
def different\_lam\_e\_rms(sigma, n\_train, n\_test, ln\_lam\_range):  
 # 训练集E\_rms与测试集E\_rms随超参数lambda的变化情况  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # n\_test为测试集大小  
 # ln\_lam\_range为超参数lambda取对数后的取值范围  
  
 # 生成训练集  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 # 生成测试集  
 x\_test, t\_test = mT.generate\_data\_with\_noise(sigma, n\_test)  
 # 存放训练集拟合优度  
 e\_rms\_train\_list = []  
 # 存放测试集拟合优度  
 e\_rms\_test\_list = []  
  
 e\_rms\_min = float('inf')  
 best\_ln\_lam = 0  
 for ln\_lam in ln\_lam\_range:  
 # 通过训练集求解得到w  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, 9), t\_train.T)  
 w = analytical\_solution.with\_punishment(np.exp(ln\_lam))  
 # 按w对训练集进行结果预测，计算拟合优度并存入列表  
 y\_train = mT.get\_predictive\_y(x\_train, w, 9)  
 e\_rms\_train\_list.append(mT.cal\_e\_rms(y\_train, t\_train.T, w, np.exp(ln\_lam)))  
 # 按w对测试集进行结果预测，计算拟合优度并存入列表  
 y\_test = mT.get\_predictive\_y(x\_test, w, 9)  
 e\_rms = mT.cal\_e\_rms(y\_test, t\_test.T, w, np.exp(ln\_lam))  
 e\_rms\_test\_list.append(e\_rms)  
 # 记录当前最优拟合情况  
 if e\_rms < e\_rms\_min:  
 e\_rms\_min = e\_rms  
 best\_ln\_lam = ln\_lam  
 # 画图操作  
 fig, axes = plt.subplots()  
 # 训练集E\_rms随超参数lambda的对数的变化情况  
 axes.plot(ln\_lam\_range, e\_rms\_train\_list, 'b-o', label="Training Set")  
 # 测试集E\_rms随超参数lambda的对数的变化情况  
 axes.plot(ln\_lam\_range, e\_rms\_test\_list, 'r-o', label="Test Set")  
 # 设置图名  
 title = "$E\_{RMS}$ for Different ln(lambda), best\_ln(lambda) : " + str(best\_ln\_lam)  
 props = {'title': title, 'xlabel': '$ln(lambda)$', 'ylabel': '$E\_{RMS}$'}  
 axes.set(\*\*props)  
 axes.legend()  
 axes.set\_ylim(bottom=0)  
 plt.show()  
  
  
def find\_lam(sigma, n\_train, n\_test, ln\_lam\_range, times):  
 # 通过多次实验寻找使拟合最优的三个超参数取值的对数  
 # sigma为噪声的标准差，默认噪声的均值为0  
 # n\_train为训练集大小  
 # n\_test为测试集大小  
 # ln\_lam\_range为超参数lambda取对数后的取值范围  
 # times为实验重复次数  
  
 best\_ln\_lam\_list = []  
 for i in range(times):  
 # 生成训练集  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 # 生成测试集  
 x\_test, t\_test = mT.generate\_data\_with\_noise(sigma, n\_test)  
  
 e\_rms\_min = float('inf')  
 best\_ln\_lam = 0  
 for ln\_lam in ln\_lam\_range:  
 # 通过训练集求解得到w  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, 9), t\_train)  
 w = analytical\_solution.with\_punishment(np.exp(ln\_lam))  
 # 对测试集进行结果预测，计算拟合优度并记录下当前最优情况  
 y\_test = mT.get\_predictive\_y(x\_test, w, 9)  
 e\_rms = mT.cal\_e\_rms(y\_test, t\_test, w, np.exp(ln\_lam))  
 if e\_rms < e\_rms\_min:  
 e\_rms\_min = e\_rms  
 best\_ln\_lam = ln\_lam  
  
 best\_ln\_lam\_list.append(best\_ln\_lam)  
 # 使拟合最优的三个超参数取值的对数及其被判为最优的实验次数  
 best\_ln\_lams = Counter(best\_ln\_lam\_list).most\_common(3)  
 print(best\_ln\_lams)  
  
  
def with\_and\_no\_punishment(sigma, n\_train, n\_fit, ln\_lam):  
 # 对比有惩罚项和无惩罚项的拟合曲线  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # n\_fit为用来拟合的数据集大小  
 # ln\_lam为超参数lambda取对数后的值  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 layout = (1, 2)  
 fig, axes = plt.subplots(\*layout)  
  
 # 无惩罚项的图  
 # 画训练集中各点  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="Training Set", ax=axes[0])  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=axes[0])  
 # 画多项式拟合曲线  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, 9), t\_train.T)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, analytical\_solution.no\_punishment(), 9), color="r",  
 label="No Punishment", ax=axes[0])  
 # 设置图名  
 axes[0].set\_title('Without Punishment', fontsize=16)  
  
 # 带惩罚项的图  
 # 画训练集中各点  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="Training Set", ax=axes[1])  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=axes[1])  
 # 画多项式拟合曲线  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, 9), t\_train.T)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, analytical\_solution.with\_punishment(np.exp(ln\_lam)), 9),  
 color="r", label="With Punishment", ax=axes[1])  
 # 设置图名  
 axes[1].set\_title('With Punishment, ' + '$ln(\\lambda) = $' + str(ln\_lam), fontsize=16)  
  
 plt.show()  
  
  
def different\_m\_with\_punishment(sigma, n\_train, n\_fit, m\_range, layout, ln\_lam):  
 # 用不同阶数分别拟合正弦函数（带惩罚项）  
 # sigma为数据噪声的标准差  
 # n\_train为用来训练的数据集大小  
 # n\_fit为用来拟合的数据集大小  
 # m\_range为不同的阶数  
 # layout为画图时排版格式（行数和列数）  
 # ln\_lam为超参数lambda取对数后的值  
  
 fig, axes = plt.subplots(\*layout)  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 对每个order分别进行拟合并画图  
 for i in range(len(m\_range)):  
 m = m\_range[i]  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, m), t\_train.T)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, analytical\_solution.with\_punishment(np.exp(ln\_lam)), m),  
 color="r", label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train)  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()  
  
  
def different\_n\_train\_with\_punishment(sigma, n\_train\_range, n\_fit, m, layout, ln\_lam):  
 # 用不同训练集大小分别训练，以拟合正弦函数（带惩罚项）  
 # sigma为数据噪声的标准差  
 # n\_train\_range为不同的用来训练的数据集大小  
 # n\_fit为用来拟合的数据集大小  
 # m为拟合时的多项式的阶数  
 # layout为画图时排版格式（行数和列数）  
 # ln\_lam为超参数lambda取对数后的值  
  
 fig, axes = plt.subplots(\*layout)  
 # 对每个n\_train分别进行训练后拟合并画图  
 for i in range(len(n\_train\_range)):  
 n\_train = n\_train\_range[i]  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 analytical\_solution = AnalyticalSolution(mT.get\_matrix\_x(x\_train, m), t\_train.T)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, analytical\_solution.with\_punishment(np.exp(ln\_lam)), m),  
 color="r", label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train)  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()

**4.gradientDescent.py**

import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
  
import myTool as mT  
import analyticalSolution as aS  
  
  
class GradientDescent(object):  
  
 def \_\_init\_\_(self, x, t, m, ln\_lam, alpha, delta=10 \*\* (-6)):  
 # 初始化行向量x,列向量T,多项式阶数m,超参数lambda,迭代步长alpha,精度delta  
 self.x = x  
 self.T = t  
 self.m = m  
 self.lam = np.exp(ln\_lam)  
 self.alpha = alpha  
 self.delta = delta  
  
 def cal\_loss(self, w):  
 # 求解当前多项式的系数w对应的拟合优度,用E\_rms体现  
 # w为列向量，其中每个元素为拟合所用的多项式的系数  
 y = mT.get\_predictive\_y(self.x, w, self.m)  
 return mT.cal\_e\_rms(y, self.T, w, self.lam)  
  
 def cal\_gradient(self, w):  
 # 求代价函数的梯度  
 # w为列向量，其中每个元素为拟合所用的多项式的系数  
 matrix\_x = mT.get\_matrix\_x(self.x, self.m)  
 return matrix\_x.T @ matrix\_x @ w - matrix\_x.T @ self.T + self.lam \* w  
  
 def solve(self, w0):  
 # 迭代求解最优解w\*  
 # w0为列向量，其中每个元素为拟合所用的多项式的系数，迭代开始时的初始值  
 w = w0  
 gradient = self.cal\_gradient(w0)  
  
 times = 0 # 记录迭代次数  
  
 times\_list = [times]  
 loss\_list = [self.cal\_loss(w0)]  
  
 while not np.all(np.absolute(gradient) <= self.delta):  
 times += 1  
 w = w - self.alpha \* gradient  
 gradient = self.cal\_gradient(w)  
 times\_list.append(times)  
 loss\_list.append(self.cal\_loss(w))  
  
 return w, np.array(times\_list), np.array(loss\_list)  
  
 def solve\_without\_loss(self, w0):  
 # 迭代求解最优解w\*  
 # w0为列向量，其中每个元素为拟合所用的多项式的系数，迭代开始时的初始值  
 w = w0  
 gradient = self.cal\_gradient(w0)  
  
 times = 0 # 记录迭代次数  
  
 times\_list = [times]  
  
 while not np.all(np.absolute(gradient) <= self.delta):  
 times += 1  
 w = w - self.alpha \* gradient  
 gradient = self.cal\_gradient(w)  
 times\_list.append(times)  
  
 return w, np.array(times\_list)  
  
  
def different\_alpha\_e\_rms(sigma, n\_train, m, ln\_lam, alpha\_range):  
 # 不同alpha取值下，E\_rms随迭代次数的变化情况  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # m为拟合时的多项式的阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # alpha\_range为超参数alpha的取值范围  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
  
 fig, axes = plt.subplots()  
  
 color\_list = ['r', 'g', 'b']  
 i = -1  
 for alpha in alpha\_range:  
 i += 1  
 color = color\_list[i]  
 # 通过梯度下降求解  
 gradient\_descent = GradientDescent(x\_train, t\_train.T, m, ln\_lam, alpha)  
 w0 = np.zeros(m + 1).T  
 w, times\_list, loss\_list = gradient\_descent.solve(w0)  
  
 axes.plot(times\_list, loss\_list, color=color, label="$\\alpha = $" + str(alpha))  
  
 title = "Iterative Results for Different alpha"  
 props = {'title': title, 'xlabel': '$Times$', 'ylabel': '$E\_{RMS}$'}  
 axes.set(\*\*props)  
 axes.legend()  
 axes.set\_ylim(bottom=0)  
 plt.show()  
  
  
def gradient\_descent\_and\_with\_punishment(sigma, n\_train, n\_fit, m, ln\_lam, alpha):  
 # 对比梯度下降和有惩罚项的拟合曲线  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # n\_fit为用来拟合的数据集大小  
 # m为拟合时的多项式的阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # alpha为超参数alpha的取值  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 fig, axes = plt.subplots()  
  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="Training Set", ax=axes)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=axes)  
 # 画多项式拟合曲线（带惩罚项的解析解）  
 analytical\_solution = aS.AnalyticalSolution(mT.get\_matrix\_x(x\_train, m), t\_train.T)  
 w1 = analytical\_solution.with\_punishment(np.exp(ln\_lam))  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w1, m), color="r", label="Analytical With Punishment",  
 ax=axes)  
 # 画多项式拟合曲线（梯度下降）  
 gradient\_descent = GradientDescent(x\_train, t\_train.T, m, ln\_lam, alpha)  
 w0 = np.zeros(m + 1).T  
 w2, times\_list = gradient\_descent.solve\_without\_loss(w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w2, m), color="purple", label="Gradient Descent", ax=axes)  
  
 axes.set\_title('M = ' + str(m) + ', N\_train = ' + str(n\_train), fontsize=16)  
 plt.show()  
  
  
def different\_n\_train(sigma, n\_train\_range, n\_fit, m, ln\_lam, alpha, layout):  
 # 用不同训练集大小分别训练，以拟合正弦函数（梯度下降）  
 # sigma为数据噪声的标准差  
 # n\_train\_range为不同的用来训练的数据集大小  
 # n\_fit为用来拟合的数据集大小  
 # m为拟合时的多项式的阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # alpha为超参数alpha的取值  
 # layout为画图时排版格式（行数和列数）  
  
 fig, axes = plt.subplots(\*layout)  
 # 对每个n\_train分别进行训练后拟合并画图  
 for i in range(len(n\_train\_range)):  
 n\_train = n\_train\_range[i]  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 gradient\_descent = GradientDescent(x\_train, t\_train.T, m, ln\_lam, alpha)  
 w0 = np.zeros(m + 1).T  
 w, times\_list = gradient\_descent.solve\_without\_loss(w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w, m), color="r", label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train) + ", Times = " + str(times\_list[-1])  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()  
  
  
def different\_m(sigma, n\_train, n\_fit, m\_range, ln\_lam, alpha, layout):  
 # 用不同多项式阶数分别拟合正弦函数（梯度下降）  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # n\_fit为用来拟合的数据集大小  
 # m\_range为拟合时的多项式的不同阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # alpha为超参数alpha的取值  
 # layout为画图时排版格式（行数和列数）  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 fig, axes = plt.subplots(\*layout)  
 # 对每个n\_train分别进行训练后拟合并画图  
 for i in range(len(m\_range)):  
 order = m\_range[i]  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 gradient\_descent = GradientDescent(x\_train, t\_train.T, order, ln\_lam, alpha)  
 w0 = np.zeros(order + 1).T  
 w, times\_list = gradient\_descent.solve\_without\_loss(w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w, order), color="r", label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(order) + ", N\_train = " + str(n\_train) + ", Times = " + str(times\_list[-1])  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()

**5.conjugateGradient.py**

import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
  
import myTool as mT  
import gradientDescent as gD  
  
  
class ConjugateGradient(object):  
 def \_\_init\_\_(self, x, t, m, ln\_lam, delta=10 \*\* (-4)):  
 # 初始化行向量X,列向量T,多项式阶数m,超参数lambda,精度delta  
 self.x = x  
 self.T = t  
 self.order = m  
 self.lam = np.exp(ln\_lam)  
 self.delta = delta  
  
 def cal\_a(self):  
 # 求解代价函数E(w)写成二次型后的矩阵A  
 matrix\_x = mT.get\_matrix\_x(self.x, self.order)  
 return matrix\_x.T @ matrix\_x + self.lam \* np.identity(len(matrix\_x.T))  
  
 def cal\_b(self):  
 # 求解代价函数E(w)写成二次型后的矩阵b  
 matrix\_x = mT.get\_matrix\_x(self.x, self.order)  
 return matrix\_x.T @ self.T  
  
 def cal\_loss(self, w):  
 # 求解当前多项式的系数w对应的拟合优度,用E\_rms体现  
 # w为列向量，其中每个元素为拟合所用的多项式的系数  
 y = mT.get\_predictive\_y(self.x, w, self.order)  
 return mT.cal\_e\_rms(y, self.T, w, self.lam)  
  
 def solve(self, a, b, w0):  
 # 迭代求解最优解w\*  
 # a为代价函数E(w)写成二次型后的矩阵A  
 # b为代价函数E(w)写成二次型后的矩阵b  
 # w0为列向量，其中每个元素为拟合所用的多项式的系数，迭代开始时的初始值  
  
 w = w0  
 times = 0  
 times\_list = [times] # 记录迭代次数  
 loss\_list = [self.cal\_loss(w)] # 记录拟合优度  
 r = [b - a @ w] # 记录残差向量  
 p = [b - a @ w] # 记录搜索方向  
  
 while True:  
 times += 1  
 times\_list.append(times)  
  
 if times == 1: # 第一次迭代搜索方向由残差向量确定  
 p.append(r[0])  
 else: # 后续迭代搜索方向需计算确定  
 p.append(  
 r[times - 1] + (r[times - 1].T @ r[times - 1]) / (r[times - 2].T @ r[times - 2]) \* p[times - 1])  
 alpha = (r[times - 1].T @ r[times - 1]) / (p[times].T @ a @ p[times])  
 w = w + alpha \* p[times]  
 loss\_list.append(self.cal\_loss(w))  
 r.append(r[times - 1] - alpha \* a @ p[times])  
  
 if np.all(np.absolute(r[times]) <= self.delta):  
 break  
  
 return w, np.array(times\_list), np.array(loss\_list)  
  
  
def gradient\_descent\_and\_conjugate\_gradient(sigma, n\_train, n\_fit, m, ln\_lam, alpha):  
 # 对比梯度下降和共轭梯度的拟合曲线  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # n\_fit为用来拟合的数据集大小  
 # m为拟合时的多项式的阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # alpha为超参数alpha的取值  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 fig, axes = plt.subplots()  
  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="Training Set", ax=axes)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=axes)  
 # 画多项式拟合曲线（共轭梯度）  
 conjugate\_gradient = ConjugateGradient(x\_train, t\_train.T, m, ln\_lam)  
 a = conjugate\_gradient.cal\_a()  
 b = conjugate\_gradient.cal\_b()  
 w0 = np.zeros(m + 1).T  
 w1, times\_list, loss\_list = conjugate\_gradient.solve(a, b, w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w1, m), color="r", label="Conjugate Gradient", ax=axes)  
 # 画多项式拟合曲线（梯度下降）  
 gradient\_descent = gD.GradientDescent(x\_train, t\_train.T, m, ln\_lam, alpha)  
 w0 = np.zeros(m + 1).T  
 w2, times\_list = gradient\_descent.solve\_without\_loss(w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w2, m), color="purple", label="Gradient Descent", ax=axes)  
 axes.set\_title('M = ' + str(m) + ', N\_train = ' + str(n\_train), fontsize=16)  
  
 plt.show()  
  
  
def different\_n\_train(sigma, n\_train\_range, n\_fit, m, ln\_lam, layout):  
 # 用不同训练集大小分别训练，以拟合正弦函数（共轭梯度）  
 # sigma为数据噪声的标准差  
 # n\_train\_range为不同的用来训练的数据集大小  
 # n\_fit为用来拟合的数据集大小  
 # m为拟合时的多项式的阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # layout为画图时排版格式（行数和列数）  
  
 fig, axes = plt.subplots(\*layout)  
 # 对每个n\_train分别进行训练后拟合并画图  
 for i in range(len(n\_train\_range)):  
 n\_train = n\_train\_range[i]  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 conjugate\_gradient = ConjugateGradient(x\_train, t\_train.T, m, ln\_lam)  
 a = conjugate\_gradient.cal\_a()  
 b = conjugate\_gradient.cal\_b()  
 w0 = np.zeros(m + 1).T  
 w, times\_list, loss\_list = conjugate\_gradient.solve(a, b, w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w, m), color="r", label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train) + ", Times = " + str(times\_list[-1])  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()  
  
  
def different\_m(sigma, n\_train, n\_fit, m\_range, ln\_lam, layout):  
 # 用不同多项式阶数分别拟合正弦函数（共轭梯度）  
 # sigma为数据噪声的标准差  
 # n\_train为训练集大小  
 # n\_fit为用来拟合的数据集大小  
 # m\_range为拟合时的多项式的不同阶数  
 # ln\_lam为超参数lambda取对数后的值  
 # layout为画图时排版格式（行数和列数）  
  
 x\_train, t\_train = mT.generate\_data\_with\_noise(sigma, n\_train)  
 fig, axes = plt.subplots(\*layout)  
 # 对每个n\_train分别进行训练后拟合并画图  
 for i in range(len(m\_range)):  
 m = m\_range[i]  
 x\_fit = np.linspace(0, 1, n\_fit)  
 # 画图操作  
 # 确定画图位置  
 ax\_target = axes[i // layout[1]][i % layout[1]]  
 # 画训练集中各点  
 # 默认情况下绘制线性回归拟合直线，用fit\_reg = False将其删除  
 sns.regplot(x=x\_train, y=t\_train, fit\_reg=False, color="b", label="training set", ax=ax\_target)  
 # 画待拟合函数  
 sns.lineplot(x=x\_fit, y=np.sin(2 \* np.pi \* x\_fit), color="g", label="$\\sin(2\\pi x)$", ax=ax\_target)  
 # 画多项式拟合曲线  
 conjugate\_gradient = ConjugateGradient(x\_train, t\_train.T, m, ln\_lam)  
 a = conjugate\_gradient.cal\_a()  
 b = conjugate\_gradient.cal\_b()  
 w0 = np.zeros(m + 1).T  
 w, times\_list, loss\_list = conjugate\_gradient.solve(a, b, w0)  
 sns.lineplot(x=x\_fit, y=mT.get\_predictive\_y(x\_fit, w, m), color="r", label="fit result", ax=ax\_target)  
 # 设置图名  
 title = "M = " + str(m) + ", N\_train = " + str(n\_train) + ", Times = " + str(times\_list[-1])  
 props = {'title': title}  
 ax\_target.set(\*\*props)  
  
 plt.show()