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)

二、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N_{train}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$sigma$）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$f: X \to Y$。对于正弦函数$\sin 2\pi x$，完全可以用多项式函数来拟合。已知训练集有N个样本点$(x_1, t_1), (x_2, t_2) \ldots (x_n, t_n)$，$m$阶多项式记为

$$y(x, \boldsymbol{w}) = \sum_{i=0}^{m} w_i x^i$$

其中多项式系数矩阵 **w** 为

$$\boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{pmatrix}$$

则用最小二乘法求解析解时的代价函数为

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, \boldsymbol{w}) - t_n\}^2$$

令

$$\boldsymbol{T} = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{pmatrix}$$

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{pmatrix}$$

则可以将代价函数改写为矩阵形式

$$E(\boldsymbol{w}) = \frac{1}{2}(\boldsymbol{Xw} - \boldsymbol{T})^T(\boldsymbol{Xw} - \boldsymbol{T})$$

展开后可得

$$E(\boldsymbol{w}) = \frac{1}{2}(\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{Xw} - 2\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{T} + \boldsymbol{T}^T\boldsymbol{T})$$

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

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

令上式为 0 得

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

（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m的增大，$\boldsymbol{w}^*$ 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$\boldsymbol{w}^*$的 2 范数没有那么大。

现修改代价函数为

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \boldsymbol{w}) - t_n\}^2 + \frac{\lambda}{2}||\boldsymbol{w}||^2$$

写成矩阵形式为

$$E(\boldsymbol{w}) = \frac{1}{2}[(\boldsymbol{Xw} - \boldsymbol{T})^T(\boldsymbol{Xw} - \boldsymbol{T}) + \lambda\boldsymbol{w}^T\boldsymbol{w}]$$

对 **w** 求导得

$$\frac{\partial E}{\partial \boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{Xw} - \boldsymbol{X}^T\boldsymbol{T} + \lambda\boldsymbol{w}$$

令上式为 0 得

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

其中 **I** 为单位矩阵。

超参数λ反映了惩罚项的重要程度，可以将代价函数看做以λ和$\boldsymbol{w}$为变量的函数$E(\boldsymbol{w}, \lambda)$，手动给定一个λ的合适范围，在该范围内取值，分别求出对应的$\boldsymbol{w}^*$，

再计算对应的$E(\boldsymbol{w}^*,\ \lambda)$，最终选取最小值对应的λ，即

$$\lambda\ = \arg min_{\lambda} E(\boldsymbol{w}, \lambda)$$

（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$E(\boldsymbol{w}) = \frac{1}{2}[(\boldsymbol{Xw}-\boldsymbol{T})^T(\boldsymbol{Xw}-\boldsymbol{T}) + \lambda\boldsymbol{w}^T\boldsymbol{w}]$其实是一个关于$\boldsymbol{w}$的多元函数，若给定一个初始点$\boldsymbol{w}_0$，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$\boldsymbol{w}^*$。

代价函数的梯度如下

$$\nabla E(\boldsymbol{w}) = \frac{\partial E}{\partial \boldsymbol{w}} = \boldsymbol{X}^T\boldsymbol{Xw} - \boldsymbol{X}^T\boldsymbol{T} + \lambda\boldsymbol{w}$$

迭代关系式如下

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha\nabla E(\boldsymbol{w}_i)$$

其中$\boldsymbol{w}_{i+1}$为下一个点，$\boldsymbol{w}_i$为当前点，α为步长。注意α的选取十分重要：过小的α会使迭代次数过多，寻找$\boldsymbol{w}^*$的速度极慢；过大的α则可能使得沿梯度反方向移动时越过$\boldsymbol{w}^*$，无法靠近$\boldsymbol{w}^*$。

和之前寻找超参数λ的方法类似，可以手动给定一个α的合适范围，在该范围内通过实验找出能使$\nabla E(\boldsymbol{w}_i)$收敛到$\boldsymbol{0}$的尽量大的$\alpha$作为超参数，最终收敛结果$\nabla E(\boldsymbol{w}^*)$对应的$\boldsymbol{w}^*$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n维空间最多走n步即可。

前面已推出

$$\nabla E(\boldsymbol{w}) = \boldsymbol{X}^T\boldsymbol{Xw} - \boldsymbol{X}^T\boldsymbol{T} + \lambda\boldsymbol{w} = \boldsymbol{Aw} - \boldsymbol{b}$$

其中$\boldsymbol{A} = \boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}$，$\boldsymbol{b} = \boldsymbol{X}^T\boldsymbol{T}$，**A** 对称且正定。

则问题转化为求解$\boldsymbol{Aw} = \boldsymbol{b}$。

迭代的过程如下：

$$\boldsymbol{w}_0 = \boldsymbol{0}$$

$$k = 0$$

$$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{Aw}$$

直到$\boldsymbol{r}_k$足够小：

$$k = k + 1$$

如果$k = 1$：

$$\boldsymbol{p}_1 = \boldsymbol{r}_0$$

否则：

$$\boldsymbol{p}_k = \boldsymbol{r}_{k-1} + \frac{\boldsymbol{r}_{k-1}^T\boldsymbol{r}_{k-1}}{\boldsymbol{r}_{k-2}^T\boldsymbol{r}_{k-2}}\boldsymbol{p}_{k-1}$$

$$\alpha_k = \frac{\boldsymbol{r}_{k-1}^T \boldsymbol{r}_{k-1}}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k}$$

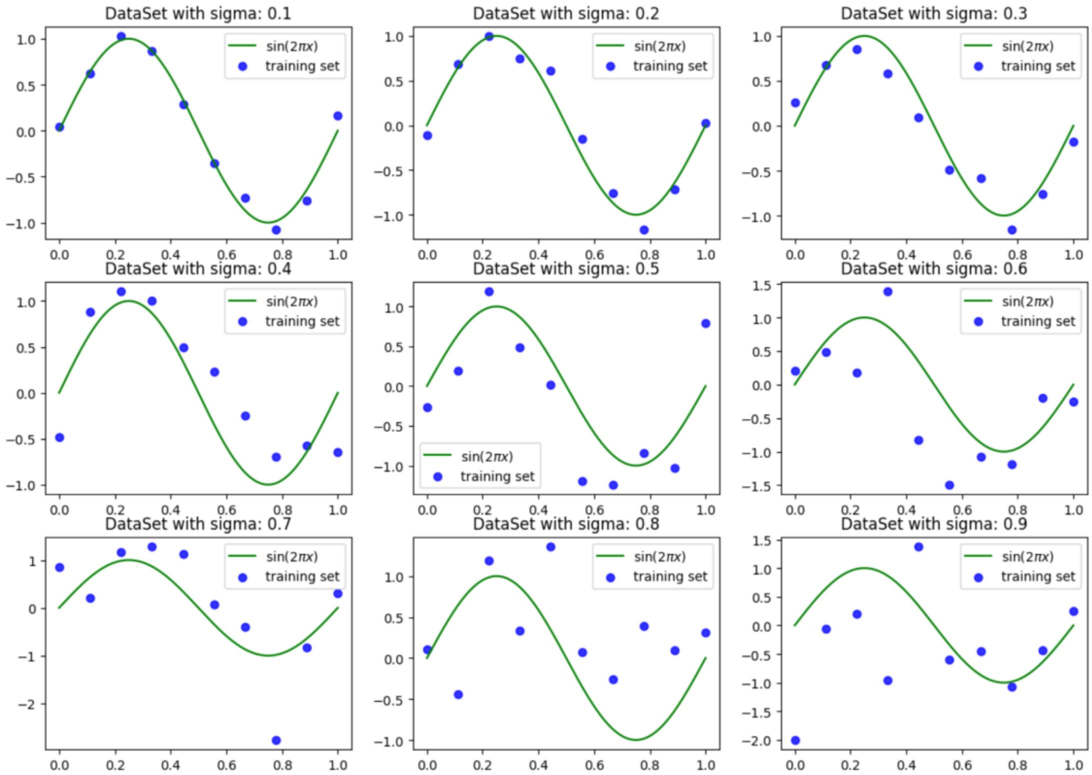
$$\boldsymbol{w}_k = \boldsymbol{w}_{k-1} + \alpha_k \boldsymbol{p}_k$$

$$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{A} \boldsymbol{p}_k$$

四、实验结果与分析

（一）生成数据

选取训练集大小N_{train}为10，再依次将高斯分布的标准差$sigma$取为0.1,0.2,0.3, ... ,0.9生成数据集，结果如下

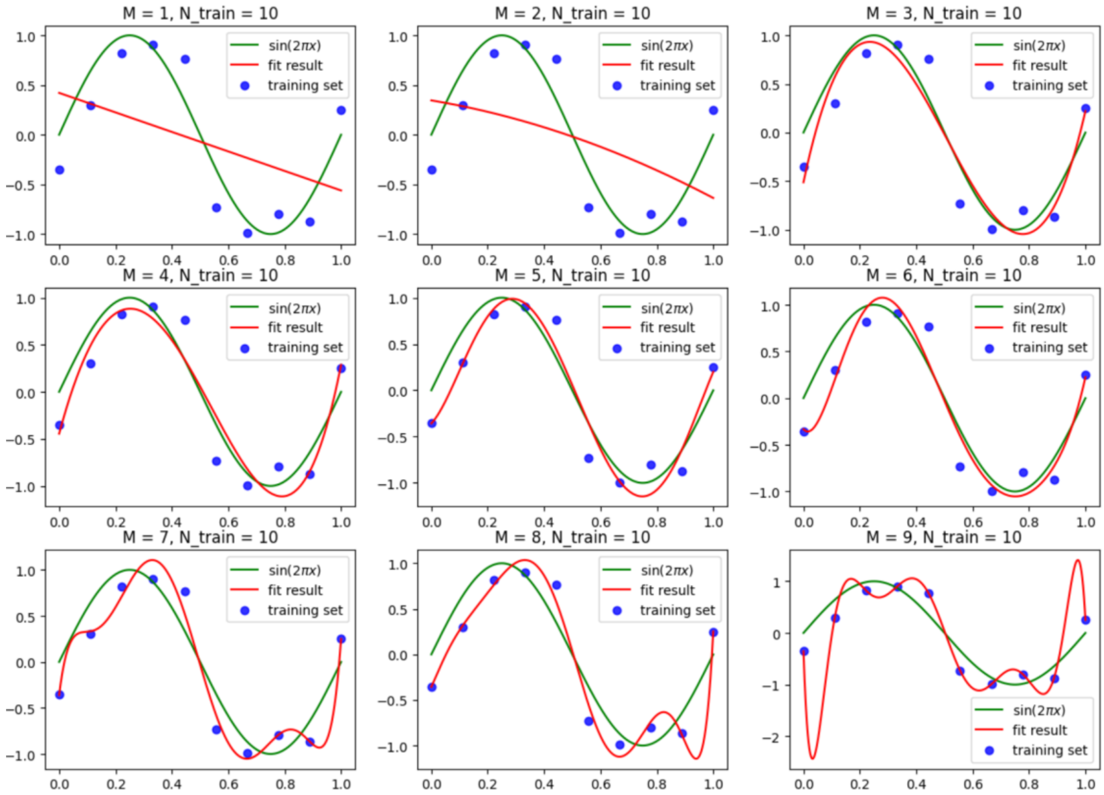


可以发现，随着标准差的增大，生成的数据偏离正弦函数越远。在本次实验中，后续数据的噪声标准差默认为0.3。

（二）最小二乘法（无惩罚项）

1.训练集大小N_{train}相同，多项式阶数m不同

选取训练集大小N_{train}为10，再依次将多项式阶数m取为1,2,3, ... ,9进行拟合，结果如下



可以发现：

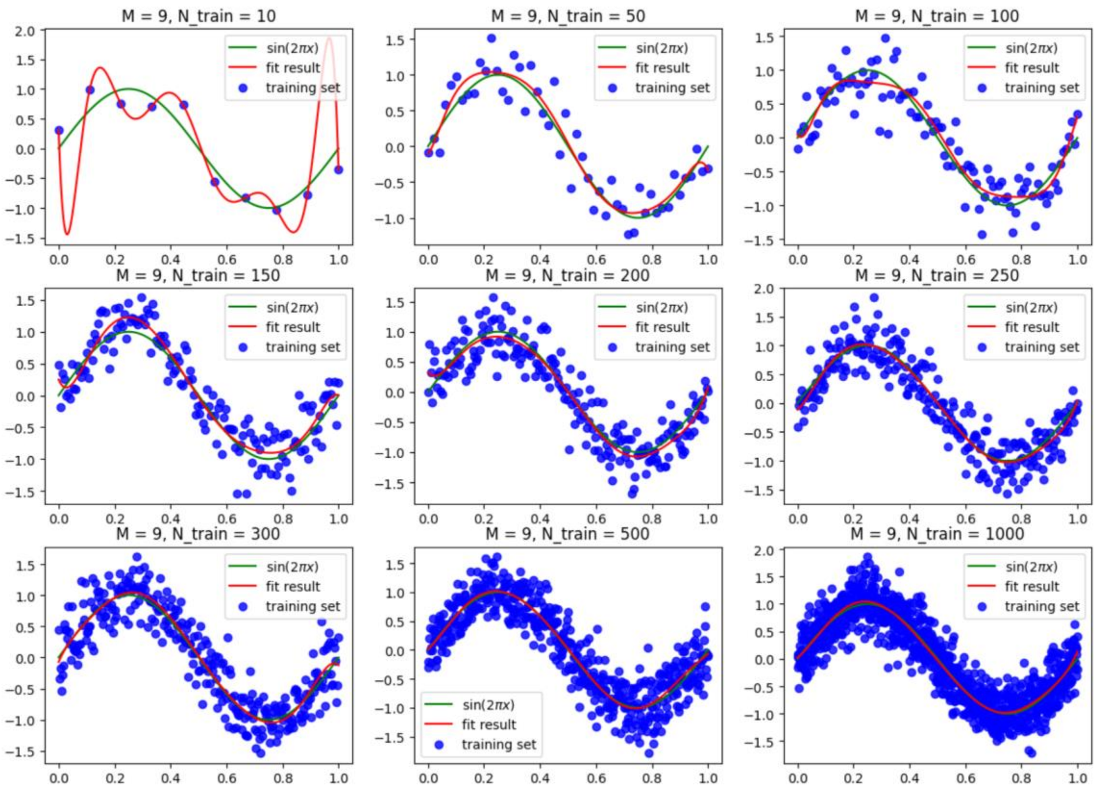
当多项式阶数m较小（m取$1\sim 2$）时，拟合效果较差。此时的模型学习能力不足，无法学习到数据集中的“一般规律”，导致泛化能力弱，被称为“欠拟合”。

当多项式阶数m适中（m取$3\sim 6$）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m较大（m取$7\sim 9$）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m相同，训练集大小N_{train}不同

选取多项式阶数m为9，再依次将训练集大小N_{train}取为10,50,100,150,200,250,300,500,1000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N_{train}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N_{train}和多项式阶数m相同，超参数λ不同

先引入对拟合优度的评价函数

$$E_{RMS}=\sqrt{\frac{2E(\boldsymbol{w}^*)}{N}}$$

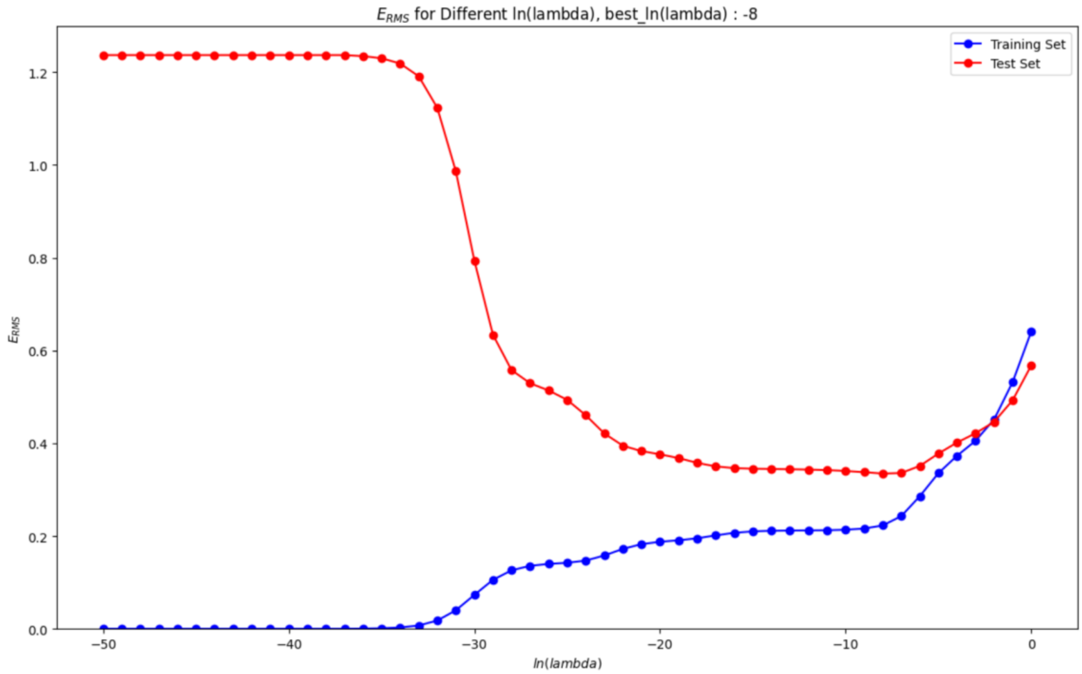
又因为已推导出

$$\lambda=\arg min_{\lambda} E(\boldsymbol{w},\lambda)$$

则可以用下式来代替

$$\lambda=\arg min_{\lambda} E_{RMS}$$

选取训练集大小N_{train}为10，多项式阶数m为9，手动给定一个λ的合适范围，在该范围内取值，分别求出对应的E_{RMS}，结果如下



可以发现：

当$\ln(\lambda)$取$-50\sim-33$时，测试集上E_{RMS}较大且几乎不变，训练集上E_{RMS}也几乎不变。此时λ过小，惩罚项比重小，模型退化为原模型。

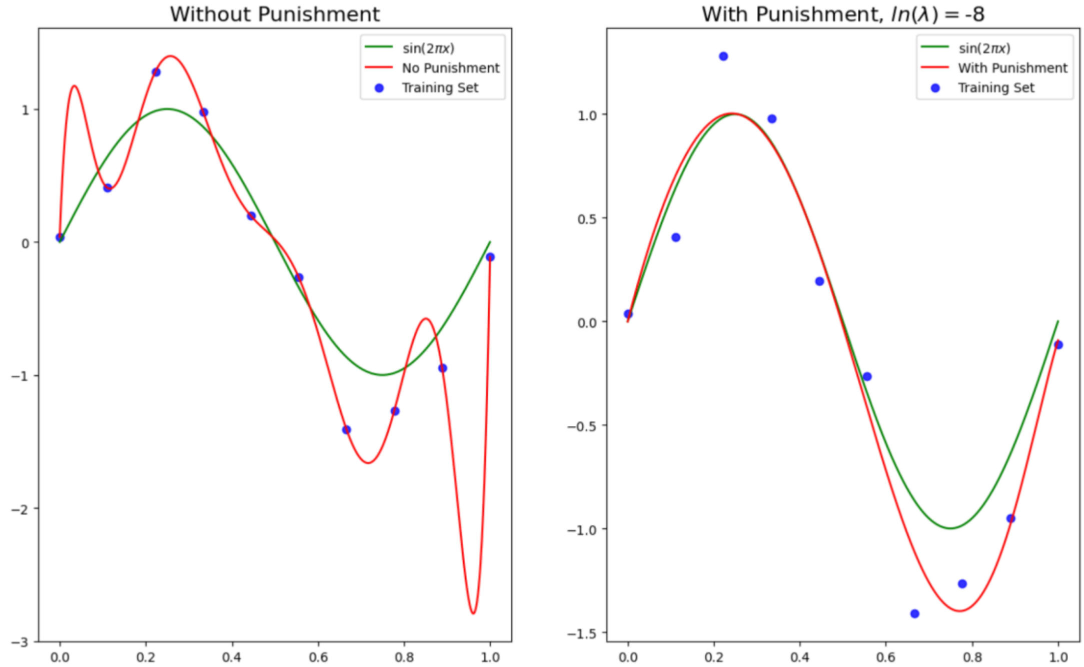
当$\ln(\lambda)$取$-32\sim-8$时，测试集上E_{RMS}开始下降，训练集上E_{RMS}有一定上升。此时λ较合适，惩罚项比重适当，模型复杂度与问题匹配。

当$\ln(\lambda)$取$-7\sim0$时，测试集上E_{RMS}又开始上升，训练集上E_{RMS}大幅上升。此时λ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λ都不一样，可以通过多次实验选择被判定为最佳λ次数最多的λ作为最终结果。通过500次实验，得到被判定为最佳λ次数最多的三个λ所对应的$\ln(\lambda)$，结果如下

```
[(-8, 116), (-9, 102), (-7, 90)]
```

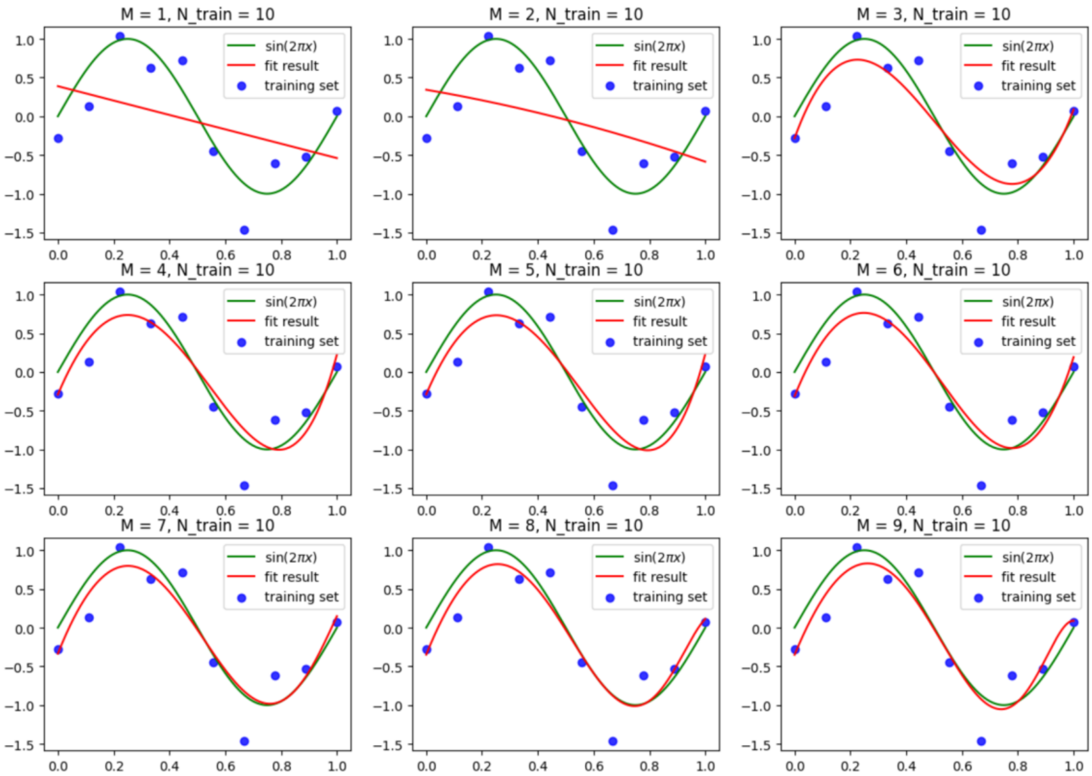
于是，得到最佳超参数$\lambda = e^{-8}$，接下来将训练集大小N_{train}为10，多项式阶数m为9，超参数λ为e^{-8}情况下的无惩罚项和带惩罚项的拟合结果进行对比，结果如下



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λ默认为e^{-8}。

2.训练集大小N_{train}和超参数λ相同，多项式阶数m不同

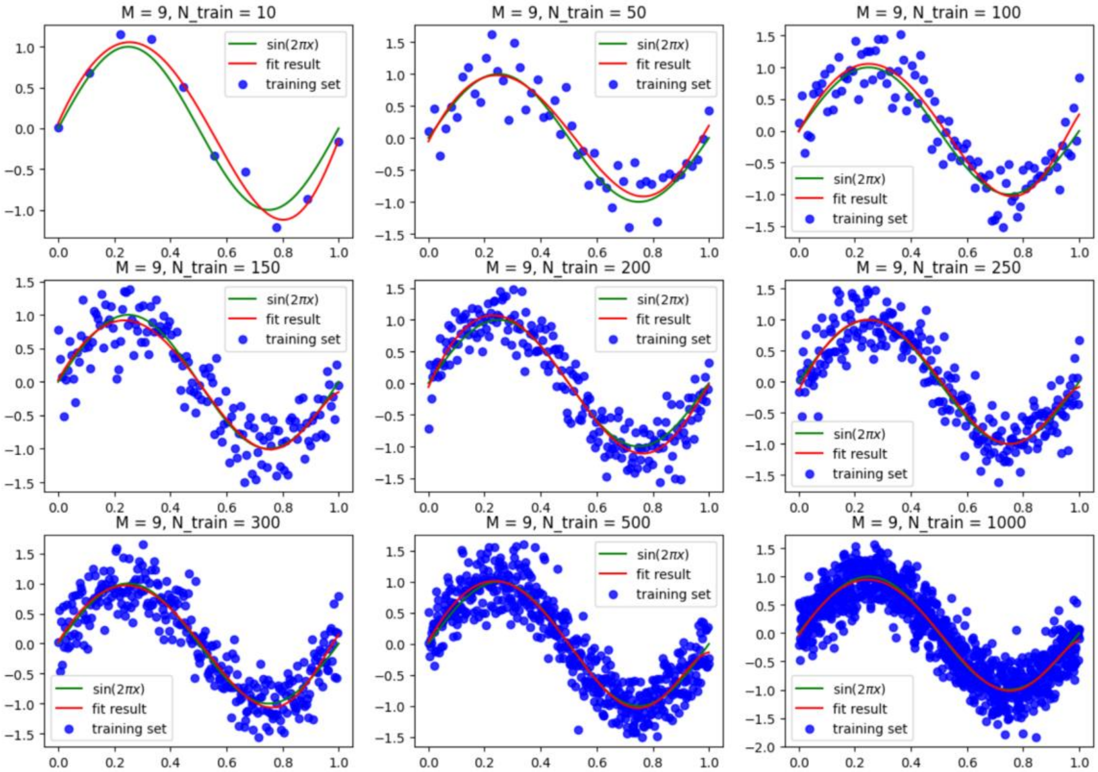
选取训练集大小N_{train}为10，超参数λ为e^{-8}，再依次将多项式阶数m取为$1,2,3,\ldots,9$进行拟合，结果如下



可以发现，由于惩罚项的加入，即使多项式阶数m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m和超参数λ相同，训练集大小N_{train}不同

选取多项式阶数m为9，超参数λ为e^{-8}，再依次将训练集大小N_{train}取为10,50,100,150,200,250,300,500,1000进行拟合，结果如下



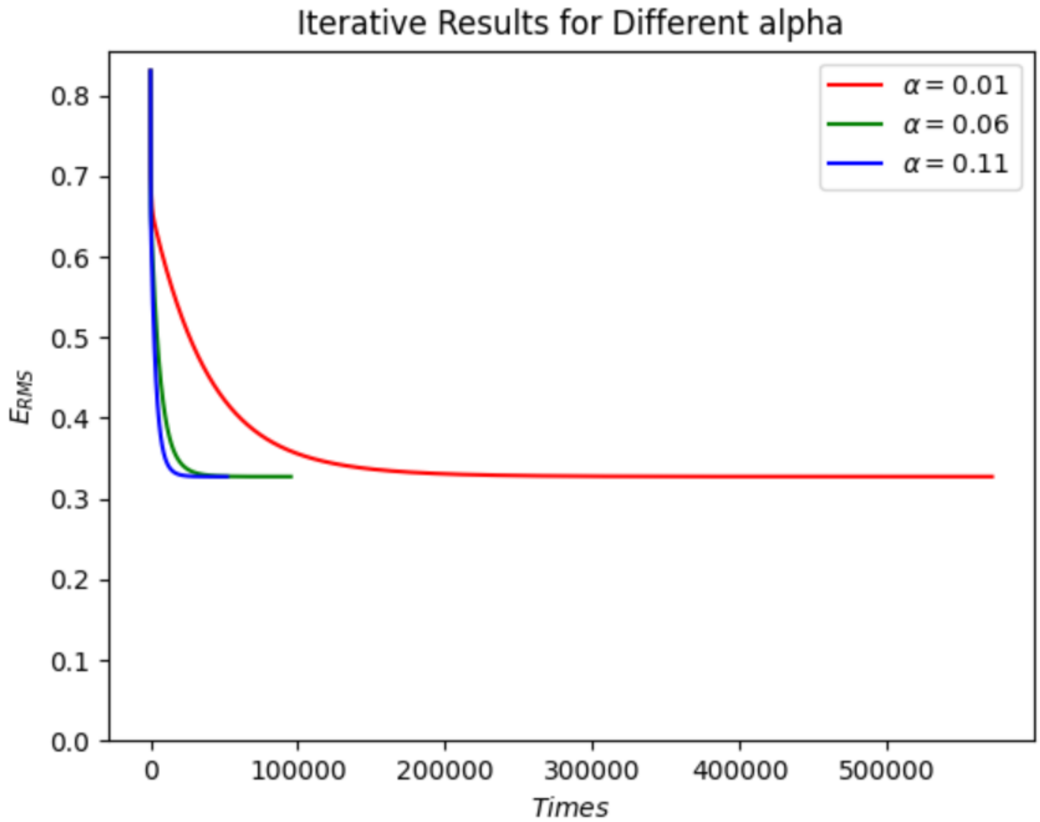
可以发现，由于惩罚项的加入，一开始就不存在过拟合现象了，随着训练集大小N_{train}的增大，拟合效果依然越来越好，越来越贴近真实的正弦曲线。

（四）梯度下降法

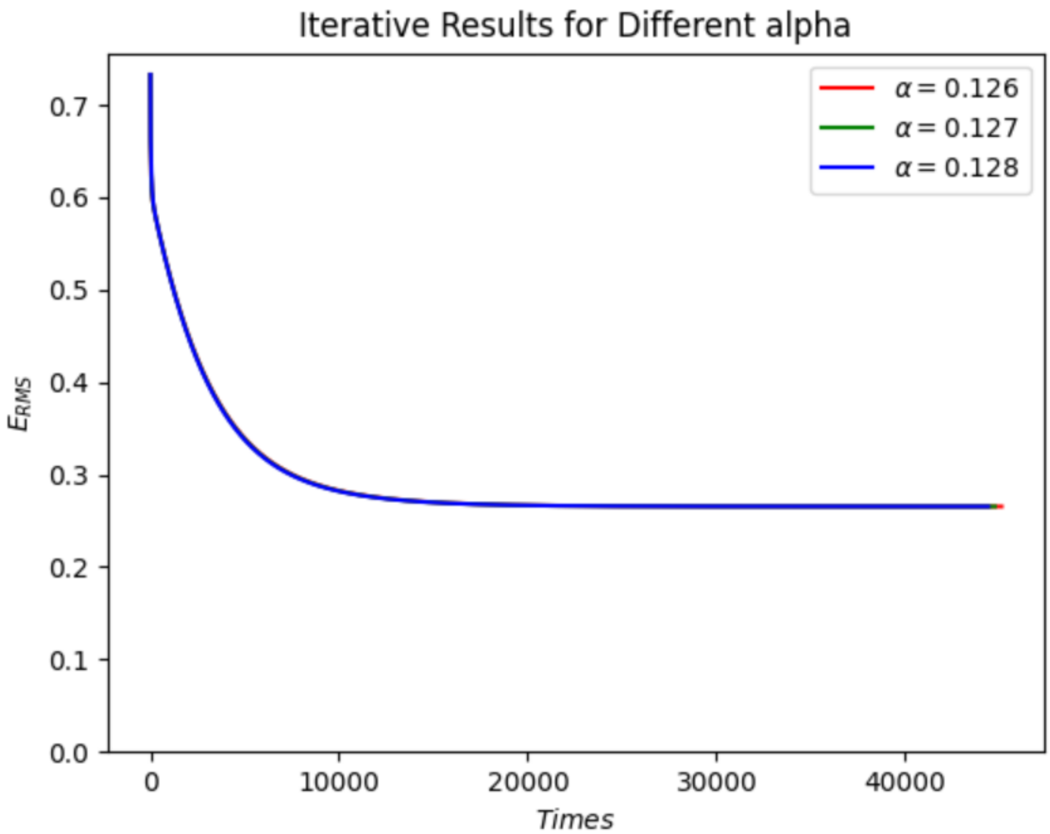
该方法的求解精度被设定为$\delta = 10^{-6}$。

1.训练集大小N_{train}和多项式阶数m相同，超参数α不同

选取训练集大小N_{train}为10，多项式阶数m为3，手动给定三个超参数α的合适值，分别求出对应的E_{RMS}，结果如下

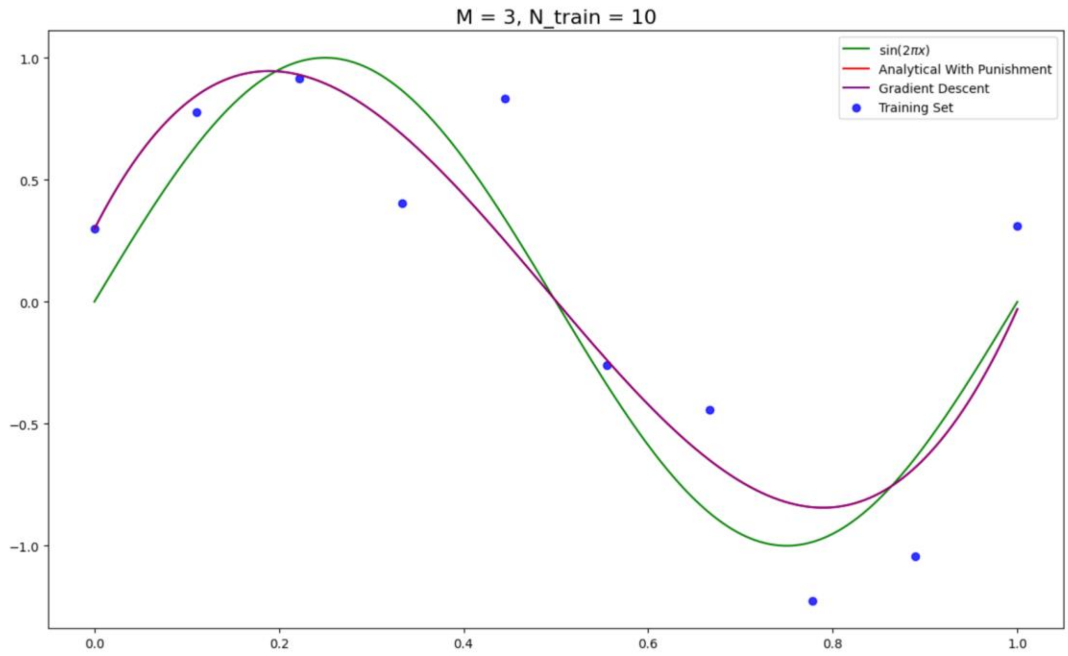


要找到使$\nabla E(\boldsymbol{w})$收敛到$\boldsymbol{0}$的尽量大的α，需要继续手动调参，直至



此时的三个超参数α对应的图像已极其接近且再增大就无法收敛，于是得到

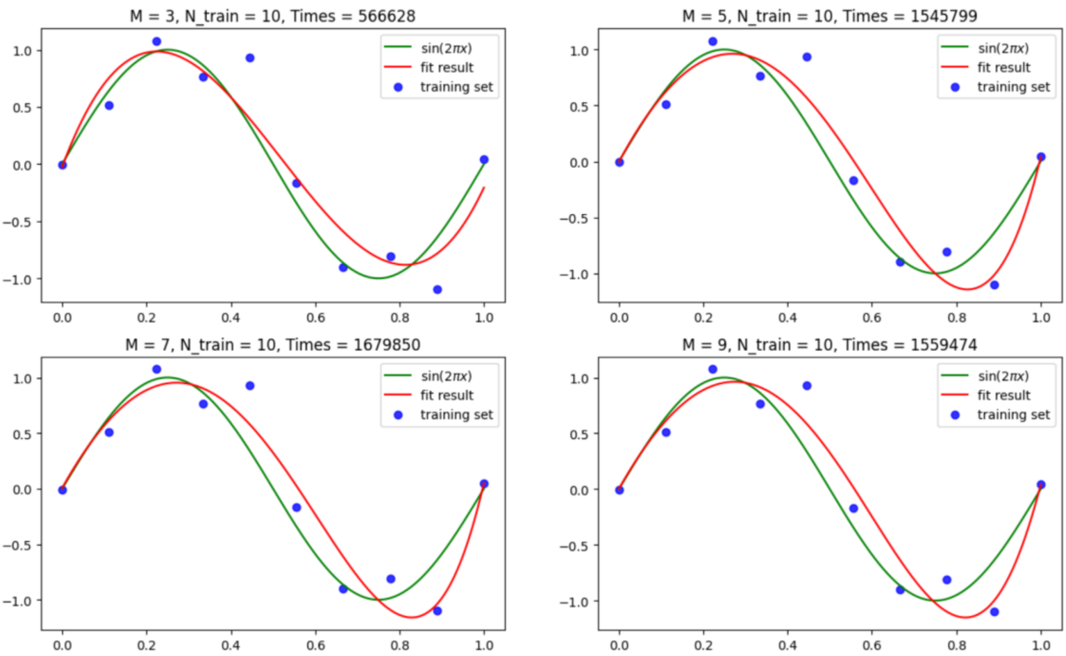
最佳超参数$\alpha=0.128$。接下来将训练集大小N_{train}为10，多项式阶数m为3，超参数λ为e^{-8}，超参数α为0.128情况下的带惩罚项最小二乘法和梯度下降法的拟合结果进行对比，结果如下



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N_{train}和超参数α相同，多项式阶数m不同

选取训练集大小N_{train}为10，超参数α为0.01，再依次将多项式阶数m取为3,5,7,9进行拟合，结果如下

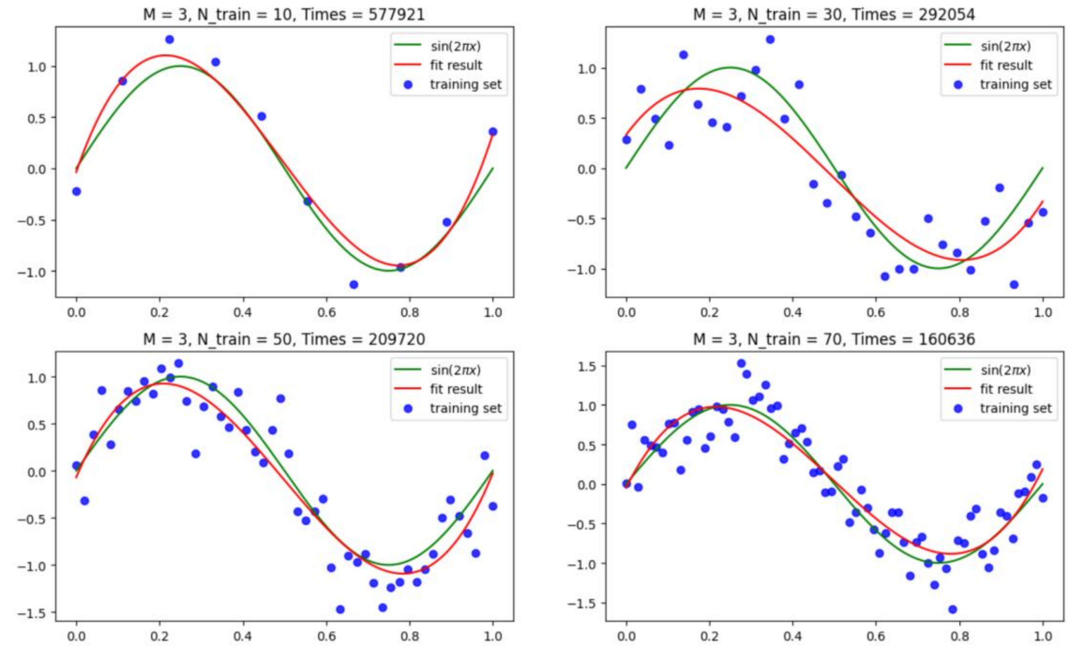


可以发现，由于梯度下降法里有惩罚项的加入，同带惩罚项的最小二乘法一样，即使多项式阶数m增大也不会出现过拟合现象，再次证明增加比重恰当的惩

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m并无明显关系。

3. 多项式阶数m和超参数α相同，训练集大小N_{train}不同

选取多项式阶数m为3，超参数α为0.01，再依次将训练集大小N_{train}取为10,30,50,70进行拟合，结果如下



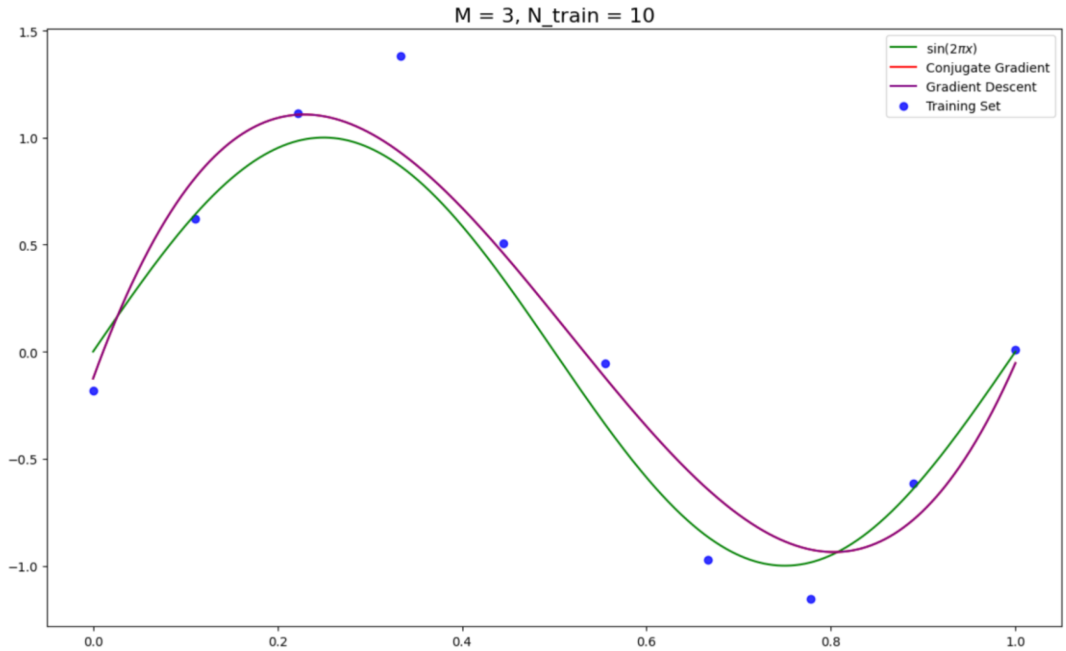
可以发现，随着训练集大小N_{train}的增大，拟合效果越来越好，越来越贴近真实的正弦曲线。还需要注意的是，随着训练集大小N_{train}的增大，迭代次数减少。

（五）共轭梯度法

该方法的求解精度被设定为$\delta = 10^{-4}$。

1. 共轭梯度法和梯度下降法的对比

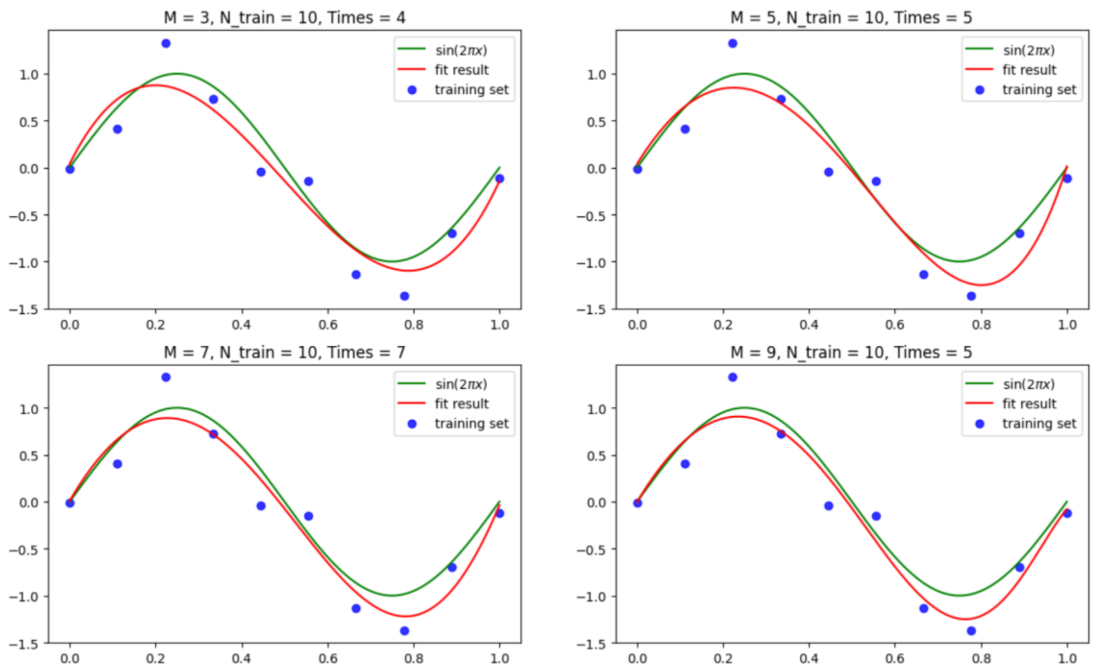
将训练集大小N_{train}为10，多项式阶数m为3，超参数α为0.128情况下的共轭梯度法和梯度下降法的拟合结果进行对比，结果如下



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N_{train}相同，多项式阶数m不同

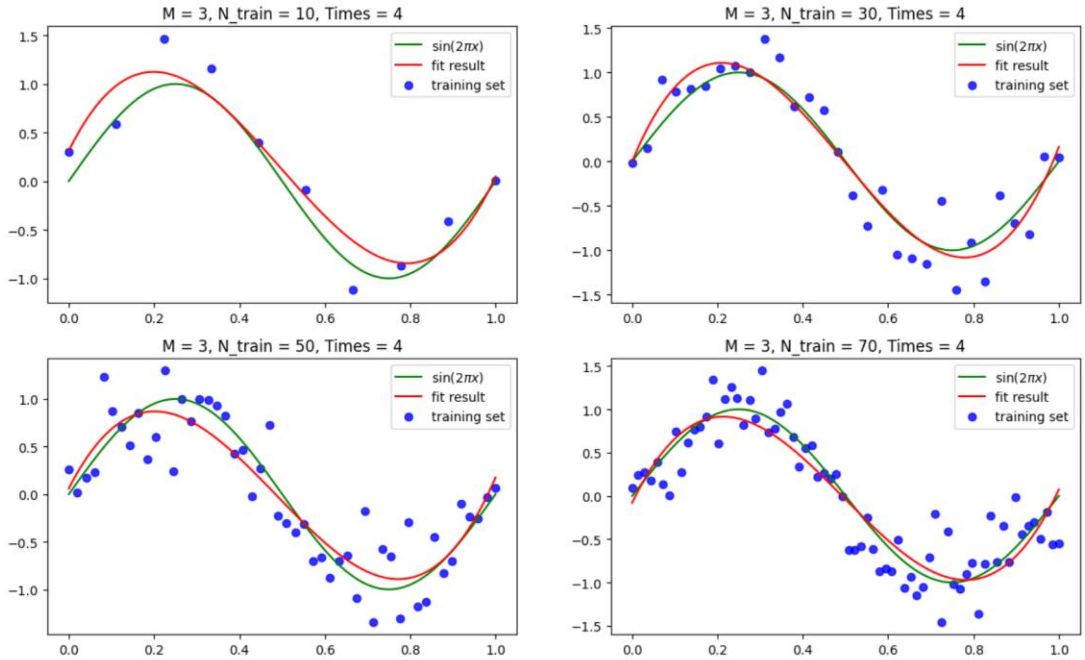
选取训练集大小N_{train}为10，再依次将多项式阶数m取为3,5,7,9进行拟合，结果如下



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m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N维空间里迭代次数最多达到N次。

3. 多项式阶数m相同，训练集大小N_{train}不同

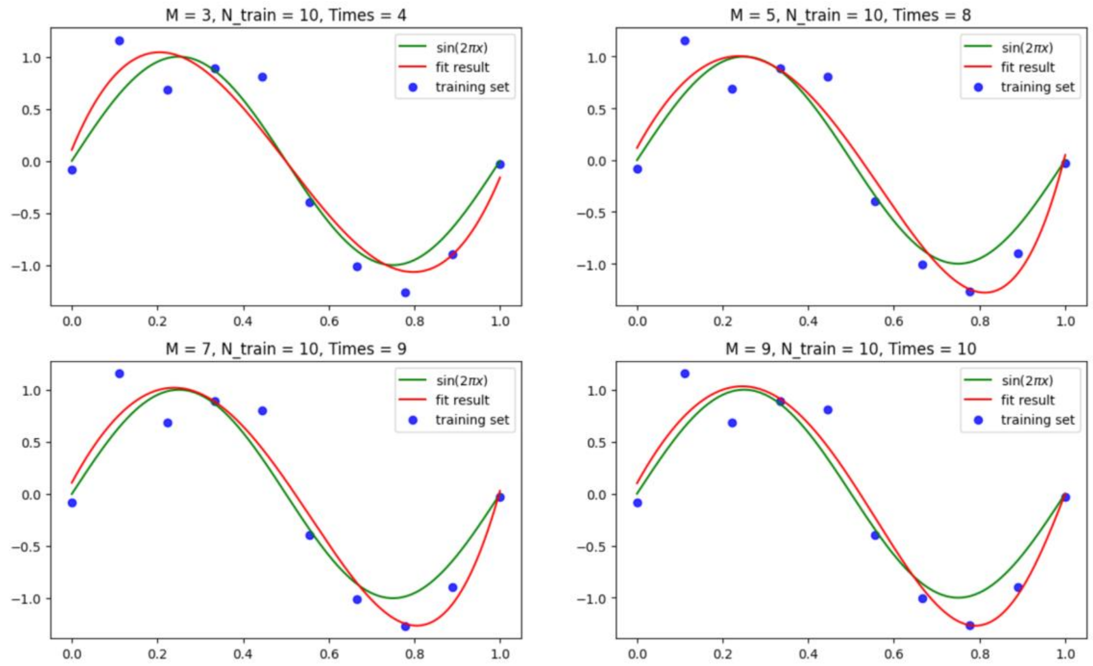
选取多项式阶数m为3，再依次将训练集大小N_{train}取为10,30,50,70进行拟合，结果如下



可以发现，随着训练集大小N_{train}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，4维空间里迭代次数最多达到4次。

补充：

共轭梯度法的迭代次数并非一定能满足N维空间里不多于N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N次，但实际上最多在迭代次数为第N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$\delta = 10^{-6}$，选取训练集大小N_{train}为10，再依次将多项式阶数m取为3,5,7,9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m取为5和7进行拟合时的迭代次数分别超过了6和8。

五、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N维空间里求解时，共轭梯度法的迭代次数最多达到N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。

六、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)
```

```python
    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)

```

```python
    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

```python
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()
```