哈尔滨工业大学计算机科学与技术学院

机器学习实验报告

姓名	徐亚楠
学号	1190201224
实验题目	多项式拟合曲线
报告时间	2021.10.1

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：

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。

实验环境：

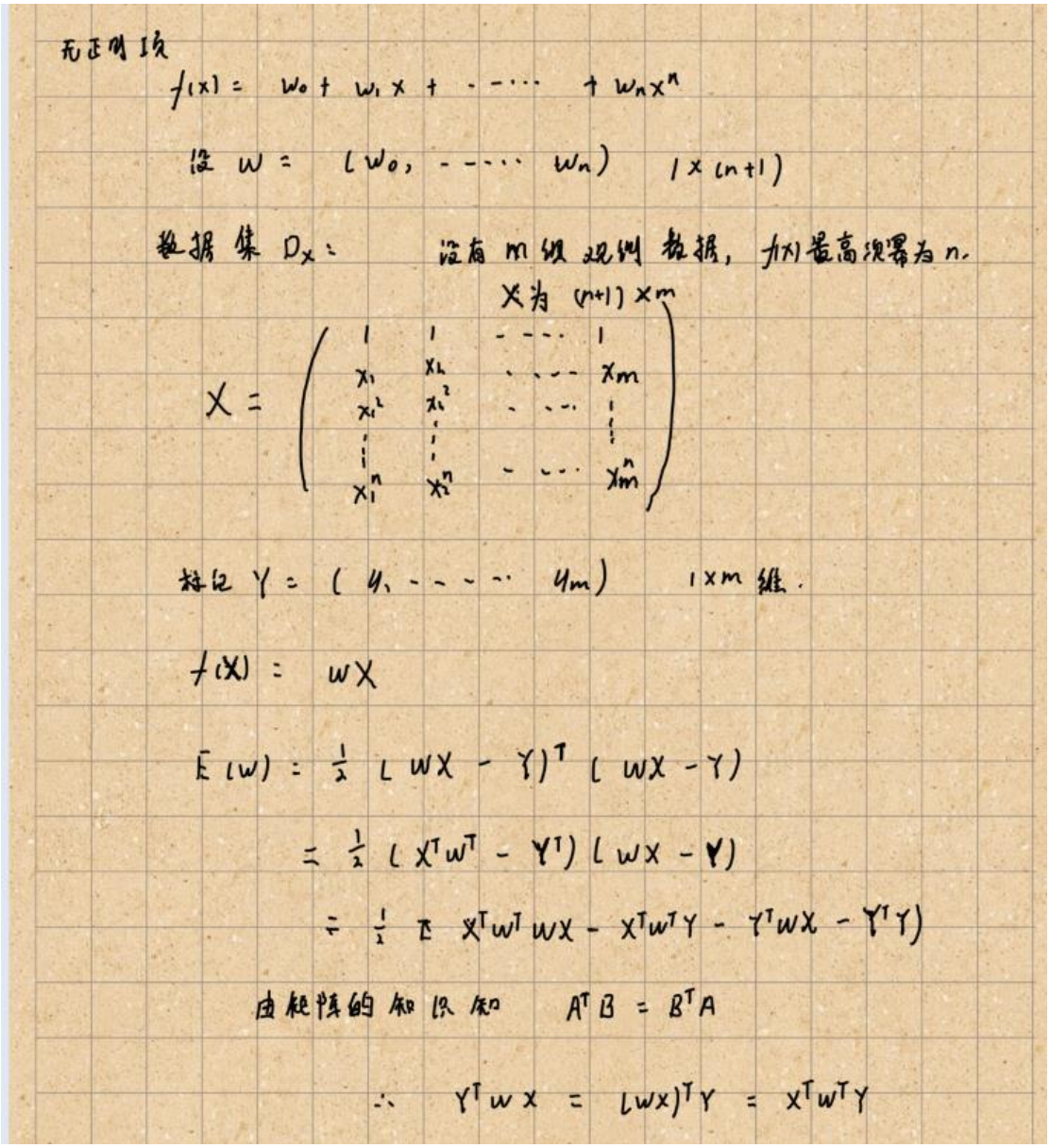
Pycharm 2021　windows10　python3.8

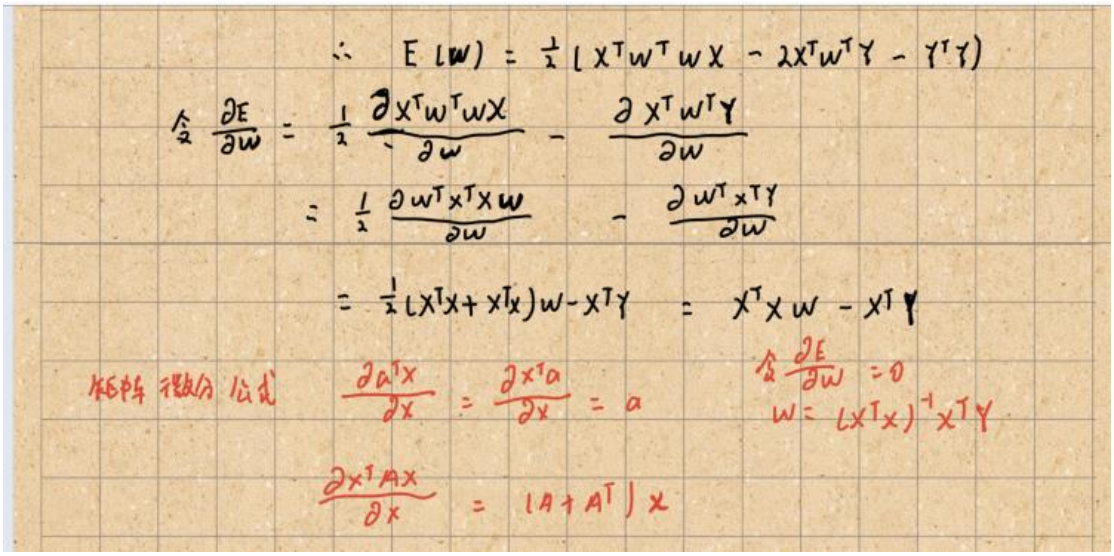
三、设计思想（本程序中的用到的主要算法及数据结构）：

泰勒公式，是一个用函数在某点的信息描述其附近取值的公式。如果函数满足一定的条件,泰勒公式可以用函数在某一点的各阶导数值做系数构建一个多项式来近似表达这个函数。

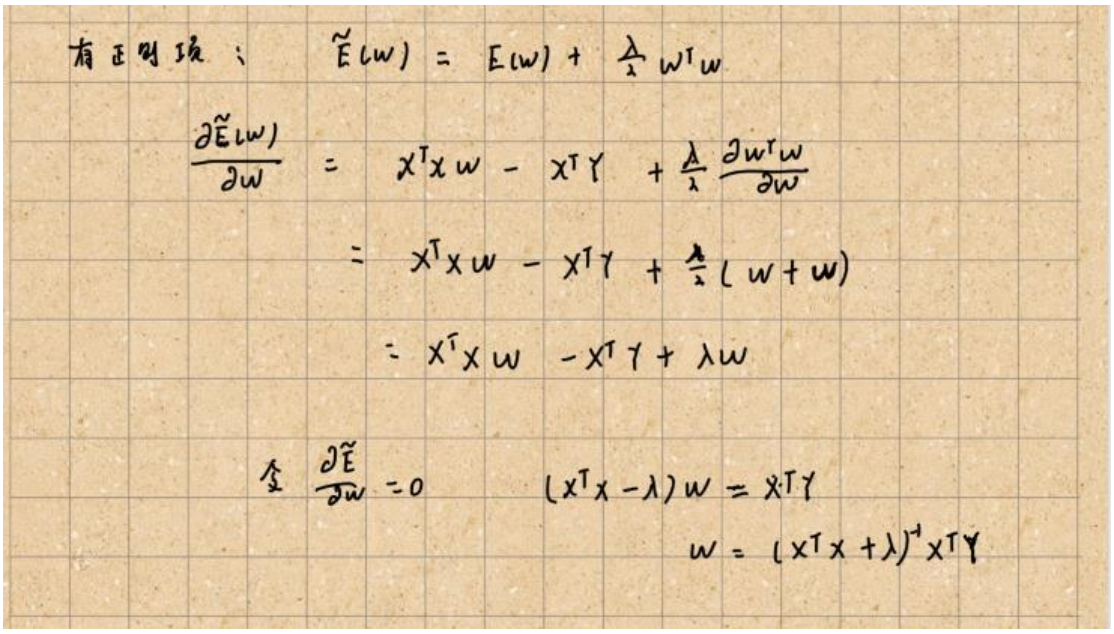
数学推导：

最小二乘法求 w 无正则项：

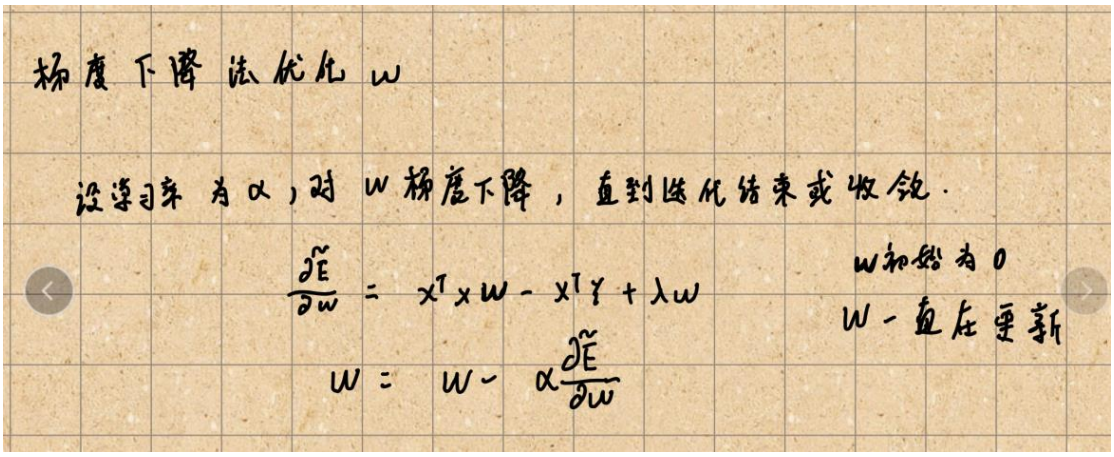




最小二乘法求 w 有正则项：



梯度下降法：



共轭梯度法：

在解大型线性方程组的时候，很少会有一步到位的精确解析解，一般都需要通过迭代来进行逼近，而 PCG 就是这样一种迭代逼近算法。
先从一种特殊的线性方程组的定义开始，比如我们需要解如下的线性方程组：Ax=b，这里的 A(n×n)是对称，正定矩阵， b(n×1)同样也是已知的列向量，我们需要通过 A 和 b 来求解 x(n×1)。

直接求解：

我们定义满足如下关系的向量为关于 矩阵 **A** 的共轭向量。

$$u^TAv=0$$

因为矩阵 A 是对称正定矩阵，所以矩阵 A 定义了一个内积空间：

$$\langle u,v\rangle_A:=\langle Au,v\rangle=\langle u,A^Tv\rangle=\langle u,Av\rangle=u^TAv$$

基于此，我们可以定义一组向量 $P=\{p_1,...,p_n\}$

其中的向量 $p_1,...,p_n$ 都是互为共轭的，那么 P 构成了 R_n 空间的一个基，上述方程的

解 x_* 可以表示成 P 中向量的线性组合：

$$\mathbf{x}_* = \sum_{i=1}^{n} \alpha_i \mathbf{p}_i$$

根据上面的表达式，我们可以得到：

$$\mathbf{A}\mathbf{x}_* = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{p}_i$$

$$\mathbf{p}_k^{\mathsf{T}} \mathbf{A}\mathbf{x}_* = \sum_{i=1}^{n} \alpha_i \mathbf{p}_k^{\mathsf{T}} \mathbf{A}\mathbf{p}_i \quad \text{(Multiply left by } \mathbf{p}_k^{\mathsf{T}}\text{)}$$

$$\mathbf{p}_k^{\mathsf{T}} \mathbf{b} = \sum_{i=1}^{n} \alpha_i \langle \mathbf{p}_k, \mathbf{p}_i \rangle_{\mathbf{A}} \qquad (\mathbf{A}\mathbf{x}_* = \mathbf{b} \text{ and } \langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{A}} = \mathbf{u}^{\mathsf{T}} \mathbf{A} \mathbf{v})$$

$$\langle \mathbf{p}_k, \mathbf{b} \rangle = \alpha_k \langle \mathbf{p}_k, \mathbf{p}_k \rangle_{\mathbf{A}} \qquad (\mathbf{u}^{\mathsf{T}} \mathbf{v} = \langle \mathbf{u}, \mathbf{v} \rangle \text{ and } \forall i \neq k : \langle \mathbf{p}_k, \mathbf{p}_i \rangle_{\mathbf{A}} = 0)$$

所以：

$$\alpha_k = \frac{\langle \mathbf{p}_k, \mathbf{b} \rangle}{\langle \mathbf{p}_k, \mathbf{p}_k \rangle_{\mathbf{A}}}$$

迭代求解：

1. 初始化,x(k)表示第 k 次迭代的解向量。d(k)表示第 k 次迭代的方向向量。r(k)表示第 k 次迭代的残差向量。直至残差满足精度。

$$x(0) = 0, d(0) = 0, r(0) = -b_n$$

进行第 k 次迭代，主要分为四个步骤

1. 计算残差向量：

$$r(k) = Ax(k-1) - b$$

2. 计算方向向量：

$$d(k) = -r(k) + \frac{r^{\mathrm{T}}(k) r(k)}{r^{\mathrm{T}}(k-1) r(k-1)} d(k-1)$$

3. 计算步长：

$$\alpha(k) = -\frac{d^{\mathrm{T}}(k) r(k)}{d^{\mathrm{T}}(k) A d(k)}$$

4. 更新解向量：

$$x(k) = x(k-1) + \alpha(k) d(k)$$

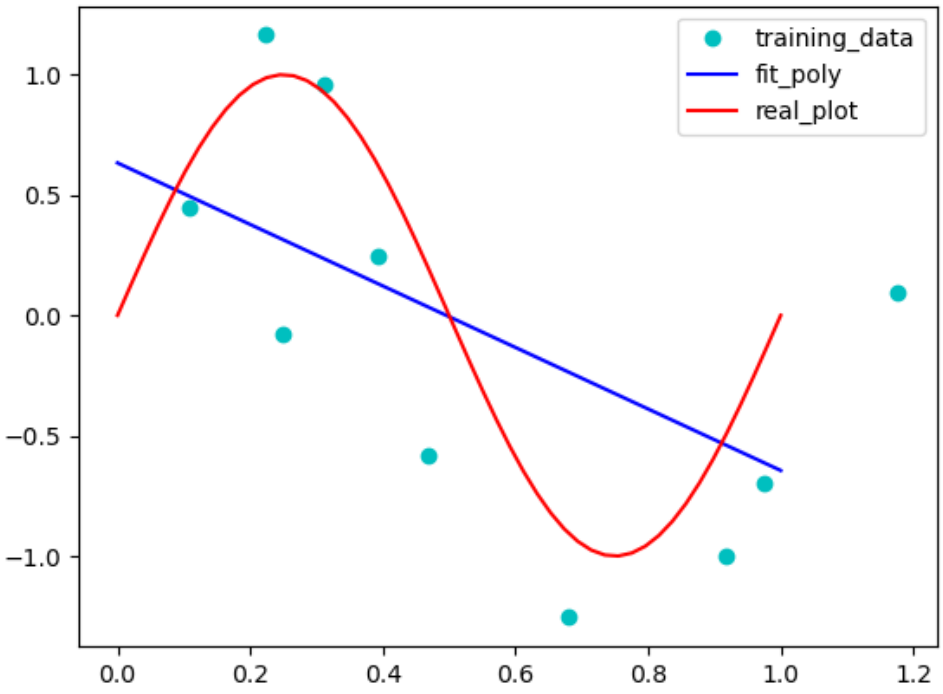
四、实验结果与分析：

4.1 不带正则项的解析解：固定训练集大小为 10

1. 多项式一阶：

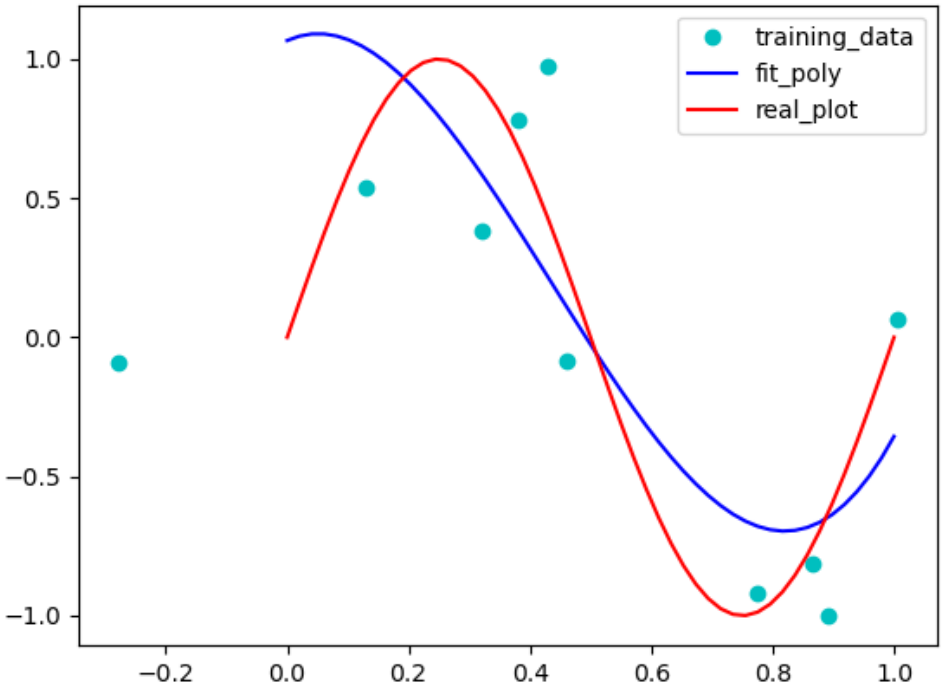
```
1.  begin, end, exponent, lamda = 0, 1, 1, 0
2.  train_x, train_y, x,y=DataMaker.generate_data(exponent,10,lamda,1)
3.  poly = DataMaker.lsm(train_x, train_y,0,exponent)
4.  DataMaker.plt_show(x,y,poly)
```

Y=-1.278 x + 0.6333



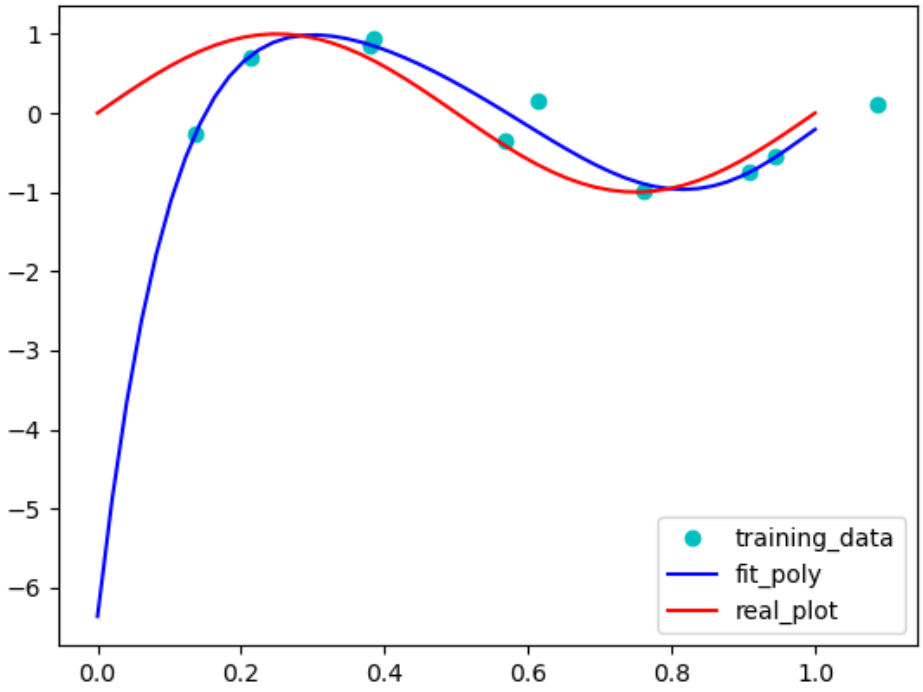
2. 多项式三阶：

Y=7.863 x^3 - 10.27 x^2+ 0.9853 x + 1.066



3. 多形式六阶：

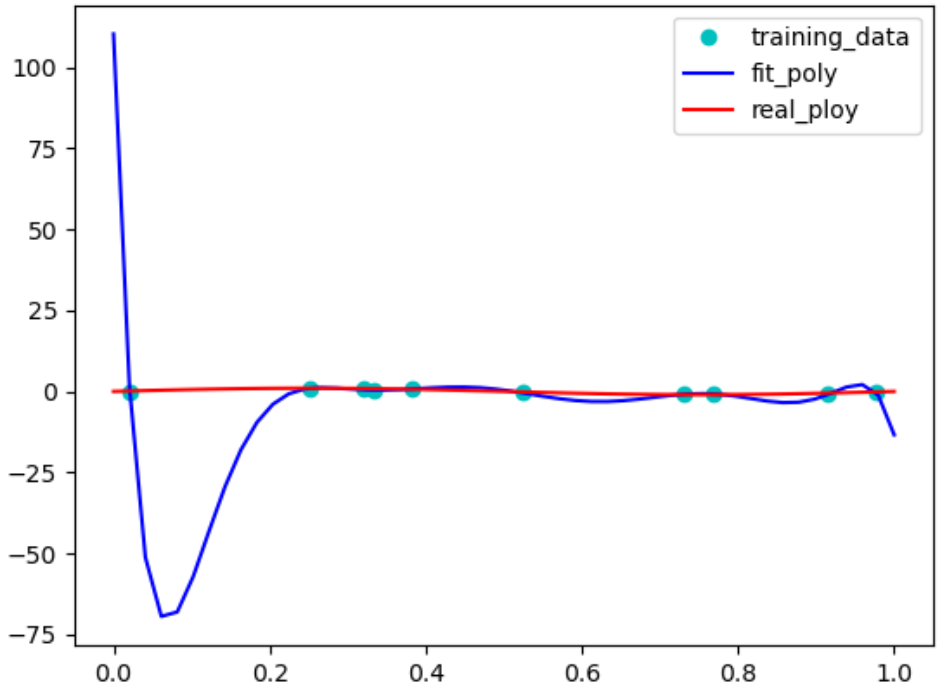
Y=-211.3 x^6 + 749.2 x^5 - 1062 x^4 + 795.9 x^3 - 344.6 x^2 + 79.38 x - 6.365



4. 多形式九阶：

Y= -2.306e+04X^9 + 1.033e+05X^8 - 1.852e+05X^7+ 1.659e+05X^6 -

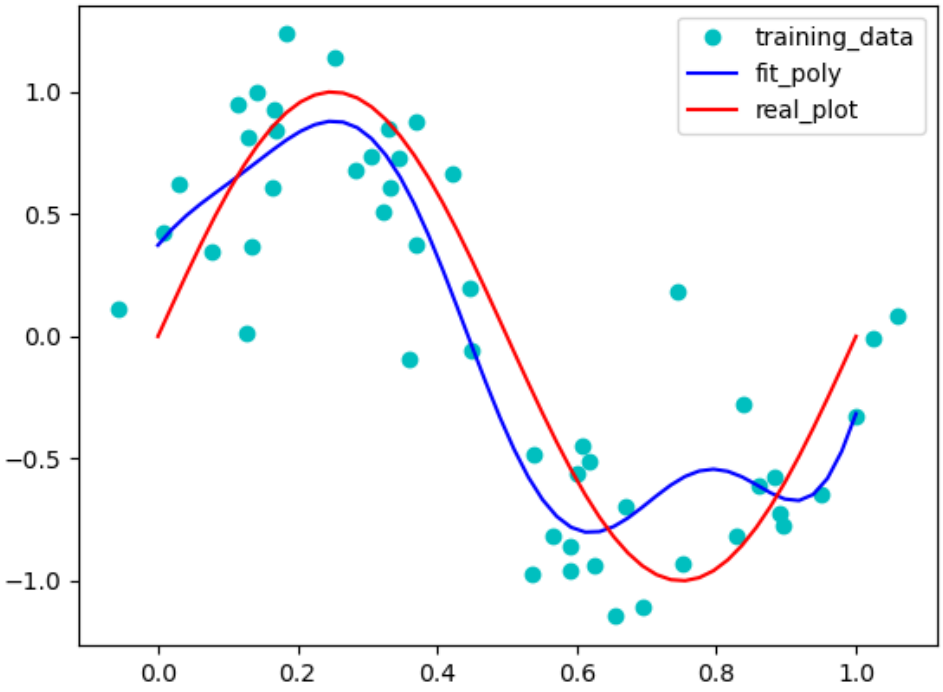
7.272e+04X^5+ 9400X^4 + 3340X^3 - 1019X^2 + 23.44X^ + 7.266



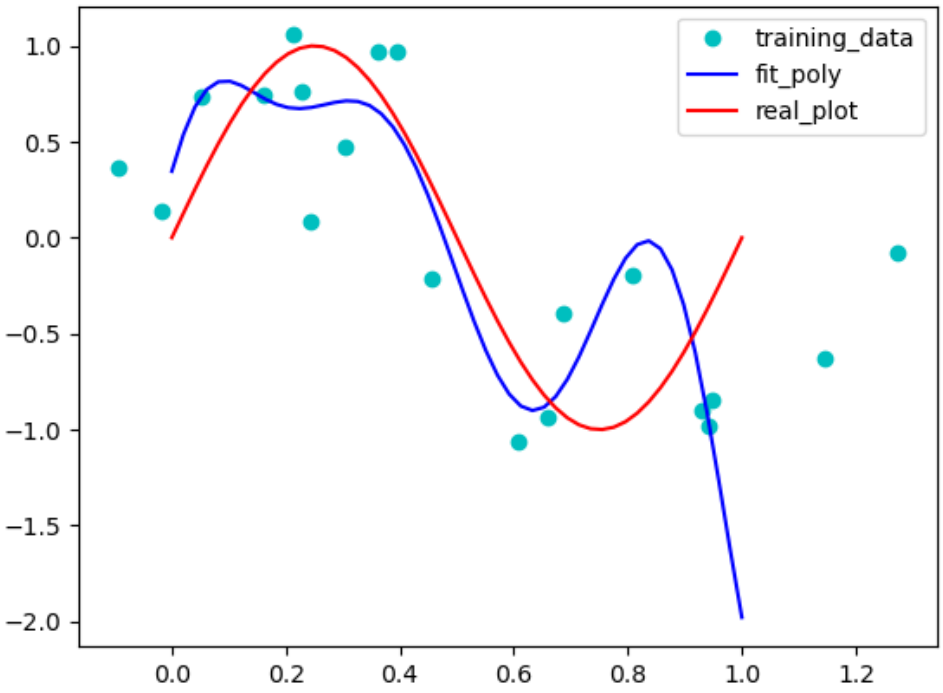
在 1 阶的时候，基本无法拟合，属于欠拟合，提高阶数，3 阶的情况下拟合效果好了些，继续提高阶数，到 9 阶时拟合函数图像完美的穿过了训练集上的大部分点，但是由于 machine 过于关注于“尽量穿过训练集上所有的点”这一条件，导致函数的波动较大，反而难以拟合数据集所代表的函数 Sin2Πx。

我们通过增加数据集降低阶数过高带来的过拟合现象的影响：

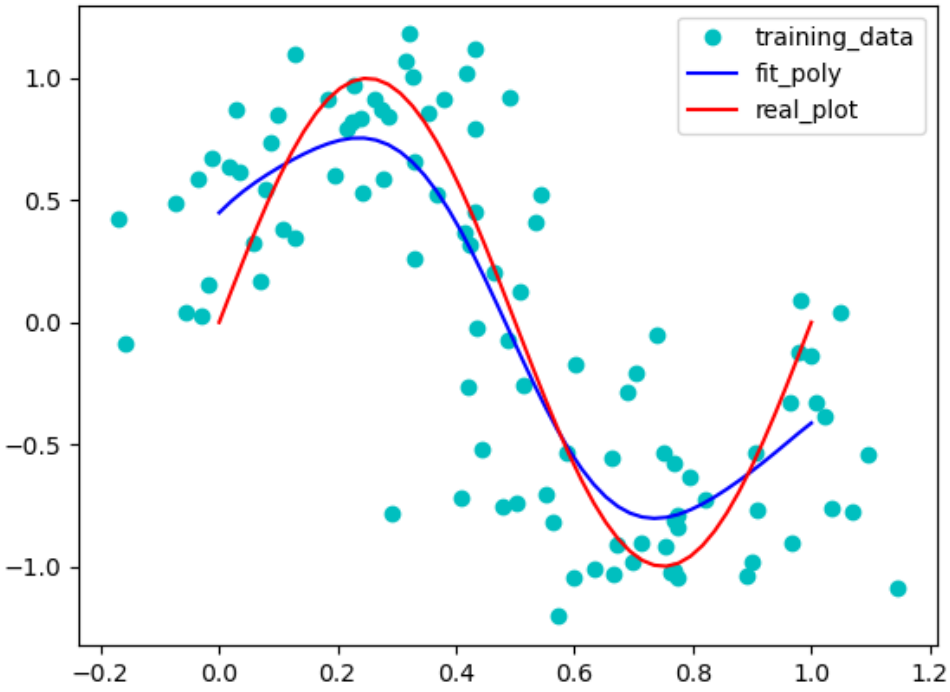
训练集大小 20：



训练集大小 50：



训练集大小 100：

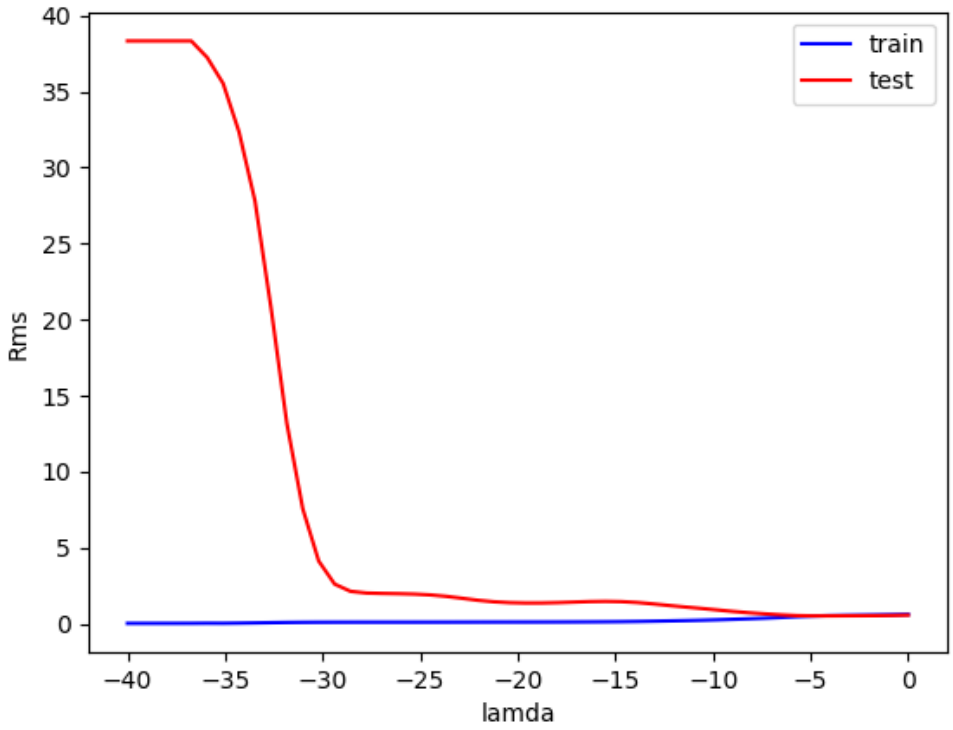


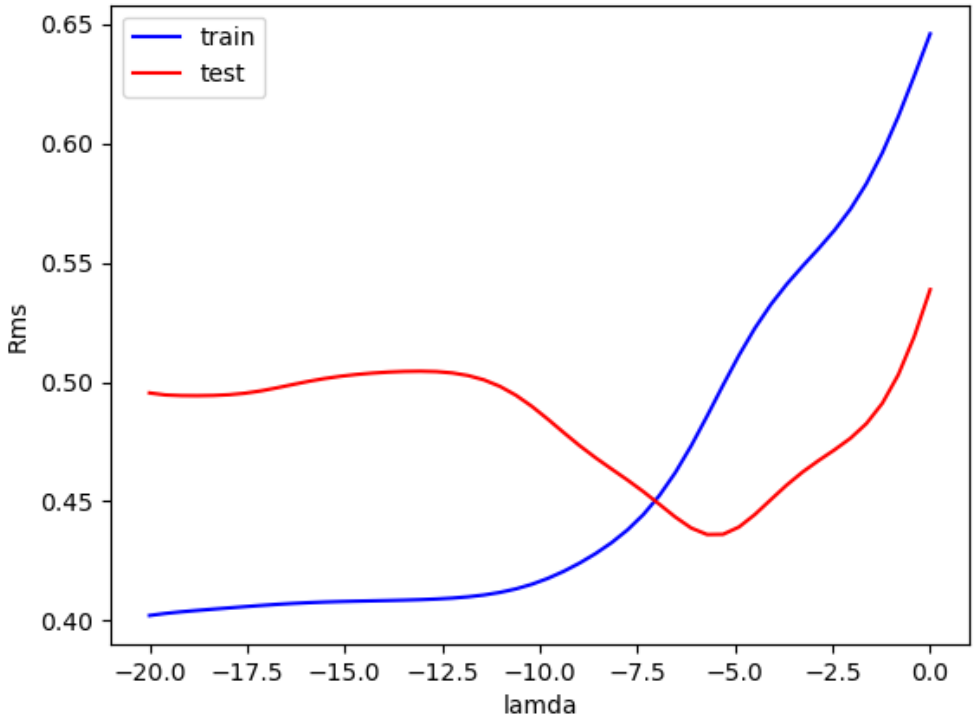
从上面几张图可以看出，在训练集大小为 10 时，过拟合现象最为严重，随着训练集的增大，过拟合现象逐渐被消除，当训练集大小为 100 时，我们生成的拟合函数逐渐靠近原数据点的函数上。

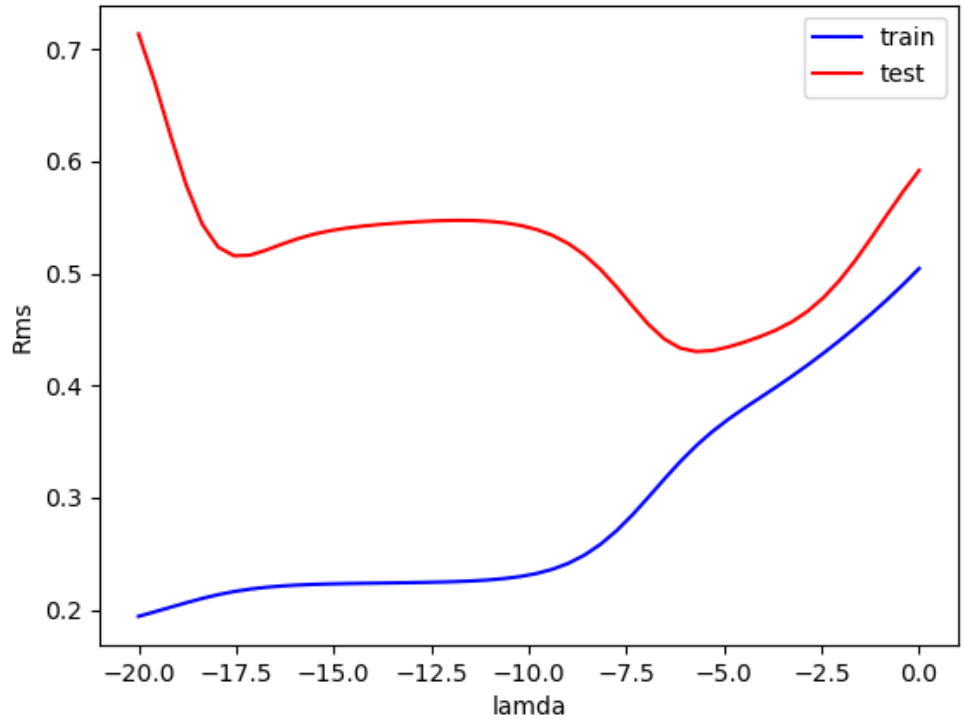
4.2 带正则项的解析解：固定训练集大小为 10 解析法

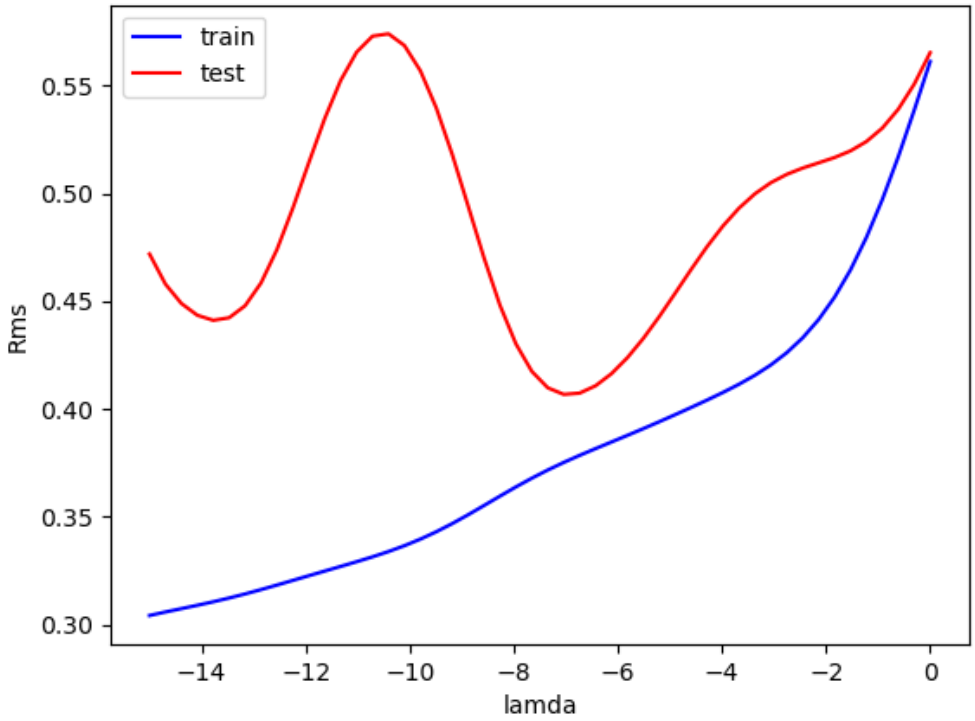
我们需要确定一个最优的 lamda，使用均方根来判断不同参数取值的拟合效果。

$$E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^{\star})/N}$$


train
test
Rms
lamda
40
35
30
25
20
15
10
5
0
−40
−35
−30
−25
−20
−15
−10
−5
0


train
test
Rms
lamda
0.65
0.60
0.55
0.50
0.45
0.40
−20.0
−17.5
−15.0
−12.5
−10.0
−7.5
−5.0
−2.5
0.0

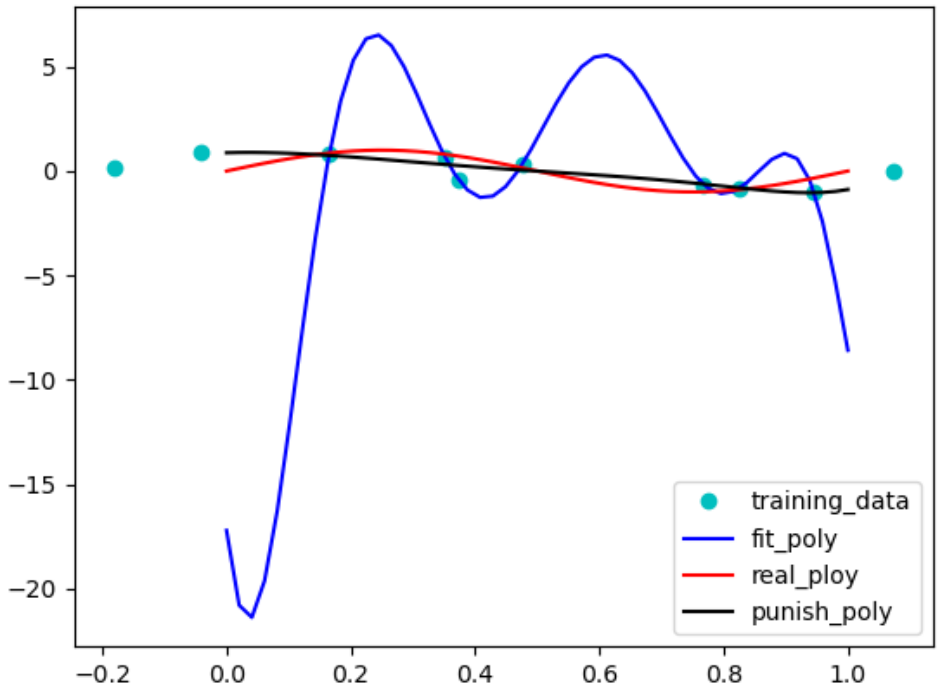


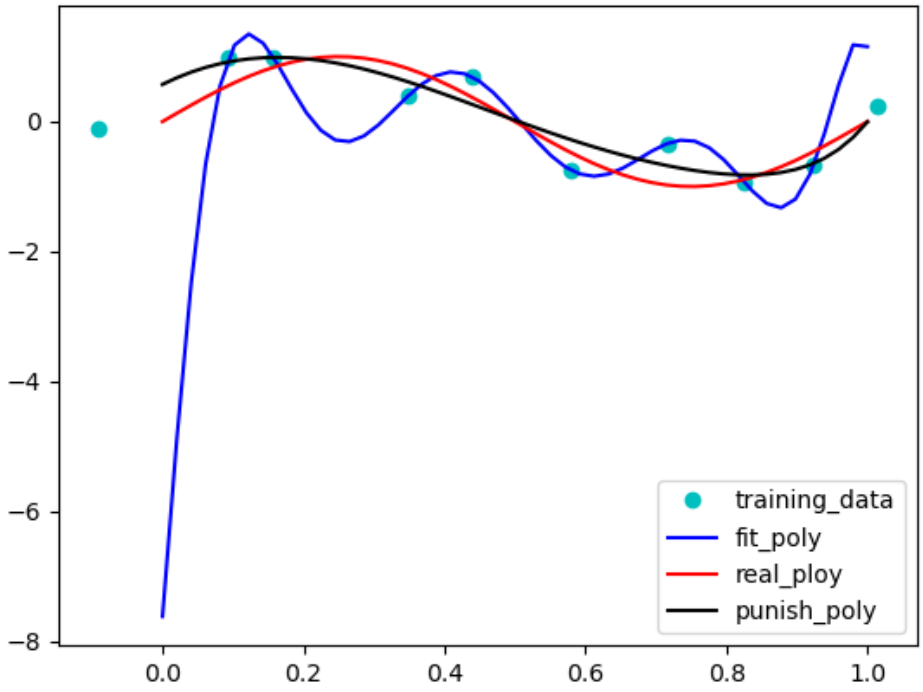


通过多次运行的结果可以看出，随着 lamda 取值的减小，拟合函数在训练集上的

错误率逐渐减小，在(e^{-40} , e^{-30})区间内的错误率几乎为 0，而在验证集上，在 lamda<e^{-7} 时，随着取 lamda 值的减小，拟合函数的错误率逐渐减小，但是在 lamda<e^{-15} 之后的验证集上的错误率开始上升，说明出现了过拟合现象。 根据多次运行的结果可得最佳的 lamda 取值范围大约为(e^{-7},e^{-11})。

取 lamda= e^{-10}, 在训练集大小为 10，阶数为 9 的条件下的带惩罚项和不带惩罚项的拟合图像比较：

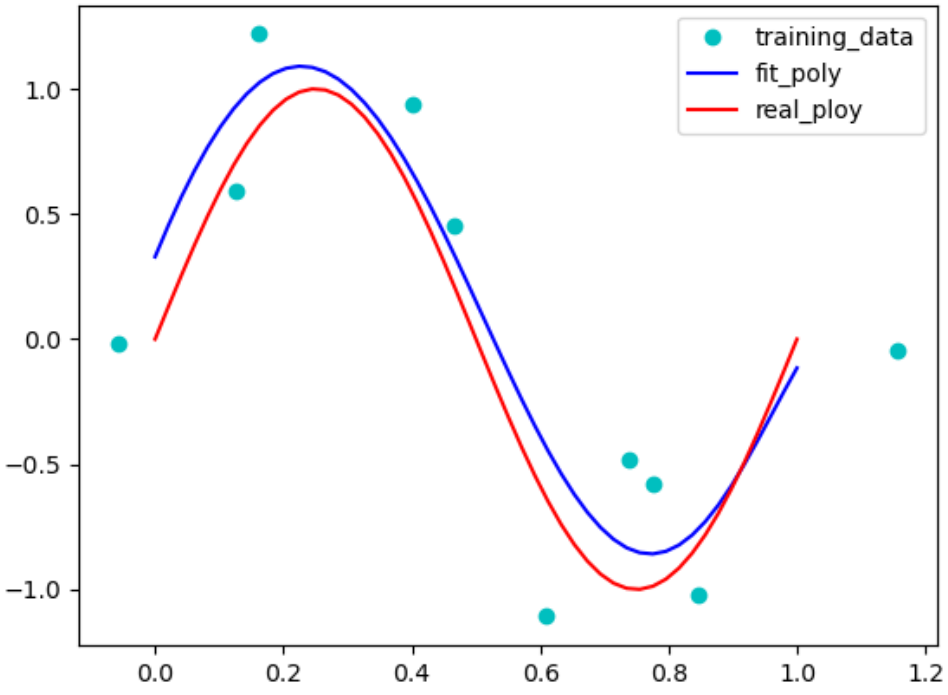


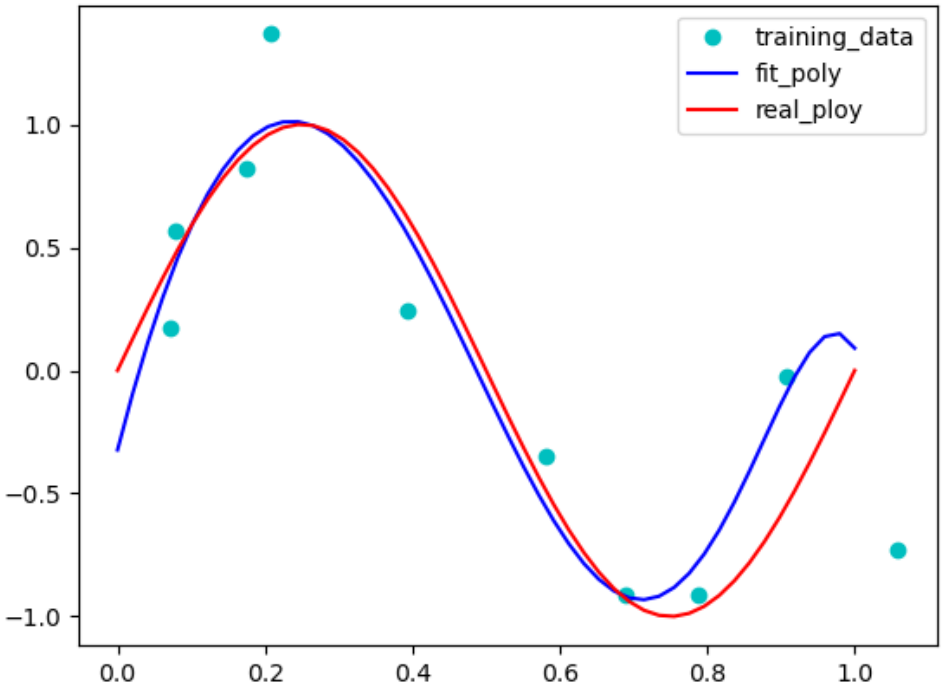


可以看出有了正则项很大程度上抑制了过拟合。

取 lamda=e^{-10}

训练集大小 10：





4.3 梯度下降求得优化解

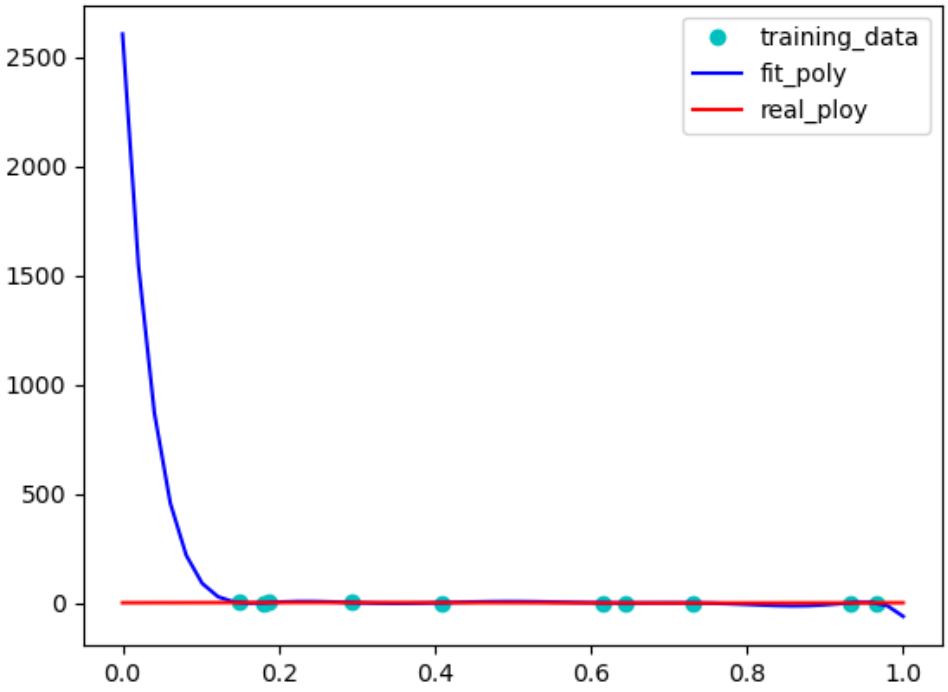
取学习率为 0.01， lamda=0 epilson=10^{-6}

多项式 9 阶

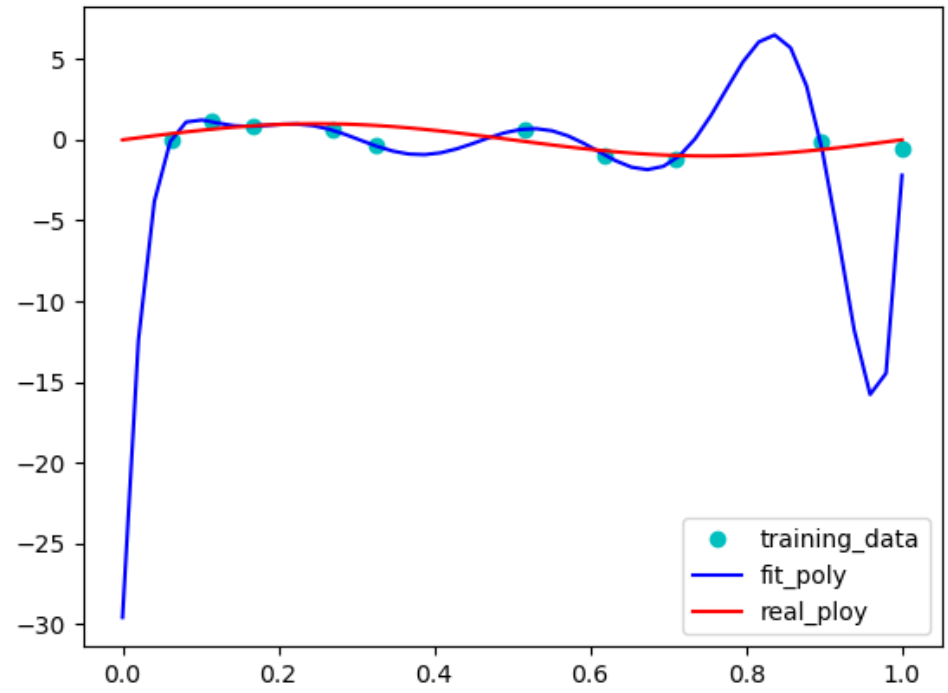
1. 给定待优化连续可微函数 $J(\Theta)$、学习率 α 以及一组初始值 $\Theta_0 = (\theta_{01}, \theta_{02}, \cdots, \theta_{0l},)$
2. 计算待优化函数梯度： $\nabla J(\Theta_0)$
3. 更新迭代公式： $\Theta^{0+1} = \Theta_0 - \alpha \nabla J(\Theta_0)$
4. 计算 Θ^{0+1} 处函数梯度 $\nabla J(\Theta_{0+1})$
5. 计算梯度向量的模来判断算法是否收敛： $\|\nabla J(\Theta)\| \leqslant \varepsilon$
6. 若收敛，算法停止，否则根据迭代公式继续迭代

```
1.  if (old_loss - new_loss < epsilon)&(np.linalg.norm(gradient_w)<=epsilon) :
```

跑十万次

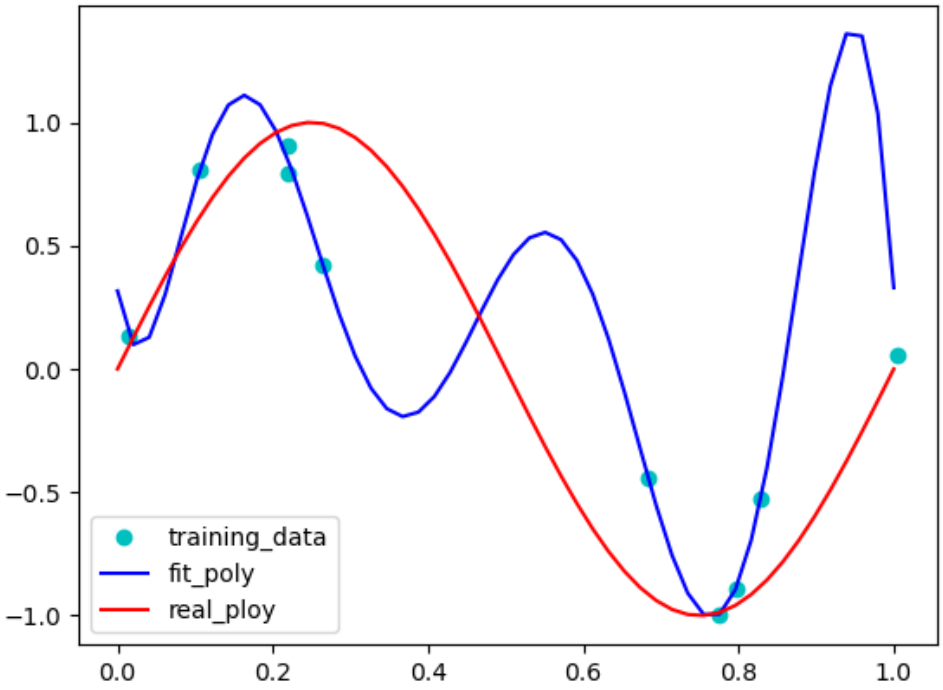

training_data
fit_poly
real_ploy
2500
2000
1500
1000
500
0
0.0
0.2
0.4
0.6
0.8
1.0

跑一百万次


5
0
−5
−10
−15
−20
−25
−30
0.0
0.2
0.4
0.6
0.8
1.0
training_data
fit_poly
real_ploy

跑一千万次

```
G:\data\python.exe G:/pythonProject1/main.py
99999999
           9             8             7             6            5
-3.845e+05 x + 1.765e+06 x - 3.419e+06 x + 3.642e+06 x - 2.33e+06 x
             4             3             2
 + 9.184e+05 x - 2.196e+05 x + 3.003e+04 x - 2044 x + 48.05
```

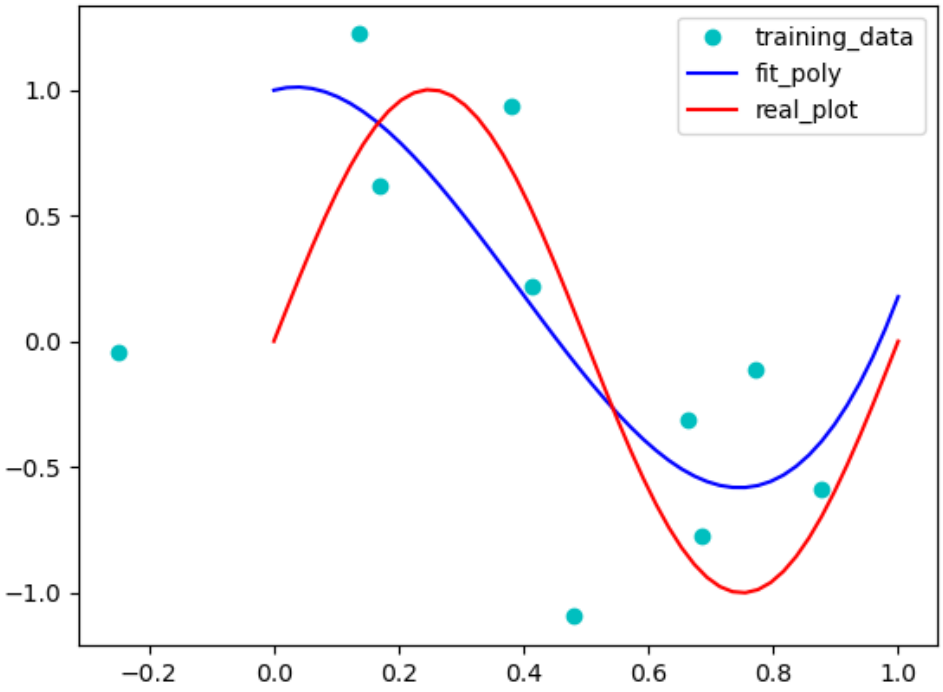


完美过拟合

梯度下降法优化过拟合

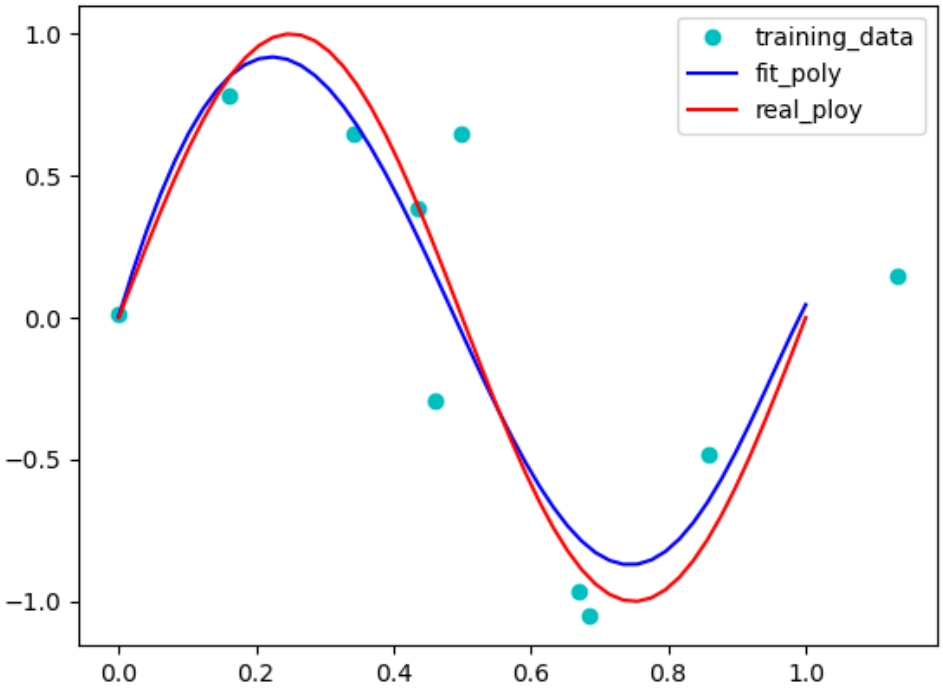
取学习率为 0.01， lamda=e^{-10} epilson=10^{-6}

3 阶：迭代次数 38800
欠拟合

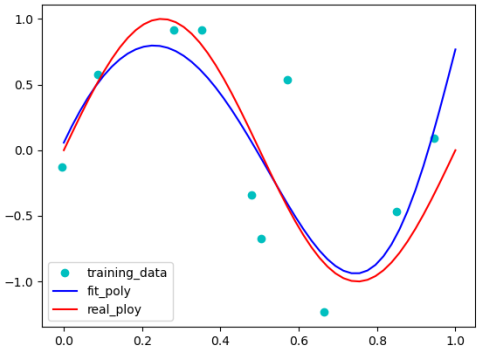


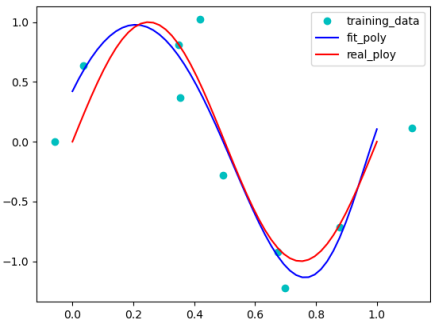
6 阶：迭代次数 100 万

```
G:\data\python.exe G:/pythonProject1/main.py
9999999
        6         5         4         3         2
-18.57 x + 21.57 x + 16.92 x - 11.5 x - 16.5 x + 8.121 x + 0.004365
```



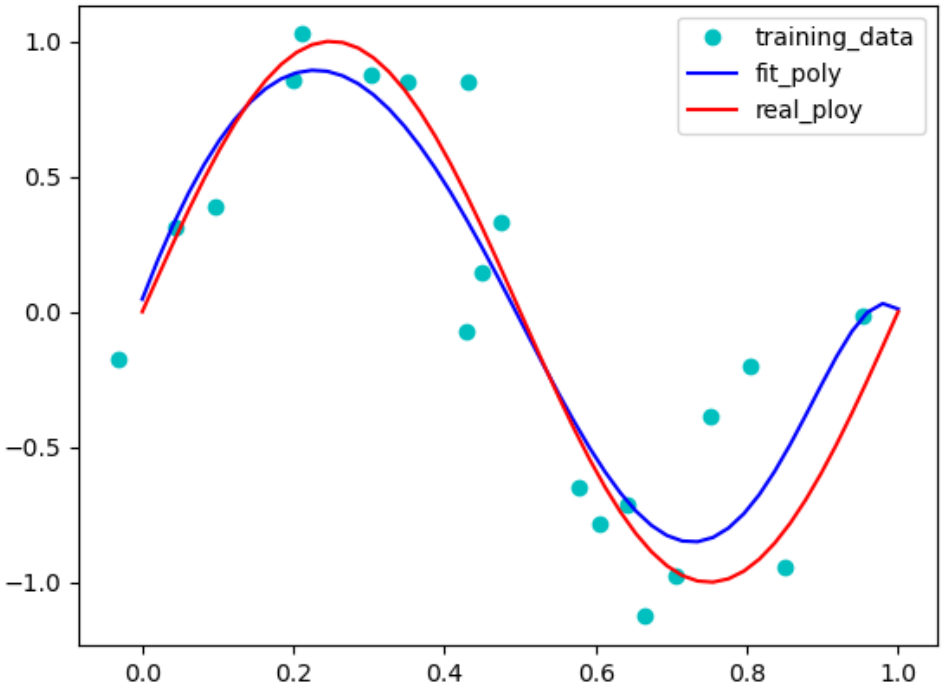
9 阶：迭代次数 75934 53943

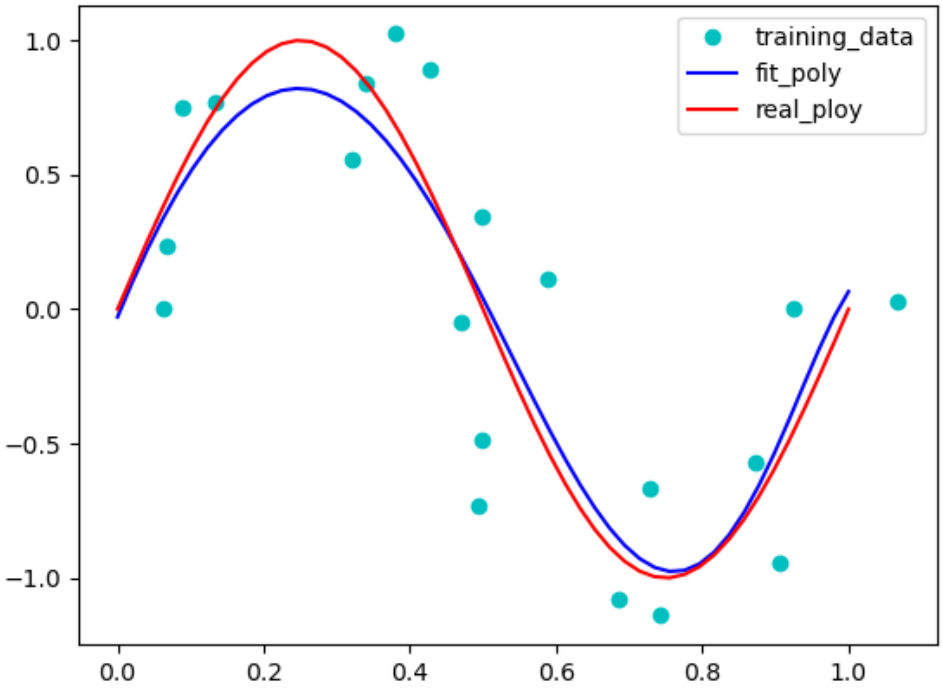




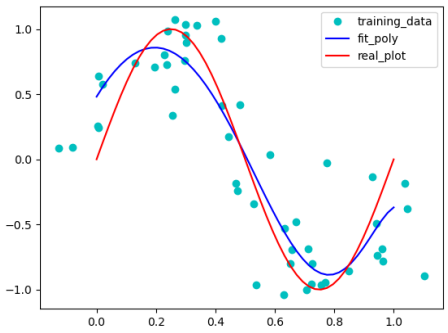
固定阶数为 9，不同训练集大小下的拟合结果：

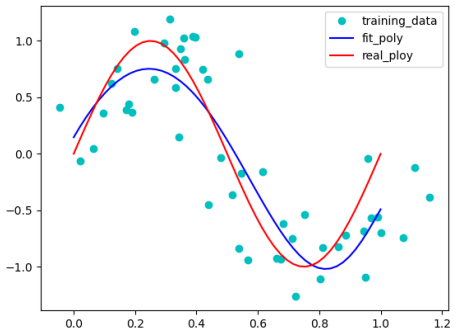
训练集大小为 20：迭代次数 20876 54192





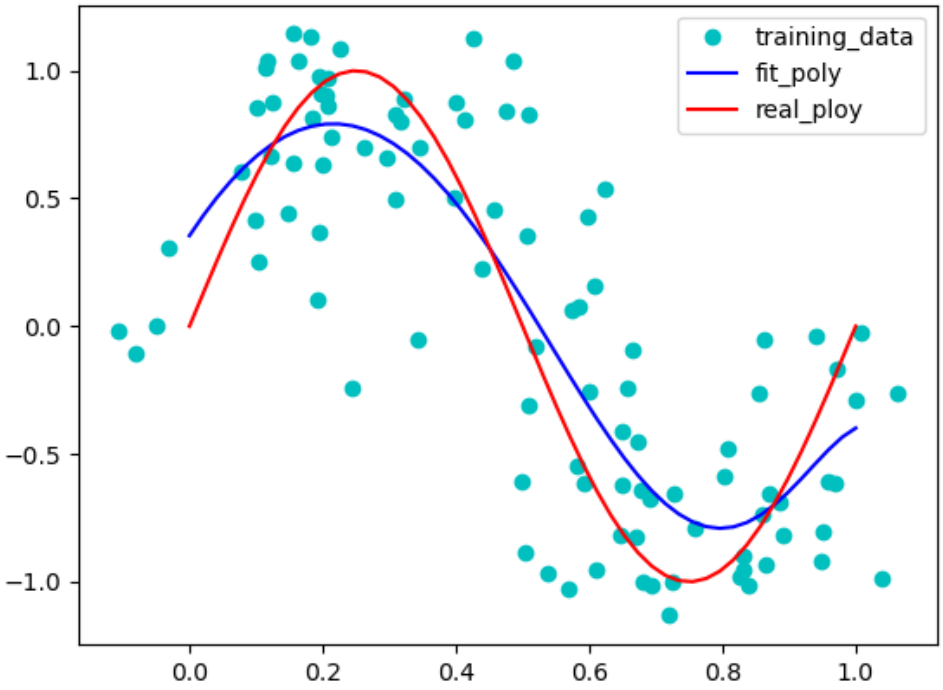
训练集大小为 50：迭代次数 12014





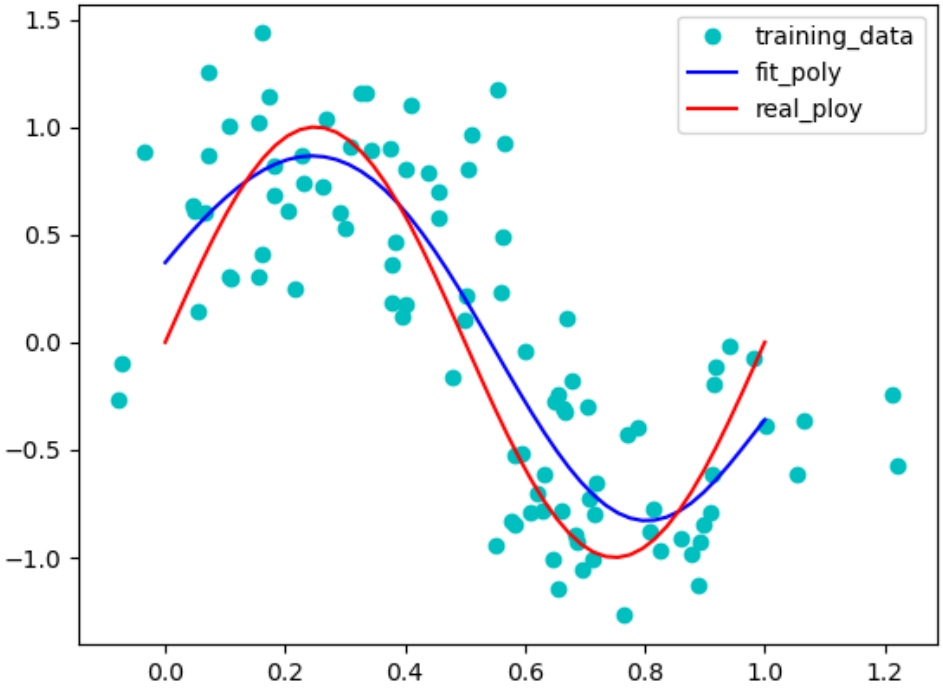
1. 训练集大小为 100：迭代次数 54080 这是判别条件只有 `(old_loss-new_loss < epsilon)`

```
G:\data\python.exe G:/pythonProject1/main.py
54080
        9          8         7         6         5         4         3
-7.469 x + 0.07784 x + 5.149 x + 6.799 x + 4.441 x - 1.214 x - 6.624 x
          2
 - 4.127 x + 2.303 x + 0.5097
```



```
1.  (old_loss-new_loss < epsilon)&(np.linalg.norm(gradient_w)<=epsilon)
```

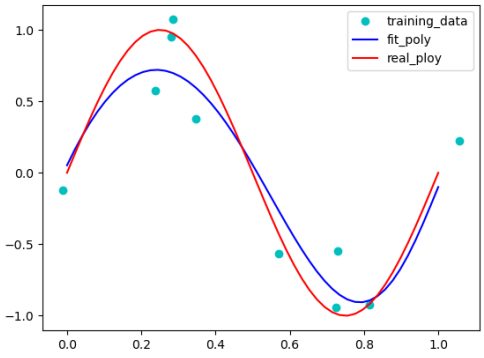
```
G:\data\python.exe G:/pythonProject1/main.py
9999999
        9         8         7         6         5         4         3
13.17 x - 27.41 x - 5.781 x + 22.08 x + 18.86 x - 13.4 x - 8.591 x
          2
 - 3.055 x + 3.399 x + 0.37
```

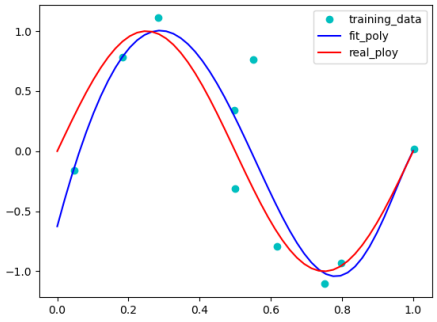


梯度下降法与共轭梯度相比其迭代次数都在 10^4 次方以上，性能很差。随着多项式阶数增加其拟合效果会变好但其拟合次数也相应增加。

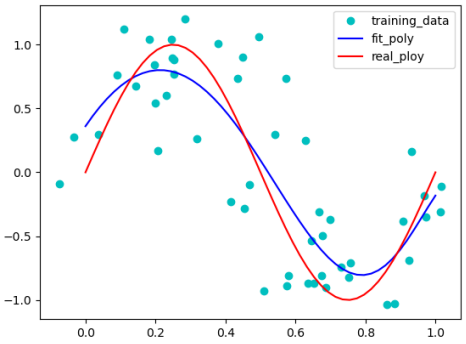
固定阶数为 9，不同精度下的拟合结果：

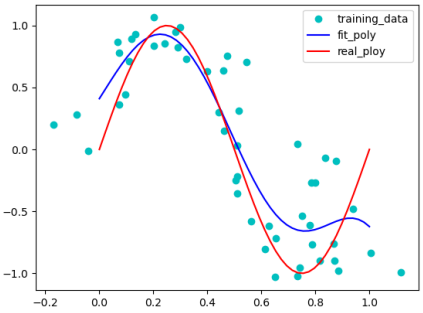
训练集大小为 10 精度分别为 10^{-6} 和 10^{-8} 迭代次数为 67445 和 344039



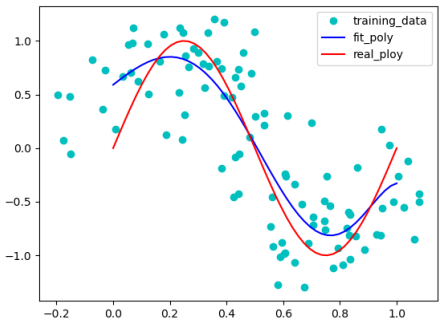


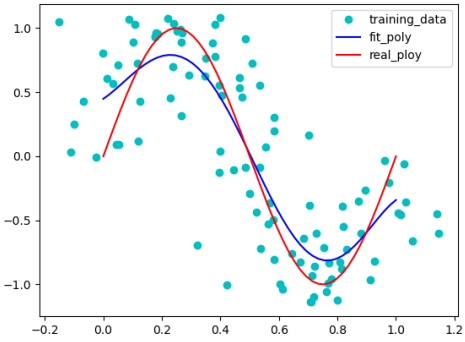
训练集大小为 50 精度分别为 10^{-6} 和 10^{-8} 迭代次数为 26627 和 1707067





训练集大小为 100 精度分别为 10^{-6} 和 10^{-8} 迭代次数为 1000 万次和一亿次





精度越高拟合效果会越好

4.4 共轭梯度求得优化解：

共轭梯度法求解析解：

$$\frac{\partial E}{\partial w} = X^TXw - X^TY + \lambda w = 0$$

则求 $(X^TX + \lambda)w = X^TY$ 的解析解

记 $A = X^TX + \lambda$　　$B = X^TY$　　则求使残差满足精度的 w

初始化 $w_0 = 0$　　$r_0 = b$　　$p_0 = r_0$

当 r_k 不满足精度时，进入循环，对第 k 次循环计算：

$$a_k = \frac{r_k^T r_k}{p_k^T A p_k}$$

$$w_k = w_k + a_k p_k$$

$$r_{k+1} = r_k - a_k A p_k$$

$$b_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

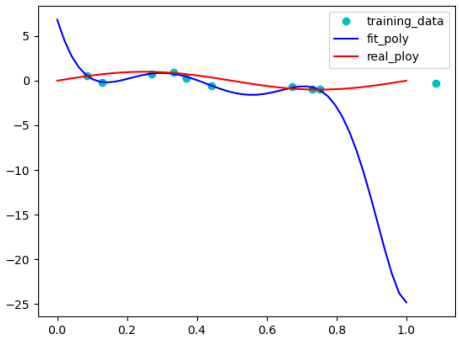
$$p_{k+1} = r_{k+1} + b_k p_k$$

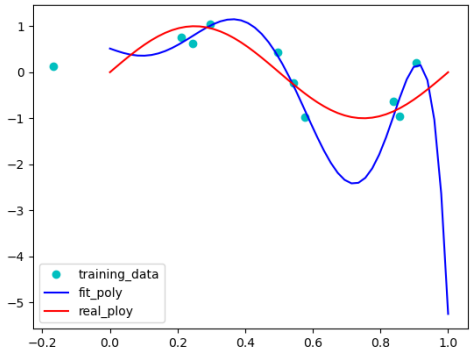
```python
def conjugate_descent(train_x, train_y,lamda,epsilon,exponent):
    # 把 loss 记为 Aw=b 的形式，其中 A = X^T * X + lamda，b = X^T * Y
    A=np.dot(train_x.T, train_x)+lamda*np.identity(exponent+1, dtype=float)

    b=np.dot(train_x.T,train_y)
    w = np.zeros((train_x.shape[1], 1))  # 初始化 w 为 n+1 * 1 的零阵
    r = b
    p = b
    i = 0
    while True:
        i = i + 1
        norm_2 = np.dot(r.T, r)
        a = norm_2 / np.dot(p.T, A).dot(p)
        w = w + a * p
        r = r - (a * A).dot(p)
        if r.T.dot(r) < epsilon:
            break
        b = np.dot(r.T, r) / norm_2
        p = r + b * p
```

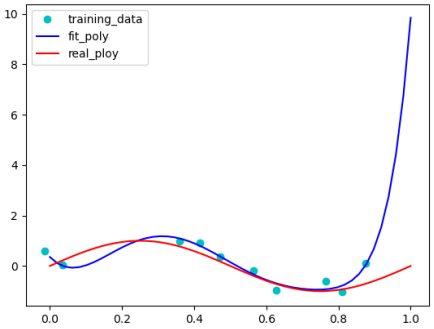
```
    print(i)
    poly_fitting = np.poly1d(w[::-1].reshape(train_x.shape[1]))
    return  poly_fitting
```

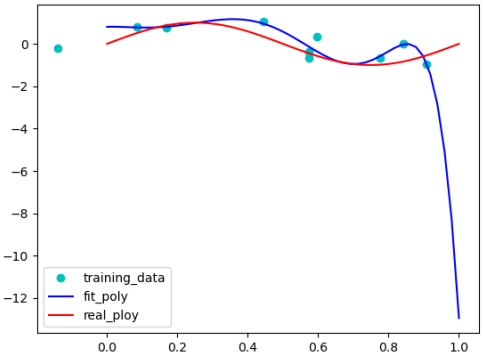
取 lamda=0 epilson=10^{-6}

迭代次数 11 11 11 9







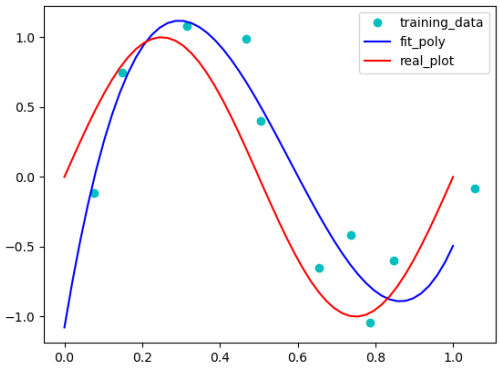


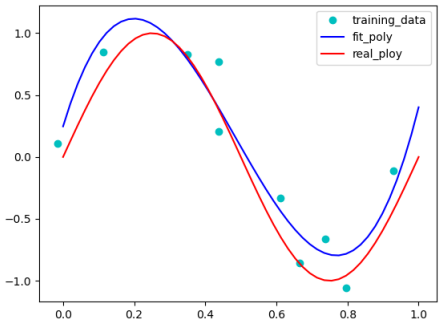
比起梯度下降的几十万次才能过拟合数据 梯度下降可以快速拟合数据

优化过拟合

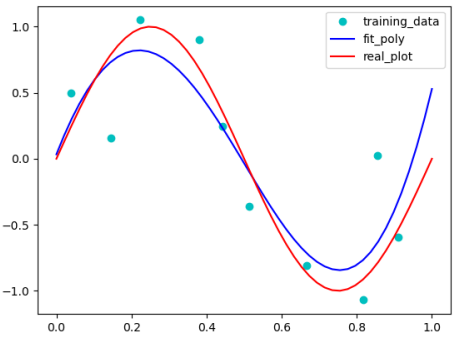
取 lamda=e^{-10} epilson=10^{-6}

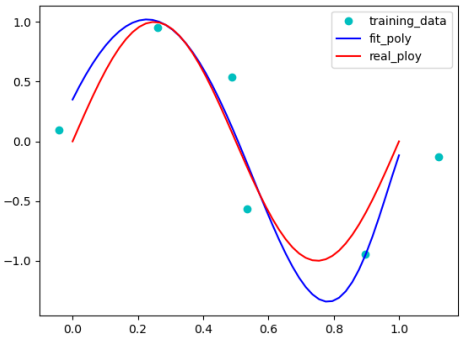
数据集 10
3 阶：迭代次数 4



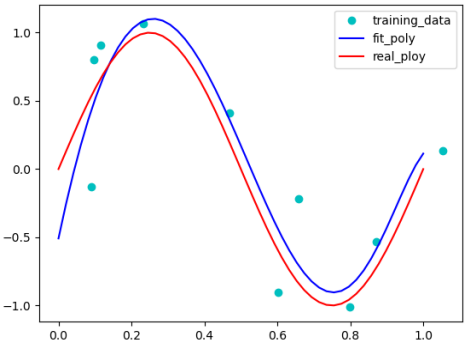


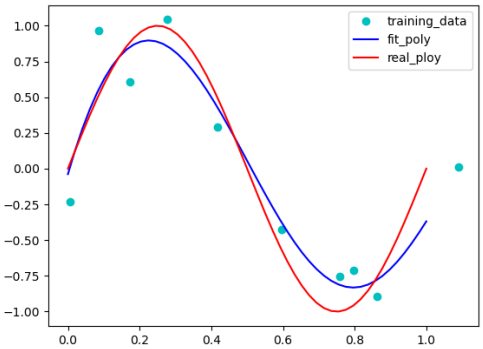
6 阶：迭代次数 5 7





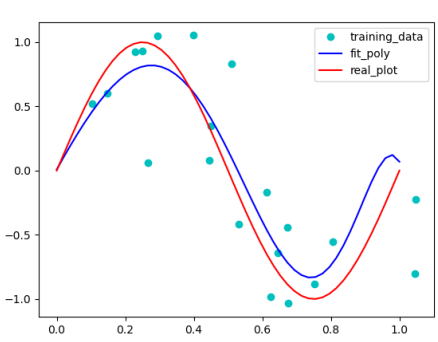
9 阶：迭代次数 5 7

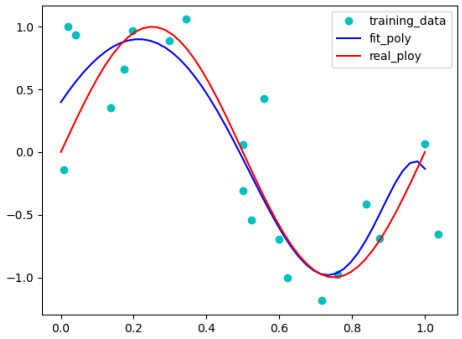




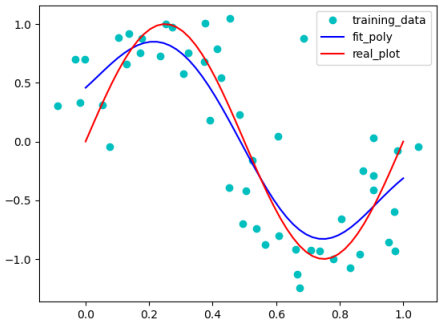
固定阶数为 9，不同训练集大小下的拟合结果：

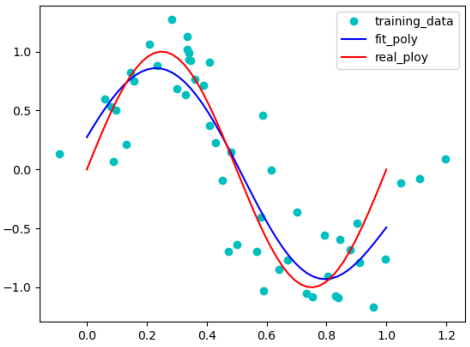
训练集大小 20：迭代次数 7 8



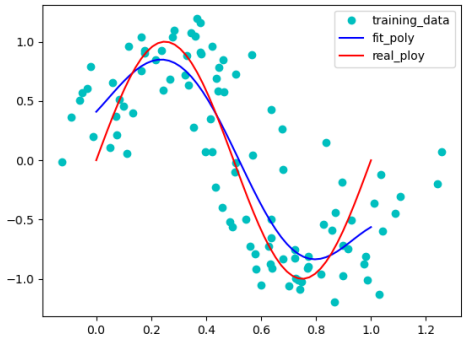


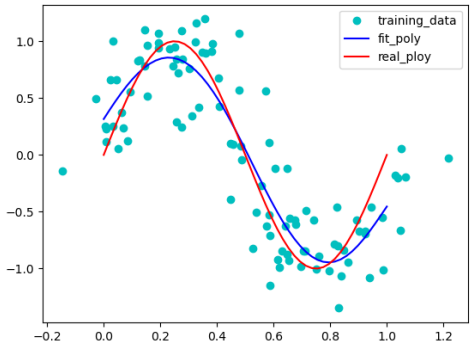
训练集大小 50：迭代次数 9 13





训练集大小 100：迭代次数 13 13





共轭梯度法的迭代次数受精度、多项式阶、训练集的大小的影响并不大 其迭代次数在 10 左右比梯度下降法速度快很多效果也更好。

五、结论：

在对正弦函数的多项式拟合中，多项式的次数越高，其模型能力越强，在不加正则项的情况下，高次多项式的拟合效果出现了过拟合，这是由于样本数量少但模型能力强导致的，模型拟合结果过分依赖数据集，而样本不足使得数据存在

一定偶然性，这种强拟合能力可能无法拟合出正弦曲线的效果。所以增大数据集可以有效地解决过拟合问题。

在目标函数中加入参数的惩罚项后，过拟合现象得到明显改善。这是由于参数增多时，往往具有较大的绝对值，加入正则项可以有效地降低参数的绝对值，从而使模型复杂度与问题匹配。所以对于训练样本限制较多的问题，增加惩罚项是解决过拟合问题的有效手段。

在使用梯度下降时，由于我们的目标函数是二次的，只有一个极值点，即最值点。如果梯度下降步长设置的比较大，那么下降结果将在目标函数最值附近逐渐向上跳动，从而无法收敛。

梯度下降相比共轭梯度收敛速度很慢，迭代次数很大，而共轭梯度的稳定性较差，更容易出现过拟合现象，但对于数据量较大复杂度较高的情况，共轭梯度显然要比梯度下降来的更优。

六、参考文献：

[1] 周志华 著. 机器学习，北京：清华大学出版社，2016 年 1 月

[2]维基百科：共轭梯度法 https://zh.wikipedia.org/wiki/%E5%85%B1%E8%BD%AD%E6%A2%AF%E5%BA%A6%E6%B3%95

七、附录：源代码（带注释）

Datamaker:

```
#根据所需数据集维数多项式阶数添加高斯噪声生成相应数据集
import numpy as np
from matplotlib import pyplot as plt
import random
import os
# 指定使用 0,1,2 三块卡
os.environ["CUDA_VISIBLE_DEVICES"] = "0,1,2"

# exponent:多项式最高次幂 size:数据集包含数据个数
def generate_data(exponent, size,begin, end):
    X = np.linspace(begin, end, size)  #
    Y = np.sin(2*X*np.pi)
    # 对输入数据加入 gauss 噪声
    # 定义 gauss 噪声的均值和方差
    mu = 0
    sigma = 0.12
```

```
    for i in range(X.size):
        X[i] += random.gauss(mu, sigma)
        Y[i] += random.gauss(mu, sigma)
    train_x = np.zeros((exponent + 1, size))  # 创建（n+1)*m 矩阵 X
    train_y = Y.reshape(size, 1)  # 标签矩阵 m*1  Y
    train_x = np.vander(X, exponent + 1, increasing=True)# 此函数生成的是转置的范德蒙德行列式
    #print(train_x)
    return train_x, train_y, X, Y
#计算 w
def cal_w(train_x, train_y, lamda,exponent):
    return np.linalg.inv(np.dot(train_x.T,train_x)+lamda*np.identity(exponent+1, dtype=float)).dot(train_x.T).dot(train_y)
 # 最小二乘法求解析解
def lsm(train_x, train_y, lamda,exponent):
     #lamda=0 时不带正则项，lamda!=0 时带正则项
    w=cal_w(train_x, train_y, lamda,exponent)
    poly_fitting = np.poly1d(w[::-1].reshape(train_x.shape[1])) #多项式函数
    return poly_fitting
#损失函数
def loss(train_x, train_y, w, lamda):
    loss = train_x.dot(w) - train_y
    loss = 1 / 2 * np.dot(loss.T, loss) + lamda / 2 * np.dot(w.T, w)
    return loss
#梯度下降
#alpha:步长 epsilon:精度 times:最高迭代次数
def gradient_descent(train_x, train_y,lamda,alpha,epsilon,exponent,times):
    #w = np.zeros((train_x.shape[1], 1))
    w=cal_w(train_x,train_y,lamda,exponent)
    new_loss = abs(loss(train_x, train_y, w, lamda))
    for i in range(times):
        old_loss = new_loss
        gradient_w=np.dot(train_x.T, train_x).dot(w) - np.dot(train_x.T, train_y) + lamda * w #损失函数对 w 求导即梯度
        old_w=w
        w -= gradient_w * alpha
        new_loss = abs(loss(train_x, train_y, w, lamda))
        gradient_w = np.dot(train_x.T, train_x).dot(w) - np.dot(train_x.T, train_y) + lamda * w
        if old_loss < new_loss:  # 不下降了，说明步长过大
            w = old_w
            alpha /= 2
        if (old_loss-new_loss < epsilon)&(np.linalg.norm(gradient_w)<=epsilon):
```

```
            break
    poly_fitting = np.poly1d(w[::-1].reshape(train_x.shape[1]))  # 多项式函数

    return poly_fitting,i
#共轭梯度下降
def conjugate_descent(train_x, train_y,lamda,epsilon,exponent):
    # 把 loss 记为 Aw=b 的形式，其中 A = X^T * X + lamda，b = X^T * Y
    A=np.dot(train_x.T, train_x)+lamda*np.identity(exponent+1, dtype=float)

    b=np.dot(train_x.T,train_y)
    w = np.zeros((train_x.shape[1], 1))  # 初始化 w 为 n+1 * 1 的零阵
    r = b
    p = b
    i = 0
    while True:
        i = i + 1
        norm_2 = np.dot(r.T, r)
        a = norm_2 / np.dot(p.T, A).dot(p)
        w = w + a * p
        r = r - (a * A).dot(p)
        if r.T.dot(r) < epsilon:
            break
        b = np.dot(r.T, r) / norm_2
        p = r + b * p
    print(i)
    poly_fitting = np.poly1d(w[::-1].reshape(train_x.shape[1]))
    return  poly_fitting
def plt_show(x, y, poly_fit):

    plot1 = plt.plot(x, y, 'co', label='training_data')
    real_x = np.linspace(0,1)
    real_y = np.sin(real_x * 2 * np.pi)
    fit_y = poly_fit(real_x)
    plot2 = plt.plot(real_x, fit_y, 'b', label='fit_poly')
    plot3 = plt.plot(real_x, real_y, 'r', label='real_ploy')
    #选择最佳位置写图像标注
    plt.legend(loc=0)
    plt.show()
    print(poly_fit)
#解析法 利用均方根判断 lamda 合适取值
# test_x, test_y 验证集合 size 是训练集大小  test_size 是测试集大小
def Rmse(train_x, train_y,train_size,test_x,test_y,exponent,test_size):

```

```
    ln_lamda = np.linspace(-10,0,50)

     rms_train = np.zeros(50)
     rms_test = np.zeros(50)
     for i in range(0, 50):
         lamda = np.exp(ln_lamda[i])
         w = cal_w(train_x, train_y, lamda,exponent)
         Ew_train = loss(train_x, train_y,w,lamda)
         rms_train[i] = np.sqrt(2 * Ew_train / train_size)
         #w_test=cal_w(test_x, test_y,lamda,exponent)
         Ew_test=loss(test_x, test_y, w,lamda)
         rms_test[i] = np.sqrt(2 * Ew_test / test_size)
     train_plot = plt.plot(ln_lamda, rms_train, 'b', label='train')
     test_plot = plt.plot(ln_lamda, rms_test, 'r', label='test')
     # 横坐标是 lamda
     plt.xlabel('lamda')
     plt.ylabel('Rms')
     plt.legend(loc=0)
     plt.show()
#有正则项与无正则项图像对比
def comparsion_show(x, y, poly_fit,punish_poly):

     plot1 = plt.plot(x, y, 'co', label='training_data')
     real_x = np.linspace(0,1)
     real_y = np.sin(real_x * 2 * np.pi)
     fit_y = poly_fit(real_x)
     plot2 = plt.plot(real_x, fit_y, 'b', label='fit_poly')
     plot3 = plt.plot(real_x, real_y, 'r', label='real_ploy')
     polt4=plt.plot(real_x,punish_poly(real_x),'k',label='punish_poly')
     #选择最佳位置写图像标注
     plt.legend(loc=0)
     plt.show()
     print(poly_fit)
```

main

```
import DataMaker
import numpy as np
epsilon = 1e-6
import matplotlib.pyplot as plt

if __name__=='__main__':
     '''''# 最小二乘法求解析解(无正则项) lamda=0
```

```
    begin, end, exponent,lamda,size = 0, 1, 50, 0,500
    train_x, train_y, x,y=DataMaker.generate_data(exponent,size,begin,end)
    poly = DataMaker.lsm(train_x, train_y,lamda,exponent)
    DataMaker.plt_show(x,y,poly)
     # 最小二乘法求解析解(有正则项)
    #梯度下降法求优化解
    begin, end, exponent, lamda, size = 0, 1, 9, np.exp(-10), 10
    alpha=0.0001
    train_x, train_y, x, y = DataMaker.generate_data(exponent, size, begin, end)
    poly,i=DataMaker.gradient_descent(train_x, train_y,lamda,alpha,epsilon,exponent,times=1000000)
    print(i)
    DataMaker.plt_show(x, y, poly)
    # 共轭梯度法求优化解'''
    '''''begin, end, exponent, lamda, size = 0, 1, 50, np.exp(-10),50
    train_x, train_y, x, y = DataMaker.generate_data(exponent, size, begin, end)
    poly=DataMaker.conjugate_descent(train_x, train_y,lamda,epsilon,exponent)
    DataMaker.plt_show(x, y, poly)'''
    '''''begin, end, exponent, lamda, size = 0, 1, 9, 0, 10
    train_x, train_y, x, y = DataMaker.generate_data(exponent, size, begin, end)
    test_x, test_y,X,Y=DataMaker.generate_data(exponent,20,0, 1)
    DataMaker.Rmse(train_x,train_y,10,test_x,test_y,exponent,20)'''
    begin, end, exponent, lamda, size = 0, 1, 9, np.exp(-10), 10
    train_x, train_y, x, y = DataMaker.generate_data(exponent, size, begin, end)
    poly = DataMaker.lsm(train_x, train_y, 0, exponent)
    punish_poly = DataMaker.lsm(train_x, train_y, lamda, exponent)
    DataMaker.comparsion_show(x,y,poly,punish_poly)
```