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：必修

实验题目：多项式拟合正弦函数

学号：1160300314

姓名：朱明彦

一、实验目的

掌握最小二乘法求解（无惩罚项的损失函数），掌握增加惩罚项（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、客服过拟合的方法（如增加惩罚项、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

- OS: Microsoft Windows 10.0.17134
- Python 3.6.4

三、设计思想(本程序中用到的主要算法及数据结构)

算法原理

1、生成数据算法

主要是利用$sin(2\pi x)$函数产生样本，其中x均匀分布在$[0,1]$之间，对于每一个目标值$t=sin(2\pi x)$增加一个0均值，方差为0.5的高斯噪声。

2、利用高阶多项式函数拟合曲线(不带惩罚项)

利用训练集合，对于每个新的$\hat{x}$，预测目标值$\hat{t}$。采用多项式函数进行学习，即利用式(1)来确定参数w，假设阶数m已知。

$$y(x,w)=w_0+w_1x+\cdots+w_mx^m=\sum_{i=0}^{m}w_ix^i \tag{1}$$

采用最小二乘法，即建立误差函数来测量每个样本点目标值t与预测函数$y(x,w)$之间的误差，误差函数即式(2)

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\{y(x_i, \mathbf{w}) - t_i\}^2 \tag{2}$$

将上式写成矩阵形式如式(3)

$$E(\mathbf{w}) = \frac{1}{2}(\mathbf{Xw} - \mathbf{T})'(\mathbf{Xw} - \mathbf{T}) \tag{3}$$

其中

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & \cdots & x_1^m \\ 1 & x_2 & \cdots & x_2^m \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^m \end{bmatrix}, \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix}, \mathbf{T} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix}$$

通过将上式求导我们可以得到式(4)

$$\frac{\partial E}{\partial \mathbf{w}} = \mathbf{X'Xw} - \mathbf{X'T} \tag{4}$$

令 $\frac{\partial E}{\partial \mathbf{w}} = 0$ 我们有式(5)即为$\mathbf{w}^*$

$$\mathbf{w}^* = (\mathbf{X'X})^{-1}\mathbf{X'T} \tag{5}$$

3、带惩罚项的多项式函数拟合曲线

在不带惩罚项的多项式拟合曲线时，在参数多时$\mathbf{w}^*$具有较大的绝对值，本质就是发生了过拟合。对于这种过拟合，我们可以通过在优化目标函数式(3)中增加$\mathbf{w}$的惩罚项，因此我们得到了式(6)。

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\{y(x_i, \mathbf{w}) - t_i\}^2 + \frac{\lambda}{2}||\mathbf{w}||^2 \tag{6}$$

同样我们可以将式(6)写成矩阵形式， 我们得到式(7)

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}[(\mathbf{Xw} - \mathbf{T})'(\mathbf{Xw} - \mathbf{T}) + \lambda\mathbf{w'w}] \tag{7}$$

对式(7)求导我们得到式(8)

$$\frac{\partial \widetilde{E}}{\partial \mathbf{w}} = \mathbf{X'Xw} - \mathbf{X'T} + \lambda\mathbf{w} \tag{8}$$

令 $\frac{\partial \widetilde{E}}{\partial \mathbf{w}} = 0$ 我们得到$\mathbf{w}^*$即式(9)，其中$\mathbf{I}$为单位阵。

$$\mathbf{w}^* = (\mathbf{X'X} + \lambda\mathbf{I})^{-1}\mathbf{X'T} \tag{9}$$

4、梯度下降法求解最优解

对于$f(\mathbf{x})$如果在$\mathbf{x_i}$点可微且有定义，我们知道顺着梯度 $\nabla f(\mathbf{x_i})$为增长最快的方向，因此梯度的反方向 $-\nabla f(\mathbf{x_i})$ 即为下降最快的方向。因而如果有式(10)对于 $\alpha > 0$ 成立,

$$\mathbf{x_{i+1}} = \mathbf{x_i} - \alpha\nabla f(\mathbf{x_i}) \tag{10}$$

那么对于序列$\mathbf{x_0}, \mathbf{x_1}, \ldots$我们有

$$f(\mathbf{x_0}) \geq f(\mathbf{x_1}) \geq \ldots$$

因此，如果顺利我们可以得到一个 $f(\mathbf{x_n})$ 收敛到期望的最小值，**当然对于我们此次实验很大可能性可以收敛到最小值。**

5、共轭梯度法求解最优解

共轭梯度法解决的主要是形如$\mathbf{Ax} = \mathbf{b}$的线性方程组解的问题，其中$\mathbf{A}$必须是对称的、正定的。 求解的方法就是我们先猜一个解$\mathbf{x_0}$，然后取梯度的反方向 $\mathbf{p_0} = \mathbf{b} - \mathbf{Ax}$，在n维空间的基中$\mathbf{p_0}$要与其与的基共轭并且为初始残差。

然后对于第k步的残差 $\mathbf{r_k} = \mathbf{b} - \mathbf{Ax}$，$\mathbf{r_k}$为$\mathbf{x} = \mathbf{x_k}$时的梯度反方向。由于我们仍然需要保证 $\mathbf{p_k}$ 彼此共轭。因此我们通过当前的残差和之前所有的搜索方向来构建$\mathbf{p_k}$，得到式(11)

$$\mathbf{p_k} = \mathbf{r_k} - \sum_{i<k} \frac{\mathbf{p_i}^T \mathbf{A} \mathbf{r_k}}{\mathbf{p_i}^T \mathbf{A} \mathbf{p_i}} \mathbf{p_i} \tag{11}$$

进而通过当前的搜索方向$\mathbf{p_k}$得到下一步优化解$\mathbf{x_{k+1}} = \mathbf{x_k} + \alpha_k \mathbf{p_k}$，其中

$$\alpha_k = \frac{\mathbf{p_k}^T (\mathbf{b} - \mathbf{A} \mathbf{x_k})}{\mathbf{p_k}^T \mathbf{A} \mathbf{p_k}} = \frac{\mathbf{p_k}^T \mathbf{r_k}}{\mathbf{p_k}^T \mathbf{A} \mathbf{p_k}}$$

算法的实现

对于数据生成、求解析解（有无正则项）都是可以利用numpy中的矩阵求逆等库变相降低了算法实现的难度，因此在这里就不再赘述，下面主要讲梯度下降法和共轭梯度法的算法实现。

1、梯度下降

此处我们利用带惩罚项的优化函数进行梯度下降法的实现。 由梯度下降主要解决的是如式(12)的线性方程组的解。

$$\mathbf{Ax} - \mathbf{b} = 0 \tag{12}$$

另由式(8)我们可以得到式(13)

$$J(\mathbf{w}) = (\mathbf{X'X} + \lambda \mathbf{I})\mathbf{w} - \mathbf{X'T} \tag{13}$$

联合式(13)与式(12)我们可以有：

$$\begin{cases} \mathbf{A} = \mathbf{X'X} + \lambda \mathbf{I} \\ \mathbf{b} = \mathbf{X'T} \end{cases} \tag{14}$$

进而我们实现算法如下，其中δ为精度要求，通常可以设置为$\delta = 1 \times 10^{-6}$：

$rate = 0.01, k = 0$
$\mathbf{w_0} = \mathbf{0}$
$\widetilde{E}(\mathbf{w}) = \frac{1}{2N}[(\mathbf{Xw} - \mathbf{T})'(\mathbf{Xw} - \mathbf{T}) + \lambda \mathbf{w}'\mathbf{w}]$
$J(\mathbf{w}) = (\mathbf{X}'\mathbf{X} + \lambda \mathbf{I})\mathbf{w} - \mathbf{X}'\mathbf{T}$
$\mathbf{loss_0} = \widetilde{\mathbf{E}}(\mathbf{w_0})$
repeat :
    $\mathbf{w_{k+1}} = \mathbf{w_k} - rate * J(\mathbf{w_k})$
    $\mathbf{loss_{k+1}} = \widetilde{\mathbf{E}}(\mathbf{w_{k+1}})$
    if $\mathbf{abs}(\mathbf{loss_{k+1}} - \mathbf{loss_k}) < \delta$ **then break loop**
    $k = k + 1$
end repeat

2、共轭梯度下降

对于共轭梯度下降，算法实现如下，参考wiki实现：

$$
\begin{aligned}
&\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0 \\
&\mathbf{p}_0 := \mathbf{r}_0 \\
&k := 0 \\
&\textbf{repeat} \\
&\quad \alpha_k := \frac{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}{\mathbf{p}_k^\mathsf{T} \mathbf{A} \mathbf{p}_k} \\
&\quad \mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k \\
&\quad \mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k \\
&\quad \text{if } r_{k+1} \text{ is sufficiently small, then exit loop} \\
&\quad \beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T} \mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k} \\
&\quad \mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\
&\quad k := k + 1 \\
&\textbf{end repeat} \\
&\text{The result is } \mathbf{x}_{k+1}
\end{aligned}
$$

其中的$\mathbf{b}$, $\mathbf{A}$均与式(14)相同。

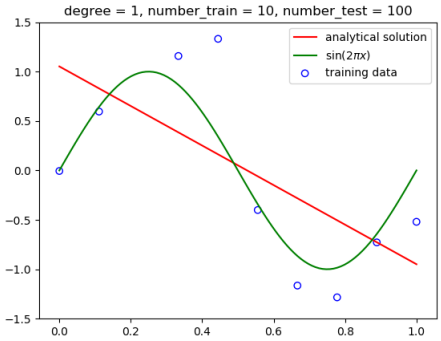
3、牛顿法

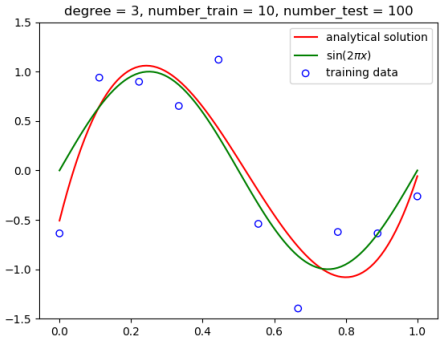
四、实验结果分析

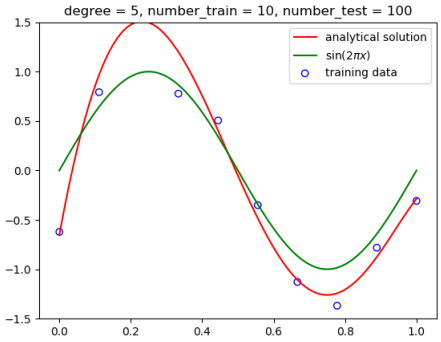
对于下面的所有的实验，在没有特别强调的情况下，所有训练样本的噪声标准差均为0.5。

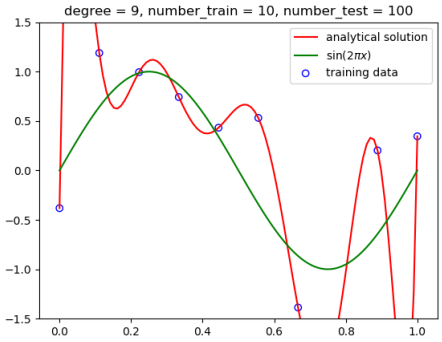
1、不带惩罚项的解析解

(1)固定训练样本的大小为10，分别使用不同多项式阶数，测试的结果如下图。





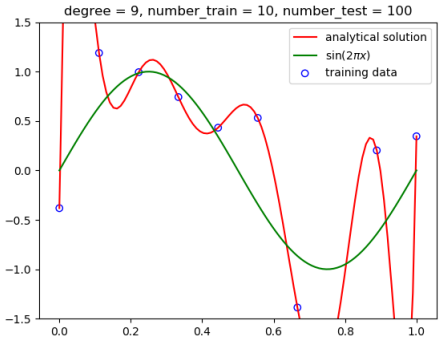


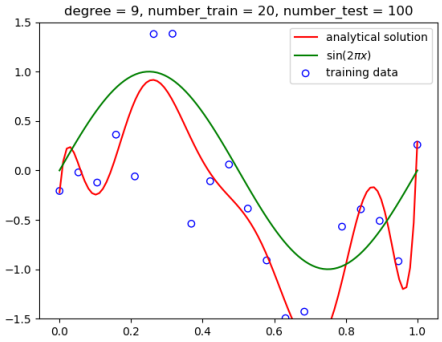


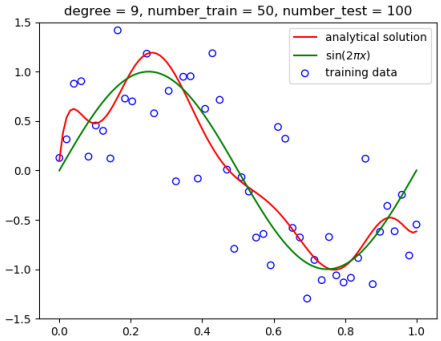
我们可以看到在固定训练样本的大小之后，在多项式阶数为3时的拟合效果已经很好。继续提高多项式的阶数，**尤其在阶数为9的时候曲线"完美的"经过了所有的节点，这种剧烈的震荡并没有很好的拟合真实的背后的函数$sin(2\pi x)$，反而将所有噪声均很好的拟合，即表现出来一种过拟合的情况。** 其出现过拟合的本质原因是，在阶数过大的情况下，模型的复杂度和拟合的能力都增强，因此可以通过过大或者过小的系数来实现震荡以拟合所有的数据点，以至于甚至拟合了所有的噪声。在这里由于我们的数据样本大小只有10，所以在阶数为9的时候，其对应的系数向量$\mathbf{w}$恰好有唯一解，因此可以穿过所有的样本点。

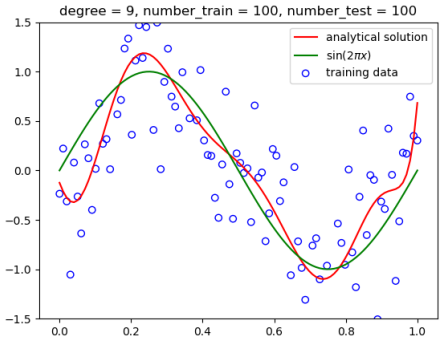
对于过拟合我们可以通过增加样本的数据或者通过增加惩罚项的方式来解决。增加数据集样本数量，使其超过参数向量的大小，就会在一定程度上解决过拟合问题。

(2)固定多项式阶数，使用不同数量的样本数据，测试的结果如下图。









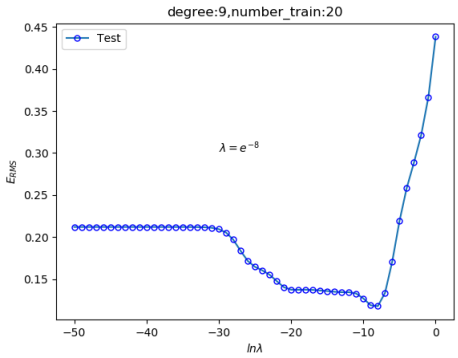
我们可以看到在固定多项式阶数为9的情况下，随着样本数量逐渐增加，过拟合的现象有所解决。特别是对比左上图与右下图的差距，可以看到样本数量对于过拟合问题是有影响的。

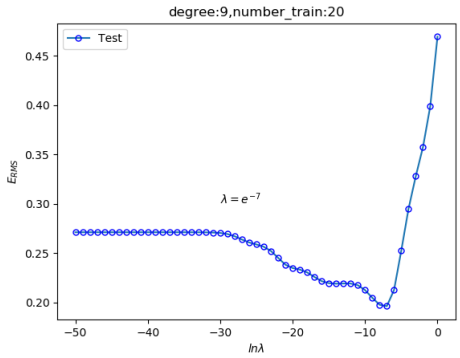
2、带惩罚项的解析解

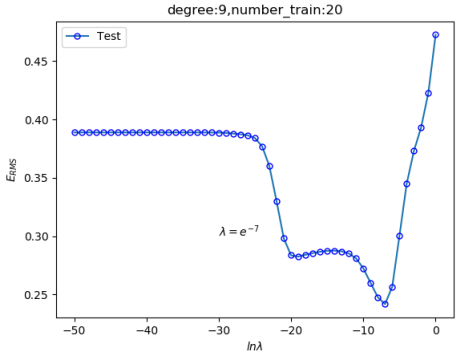
首先根据式(6)我们需要确定最佳的超参数λ，因此我们通过PRML中提到的根均方(RMS)误差来确定，其中RMS的定义如式(15)。

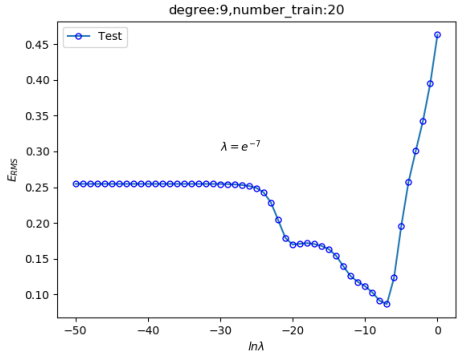
$$E_{RMS} = \sqrt{\frac{2E(\mathbf{w}^*)}{N}} \tag{15}$$

在这里我们抽取了100次试验中的四次将实验结果放在了下图中，这四次中恰好有三次的最优的超参数均为$\lambda = e^{-7}$。







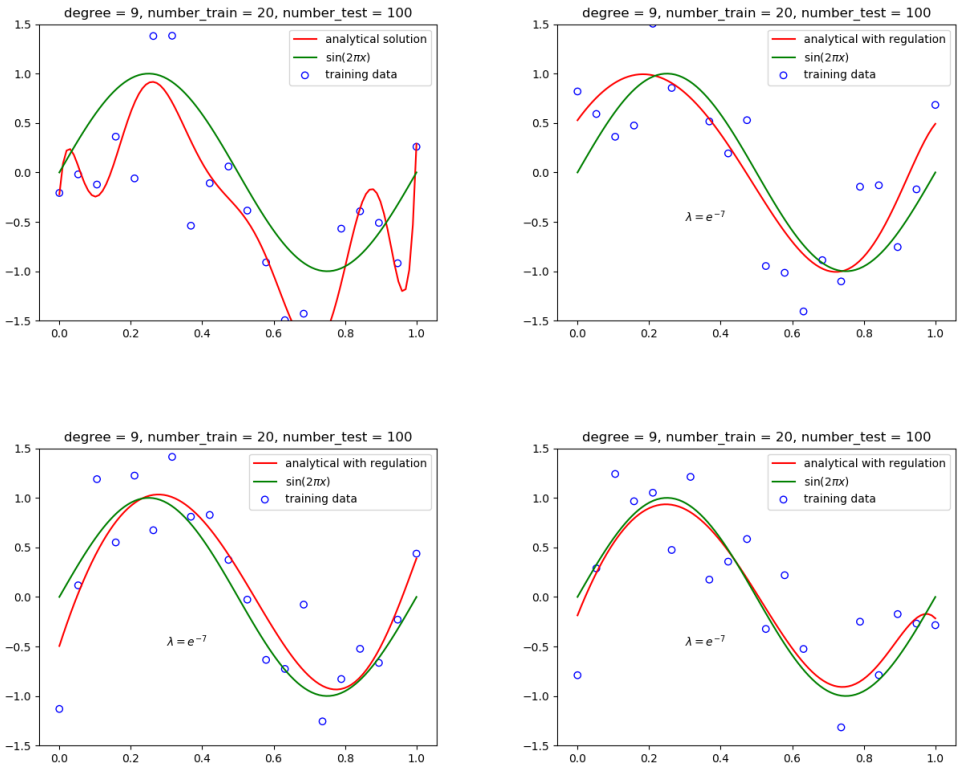


观察上面的四张图， 我们可以发现对于超参数λ的选择，在(e^{-50}, e^{-30})左右保持一种相对稳定的错误率；但是在(e^{-30}, e^{-5})错误率有一个明显的下降，所以下面在下面的完整100次实验中我们可以看到最佳参数的分布区间也大都在这个范围内；在大于e^{-5}的区间内，错误率有一个急剧的升高。

完整的100次实验的数据如下表格，我们可以看到最佳的超参数范围在(e^{-9}, e^{-6})之间，**因此在接下来的实验中我们将选取$\lambda = \mathbf{e}^{-7}$作为我们剩余中使用的最佳的超参数。**

$ln\lambda$	-25	-22	-17	-14	-13	-11	-10	-9	-8	-7	-6	-5	-3
$count$	1	1	2	4	2	5	6	11	26	29	11	1	1

基于我们上面选择的超参数，我们进行对照实验，测试在多项式阶数为9，训练样本数量为20个的情况下进行对照实验，其中左上的图片是没有增加惩罚项的测试结果，其余三张为增加惩罚项后的测试结果。



我们可以看到增加正则项的后三张图片相比没有正则项的第一张图片，过拟合的现象得到了很好的解决，**这一结果验证了增加惩罚项对于过拟合的作用，以及经过上面实验得到的超参数λ是符合我们的要求的。**

3、优化解

在这里主要是利用梯度下降(Gradient descent)和共轭梯度法(Conjugate gradient)两种方法来求优化解。由于该问题是有解析解存在，并且在之前的实验报告部分已经列出，所以在此处直接利用上面的解析解进行对比。**此处使用的学习率固定为$rate = 0.01$，停止的精度要求为1×10^{-6}。**

对比实验主要分为2个变量的对比：**多项式的阶数**和**训练样本的数量** 。测试的结果的**迭代次数**全表如下：

行号	多项式阶数	训练样本数量	梯度下降	共轭梯度
1	3	10	47707	4
2	3	20	31010	4
3	3	50	19708	4
4	6	10	11368	5
5	6	20	8195	5
6	6	50	4209	7

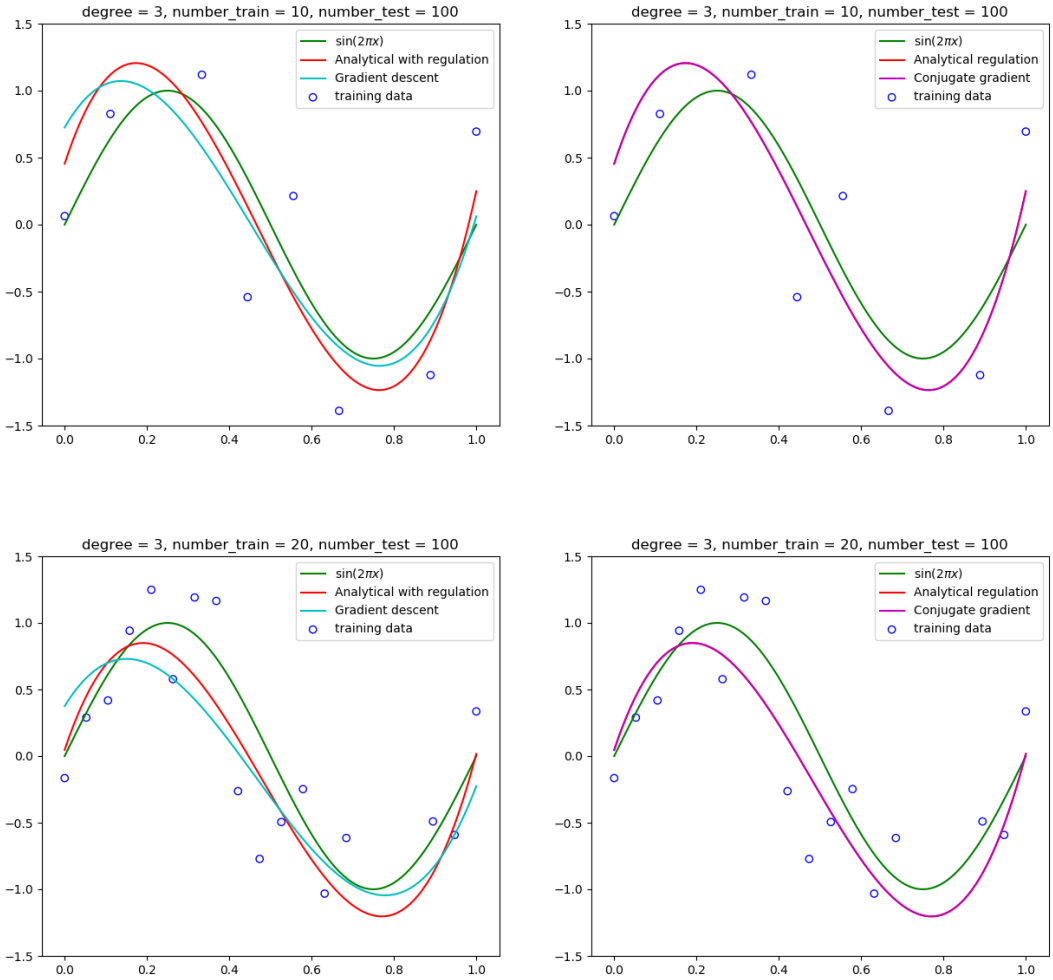
行号	多项式阶数	训练样本数量	梯度下降	共轭梯度
7	9	10	28135	7
8	9	20	19348	7
9	9	20	2134	7

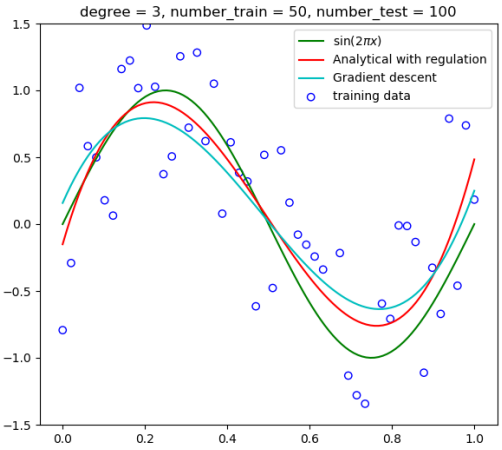
首先在固定多项式阶数的情况下，**随着训练样本的增加，梯度下降的迭代次数均有所下降，但是对于共轭梯度迭代次数变化不大。**

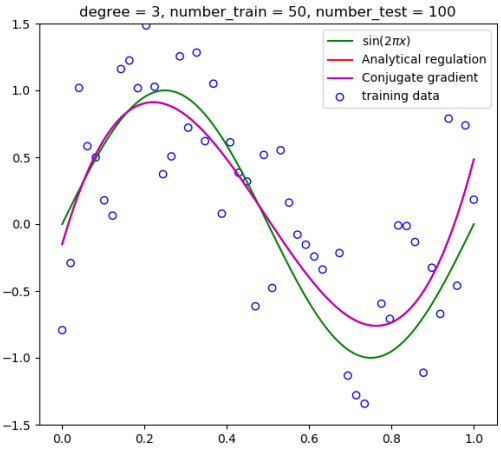
其次在固定训练样本的情况下，**梯度下降迭代次数的变化，对于degree = 3的情况下多于degree = 9的情况，均多于degree = 6的情况。对于共轭梯度的而言，迭代次数仍然较少。**

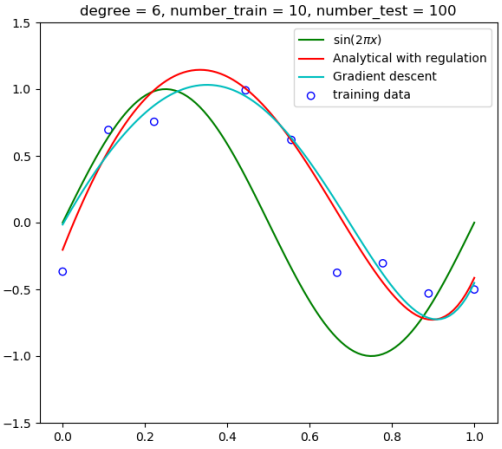
总的来说，**对于梯度下降法，迭代次数多在10000次以上；而对于共轭梯度下降，则需要的迭代次数均不超过10次。**

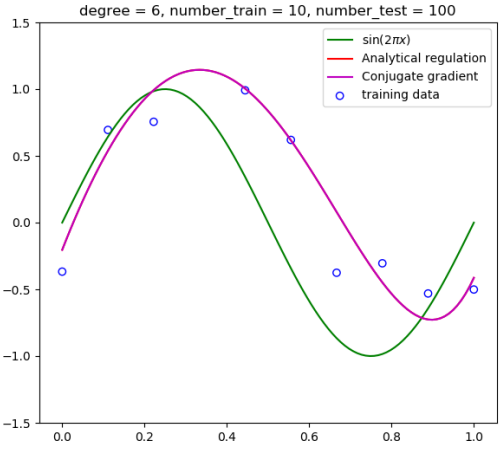
下面给出这些实验的数据图，如下：

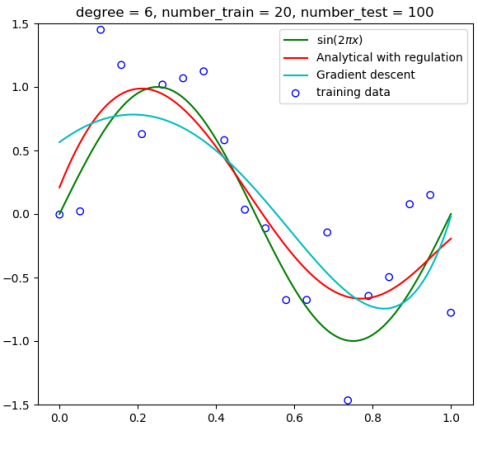


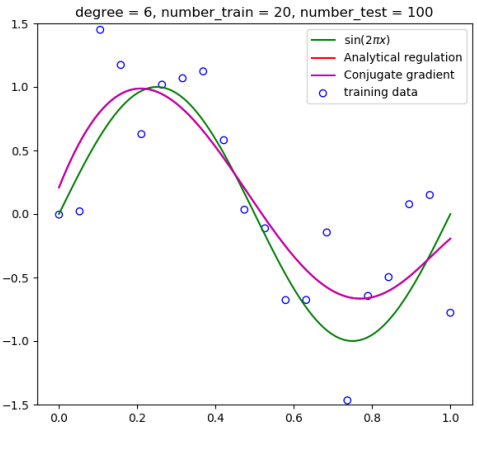

degree = 3, number_train = 50, number_test = 100
sin(2πx)
Analytical with regulation
Gradient descent
training data

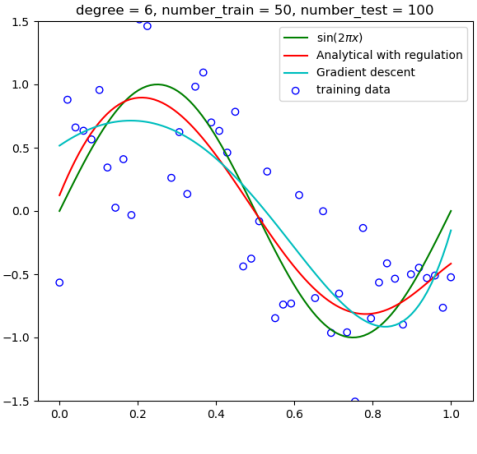

degree = 3, number_train = 50, number_test = 100
sin(2πx)
Analytical regulation
Conjugate gradient
training data

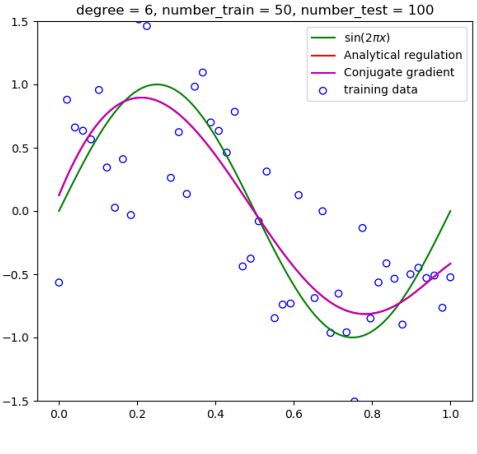

degree = 6, number_train = 10, number_test = 100
sin(2πx)
Analytical with regulation
Gradient descent
training data

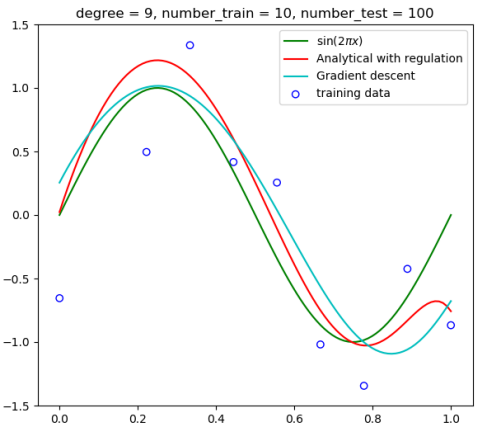

degree = 6, number_train = 10, number_test = 100
sin(2πx)
Analytical regulation
Conjugate gradient
training data

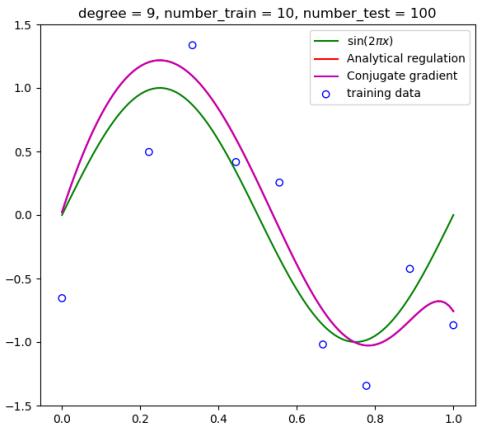

degree = 6, number_train = 20, number_test = 100
sin(2πx)
Analytical with regulation
Gradient descent
training data

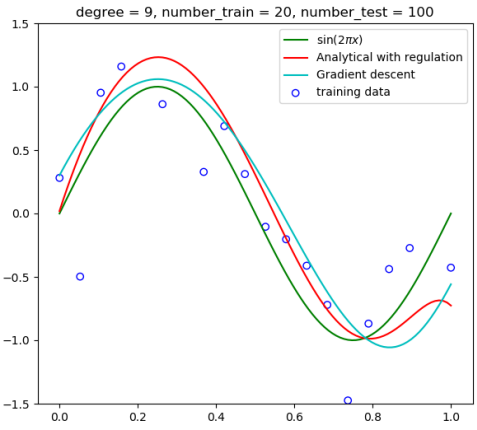

degree = 6, number_train = 20, number_test = 100
sin(2πx)
Analytical regulation
Conjugate gradient
training data

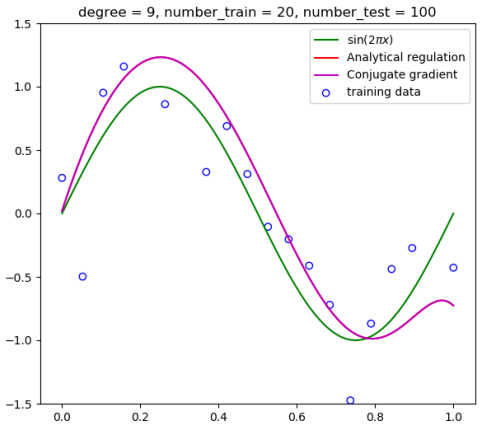

degree = 6, number_train = 50, number_test = 100
sin(2πx)
Analytical with regulation
Gradient descent
training data

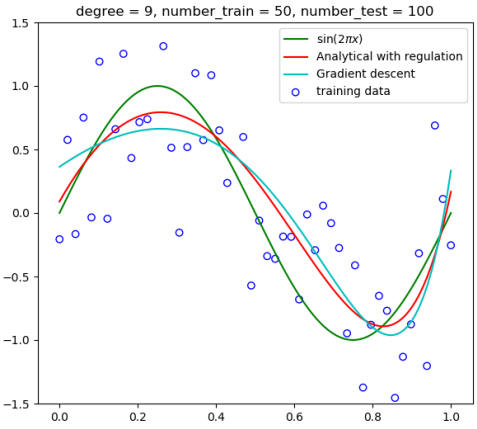

degree = 6, number_train = 50, number_test = 100
sin(2πx)
Analytical regulation
Conjugate gradient
training data

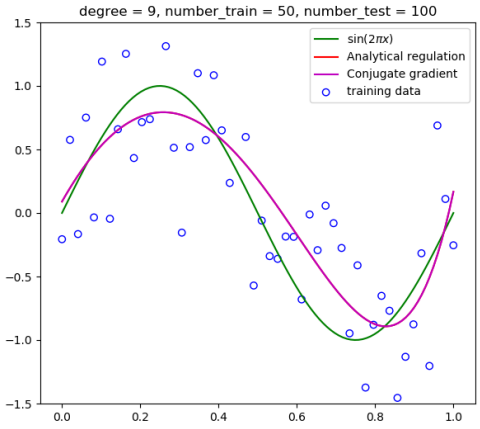












综合以上几次实验的结果，**我们可以发现对于梯度下降法，其与解析解相比拟合的效果较差，而且迭代的次数比较多；而对于共轭梯度下降，在右边的图我们可以发现，其几乎与解析解表现相同，甚至不能明显的区分二者。**

五、结论

- **增加训练样本的数据可以有效的解决过拟合的问题。**
- 对于训练样本限制较多的问题，**通过增加惩罚项仍然可以有效解决过拟合问题。**
- 对于梯度下降法和共轭梯度法而言，**梯度下降收敛速度较慢，共轭梯度法的收敛速度快；且二者相对于解析解而言，共轭梯度法的拟合效果解析解的效果更好。**

六、参考文献

- Pattern Recognition and Machine Learning.
- Gradient descent wiki
- Conjugate gradient method wiki
- Shewchuk J R. An introduction to the conjugate gradient method without the agonizing pain[J]. 1994.

七、附录:源代码(带注释)

主程序 lab1.py

```
import numpy as np
from matplotlib import pyplot as plt
import newton_method
import analytical_solution
import gradient_descent
import conjugate_gradient


def generateData(number, scale=0.3):
    """ Generate training or test data.
    Args:
        number: data number you want which is an integer
        scale: the variance of Gaussian diribution for noise.
    Returns:
        X: a one-dimensional array containing all uniformly distributed x.
        T: sin(2 * pi * x) with Gaussian distribution noise with variance
            of scale.
    """
    assert isinstance(number, int)
    assert number > 0
    assert scale > 0
    X = np.linspace(0, 1, num=number)
    T = np.sin(2 * np.pi * X) + np.random.normal(scale=scale, size=X.shape)
    return X, T


def transform(X, degree=2):
    """
    Transform an array to (len(X), degree + 1) matrix.
    Args:
        X: an ndarray.
        degree:int, degree for polynomial.
    Returns:
        for example, [a b] -> [[1 a a^2] [1 b b^2]]
    """
    assert isinstance(degree, int)
    assert X.ndim == 1
```

```
    X_T = X.transpose()
    X = np.transpose([X])
    features = [np.ones(len(X))]
    for i in range(0, degree):
        features.append(np.multiply(X_T, features[i]))

    return np.asarray(features).transpose()


def predict(X_Test, w):
    """
    用于拟合函数
    Args:
        X_Test 为 (m+1, m+1)的矩阵
        w 为求解得到的系数向量,其维度为m
    """
    return np.dot(X_Test, w)


number_train = 10  # 训练样本的数量
number_test = 100  # 测试样本的数量
degree = 9  # 多项式的阶数
X_training, T_training = generateData(number_train)
X_test = np.linspace(0, 1, number_test)
X_Train = transform(X_training, degree=degree)
X_Test = transform(X_test, degree=degree)
Y = np.sin(2 * np.pi * X_test)


plt.figure(figsize=(20, 10))
title = "degree = " + str(degree) + ", number_train = " + \
    str(number_train) + ", number_test = " + str(number_test)
plt.title(title)
# 用于解析解(不带正则项)的实验
plt.subplot(231)
plt.ylim(-1.5, 1.5)
plt.scatter(X_training, T_training, facecolor="none",
            edgecolor="b", label="training data")
anaSolution = analytical_solution.AnalyticalSolution(X_Train, T_training)
plt.plot(X_test, Y, "g", label="$\sin(2\pi x)$")
plt.plot(X_test, predict(X_Test, anaSolution.fitting()),
         "r", label="analytical solution")
plt.title(title)
plt.legend()


# 用于解析解(带正则项)的实验 寻找最优的超参数
# 经过100次实验(具体的测试数据见实验报告)最终得到的最优参数为e^-7
anaSolution = analytical_solution.AnalyticalSolution(X_Train, T_training)
hyperTestList = []
hyperTrainList = []
hyperList = range(-50, 1)
for hyper in hyperList:
    w_analytical_with_regulation = anaSolution.fitting_with_regulation(
        np.exp(hyper))
```

```
    T_test = predict(X_Test, w_analytical_with_regulation)
    hyperTestList.append(anaSolution.E_rms(T_test, Y))
    hyperTrainList.append(anaSolution.E_rms(T_training, predict(
        transform(X_training, degree=degree), w_analytical_with_regulation)))
bestHyper = hyperList[np.where(hyperTestList ==
                               np.min(hyperTestList))[0][0]]
print("bestHyper:", bestHyper, np.min(hyperTestList))
annotate = "$\lambda = e^{" + str(bestHyper) + "}$"
plt.subplot(232)
plt.ylabel("$E_{RMS}$")
plt.ylim(0, 1)
plt.xlabel("$ln \lambda$")
plt.annotate(annotate, xy=(-30, 0.8))
plt.plot(hyperList, hyperTestList, 'o-', mfc="none", mec="b", ms=5,
         label="Test")
plt.plot(hyperList, hyperTrainList, 'o-',
         mfc="none", mec="r", ms=5, label="Training")
plt.legend()


# 此处用于确认带有惩罚项的解析解的正确性实验
bestHyper = -7  # 此处的最佳的超参数是经过上面提到的实验中确定的
# 求解解析解
anaSolution = analytical_solution.AnalyticalSolution(X_Train, T_training)
w_analytical_with_regulation = anaSolution.fitting_with_regulation(
    np.exp(bestHyper))

print("w_analytical_with_regulation(Analytical solution):\n",
      w_analytical_with_regulation)

annotate = "$\lambda = e^{" + str(bestHyper) + "}$"
plt.subplot(233)
plt.ylim(-1.5, 1.5)
plt.scatter(X_training, T_training, facecolor="none",
            edgecolor="b", label="training data")
plt.plot(X_test, predict(X_Test, w_analytical_with_regulation),
         "r", label="analytical with regulation")
plt.plot(X_test, Y, "g", label="$\sin(2\pi x)$")
plt.annotate(annotate, xy=(0.3, -0.5))
plt.legend()


# 用于测试梯度下降法 并与解析解相对比
# 梯度下降求解
gd = gradient_descent.GradientDescent(X_Train, T_training, np.exp(bestHyper))
w_gradient = gd.fitting(np.zeros(degree + 1))

print("w_gradient(Gradient descent):\n", w_gradient)

plt.subplot(234)
plt.ylim(-1.5, 1.5)
plt.scatter(X_training, T_training, facecolor="none",
            edgecolor="b", label="training data")
plt.plot(X_test, Y, "g", label="$\sin(2\pi x)$")
plt.plot(X_test, predict(X_Test, w_analytical_with_regulation),
```

```
        "r", label="Analytical with regulation")
plt.plot(X_test, predict(X_Test, w_gradient), "c", label="Gradient descent")
plt.legend()


# 测试共轭梯度下降 并与解析解对比
# 共轭梯度求解
cg = conjugate_gradient.ConjugateGradient(
    X_Train, T_training, np.exp(bestHyper))
w_conjugate = cg.fitting(np.zeros(degree + 1))

print("w_conjugate(Conjugate gradient):\n", w_conjugate)

plt.subplot(235)
plt.ylim(-1.5, 1.5)
plt.scatter(X_training, T_training, facecolor="none",
            edgecolor="b", label="training data")
plt.plot(X_test, Y, "g", label="$\sin(2\pi x)$")
plt.plot(X_test, predict(X_Test, w_analytical_with_regulation),
         "r", label="Analytical regulation")
plt.plot(X_test, predict(X_Test, w_conjugate), "m",
         label="Conjugate gradient")
plt.legend()


# 测试牛顿法 并与解析解对比
# 求解牛顿法的解
nm = newton_method.NewtonMethond(X_Train, T_training, np.exp(bestHyper))
w_newton = nm.fitting(np.ones(degree + 1))

print("w_analytical_with_regulation(Analytical solution):\n",
      w_analytical_with_regulation)
print("w_newton(Newton method):\n", w_newton)

plt.subplot(236)
plt.ylim(-1.5, 1.5)
plt.scatter(X_training, T_training, facecolor="none",
            edgecolor="b", label="training data")
plt.plot(X_test, Y, "g", label="$\sin(2\pi x)$")
plt.plot(X_test, predict(X_Test, w_analytical_with_regulation),
         "r", label="Analytical regulation")
plt.plot(X_test, predict(X_Test, w_newton), "k", label="Newton method")
plt.legend()
plt.show()
```

解析解 analytical_solution.py

```
import numpy as np


class AnalyticalSolution(object):
    def __init__(self, X, T):
```

```
        """ 求方程的解析解 """
        self.X = X
        self.T = T

    def fitting(self):
        """ 不带惩罚项的解析解 """
        return np.linalg.pinv(self.X) @ self.T

    def fitting_with_regulation(self, hyper):
        """ 带惩罚项的解析解 """
        return np.linalg.solve(np.identity(len(self.X.T)) * hyper + \
        self.X.T @ self.X, self.X.T @ self.T)

    def E_rms(self, x, y):
        """ 根均方(RMS)误差 """
        return np.sqrt(np.mean(np.square(x-y)))
```

梯度下降法 gradient_descent.py

```
import numpy as np


class GradientDescent(object):
    def __init__(self, X, T, hyper, rate=0.1, delta=1e-6):
        """ Args:
            X, T训练集, 其中X为(number_train, degree + 1)的矩阵
            T为(number_train, 1)的向量
            hyper 为超参数
            rate 为学习率, delta 为停止迭代的条件
        """
        self.X = X
        self.T = T
        self.hyper = hyper
        self.rate = rate
        self.delta = delta

    def __loss(self, w):
        """
        用于求解loss 在本次实验中loss的公式为
        $E(w) = \frac{1}{2N}[(Xw - T)'(Xw - T) + \lambda w'w]$
        """
        wt = np.transpose([w])
        temp = self.X @ w - self.T
        return 0.5 * np.mean(temp.T @ temp + self.hyper * w @ wt)

    def __derivative(self, w):
        """
        一阶函数求导 """
        return np.transpose(self.X) @ self.X @ w + self.hyper * w -\
         self.X.T @ self.T

    def fitting(self, w_0):
```

```
        """
        用于多项式函数使用梯度下降拟合
        Args:
            w_0 初始解，通常以全零向量
         """
        loss0 = self.__loss(w_0)
        k = 0
        w = w_0
        while True:
            wt = w - self.rate * self.__derivative(w)
            loss = self.__loss(wt)
            if np.abs(loss - loss0) < self.delta:
                break
            else:
                k = k + 1
                # print(k) # 记录迭代次数
                # print("loss:", loss)
                if loss > loss0:
                    # 当loss函数相比上次增大 学习率衰减为之前的一半
                    self.rate *= 0.5
                loss0 = loss
                w = wt
        return w
```

共轭梯度法 conjugate_gradient.py

```
import numpy as np


class ConjugateGradient(object):
    def __init__(self, X, T, hyper, delta=1e-6):
        """ 共轭梯度下降
        Args:
            X, T 训练集 其中X为(m+1, m+1)的矩阵 T为(N, 1)的向量
            hyper 超参数
            delta 迭代停止条件 """
        self.X = X
        self.T = T
        self.hyper = hyper
        self.delta = delta
        self.A = X.T @ X + np.identity(len(X.T)) * hyper
        # 将问题转化为 Ax = b的求解
        # 因此将其中的A和b转为实验带求的方程的参数
        self.b = X.T @ T

    def fitting(self, w_0):
        r_0 = self.b - self.A @ w_0
        w = w_0
        p = r_0
        k = 0
        while True:
            k = k + 1
```

```
            alpha = np.linalg.norm(r_0) ** 2 / (np.transpose(p) @ self.A @ p)
            w = w + alpha * p
            r = r_0 - alpha * self.A @ p # 当前的残差
            # print(k, r)
            if(np.linalg.norm(r) ** 2 < self.delta):
                break
            beta = np.linalg.norm(r)**2 / np.linalg.norm(r_0)**2
            p = r + beta * p # 下次的搜索方向
            r_0 = r
        return w

    def fitting_standford(self, w_0):
        w = w_0
        r = self.b - self.A @ w_0
        M = np.linalg.inv(self.A)
        p = r
        z = M @ r
        rho_0 = r.T @ z
        for i in range(len(w_0)):
            omega = self.A @ p
            alpha = rho_0 / (omega @ p)
            w = w + alpha * p
            print(i, w)
            r = r - alpha * omega
            z = M @ r
            rho = z.T @ r
            p = z + rho / rho_0 * p
            rho_0 = rho
        return w
```

牛顿法 newton_method.py

```
import numpy as np


class NewtonMethond(object):
    def __init__(self, X, T, hyper, delta=1e-6):
        """
        Args:
            X, T训练集, 其中X为(number_train, degree + 1)的矩阵
            T为(number_train, 1)的向量
            hyper 为超参数
            delta 为迭代停止条件
        """
        self.X = X
        self.T = T
        self.hyper = hyper
        self.delta = delta

    def __derivative(self, w):
        """
        求函数的一阶导数 即 $J(w) = (X'X + \lambda I)w - X'T$
```

```
    """
    return np.transpose(self.X) @ self.X @ w + self.hyper * w \
    - self.X.T @ self.T

def __second_derivative(self):
    """
    求函数的二阶导数的逆 即hessian矩阵的逆 """
    return np.linalg.pinv(self.X.T @ self.X + self.hyper \
        * np.identity(len(self.X.T[0])))

def fitting(self, w_0):
    """
    用于牛顿法求解方程的解 """
    w = w_0
    while True:
        gradient = self.__derivative(w) # 梯度
        if np.linalg.norm(gradient) < self.delta:
            break
        wt = w - self.__second_derivative() @ gradient
        w = wt
    return w
```