机器学习实验报告一——多项式拟合正弦函数

李一鸣
2018 年 9 月 25 日

网址：https://upupming.site/Lab1-polynomial-curve-fitting/README.html
源代码：https://github.com/upupming/Lab1-polynomial-curve-fitting

多项式拟合正弦函数

班号	学号	姓名
1603103	1160300625	李一鸣

〇、数学符号格式规范

本文格式参考了 [Bishop 2006]，若不加特殊说明，定义向量时均是列向量。

数据类型	格式规范	MathJax 写法	实际效果
向量	小写加粗罗马字母	`\mathrm{\mathbf{w}}`	$\mathrm{\mathbf{w}}$
转置	右上标 T	`(w_0, w_1, ..., w_M)^T`	$(w_0, w_1, ..., w_M)^T$
随机变量分布类型	书法字母 (calligraphic letters)	`\mathcal{N}`	$\mathcal{N}$
矩阵	大写加粗罗马字母	`\mathrm{\mathbf{X}}`	$\mathrm{\mathbf{X}}$

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

- [x] 生成数据，加入噪声
- [x] 用高阶多项式函数拟合曲线
- [x] 用解析解求解两种 loss 的最优解（无正则项和有正则项）
- [x] 优化方法求解最优解（梯度下降，共轭梯度）
- [x] 用你得到的实验数据，解释过拟合
- [x] 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

实验环境

硬件

- Windows 10 64-bit
- Python 3.7.0

软件

- Matplotlib

  Python 2D 绘图库

- NumPy

  矩阵运算

三、设计思想

本次实验的目标是对正弦函数曲线 $sin(2\pi x)$ 进行拟合。简单来讲，就是根据已有数据集 (x_i, y_i) 找到一条曲线，使其能够最好地预测真实情况下给定 x 后计算出的 y 值。正弦函数只是一个具体的例子，理解了算法的思想之后，我们可以对任意曲线进行拟合。

算法原理

向量、矩阵均采用数据结构均是 Numpy 中提供的数组结构。

生成数据

编写一个函数，根据用户传入的函数生成指定数量的数据，利用 Numpy 库提供的 `numpy.random.normal` 加入以 0 为均值、用户指定方差的噪声。以字典 `{'xArray': array, 'tArray': array}` 的形式返回给用户。

最小二乘法

误差函数 $E(\mathbf{w})$

最小二乘法通过**最小化误差的平方和**寻找数据的最佳函数匹配。

我们使用下面的多项式函数来拟合数据:

$$y(x,\mathbf{w})=w_0+w_1x+w_2x^2+...+w_Mx^M=\sum_{j=0}^{M}w_jx^j \tag{1}$$

其中 M 为多项式阶数，多项式系数向量 $\mathbf{w}=(w_0,w_1,...,w_M)^T$。

显然

$$y(x,\mathbf{w})=\begin{pmatrix}1 & x & ... & x^M\end{pmatrix}\mathbf{w} \tag{2}$$

其中

为了求得最优的解析解，我们需要最小化 N 个数据误差的平方和，也就是最小化下面的误差函数：

$$E(\mathbf{w})=\frac{1}{2}\sum_{n=1}^{N}\{y(x_n,\mathbf{w})-t_n\}^2 \tag{3}$$

误差函数中的 $\frac{1}{2}$ 是利用高斯分布进行最大似然估计得来的。假定在 x 已知的情况下，相应的 t 服从以 $y(x,\mathbf{w})$ 为均值、以 β^{-1} 为方差的高斯分布，即：

$$\begin{aligned}p(t|x,\mathbf{w},\beta)&=\mathcal{N}(t|y(x,\mathbf{w}),\beta^{-1})\\&=\frac{\exp-\frac{[t-y(x,\mathbf{w})]^2}{2\beta^{-1}}}{(2\pi\beta^{-1})^{\frac{1}{2}}}\end{aligned} \tag{4}$$

由于每个数据都是独立同分布的，根据乘法公式，对于训练数据集 $\mathbf{x}=(x_1,x_2,...,x_N)^T$ 和 $\mathbf{t}=(t_1,t_2,...,t_N)^T$ 则有：

$$\begin{aligned}p(\mathbf{t}|\mathbf{x},\mathbf{w},\beta)&=\prod_{n=1}^{N}\mathcal{N}(t_n|y(x_n,\mathbf{w}),\beta^{-1})\\&=\prod_{n=1}^{N}\frac{\exp-\frac{[y(x_n,\mathbf{w})-t_n)]^2}{2\beta^{-1}}}{(2\pi\beta^{-1})^{\frac{1}{2}}}\end{aligned} \tag{5}$$

这就是似然函数，即在 $\mathbf{w}$ 已知的条件下，给定 $\mathbf{x}$ 利用拟合曲线得到的估计值正好是 $\mathbf{t}$ 的概率。我们的任务就是最大化似然函数，由于大量小概率的乘积很容易下溢，于是我们转而计算概率的对数和：

$$\begin{aligned}\ln p(\mathbf{t}|\mathbf{x},\mathbf{w},\beta)&=\sum_{n=1}^{N}\{-\frac{\beta}{2}[y(x_n,\mathbf{w})-t_n)]^2-\frac{1}{2}\ln\frac{2\pi}{\beta}\}\\&=-\frac{\beta}{2}\sum_{n=1}^{N}[y(x_n,\mathbf{w})-t_n)]^2+\frac{N}{2}\ln\beta-\frac{N}{2}\ln 2\pi\end{aligned} \tag{6}$$

现假设系数向量 $\mathbf{w}$ 取 $\mathbf{w_{ML}}$ 时似然函数达到最大值。为了求解 $\mathbf{w_{ML}}$，可以忽略后两项，同时将第一项中的 $-\frac{\beta}{2}$ 替换为 $\frac{1}{2}$ 也不会影响计算结果。也就是要最大化：

$$-\frac{1}{2}\sum_{n=1}^{N}[y(x_n,\mathbf{w})-t_n)]^2 \tag{7}$$

这等价于最小化误差函数 (3)。

最小化误差函数求得最优解 $\mathbf{w}_{\mathrm{ML}}$

下面对 (3) 做一些变形：

$$
\begin{aligned}
E(\mathbf{w}) &= \frac{1}{2}\sum_{n=1}^{N}\left[y(x_n,\mathbf{w})-t_n\right]^2 \\
&= \frac{1}{2}\left\|\begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^M \\ 1 & x_2 & x_2^2 & \dots & x_2^M \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_N & x_N^2 & \dots & x_N^M \end{pmatrix}_{N\times(M+1)} \mathbf{w}_{(M+1)\times 1} - \mathbf{t}_{\mathrm{N}\times 1}\right\|^2
\end{aligned}
$$

令

$$
\mathbf{X} = \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^M \\ 1 & x_2 & x_2^2 & \dots & x_2^M \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_N & x_N^2 & \dots & x_N^M \end{pmatrix}_{N\times(M+1)} \tag{8}
$$

则

$$
\begin{aligned}
E(\mathbf{w}) &= \frac{1}{2}\left\|\mathbf{X}\mathbf{w}-\mathbf{t}\right\|^2 \\
&= \frac{1}{2}(\mathbf{X}\mathbf{w}-\mathbf{t})^T(\mathbf{X}\mathbf{w}-\mathbf{t}) \\
&= \frac{1}{2}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}-\mathbf{w}^T\mathbf{X}^T\mathbf{t}-\mathbf{t}^T\mathbf{X}\mathbf{w}+\mathbf{t}^T\mathbf{t})
\end{aligned} \tag{9}
$$

接下来我们需要求导，在下面的计算过程中要十分注意矩阵求导是该采用分子布局还是分母布局，下面引用一段 [Matrix calculus | Wikipedia] 中的原话：

> 1. If we choose numerator layout for $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$, we should lay out the gradient $\frac{\partial y}{\partial \mathbf{x}}$ as a row vector, and $\frac{\partial \mathbf{y}}{\partial x}$ as a column vector.
> 2. If we choose denominator layout for $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$, we should lay out the gradient $\frac{\partial y}{\partial \mathbf{x}}$ as a column vector, and $\frac{\partial \mathbf{y}}{\partial x}$ as a row vector.
> 3. In the third possibility above, we write $\frac{\partial y}{\partial \mathbf{x}'}$ and $\frac{\partial \mathbf{y}}{\partial x}$, and use numerator layout.

我们在接下来的计算中选择第 1 种规约 (Consistent numerator layout)，$\frac{\partial \mathbf{y}}{\partial x}$ 以 $\mathbf{y}$ 布局，$\frac{\partial y}{\partial \mathbf{x}}$ 以 $\mathbf{x}^T$ 布局，那么就有

$$
\frac{\partial \mathbf{w}^T\mathbf{A}\mathbf{w}}{\partial \mathbf{w}} = 2\mathbf{w}^T\mathbf{A}
$$

$$
\frac{\partial \mathbf{w}^T\mathbf{A}}{\mathbf{w}} = \frac{\partial \mathbf{A}^{\mathrm{T}}\mathbf{w}}{\mathbf{w}} = \mathbf{A}^T
$$

（注：当分母、分子布局一样应该将分母进行转置之后再计算；这两个求导法则非常经典，读者可以尝试推导一下）

(10)

对 $\mathbf{w}$ 求导，得：

$$\begin{aligned}\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} &= \frac{\partial \frac{1}{2}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{t} - \mathbf{t}^T\mathbf{X}\mathbf{w} + \mathbf{t}^T\mathbf{t})}{\partial \mathbf{w}} \\ &= \frac{1}{2}(2\mathbf{w}^T\mathbf{X}^T\mathbf{X} - 2\mathbf{t}^T\mathbf{X}) \\ &= \mathbf{w}^T\mathbf{X}^T\mathbf{X} - \mathbf{t}^T\mathbf{X}\end{aligned} \tag{11}$$

令导数为 0 解得：

$$\mathbf{w_{ML}} = (\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{t}) \tag{12}$$

总结：对于已知训练数据集 $\mathbf{x}$、$\mathbf{t}$，规定一个多项式次数 M，根据相应的范德蒙德矩阵 (8) 以及式 (12) 就可以解出最优解 $\mathbf{w_{ML}}$。再根据 (2) 即可计算估计值从而获得拟合曲线。

然而，当多项式次数 M 刚好等于 $N-1$ 时，能够使训练数据集的误差函数取值为 0（例：2 次函数可以完全拟合 3 个点），但对测试数据集却不能很好地拟合，这就是**过拟合**。经过查阅资料，主要有这几种解决方法：

1. 增大数据量
2. 使用贝叶斯方法，根据数据集的规模自动调节有效参数数量
3. 修正误差函数，加入惩罚项，对其进行正则化 (regularization)

下面讨论加入惩罚项的最小二乘法。

带惩罚项的最小二乘法

修正误差函数，在 (3) 式的基础上加入正则项：

$$\begin{aligned}\widetilde{E}(\mathbf{w}) &= \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2 \\ &= \frac{1}{2}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{t} - \mathbf{t}^T\mathbf{X}\mathbf{w} + \mathbf{t}^T\mathbf{t}) + \frac{\lambda}{2}(\mathbf{w}^T\mathbf{w})\end{aligned} \tag{13}$$

其中 λ 调节正则项、平方和两者之间的比例。

将其对 $\mathbf{w}$：

$$\frac{\partial \widetilde{E}(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{w}^T\mathbf{X}^T\mathbf{X} - \mathbf{t}^T\mathbf{X} + \frac{\lambda}{2}\frac{\partial(\mathbf{w}^T\mathbf{w})}{\partial \mathbf{w}}$$

这里同样需要用到 [Matrix calculus | Wikipedia] 中的矩阵求导的一个 (scalar-by-vector) 公式：

$$\frac{\partial(\mathbf{w}^T\mathbf{w})}{\partial \mathbf{w}} = 2\mathbf{w}^T \tag{14}$$

于是得到：

$$\frac{\partial \widetilde{E}(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{w}^T\mathbf{X}^T\mathbf{X} - \mathbf{t}^T\mathbf{X} + \lambda\mathbf{w}^T \tag{15}$$

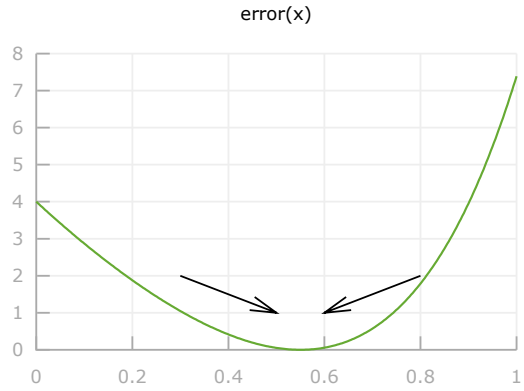
令导数为 0，解得：

$$\mathbf{w_{ML}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_{(M+1)(M+1)})^{-1}(\mathbf{X}^T\mathbf{t}) \tag{16}$$

总结：带惩罚项的最小二乘法解析解与普通最小二乘法相比，相差不大，只是在左边乘积项中多了一个单位矩阵的 λ 倍。

梯度下降法

说到梯度下降法，我们可以先来看看**导数下降法**。对于下面的函数 f：



我们可以用这种方法求解 $f(x) = 0$ 的解：

1. 初始化 $x = 0$（任意值都可以），给定精确度 $\alpha = 0.2$
2. $f'(0) < 0$，令 $x = x + \alpha = 0.2$
3. $f'(0.2) < 0$，令 $x = x + \alpha = 0.4$
4. $f'(0.4) < 0$，令 $x = x + \alpha = 0.6$
5. $f'(0.6) > 0$，令 $x = x - \alpha = 0.4$

可以看到，按照这个计算过程 x 会一直在 0.4 和 0.6 两个数之间变化。实际计算时应该减去导数值与 α（学习率）的乘积，如果我们将 α 取的足够小，并让这个迭代过程在可接受的误差范围内停止的话，我们最终将得到方程的解。当然，α 也不宜太小，否则会导致误差下降缓慢，计算耗时太长。

梯度下降法则是对向量函数进行类似的处理。梯度的方向是函数上升最快的方向，其反方向则是函数下降最快的方向。我们的目的就是找到误差函数 $\widetilde{E}(\mathbf{w})$ 取最小值的点，我们可以从任意点开始沿着其梯度反方向以学习率为步长进行迭代，最终在误差许可范围内停止。

对误差函数 $\widetilde{E}(\mathbf{w})$ 求梯度得：

$$\begin{aligned}\nabla\widetilde{E}(\mathbf{w}) &= \nabla[\frac{1}{2}\left\|\mathbf{X}\mathbf{w} - \mathbf{t}\right\|^2 + \frac{\lambda}{2}(\mathbf{w}^T\mathbf{w})] \\ &= \mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{t}) + \lambda\mathbf{w}\end{aligned} \tag{17}$$

共轭梯度法

共轭梯度法是求解系数矩阵为对称正定矩阵的线性方程组的数值解的方法。共轭梯度法是一个迭代方法，它适用于系数矩阵为稀疏矩阵的线性方程组，因为使用像 Cholesky 分解这样的直接方法求解这些系统所需的计算量太大了。这种方程组在数值求解偏微分方程时很常见。[Conjugate gradient method | Wikipedia]

现有正定对称矩阵 $\mathbf{A}$，如果

$$\mathbf{u}^T\mathbf{A}\mathbf{v} = 0 \tag{18}$$

成立，则称 $\mathbf{u}$、$\mathbf{v}$ 关于 $\mathbf{A}$ 共轭,此时上式可用 $\langle\mathbf{u}, \mathbf{v}\rangle_{\mathbf{A}}$ 表示。

在上文中我们看到的 $\mathbf{X}^T\mathbf{X}$ 显然是满足正定且对称这个条件的，因为：

$$\mathbf{P}^T\mathbf{X}^T\mathbf{X}\mathbf{P} = \|\mathbf{X}\mathbf{P}\|^2 > 0 \quad \text{(正定性)}$$

$$(\mathbf{X}^T\mathbf{X})^T = \mathbf{X}^T\mathbf{X} \quad \text{(对称性)}$$

由式(15)在 $\mathbf{w} = \mathbf{w_{ML}}$ 时为 0 可得:

$$\begin{aligned}(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_{(M+1)\times(M+1)})\mathbf{w_{ML}} &= \mathbf{X}^T\mathbf{t} \\ \mathbf{B}\mathbf{w_{ML}} &= \mathbf{X}^T\mathbf{t}\end{aligned} \tag{19}$$

其中

$$\mathbf{B} = \mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_{(M+1)\times(M+1)} \tag{20}$$

显然 $\mathbf{B}$ 也是正定、对称的。

定义矩阵 $\mathbf{P} = (\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_{M+1}})$，其中的每一个向量都是关于 $\mathbf{B}$ 共轭的，并且构成了一组基底，任何向量都可以由它们组合产生。先假设我们要求的 $\mathbf{w_{ML}}$ 为:

$$\mathbf{w_{ML}} = \sum_{i=1}^{M+1} \alpha_i \mathbf{p_i} \tag{21}$$

两边同乘 $\mathbf{B}$ 有:

$$\mathbf{B}\mathbf{w_{ML}} = \sum_{i=1}^{M+1} \alpha_i \mathbf{B}\mathbf{p_i} \tag{22}$$

再在两边同乘 $\mathbf{p_k}^T$，有:

$$\mathbf{p_k}^T\mathbf{B}\mathbf{w_{ML}} = \sum_{i=1}^{M+1} \alpha_i \mathbf{p_k}^T\mathbf{B}\mathbf{p_i} \tag{23}$$

由式(19)我们知道 $\mathbf{B}\mathbf{w_{ML}} = \mathbf{X}^T\mathbf{t}$，从而:

$$\mathbf{p_k}^T\mathbf{X}^T\mathbf{t} = \sum_{i=1}^{M+1} \alpha_i \langle \mathbf{p_k}, \mathbf{p_i} \rangle_{\mathbf{B}} \tag{24}$$

用 $\langle \mathbf{u}, \mathbf{v} \rangle$ 表示 $\mathbf{u}^T\mathbf{v}$，同时使用共轭的已知条件 $\forall i = \not{k}: \langle \mathbf{p_k}, \mathbf{p_i} \rangle_{\mathbf{B}} = 0$，可以得到:

$$\langle \mathbf{p_k}, \mathbf{X}^T\mathbf{t} \rangle = \alpha_k \langle \mathbf{p_k}, \mathbf{p_k} \rangle_{\mathbf{B}} \tag{25}$$

从而解得:

$$\alpha_k = \frac{\langle \mathbf{p_k}, \mathbf{X}^T\mathbf{t} \rangle}{\langle \mathbf{p_k}, \mathbf{p_k} \rangle_{\mathbf{B}}} \tag{26}$$

总结：利用共轭梯度法我们可以从 $\mathbf{p}_0$ 开始逐渐向初始点的共轭方向逼近，加快求解效率。最优解由式 (20)、式 (21)、式 (26) 给出。

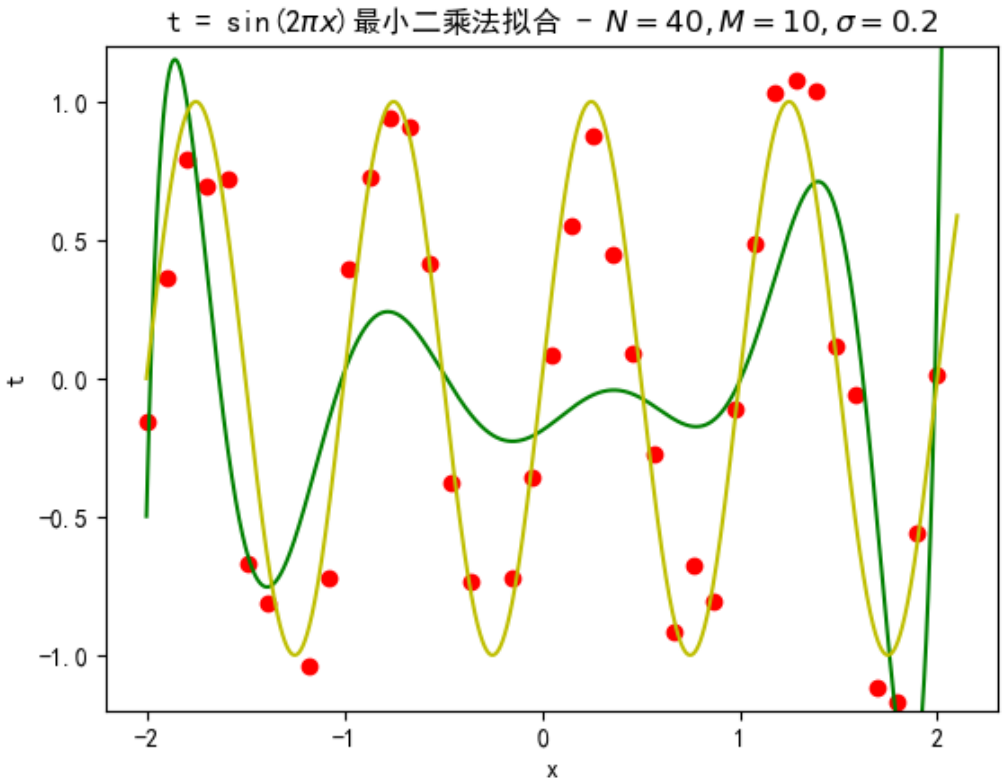
四、实验结果与分析

最小二乘法

运行命令：

```
$ make least_squares
```

N = 40, M = 10

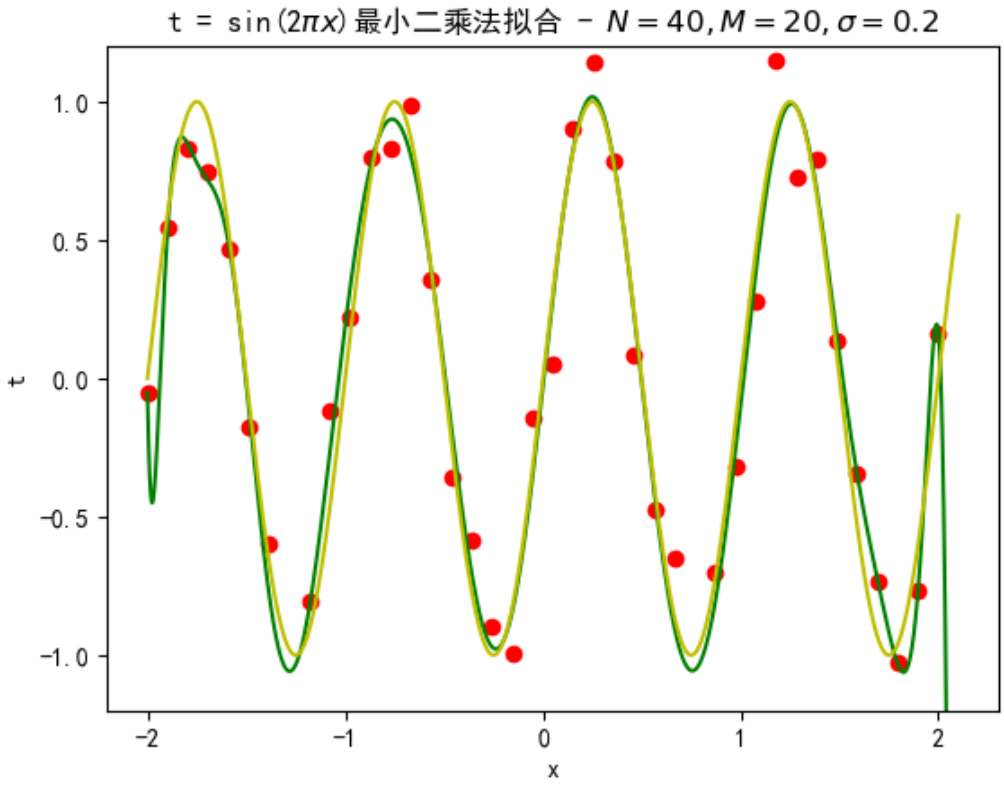


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_10] =
        [-0.18852093  0.46764938  0.96165658 -3.19320603 -1.43929633  4.26944798
  0.89267684 -1.78949764 -0.24462765  0.22946408  0.02422608]
```

拟合效果不太好，很多点相距曲线较远。同时相对正弦函数有一些误差。

N = 40, M = 20

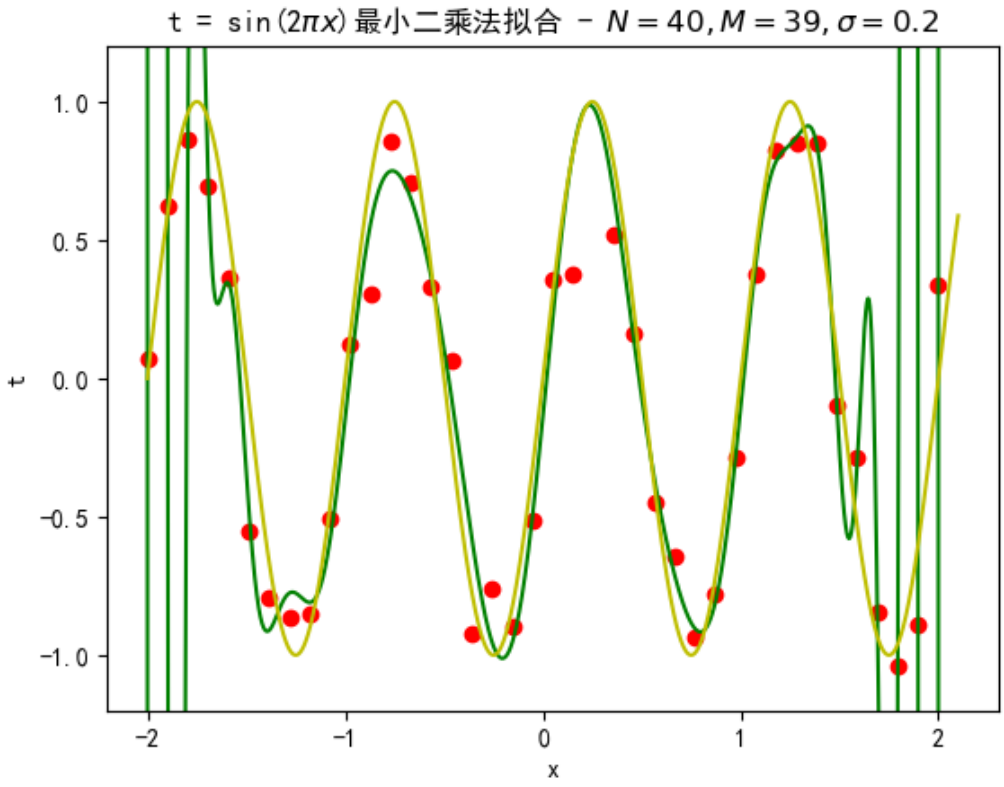


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_20] =
        [-2.83189643e-02  6.44486656e+00  1.14232907e+00 -4.53984181e+01
 -6.16291034e+00  1.04519205e+02  9.79506324e+00 -1.31726786e+02
 -2.30323063e+00  1.10139919e+02 -8.33376050e+00 -6.30102026e+01
  9.65748753e+00  2.37478078e+01 -4.83955875e+00 -5.53899824e+00
  1.28735286e+00  7.18432309e-01 -1.77531943e-01 -3.94562349e-02
  1.00096458e-02]
```

对训练数据拟合度比较高，同时拟合曲线比较符合正弦曲线的轨迹。

N = 40, M = 39

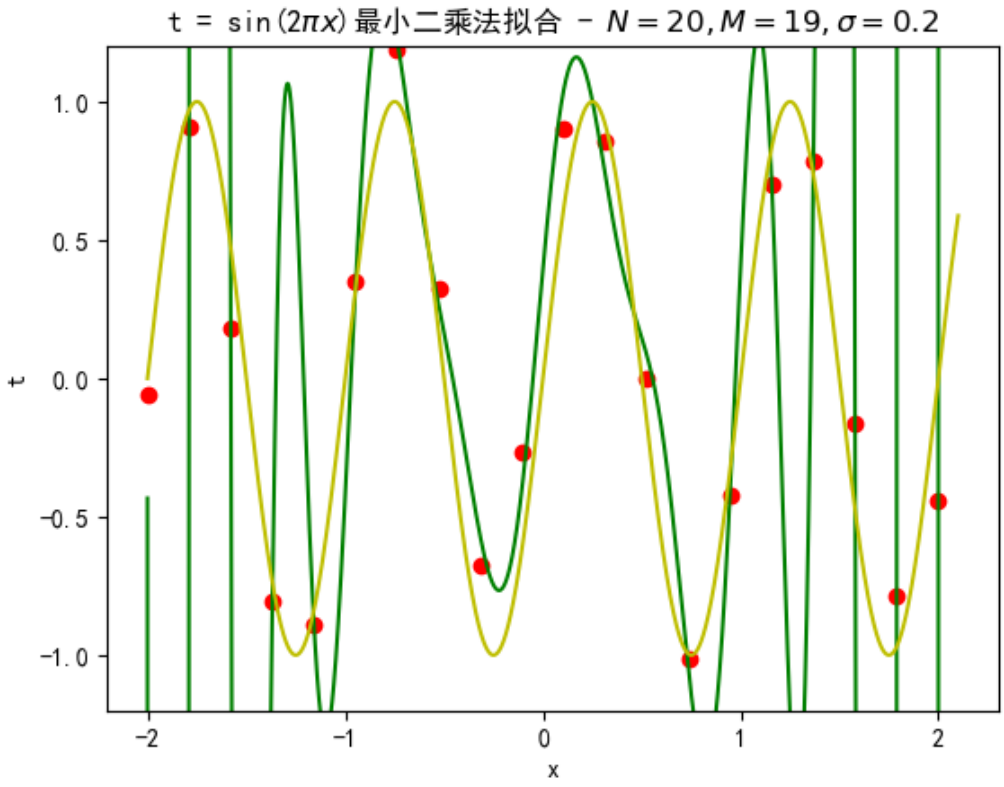


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_39] =
       [-1.37730928e-01  7.20011167e+00  3.78940964e+00 -6.54217300e+01
 -2.97538302e+01  2.27998662e+02  1.15015360e+02 -4.41036415e+02
 -2.57986663e+02  4.68833665e+02  3.48980403e+02 -2.08874608e+02
 -2.84712598e+02 -5.19293571e+01  1.31497249e+02  9.01931307e+01
 -2.52221608e+01 -2.59461545e+01 -3.46148449e+00 -2.44292100e+00
  1.78139942e+00  1.03220160e+00  2.20023558e-01  5.10055655e-01
 -1.53982743e-01 -7.15784584e-02  2.70768483e-02 -1.61203786e-02
 -1.01308220e-02 -1.24368070e-02  1.54215458e-03  1.97230898e-03
  6.01516254e-04  1.40507724e-03 -1.34604119e-04 -1.93954562e-04
 -1.06634393e-05 -5.04452632e-05  2.81865346e-06  8.14970835e-06]
```

可以看到几乎所有点都被拟合了，但是图像与正弦函数相比有较大的误差，有些地方曲线波动非常大。这就是**过拟合**现象。

N = 20, M = 19



$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_19] =
        [ 3.94600350e-01  7.81823176e+00 -6.89831144e+00 -9.52288036e+01
  5.15981932e+01  4.99806633e+02 -1.65924216e+02 -1.44166505e+03
  2.74554816e+02  2.33489154e+03 -2.56802371e+02 -2.18417776e+03
  1.40385217e+02  1.19842711e+03 -4.41962237e+01 -3.78496021e+02
  7.38475847e+00  6.33790112e+01 -5.04968006e-01 -4.33951133e+00]
```

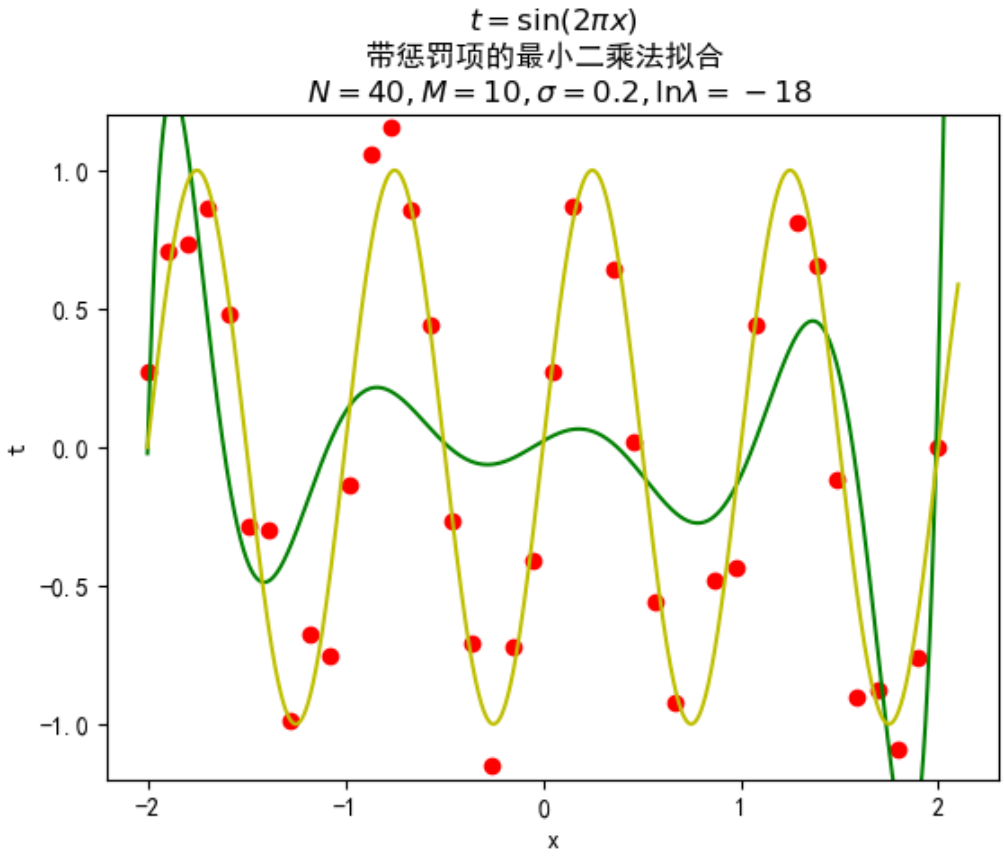
跟上一种情况类似，几乎所有点都被拟合了，但是图像与正弦函数相比有较大的误差，有许多波动大的地方。同样出现了过拟合现象。

带惩罚项的最小二乘法

运行命令：

```
$ make least_squares_regularization
```

N = 40, M = 10

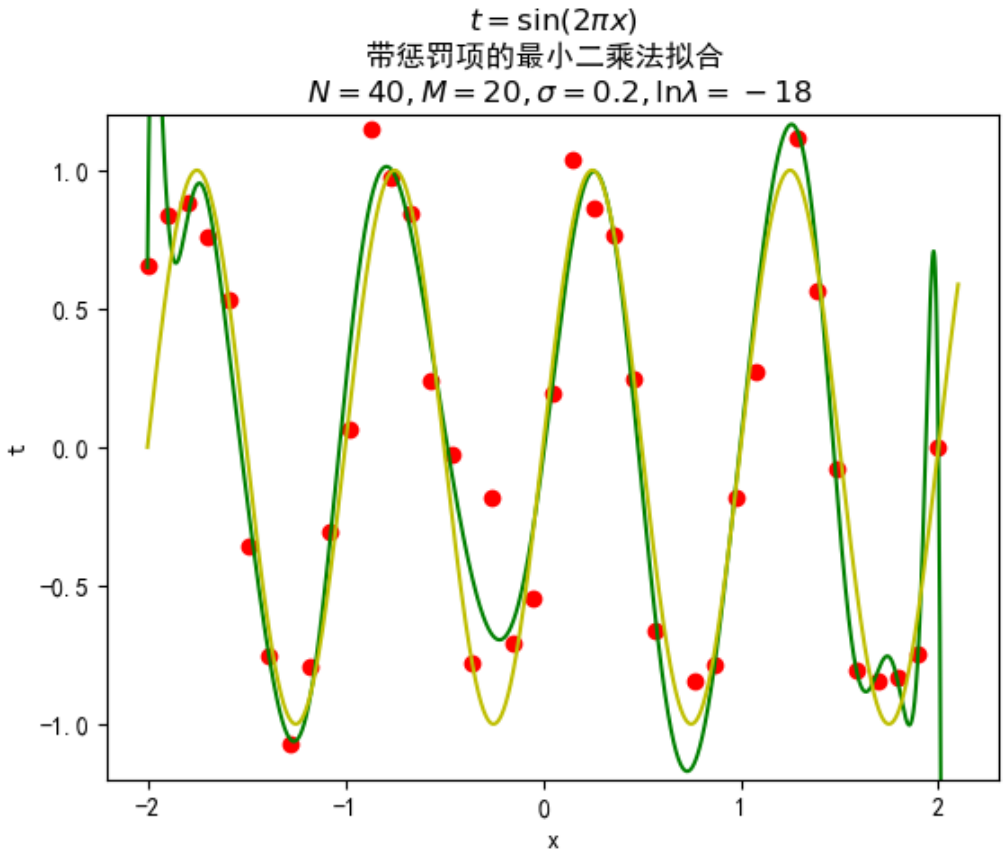


$\mathbf{w}_{\mathrm{ML}} =$

```
[w_0 w_1 ... w_10] =
        [ 0.0217192   0.40841178 -0.43229322 -2.95248222  0.846873    3.80375491
 -0.55453236 -1.57676222  0.14005201  0.20130217 -0.01178719]
```

对训练数据的拟合效果较差，同时相比于正弦函数误差较大，需要增加多项式系数来更好地拟合。

N = 40, M = 20

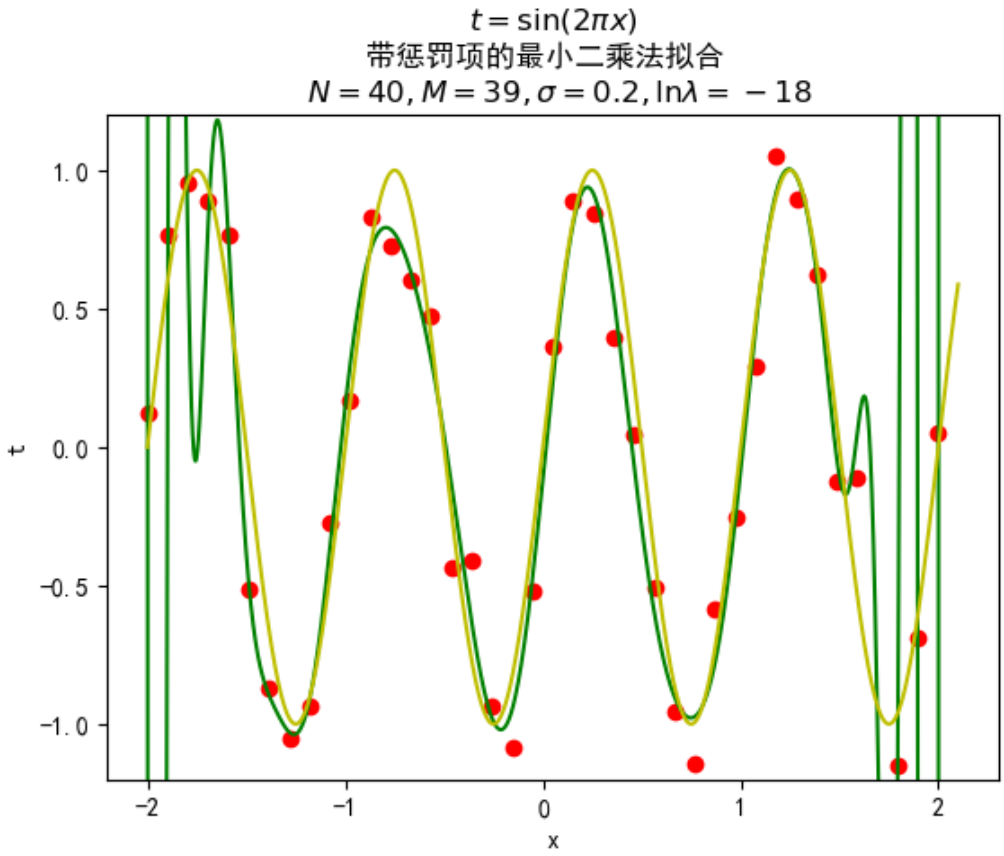


$\mathbf{w}_{\mathrm{ML}} =$

```
[w_0 w_1 ... w_20] =
        [-4.38887174e-02  5.39145965e+00  5.68850337e+00 -3.69229225e+01
 -4.87387946e+01  7.41077662e+01  1.54486407e+02 -7.54974009e+01
 -2.50388565e+02  5.12815965e+01  2.36667326e+02 -2.58574305e+01
 -1.37960721e+02  9.27702058e+00  5.02125993e+01 -2.13398723e+00
 -1.11008108e+01  2.75002306e-01  1.36209559e+00 -1.49529033e-02
 -7.10831890e-02]
```

拟合效果比较好，有一些点都没有落在拟合曲线上。拟合曲线比较符合正弦的轨迹。

N = 40, M = 39

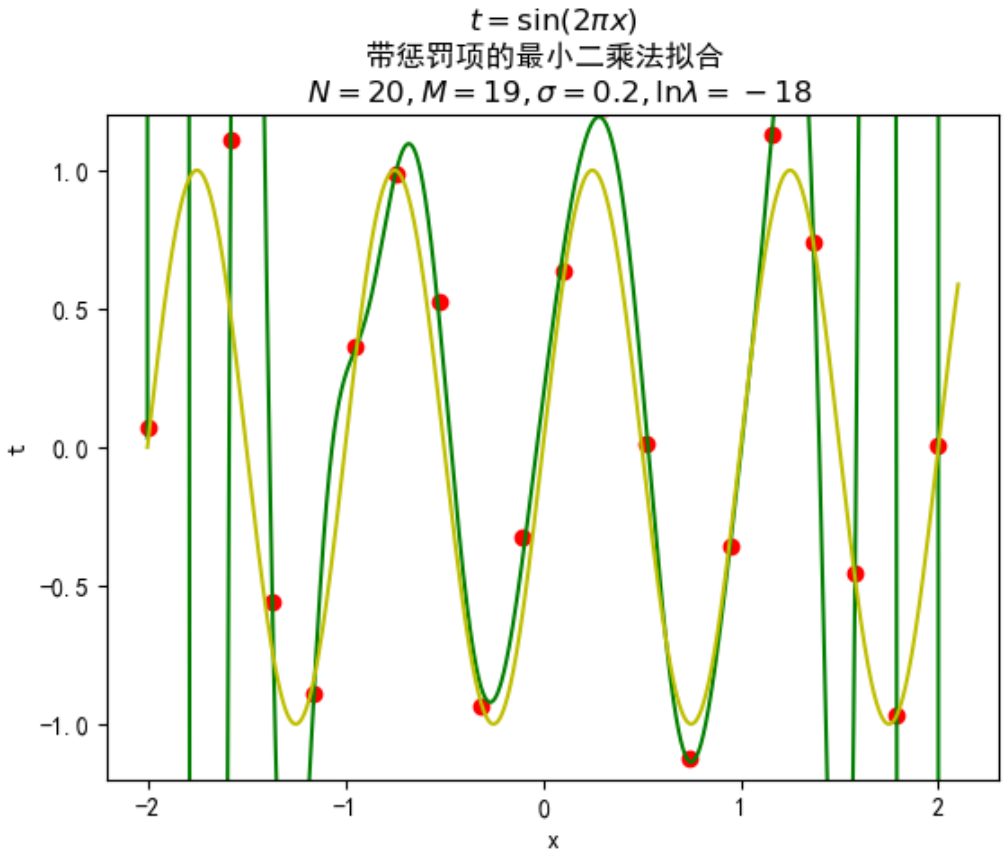


$\mathbf{w}_{\mathrm{ML}} =$

```
[w_0 w_1 ... w_39] =
        [-1.15793582e-01  7.10902374e+00  2.67462073e+00 -6.45744462e+01
 -2.81845074e+01  2.17658536e+02  1.16936951e+02 -4.13028697e+02
 -2.62541864e+02  4.64199909e+02  3.57282149e+02 -2.88655110e+02
 -3.00498355e+02  7.49773090e+01  1.48145956e+02  9.86797190e+00
 -3.40429061e+01 -7.65234795e+00 -1.45985338e+00 -5.53396128e-01
  1.86437391e+00  4.53977265e-01  5.65314405e-02  9.69407791e-02
 -8.36397903e-02 -1.46452268e-03  1.02812669e-03 -1.37301140e-02
 -5.03505365e-03 -2.62872028e-03  1.43656293e-03  1.08385366e-03
  5.02223439e-04  3.30782203e-04 -1.11968993e-04 -9.61001656e-05
 -1.62210540e-05 -2.46803086e-06  3.43505060e-06  1.39277743e-06]
```

拟合效果比较好，有一些点都没有落在拟合曲线上。拟合曲线也比较符合正弦的轨迹，但不如 $M = 20$ 时，可以考虑增大 λ。

N = 20, M = 19



$\mathbf{w}_{\mathrm{ML}} =$

```
[w_0 w_1 ... w_19] =
        [ 1.72968645e-01  5.61466341e+00 -2.26815059e+00 -1.93159450e+01
  3.29354919e+01 -8.49558296e+01 -1.43903761e+02  4.51413584e+02
  2.78872574e+02 -8.27438168e+02 -2.84730808e+02  7.97858067e+02
  1.63757378e+02 -4.41115349e+02 -5.31400413e+01  1.39581825e+02
  9.04516116e+00 -2.33855863e+01 -6.25696969e-01  1.60162205e+00]
```

拟合效果很好，基本都落在拟合曲线上。拟合曲线有些许波动，不过比不加正则项要好许多。可以考虑进一步增大 λ 来降低误差。

梯度下降法

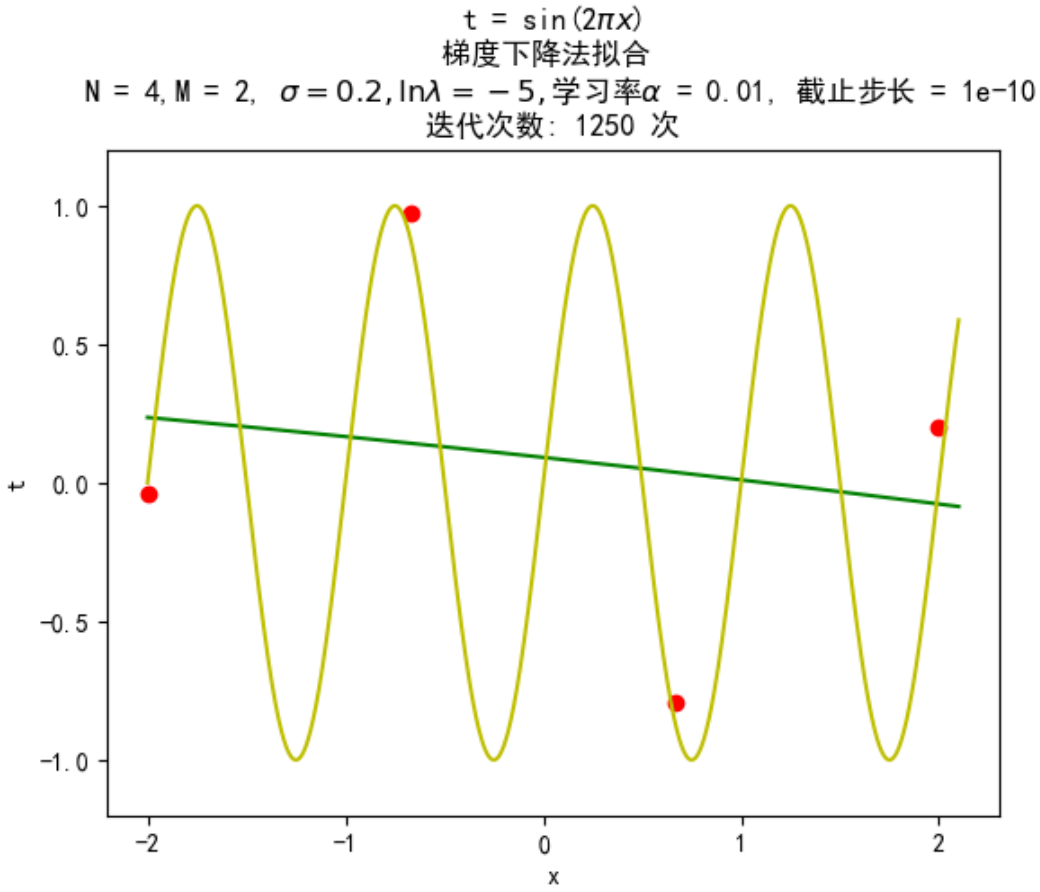
由于 $\lambda = 0$ 时等价于没有惩罚项，而确实在惩罚项存在时过拟合能被较好地避免，下面只考虑 $\ln\lambda = -0.5$ 的情况如果感兴趣，可自行在 `gradient_descent` 增加指定参数的测试用例。

通过实验，我可算是理解到了梯度下降的本质：不断地调节学习率，既不能太大以确保收敛，又不能太小让计算机能在有效时间内给出结果。

运行命令：

```
$ make gradient_descent
```

N = 4, M = 2

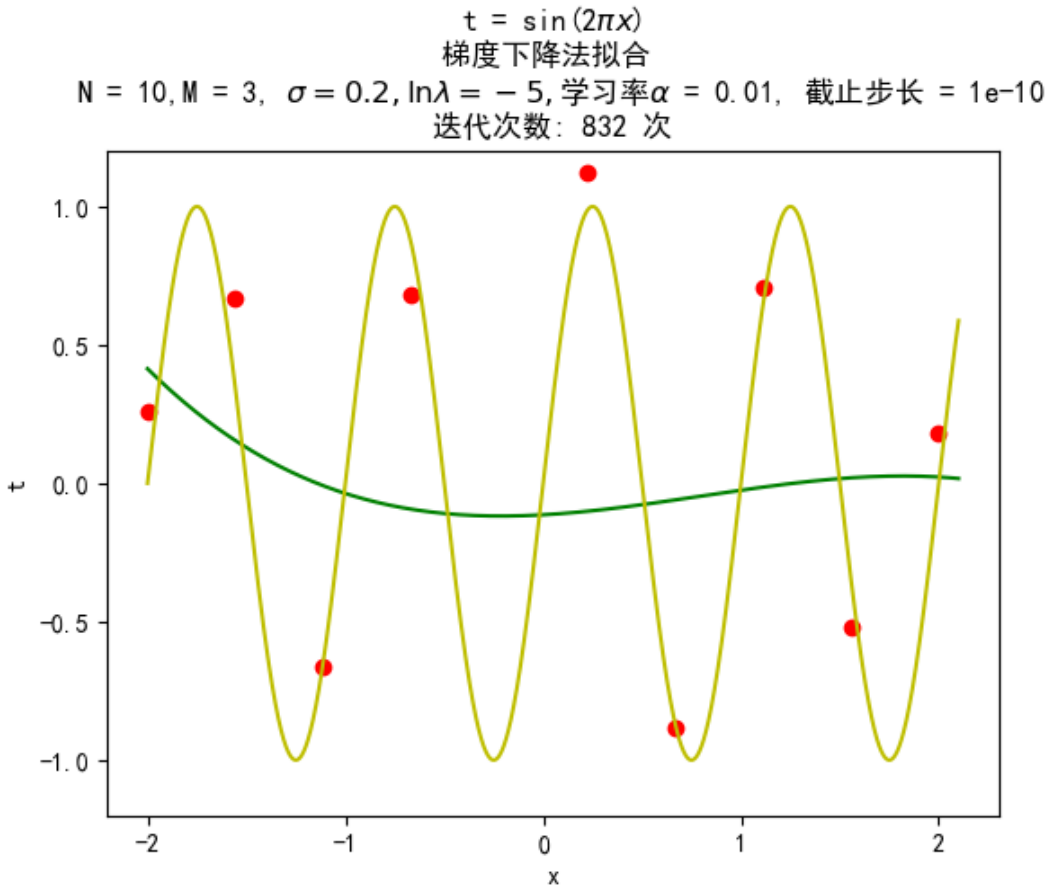


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_2] =
        [ 0.09188777 -0.07816481 -0.00300807]
```

因为数据点比较少，拟合效果一般，也不太符合正弦曲线的特征。

N = 10, M = 3

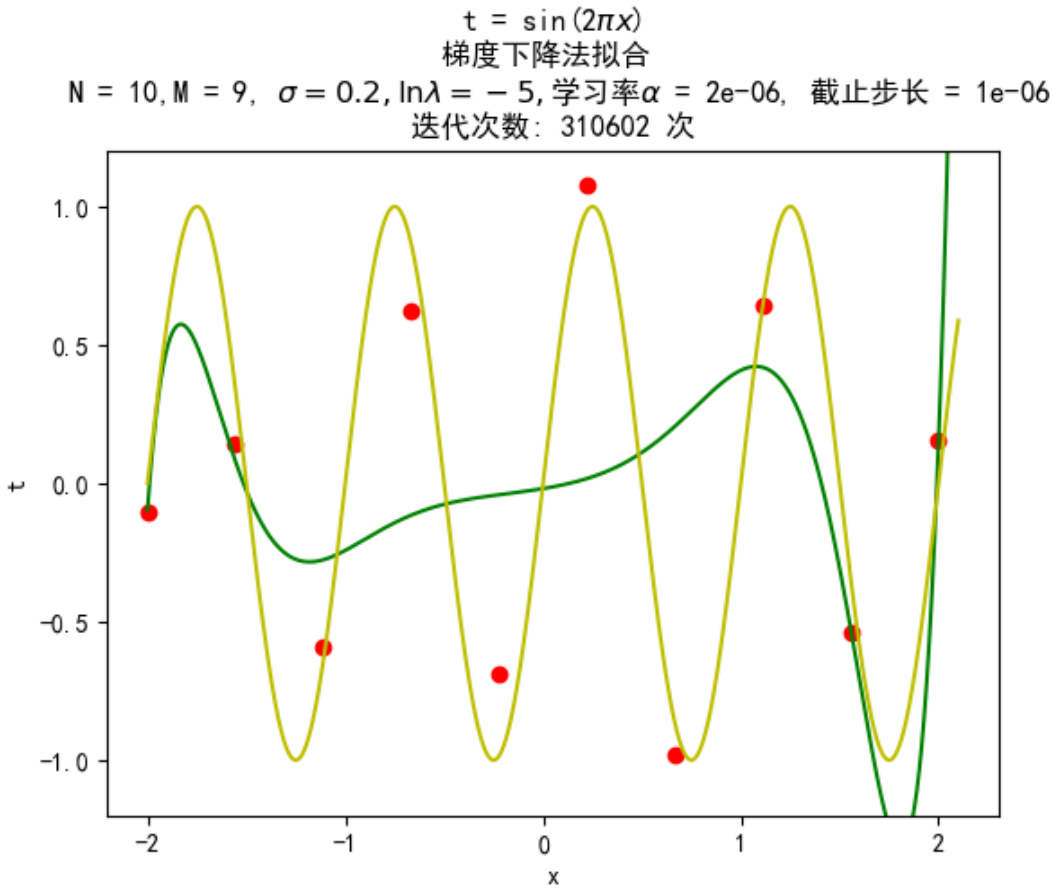


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_3] =
        [-0.11386564  0.04037396  0.08304618 -0.03454554]
```

拟合效果不太好，也不太符合正弦曲线的特征。

N = 10, M = 9



$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_9] =
        [-0.01917939  0.11826875  0.13976281  0.25582862  0.06287913  0.1387243
 -0.12620832 -0.23266613  0.02562578  0.04528182]
```

拟合效果比较好，同时曲线较好地吻合了正弦曲线。

共轭梯度法

共轭梯度法在实际求解中是对 $\mathbf{w}$ 不断进行迭代的，算法描述如下：

$\mathbf{B} = \mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I}_{(\mathrm{M}+1)\times(\mathrm{M}+1)}$
$\mathbf{w_0} = \mathbf{0}$
$\mathbf{r_0} = \mathbf{X}^T\mathbf{t} - \mathbf{B}\mathbf{w_0}$
$\mathbf{p_0} = \mathbf{r_0}$
$k = 0$

$whileTrue$:

$\alpha_k = \frac{\mathbf{r_k}^{\mathrm{T}}\mathbf{r_k}}{\mathbf{p_k}^T\mathbf{B}\mathbf{p_k}}$
$\mathbf{w_{k+1}} = \mathbf{w_k} + \alpha_k\mathbf{p_k}$
$\mathbf{r_{k+1}} = \mathbf{r_k} - \alpha_k\mathbf{B}\mathbf{p_k}$
如果$\mathbf{r_{k+1}}$足够小，停止迭代
$\beta_k = \frac{\mathbf{r_{k+1}}^{\mathrm{T}}\mathbf{r_{k+1}}}{\mathbf{r_k}^{\mathrm{T}}\mathbf{r_k}}$
$\mathbf{p_{k+1}} = \mathbf{r_{k+1}} + \beta_k\mathbf{p_k}$
$k = k + 1$

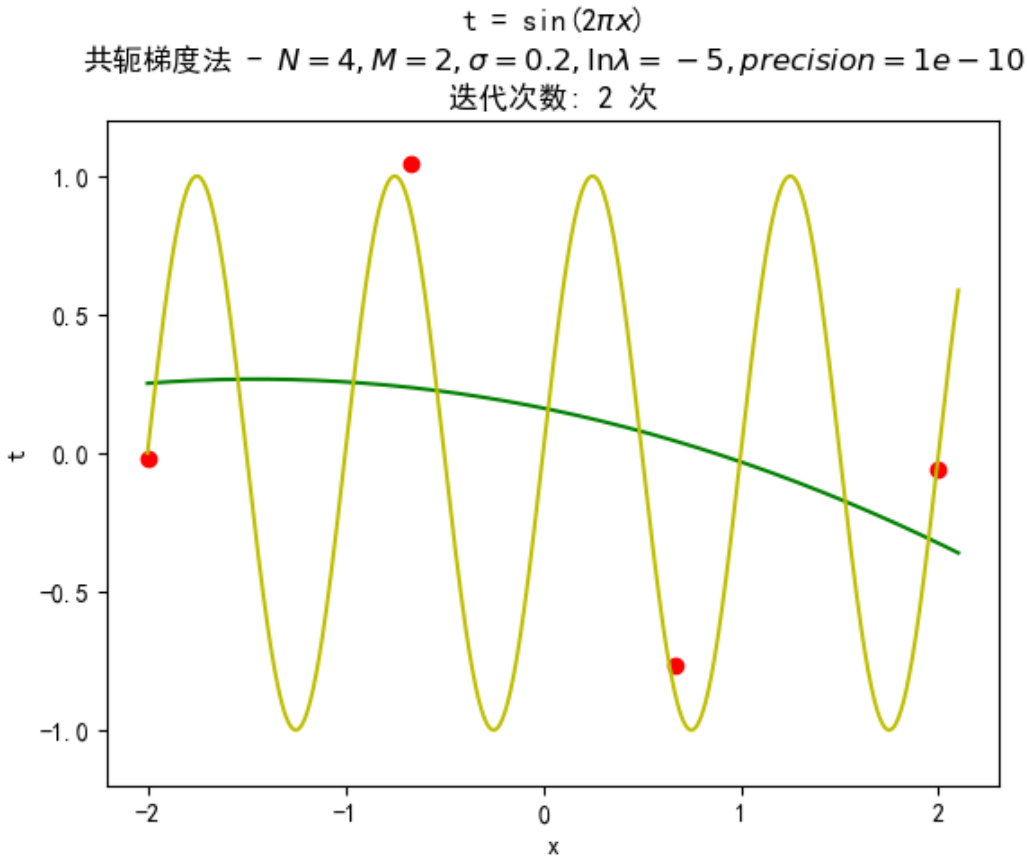
输出结果 $\mathbf{w_{k+1}}$

实际运行时发现相对于梯度下降法，迭代次数少了许多，速度明显增加。

运行命令:

```
$ make conjugate_gradient
```

N = 4, M = 2

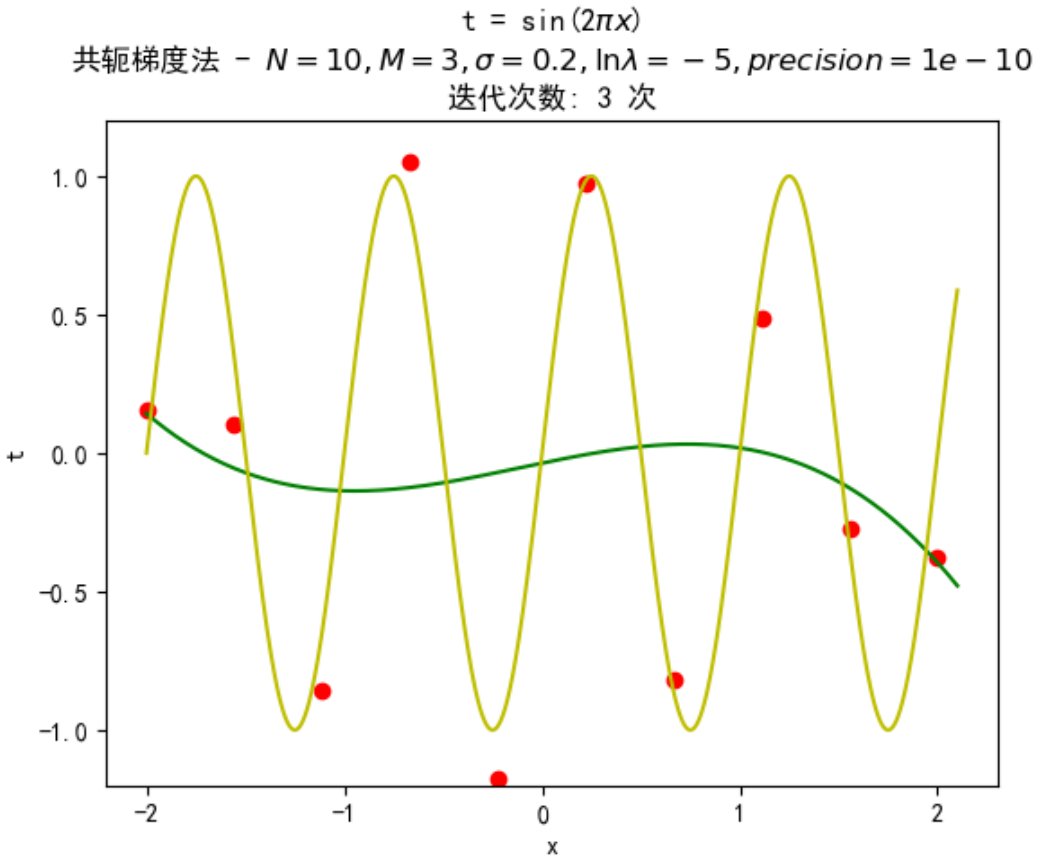


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_2] =
        [ 0.16231693 -0.14439352 -0.04983526]
```

拟合效果不太好，也不太符合正弦曲线的特征。与梯度下降法的误差相差不大。

N = 10, M = 3

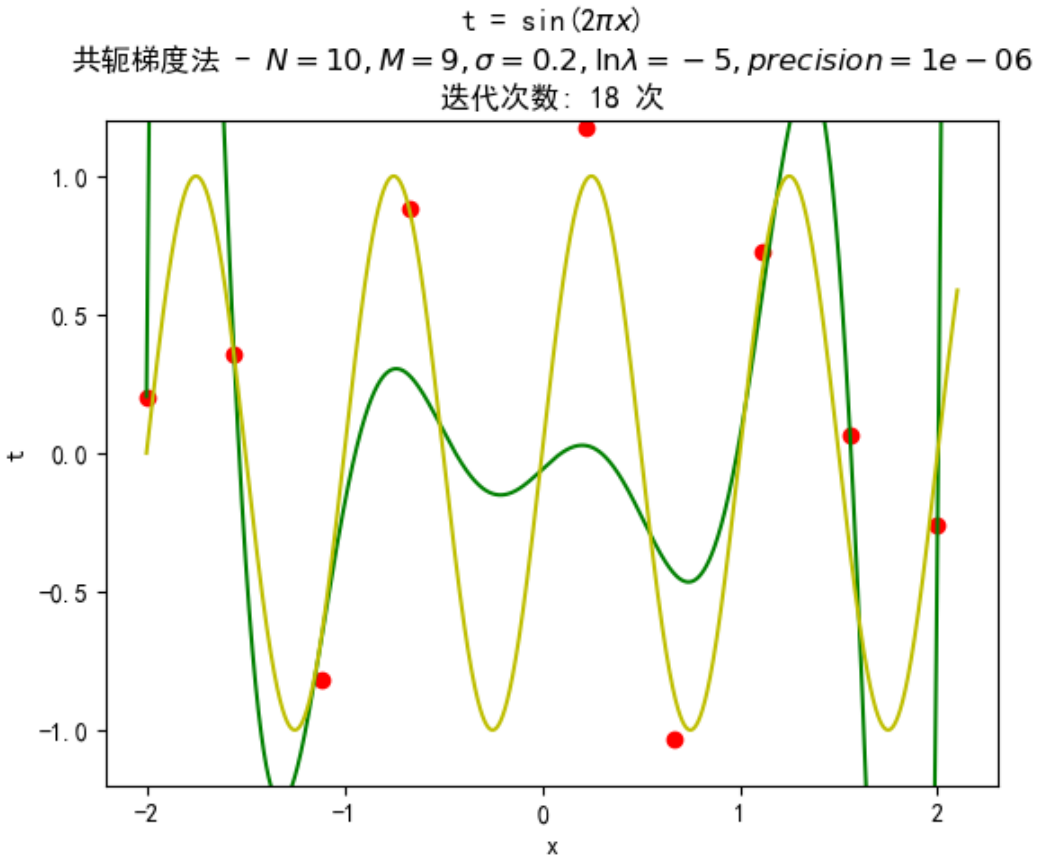


$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_3] =
        [-0.03646811  0.14819311 -0.02249006 -0.0706984 ]
```

同样拟合效果不太好，也不太符合正弦曲线的特征。与梯度下降法的误差相差不大。

N = 10, M = 9



$\mathbf{w}_{ML}$ =

```
[w_0 w_1 ... w_9] =
        [-0.05972061  0.66720683 -0.045204   -5.82065486 -0.0254253   8.64213249
  0.07935999 -3.88690431 -0.01742728  0.51947691]
```

拟合效果很好，符合正弦曲线的特征。与梯度下降法的误差相差不大。

五、结论

1. 最小二乘法需要计算矩阵的逆效率较慢。
2. 最小二乘法中多项式阶数 M 越大，拟合效果越好，但可能会出现过拟合现象，主要的解决办法有：
   - 增加数据数量
   - 采用贝叶斯方法自动调节有效参数数量
   - 加入惩罚项 ✔
3. 梯度下降法避免了矩阵求逆过程，不过梯度的选取需要反复测试。当学习率较小的时候，运行的时候速度与最小二乘法相比慢得多；当学习率较大的时候，又可能出现不收敛的情况。
4. 相比于梯度下降法，共轭梯度法效率要快得多。

六、参考文献

1. **[Bishop 2006]** Christopher M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.
2. **[Least squares | Wikipedia]**
3. **[Linear least squares | Wikipedia]**
4. **[Numpy and Scipy Documentation]**
5. **[Matrix calculus | Wikipedia]**
6. **[ctmakro 梯度下降]** Gradient Descent 梯度下降法
7. **[Gradient descent | Wikipedia]**
8. **[Gradient descent]**
9. **[MATT NEDRICH 2014]**
10. **[Conjugate gradient method | Wikipedia]**

七、附录：源代码

data_generator.py

用于生成数据，函数可以自定义，默认使用正弦函数。其他参数参见函数前面的 docstring。

```
import numpy
    from matplotlib.pyplot import rcParams


    def sin2PiX(x):
        return numpy.sin(2 * numpy.pi * x)


    def generateData(func=sin2PiX, N=5000, sigma=1.0, start=-2.0, stop=2.0):
        """Generate data with random Gaussian noise whose expectation is 0.

        Parameters:
            func -- the function used to generate data points(default sin(2pi x))
            N -- the number of data points you want to generate (default 50)
            sigma -- the standard deviation of the Gaussian noise (default 1.0)
            start -- Start of interval. (default -2.0)
            stop -- End of interval. (default 2.0)
        Returns:
            A dict {'xArray': array, 'tArray': array} in which each array is
            the data points' X-axis and Y-axis
        """
        # 用来正常显示中文标签
        rcParams['font.family'] = 'sans-serif'
        rcParams['font.sans-serif'] = ['SimHei', 'Helvetica', 'Calibri']
        # 用来正常显示负号
        rcParams['axes.unicode_minus'] = False

        # For N points, there is only have N -1 intervals
        xArray = numpy.arange(start, stop + 0.001, (stop - start)/(N - 1))
        yArray = numpy.array(list(map(func, xArray)))

        # Random Gaussian noise
        noise = numpy.random.normal(0.0, sigma, N)
```

```
        # Show the noise
        # import matplotlib.pyplot as plt
        # count, bins, ignored = plt.hist(noise, 30, density=True)
        # plt.plot(bins, 1/(sigma * numpy.sqrt(2 * numpy.pi)) *
        #     numpy.exp( - (bins - 0)**2 / (2 * sigma**2) ),
        #         linewidth=2, color='r')
        # plt.show()

        yArray += noise
        return {'xArray': xArray, 'yArray': yArray}
```

least_squares.py

最小二乘法

```
import sys
    from data_generator import generateData
    from data_generator import sin2PiX
    from numpy import *
    from numpy.linalg import *
    from matplotlib.pyplot import *


    def train(numOfTraningDataPoints, orderOfPolynomial, sigmaOfNoise):
        """A train function you can customise"""
        print(f'Least squares: N ={numOfTraningDataPoints}, M = {orderOfPolynomial}, sigma = {sigmaOfNo
ise} is plotting...')
        data = generateData(N=numOfTraningDataPoints, sigma=sigmaOfNoise)
        vectorX_T = data['xArray']
        vectorT_T = data['yArray']

        # Plot the training data points
        # Plot('x', 't', '', data={'x': vectorX_T, 'y': vectorT_T})
        plot(vectorX_T, vectorT_T, 'ro')
        xlabel('x')
        ylabel('t')
        ylim(bottom=-1.2, top=1.2)
        title(f't = sin(2$\pi x$)'
            f'最小二乘法拟合 - $N = {numOfTraningDataPoints}, '
            f'M = {orderOfPolynomial}, \sigma = {sigmaOfNoise}$')

        # Get Vandermonde matrix X, see equation (8)
        matrixX = vander(vectorX_T, orderOfPolynomial+1, True)

        # Get the transpose of w_ML, see equation (12)
        w_ML_T = matmul(
            matmul(vectorT_T, matrixX),
            inv(matmul(transpose(matrixX), matrixX))
        )

        # Print the solution for polynomial coefficients to file
        with open(f'training_results/least-squares-{numOfTraningDataPoints}-{orderOfPolynomial}.txt',
'w+') as training_results:
```

```
            training_results.write(f'[w_0 w_1 ... w_{orderOfPolynomial}] = \n\t' + str(w_ML_T) + '\n\n'
)

        # Generate shorter intervals than vectorX_T
        vectorFittingX = arange(-2.0, 2.1, 0.000001)
        matrixFittingX = vander(vectorFittingX, orderOfPolynomial+1, True)
        # Plot the fitting curve, see equation (2)
        vectorY = transpose(matmul(matrixFittingX, transpose(w_ML_T)))
        plot(vectorFittingX, vectorY, 'g')
        # Plot sin(2 * pi * x)
        vector2PiX = array(list(map(sin2PiX, vectorFittingX)))
        plot(vectorFittingX, vector2PiX, 'y')

        # Save to /images
        savefig(f'images/least-squares-{numOfTraningDataPoints}-{orderOfPolynomial}.png', bbox_inches=
'tight')
        close()


    # Run training

    # Case 1
    train(numOfTraningDataPoints=40, orderOfPolynomial=10, sigmaOfNoise=0.2)
    # Case 2
    train(numOfTraningDataPoints=40, orderOfPolynomial=20, sigmaOfNoise=0.2)
    # Case 3
    train(numOfTraningDataPoints=40, orderOfPolynomial=39, sigmaOfNoise=0.2)
    # Case 4
    train(numOfTraningDataPoints=20, orderOfPolynomial=19, sigmaOfNoise=0.2)
```

least_squares_regularization.py

带惩罚项的最小二乘法

```
import sys
    from data_generator import generateData
    from data_generator import sin2PiX
    from numpy import *
    from numpy.linalg import *
    from matplotlib.pyplot import *


    def train(numOfTraningDataPoints, orderOfPolynomial, sigmaOfNoise, lnOfLambda):
        """A train function you can customise"""
        print(f'Least squares regularization: N ={numOfTraningDataPoints}, M = {orderOfPolynomial}, sig
ma = {sigmaOfNoise}, ln(lambda) = {lnOfLambda} is plotting...')
        data = generateData(N=numOfTraningDataPoints, sigma=sigmaOfNoise)
        vectorX_T = data['xArray']
        vectorT_T = data['yArray']

        # Plot the training data points
        # Plot('x', 't', '', data={'x': vectorX_T, 'y': vectorT_T})
        plot(vectorX_T, vectorT_T, 'ro')
```

```
        xlabel('x')
        ylabel('t')
        ylim(bottom=-1.2, top=1.2)
        title(f'$t = \sin(2\pi x)$\n'
            f'带惩罚项的最小二乘法拟合 \n $N = {numOfTraningDataPoints}, '
            f'M = {orderOfPolynomial}, \sigma = {sigmaOfNoise}, \ln\lambda = {lnOfLambda}$')

        # Get Vandermonde matrix X, see equation (8)
        matrixX = vander(vectorX_T, orderOfPolynomial+1, True)

        # Get the transpose of w_ML, see equation (16)
        w_ML_T = matmul(
            matmul(vectorT_T, matrixX),
            inv(
                matmul(transpose(matrixX), matrixX) +
                exp(lnOfLambda)*identity(orderOfPolynomial + 1)
            ),
        )

        # Print the solution for polynomial coefficients to file
        with open(f'training_results/least-squares-regularization-{numOfTraningDataPoints}-{orderOfPoly
nomial}.txt', 'w+') as training_results:
            training_results.write(f'[w_0 w_1 ... w_{orderOfPolynomial}] = \n\t' + str(w_ML_T) + '\n\n'
)

        # Generate shorter intervals than vectorX_T
        vectorFittingX = arange(-2.0, 2.1, 0.000001)
        matrixFittingX = vander(vectorFittingX, orderOfPolynomial+1, True)
        # Plot the fitting curve, see equation (2)
        vectorY = transpose(matmul(matrixFittingX, transpose(w_ML_T)))
        plot(vectorFittingX, vectorY, 'g')
        # Plot sin(2 * pi * x)
        vector2PiX = array(list(map(sin2PiX, vectorFittingX)))
        plot(vectorFittingX, vector2PiX, 'y')

        # Save to /images
        savefig(f'images/least-squares-regularization-{numOfTraningDataPoints}-{orderOfPolynomial}.png'
, bbox_inches='tight')
        close()


    # Run training

    # Case 1
    train(numOfTraningDataPoints=40, orderOfPolynomial=10, sigmaOfNoise=0.2, lnOfLambda=-18)
    # Case 2
    train(numOfTraningDataPoints=40, orderOfPolynomial=20, sigmaOfNoise=0.2, lnOfLambda=-18)
    # Case 3
    train(numOfTraningDataPoints=40, orderOfPolynomial=39, sigmaOfNoise=0.2, lnOfLambda=-18)
    # Case 4
    train(numOfTraningDataPoints=20, orderOfPolynomial=19, sigmaOfNoise=0.2, lnOfLambda=-18)
```

gradient_descent.py

梯度下降法

```
import sys
    from data_generator import generateData
    from data_generator import sin2PiX
    from numpy import *
    from numpy.linalg import *
    from matplotlib.pyplot import *


    def train(numOfTraningDataPoints, orderOfPolynomial, sigmaOfNoise, lnOfLambda, learningRate, precis
ion):
        """A train function you can customise (use gradient descient)"""
        print(f'Gradient descient: N ={numOfTraningDataPoints}, '
            f'M = {orderOfPolynomial}, '
            f'sigma = {sigmaOfNoise} is plotting...')
        data = generateData(N=numOfTraningDataPoints, sigma=sigmaOfNoise)
        vectorX_T = data['xArray']
        vectorT_T = data['yArray']

        # Plot the training data points
        # Plot('x', 't', '', data={'x': vectorX_T, 'y': vectorT_T})
        plot(vectorX_T, vectorT_T, 'ro')
        xlabel('x')
        ylabel('t')
        ylim(bottom=-1.2, top=1.2)

        # Get Vandermonde matrix X, see equation (8)
        matrixX = vander(vectorX_T, orderOfPolynomial+1, True)

        # Gradient function
        def gradient(w):
            return matmul(
                transpose(matrixX),
                matmul(matrixX, w) - vectorT_T.reshape(-1, 1)
                ) + exp(lnOfLambda) * w

        # Initialize the polynomial with ones
        cur_w = ones((orderOfPolynomial + 1, 1))
        previous_step_size = 1
        iters = 0
        while previous_step_size > precision:
            learning = gradient(cur_w) * learningRate
            cur_w -= learning
            previous_step_size = linalg.norm(learning)
            print('Current learning: ', previous_step_size)
            iters += 1


        title(f't = sin(2$\pi x$)\n'
            f'梯度下降法拟合 \n N = {numOfTraningDataPoints},'
            f'M = {orderOfPolynomial}, $\sigma$ = {sigmaOfNoise}, 学习率 $\\alpha$ = {learningRate}, 截
止步长 = {precision}\n'
            f'迭代次数: {iters} 次')

        # Print the solution for polynomial coefficients to file
```

```
        with open(f'training_results/gradient-descent-{numOfTraningDataPoints}-{orderOfPolynomial}.txt'
, 'w+') as training_results:
            training_results.write(f'[w_0 w_1 ... w_{orderOfPolynomial}] = \n\t' + str(transpose(cur_
w).reshape(-1)) + '\n\n')

        # Generate shorter intervals than vectorX_T
        vectorFittingX = arange(-2.0, 2.1, 0.000001)
        matrixFittingX = vander(vectorFittingX, orderOfPolynomial+1, True)
        # Plot the fitting curve, see equation (2)
        vectorY = transpose(matmul(matrixFittingX, cur_w)).reshape(-1)
        plot(vectorFittingX, vectorY, 'g')
        # Plot sin(2 * pi * x)
        vector2PiX = array(list(map(sin2PiX, vectorFittingX)))
        plot(vectorFittingX, vector2PiX, 'y')

        # Save to /images
        savefig(f'images/gradient-descent-{numOfTraningDataPoints}-{orderOfPolynomial}.png', bbox_inche
s='tight')
        close()
        print(f'Done! iteration times: {iters}')


    # Run training

    # Case 1
    train(numOfTraningDataPoints=4, orderOfPolynomial=2, sigmaOfNoise=0.2, lnOfLambda=-5, learningRate=
0.01, precision=1e-10)

    # Case 2
    train(numOfTraningDataPoints=10, orderOfPolynomial=3, sigmaOfNoise=0.2, lnOfLambda=-5, learningRate
=0.01, precision=1e-10)

    # Case 3
    train(numOfTraningDataPoints=10, orderOfPolynomial=9, sigmaOfNoise=0.2, lnOfLambda=-5, learningRate
=0.000002, precision=1e-6)
```

conjugate_gradient.py

共轭梯度法

```
import sys
    from data_generator import generateData
    from data_generator import sin2PiX
    from numpy import *
    from numpy.linalg import *
    from matplotlib.pyplot import *


    def train(numOfTraningDataPoints, orderOfPolynomial, sigmaOfNoise, lnOfLambda, precision):
        """A train function you can customise (use conjugate gradient)"""
        print(f'Least squares: N ={numOfTraningDataPoints}, M = {orderOfPolynomial}, sigma = {sigmaOfNo
ise} is plotting...')
        data = generateData(N=numOfTraningDataPoints, sigma=sigmaOfNoise)
        vectorX_T = data['xArray']
```

```
        vectorT_T = data['yArray']

        # Plot the training data points
        # Plot('x', 't', '', data={'x': vectorX_T, 'y': vectorT_T})
        plot(vectorX_T, vectorT_T, 'ro')
        xlabel('x')
        ylabel('t')
        ylim(bottom=-1.2, top=1.2)

        # Get Vandermonde matrix X, see equation (8)
        matrixX = vander(vectorX_T, orderOfPolynomial+1, True)

        # Get matrix B, see equation (20)
        matrixB = matmul(transpose(matrixX), matrixX) + exp(lnOfLambda) * identity(orderOfPolynomial +
1)
        # Initialize variables
        w = zeros((orderOfPolynomial+1, 1))
        r = matmul(transpose(matrixX), vectorT_T.reshape(-1, 1)) - matmul(matrixB, w)
        p = r
        k = 0
        # Begin iterating
        while True:
            alpha = matmul(transpose(r), r) / matmul(matmul(transpose(p), matrixB), p)
            new_w = w + alpha * p
            new_r = r - alpha * matmul(matrixB, p)
            # Exit if new_r is small enough
            if(linalg.norm(new_r) < precision):
                w = new_w
                break
            beta = matmul(transpose(new_r), new_r) / matmul(transpose(r), r)
            new_p = new_r + beta * p

            w = new_w
            r = new_r
            p = new_p
            k = k+1

        # Print the solution for polynomial coefficients to file
        with open(f'training_results/conjugate-gradient-{numOfTraningDataPoints}-{orderOfPolynomial}.tx
t', 'w+') as training_results:
            training_results.write(f'[w_0 w_1 ... w_{orderOfPolynomial}] = \n\t' + str(transpose(w)) +
'\n\n')

        title(f't = sin(2$\pi x$)\n'
            f'共轭梯度法 - $N = {numOfTraningDataPoints}, '
            f'M = {orderOfPolynomial}, \sigma = {sigmaOfNoise}, \ln\lambda = {lnOfLambda}, precision =
{precision}$\n'
            f'迭代次数: {k} 次')
        # Generate shorter intervals than vectorX_T
        vectorFittingX = arange(-2.0, 2.1, 0.000001)
        matrixFittingX = vander(vectorFittingX, orderOfPolynomial+1, True)
        # Plot the fitting curve, see equation (2)
        vectorY = transpose(matmul(matrixFittingX, w.reshape(-1)))
        plot(vectorFittingX, vectorY, 'g')
        # Plot sin(2 * pi * x)
        vector2PiX = array(list(map(sin2PiX, vectorFittingX)))
        plot(vectorFittingX, vector2PiX, 'y')
```

```
        # Save to /images
        savefig(f'images/conjugate-gradient-{numOfTraningDataPoints}-{orderOfPolynomial}.png', bbox_inc
hes='tight')
        close()


    # Run training

    # Case 1
    train(numOfTraningDataPoints=4, orderOfPolynomial=2, sigmaOfNoise=0.2, lnOfLambda=-5, precision=1e-
10)

    # Case 2
    train(numOfTraningDataPoints=10, orderOfPolynomial=3, sigmaOfNoise=0.2, lnOfLambda=-5, precision=1e
-10)

    # Case 3
    train(numOfTraningDataPoints=10, orderOfPolynomial=9, sigmaOfNoise=0.2, lnOfLambda=-5, precision=1e
-6)
```