哈尔滨工业大学计算机科学与技术学院

2016 年秋季学期《机器学习概论》

多项式拟合曲线

姓名	班级/学号	联系方式
匡盟盟	2 班/1143220116	kuangmeng@msn.com

目 录

实验要求

实验背景

生活中，我们经常会听到或者自己也在实践着一个词——“预测”。我们人类实现这一过程，可以通过一系列不为人知甚至真的是“猜”的过程来得到猜测的结果。计算机不像我们一样有一颗可以自由转动的大脑，他们只会盲目地受人的控制地不停地计算。

为了预测事物的发展方向，我们需要先了解一些已经在给定条件下发生的事件的结果，然后形成自己的经验，之后再发生相似的情况，我们就可以通过预测来决定采取什么样的策略或者预测事态的发展。

计算机为了能够预测事件结果的走向，同样需要读取足够多的数据，然后通过归纳总结出规律、进而对以后的走向做预测。虽然计算机不像我们人一样能够很快就发现规律，但是它能不停的计算呀！(在后文梯度下降法中就有体现)。通过不停地在限定条件下的计算，可以得出一个近似的解。

正是利用计算机的这种特性，我们才能在给定部分点之后，让计算机自动地为我们生成目标曲线。

实验目标

掌握最小二乘法（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。

实验过程

- 生成数据、加入噪声；
- 用高阶多项式函数拟合曲线；
- 优化方法评价（梯度下降、共轭梯度、最小二乘求解）；
- 用实验数据解释过拟合。

待求问题重述

本次实验的待求问题大致可以分为如下几个方面：

- 输入：这个实验的数据采用程序随机生成的方式（满足“生成数据”的要求）；
- 输出：通过对四种不同的拟合方法的 Python 编程实现，得到相应的拟合曲线；
- 比较：通过以上四个方法得出的曲线对比，对不同的拟合方法做出不同的评价；
- 分析：通过以上得到的实验数据，对过拟合现象做相应的解释。

算法设计

由于本次实验是要对曲线进行拟合，说白了就是画一条线（二维空间 y= ϕ(x)）来尽可能与散点（matrix[x,y]）所符合的函数关系（假定为 y=f(x)）相匹配，于是我的数据结构便也相适应的采用了“数组”、“二维数组”等比较简介明了，又合乎实际的数据结构。

同时针对不同的拟合方法，我有不同的拟合算法：

最小二乘法

所谓最小二乘法，就是按照偏差平方和[1]最小的原则采用二项式方程来拟合曲线的方法。查阅资料发现，虽然有了最小二乘法以后，我们已经可以对数据点进行拟合，但由于最小二乘法需要计算逆矩阵，计算量很大，因此特征个数多时计算会很慢，只适用于特征个数小于 100000 的情况；当特征个数大于 100000 时就显得有些吃力，就需要下面的其他几种方法来弥补。

算法详述：

[1] 假定拟合的多项式为：$y=a+a_1x+a_2x^2+\cdots\cdots+a_nx^n$;

[2] 偏差平方和计算：
$$r^2 \equiv \sum_{i=1}^{n}[y_i - (a_0 + a_1x_i + \cdots + a_nx_i^n)]^2$$
;

[3] 分别对 a_i 求偏导数，可以得到如下的方程组：

$$0 = \sum_{i=1}^{n}[y - (a_0 + a_1x + \cdots + a_nx^n)]$$

$$0 = x\sum_{i=1}^{n}[y - (a_0 + a_1x + \cdots + a_nx^n)]$$

……

$$0 = x^n\sum_{i=1}^{n}[y - (a_0 + a_1x + \cdots + a_nx^n)]$$

[4] 化简可以得到：

$$a_0n + a_1\sum_{i=1}^{n}x_i + \cdots + a_n\sum_{i=1}^{n}x_i^n = \sum_{i=1}^{n}y_i$$

[1] 偏差平方和的计算：$\min_{\varphi}\sum_{i=1}^{m}\delta_i^2 = \sum_{i=1}^{m}(\varphi(x_i)-y_i)^2$

$$a_0\sum_{i=1}^{n}x_i+a_1\sum_{i=1}^{n}x_i^2+\cdots+a_n\sum_{i=1}^{n}x_i^{n+1}=\sum_{i=1}^{n}x_iy_i$$

……

$$a_0\sum_{i=1}^{n}x_i^n+a_1\sum_{i=1}^{n}x_i^{n+1}+\cdots+a_n\sum_{i=1}^{n}x_i^{2n}=\sum_{i=1}^{n}x_i^ny_i$$

[5] 这个形式就比较像我们学过的矩阵了，化成矩阵的形式，得到：

$$\begin{bmatrix} n & \cdots & \sum_{i=1}^{n}x_i^n \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{n}x_i^k & \cdots & \sum_{i=1}^{n}x_i^{2k} \end{bmatrix}\begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix}=\begin{bmatrix} \sum_{i=1}^{n}y_i \\ \vdots \\ \sum_{i=1}^{n}x_i^ny_i \end{bmatrix}$$

[6] 化简成范德蒙德矩阵后：

$$\begin{bmatrix} 1 & \cdots & x_1^n \\ \vdots & \ddots & \vdots \\ 1 & \cdots & x_n^n \end{bmatrix}\begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix}=\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

[7] 于是，我们需要求得的：XA=Y，$A=(X^T*X)^{-1}X^T*Y$；

[8] 通过 Python，我就得到了相应的拟合曲线。

加惩罚项最小二乘法

根据奥卡姆剃刀原理，在所有的模型中，能很好的解释已知的数据并且十分简单才是最好的模型。因此我们再在优化目标上加上惩罚项。

加入惩罚项，在以上最小二乘法的基础上，在第二步求偏差平方和时，不再像之前那样 $r^2\equiv\sum_{i=1}^{n}[y_i-(a_0+a_1x_i+\cdots+a_nx_i^n)]^2$，而是在右侧加上 $\lambda||w||_2$，得到：$r^2\equiv\sum_{i=1}^{n}[y_i-(a_0+a_1x_i+\cdots+a_nx_i^n)]^2+\lambda||w||_2$ 作为新的偏差，其中 w 为：$[a_0\ \ a_1\ldots a_n]^T$。$||w||_2$ 的计算公式为：$||w||_2=\sqrt{\sum_i w_i^2}$。

其他步骤与之前的最小二乘法相似，在此不再赘述。

梯度下降法

顾名思义，假如你在一座高山的顶端，需要向山下走，而你的周围都可以走，那么你会怎么选择眼前的一小步，使你下山的速度最快呢？梯度下降法就是来解决这个“下山问题”的有效方法，而你所选定的走的方向，就称为“梯度方向”。高中的梯度知识告诉我们 :梯度的方向是函数上升最快的方向。那么函数下降最快的方向，

也就是梯度的反方向。

与最小二乘法相似，该方法也是用来求最优，但是相比最小二乘法的“全局最优”思想，梯度下降法则使用迭代实现每一步的“局部最优”从而得出最终的最优解。适用于凸二次函数，收敛较快，在优化带有平方和形式的惩罚时采用这种方法。

梯度下降法有两种：批量梯度下降法与随机梯度下降法。

算法详述：(这里选用批量梯度下降法)

[1] 设定拟合函数（$h_\theta(x)$）：

按照教材，我们的拟合函数应当为：$h_\theta(x)=\theta^T X$=a+a_1x+a_2x^2+……+a_nx^n.

[2] 设定代价函数（J(Θ)），即为预测误差:

$$J(\Theta)=\frac{1}{2}\sum_{i=1}^{m}\left(h_\theta(x^i)-y^i\right)$$

$$C(h_\theta(x),y)=-y\log h_\theta(x)-(1-y)\log(1-h_\theta(x))$$

[3] 利用梯度下降的思想来调整Θ，使代价最小:

$$\theta_j=\theta_j-\frac{a}{n}\sum_{i=1}^{n}(h_\theta(x^i)-y^i)x_j^i$$

[4] 步长的选择：

步长的选择是一件很棘手的事情，步长如果设置过大，就会偏离超过目标函数的最小值太多，导致目标函数不能达到最优解，选择出来的参数所描述的函数自然也会和样本点偏离较大。如果步长选择较小，会得到较大的结果。但是收敛速度过慢。

最终经过反复的尝试，将步长确定为 0.000001。

对于随机梯度下降法：

此时需要把 n 个样本全部代入计算，与上一种方法相比，只是在调整Θ时有不同：

$$\theta_j=\theta_j-\lambda\left(h_\theta(x^i)-y^i\right)x_j^i$$

同时在运算时对上式进行一个循环，循环次数为数据的项数。

梯度下降法由于采用的是不停地迭代，所以会很耗时，而且即使能用多项式函数完全拟合的图像，使用此方法也会有些偏差，该方法主要用来估计。

加惩罚项梯度下降法

为了得到更好的比较好的拟合训练数据的非“过拟合”多项式函数，我在梯度下降法完成的基础上添加惩罚项，具体就是在求偏差时加上像之前在最小二乘法中所实现的$\lambda||w||_2$,得到：作为新的偏差，其中 w 为：$[a_0\ \ a_1\ldots a_n]^T$。||w||$_2$的计算公式为：$||w||_2=\sqrt{\sum_i w_i^2}$。

其他步骤与之前的梯度下降法相似，在此不再赘述。

共轭梯度法

共轭梯度法是一种介于梯度下降法和牛顿法之间的无约束优化算法，具有超线性收敛速度，而且结构简单。共轭梯度法只用到了目标函数及其梯度值，避免了二阶导数的计算，降低了计算量和存储量。在各种优化算法中，共轭梯度法是非常重要的一种。其优点是所需存储量小，具有步收敛性，稳定性高，而且不需要任何外来参数。[2]

共轭梯度法将方向限制在初始点的共轭方向空间内，而不是像梯度下降法那样随意，于是他有更好的优化效果。

算法详述：

[1] 待求函数定义为：

$$F = X^T G X + b^T X + c$$

[2] 参考 FR 共轭梯度法可以知道该方法主要建立在以下的迭代公式基础上：

$$\begin{cases} d_0 = -g_0 \\ d_{k+1} = -g_{k+1} + \beta_k d_k \\ \beta_k = \dfrac{g_{k+1}^T g_{k+1}}{g_k^T g_k} \end{cases}$$

[3] 具体步骤：

a) 选定相关系数的初值；
b) 通过迭代公式得到一系列 d
c) 再通过迭代公式 3 得到一系列 β
d) 然后通过 $X_k=X_{k-1}+\beta_k d_k$；$g_k=g_{k-1}-\beta_k G d_k$ 得到系列 X

加惩罚项共轭梯度法

同样为了得到更好的比较好的拟合训练数据的非“过拟合”多项式函数，我在共轭梯度法完成的基础上添加惩罚项，具体就是在求偏差时加上像之前在最小二乘法中所实现的$\lambda||w||_2$,得到：作为新的偏差，其中 w 为：$[a_0 \ a_1 \dots a_n]^T$。$||w||_2$ 的计算公式为：$||w||_2=\sqrt{\sum_i w_i^2}$。

其他迭代与之前的共轭梯度法相似，在此不再赘述。

实验结果

程序运行截图：

[2] 节选自百度百科：http://www.e1v.cn/91c

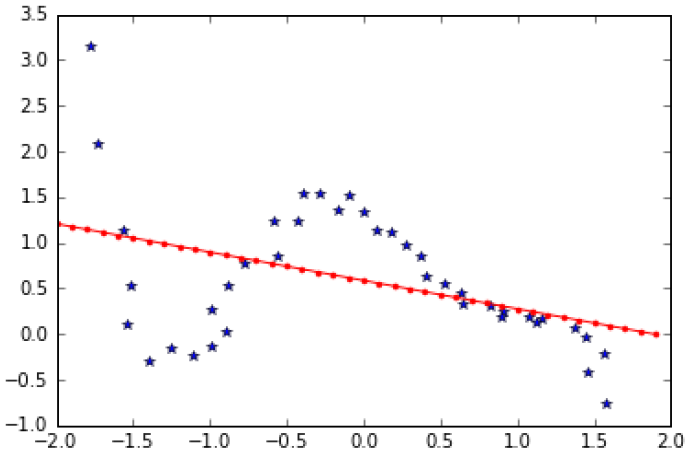
```
Spyder Editor
Created on Wed Oct 15 16:05:28 2016
@author: 匡盟盟
1：最小二乘法
2：加惩罚项最小二乘法
3：梯度下降法求正弦函数
4：梯度下降法（可加惩罚项）
5：共轭梯度法（可加惩罚项）


请输入所要使用的拟合函数：
```

程序运行结果截图

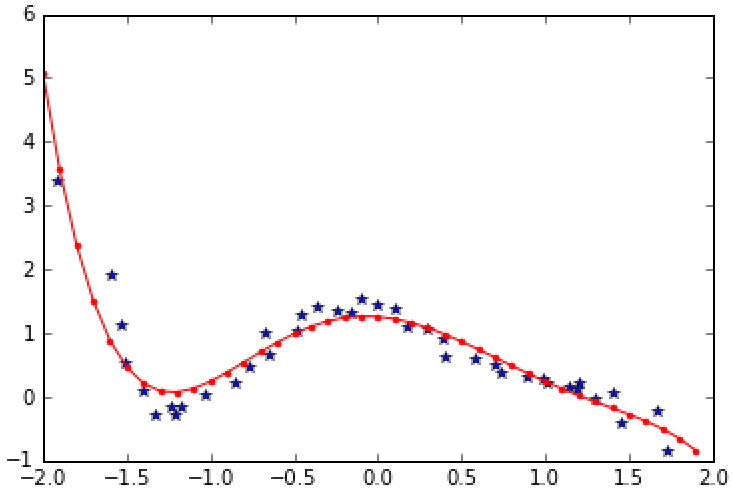
使用最小二乘法来求多项式拟合曲线

- **将拟合函数最高项数限定为 1 时（线性）：**



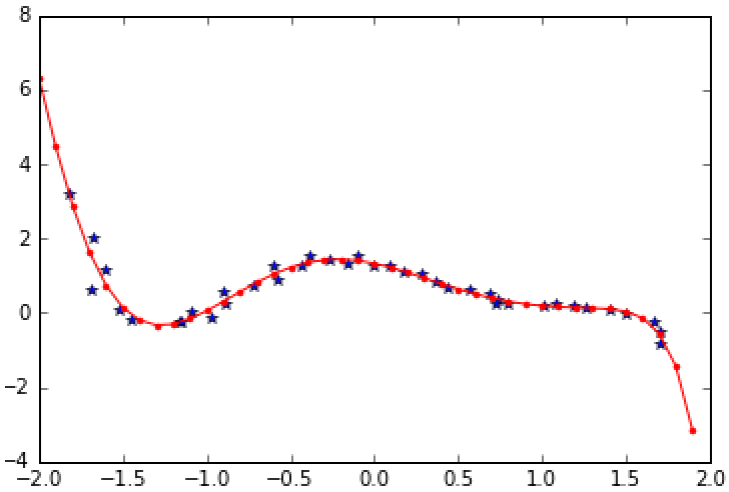
此时，系数矩阵：[0.58604315 -0.31021821]

- **将拟合函数最高项数限定为 5 时：**



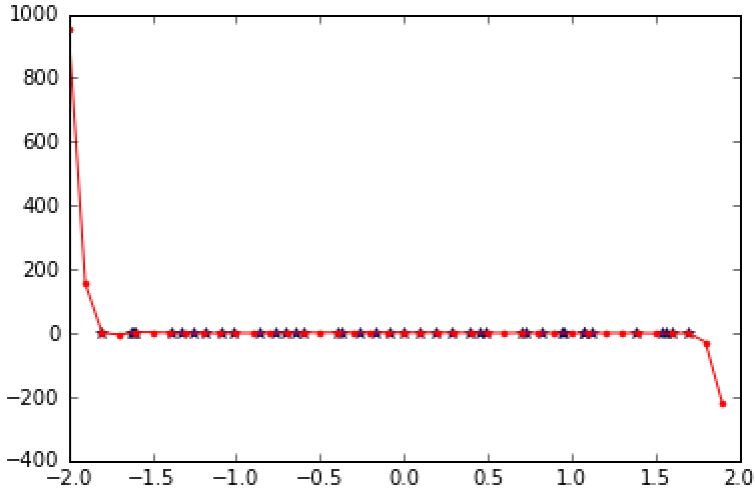
此时，系数矩阵：[1.25956531 -0.20830051 -1.4168774 0.3899689
0.40012266 -0.18082773]

- **将拟合函数最高项数限定为 10 时：**



此时，系数矩阵：[1.32788164 -0.9631046 -1.75073191 1.7263975
0.62046254 -0.84327472 -0.14691377 0.17573518 0.08424992
-0.02682822 -0.01605675]

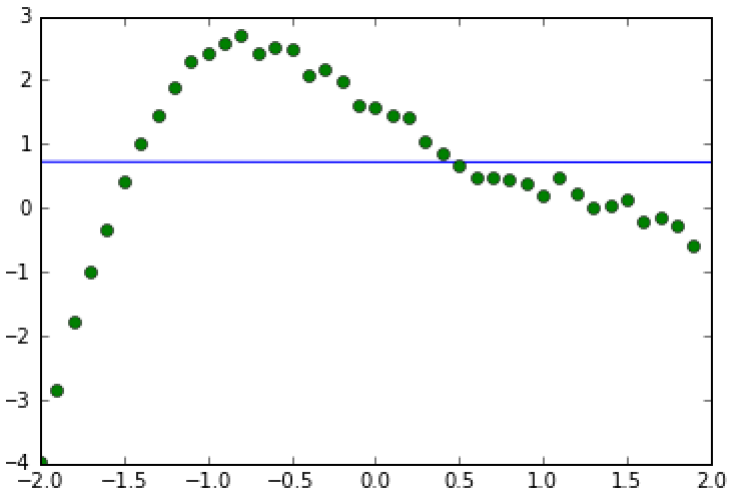
- **将拟合函数最高项数限定为 15 时：**



此时，系数矩阵：[1.28030822 0.95603071 0.79235105 -23.20766698 -14.8934258 96.9890972 33.48394794 -170.94930453 -33.73667368 153.8121295 16.97311192 -73.67673201 -4.07979913 17.79544159 0.3674746 -1.69900624]

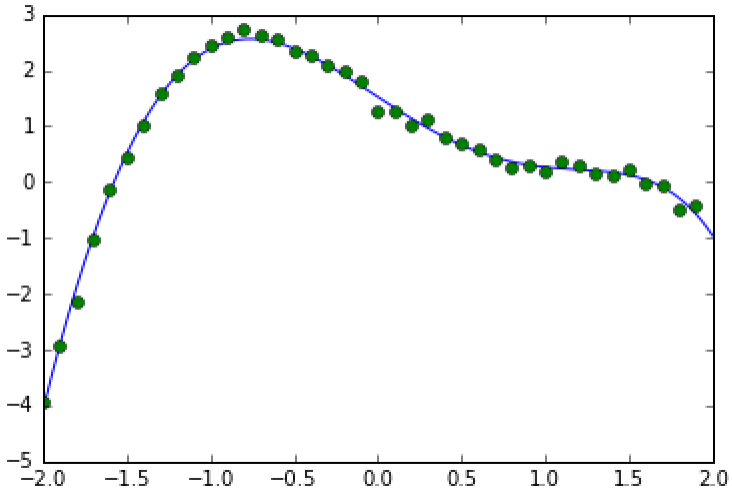
带惩罚项最小二乘法来求多项式拟合曲线（使用第三方库）

- **将拟合函数最高项数限定为 1 时（线性）：**



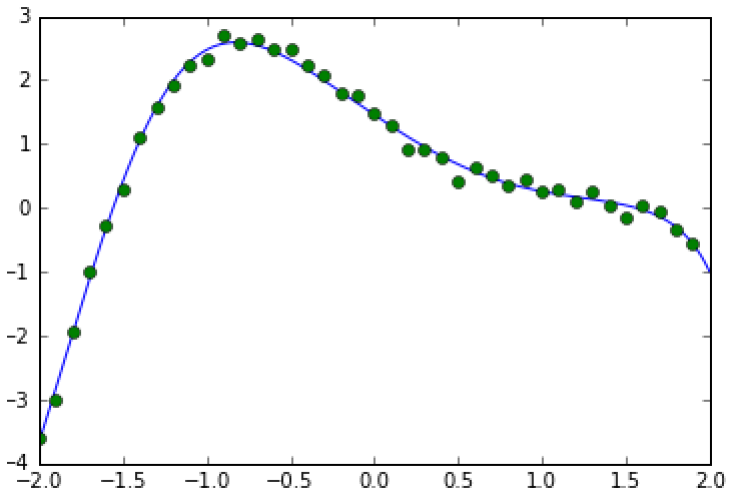
此时，系数矩阵：[-0.00184513 0.71858648]

- **将拟合函数最高项数限定为 5 时：**



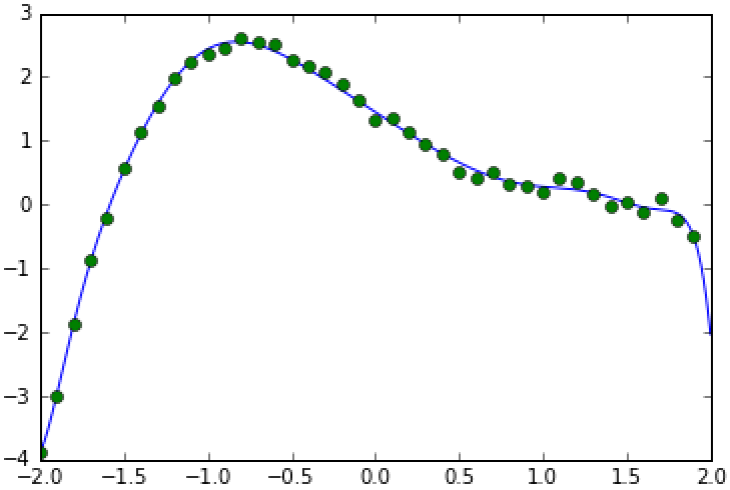
此时，系数矩阵：[-0.07531613 -0.27093801 0.98983727 0.07220885 -1.97159851 1.52367795]

- **将拟合函数最高项数限定为 10 时：**



此时，系数矩阵：[-0.0055896 0.00244632 0.05411642 -0.05870907 -0.13851679 0.22323779 -0.25474544 0.50162151 0.25534252 -1.77606386 1.45592447]

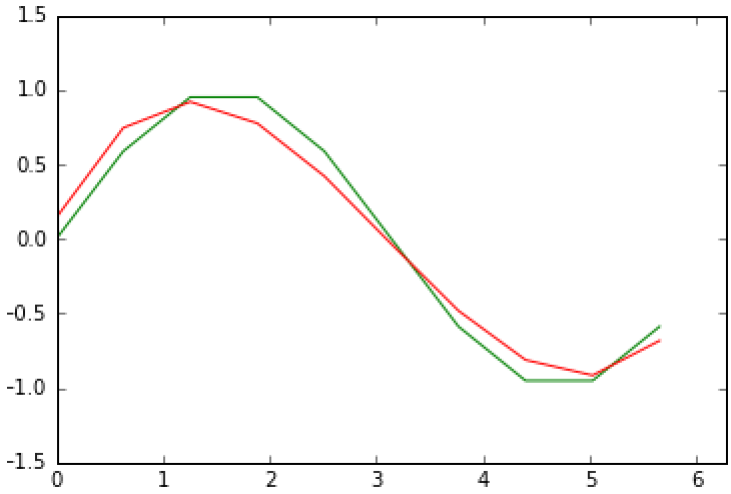
- **将拟合函数最高项数限定为 15 时：**



此时，系数矩阵：[-4.61805672e-05 1.47340093e-03 -8.07686927e-03 -2.12847435e-02 7.30283666e-02 9.50404386e-02 -1.95918284e-01 -1.24187773e-01 5.23739227e-02 -1.09210899e-01 4.32245599e-01 -1.45054522e-02 2.80855814e-01 8.06307636e-02 -1.71307088e+00 1.45015689e+00]

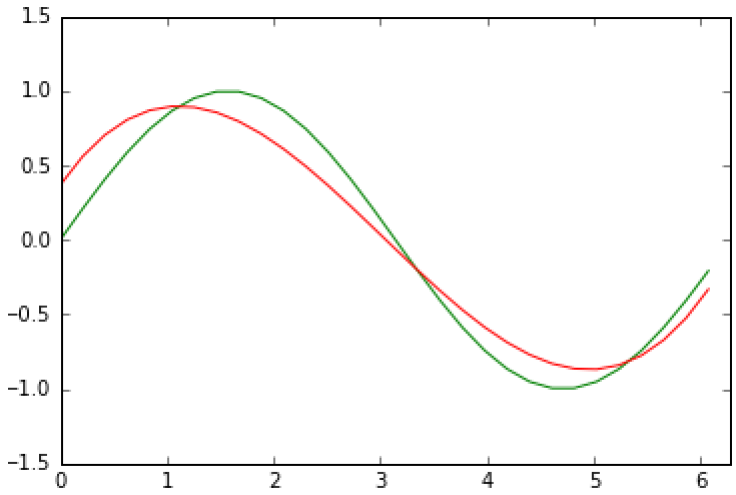
使用梯度下降法来求凸二次函数（拟合正弦函数）

- **将拟合点设置为 10 时：(红色为预测)**



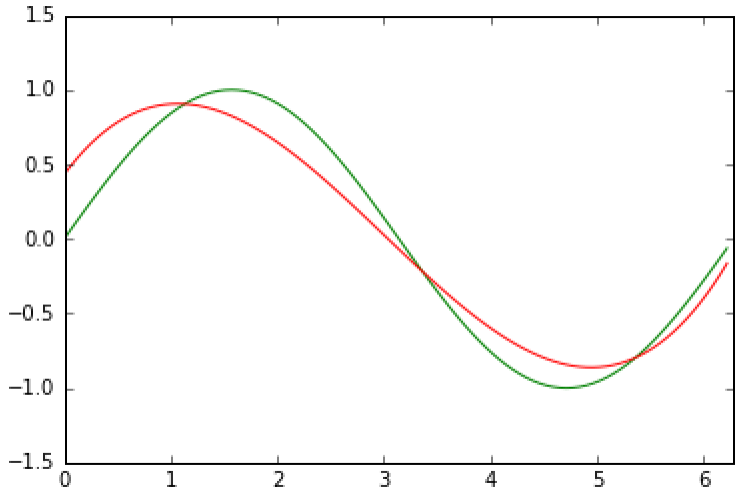
此时系数为：[0.14190868 1.3587761 -0.68088078 0.0733555]

- **将拟合顶点设置为 30 时：**



此时系数为： [0.3678567 1.02946615 -0.56703466 0.06233046]

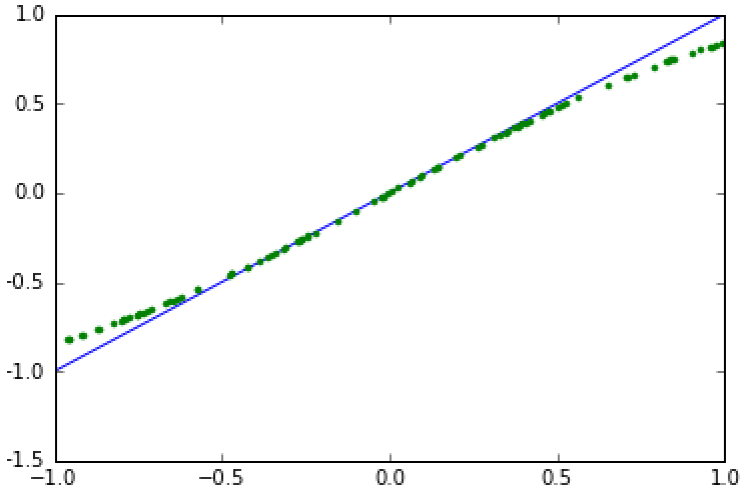
- **将拟合顶点设置为 100 个时：**



此时系数为： [0.43211893 0.95840701 -0.54619102 0.06055968]

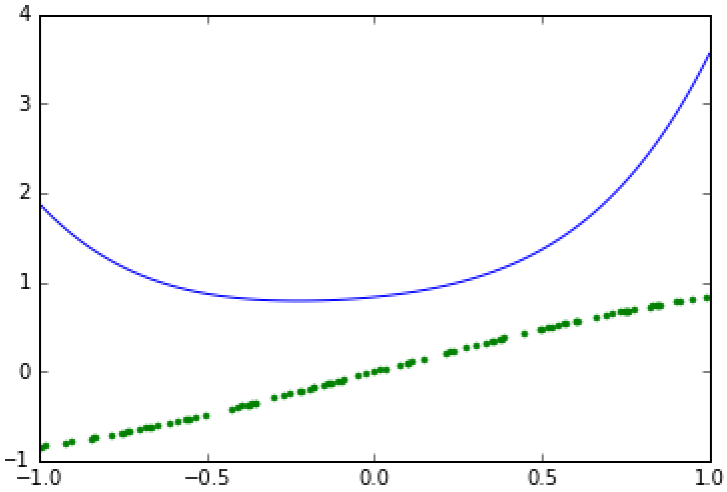
使用梯度下降法来求多项式拟合曲线

- **将函数次数设置为 1（线性）时：**



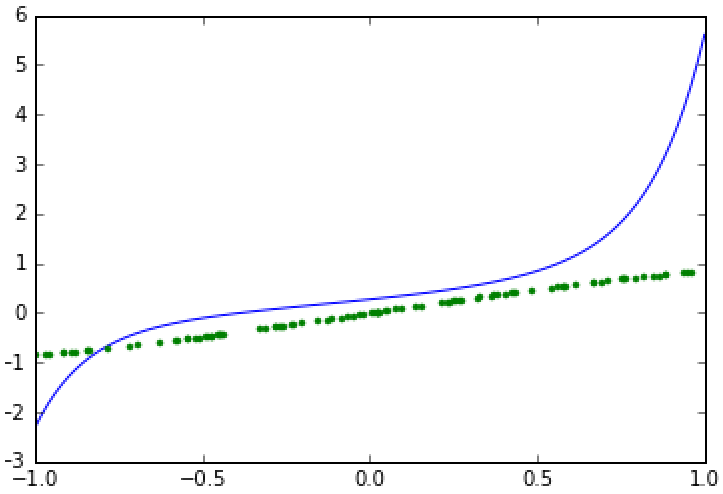
此时系数为：[1.0, 0.999990031489799]

- **将节点设置为 4 时：**



此时系数为：[1.0, 0.45778818054642445, 0.8877827548847669, 0.37988879468324727, 0.8320918150773232]

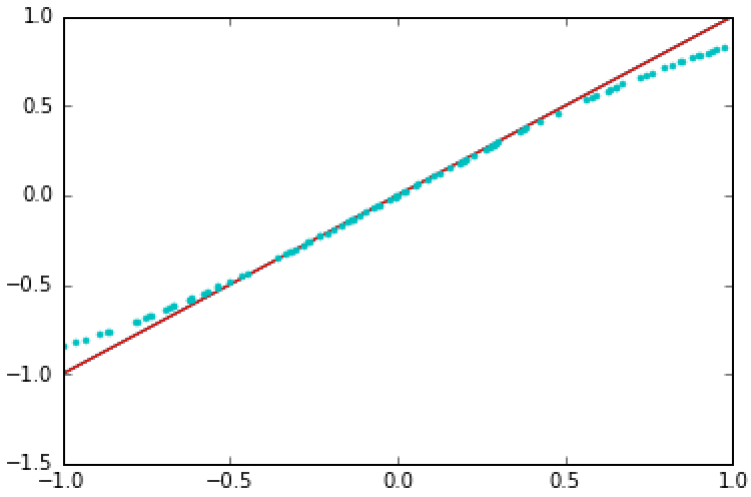
- **将函数次数设置为 9 时：**



此时系数为：[1.0, 0.4404296698479259, 0.8106586151699275, 0.3477253917483972, 0.7312066431626252, 0.30467438397317603, 0.7091106703575671, 0.28231539228763325, 0.7112836207486094, 0.27059129861651887]

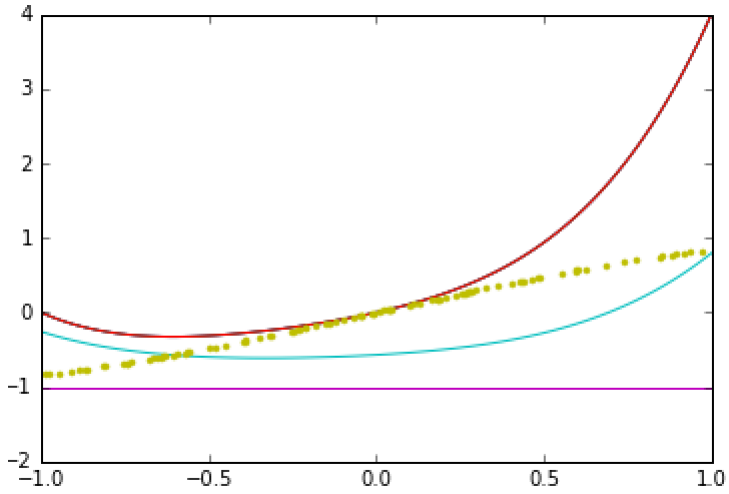
加入惩罚项的梯度下降法来求多项式拟合曲线

- **将函数次数设置为 1 时：**



此时系数为：[0.9999858578643762, 0.9999758405428913]

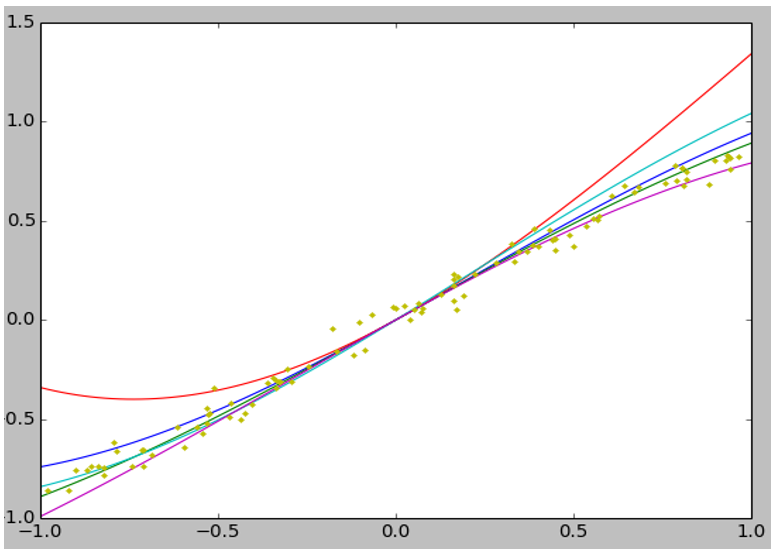
- **将节点设置为 4 时：**



此时系数为：[-4.307389416356838e-06, -8.409209497250236e-06, -1.471591613428924e-05, -5.654484212730725e-06, -6.71229949886378e-06]

最后一次生成的拟合曲线为中间的一条，其他两条同样为加了惩罚项的拟合曲线，可见相对原来的相同次数的拟合曲线来看，明显拟合性变强了很多。

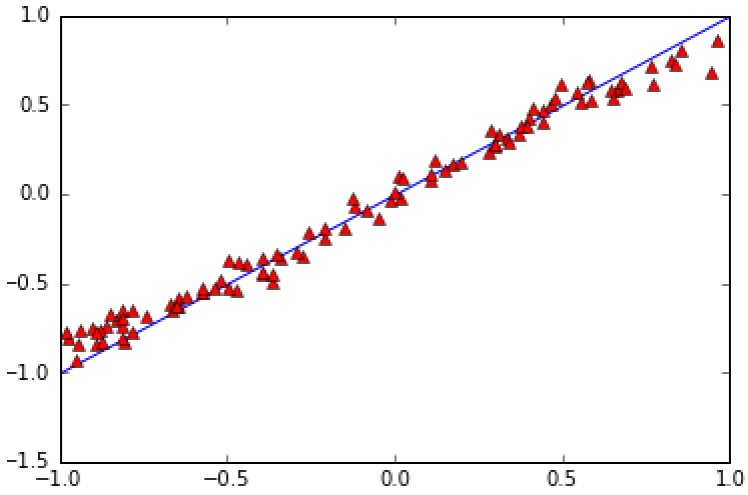
- **将函数次数设置为 9 时：**



此时系数为：[-0.00011266379542634942, -4.7672893761172236e-05, 1.7341765037565 44e-05, 4.229831313337224e-05, 5.1415666905070225e-05, 6.998377164467505e-05, 5.889869270073861e-05, 7.239587189052722e-05, 5.39136933832711e-05, 6.386527640492683e-05]

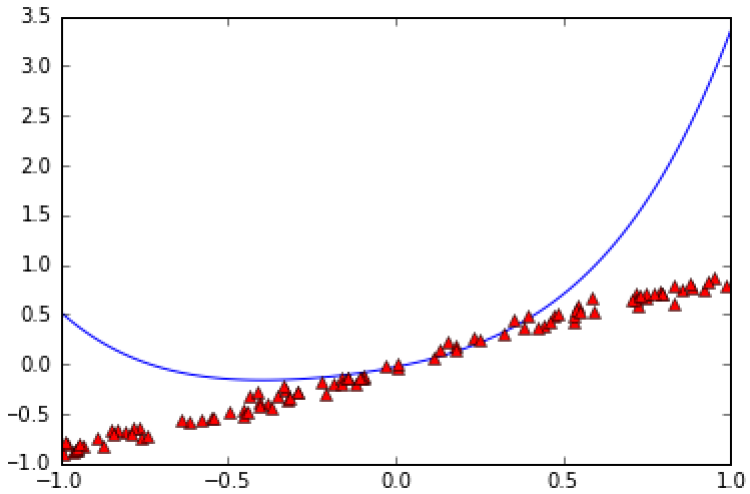
使用共轭梯度法来求多项式拟合曲线

- **将函数次数设置为 1 时：**



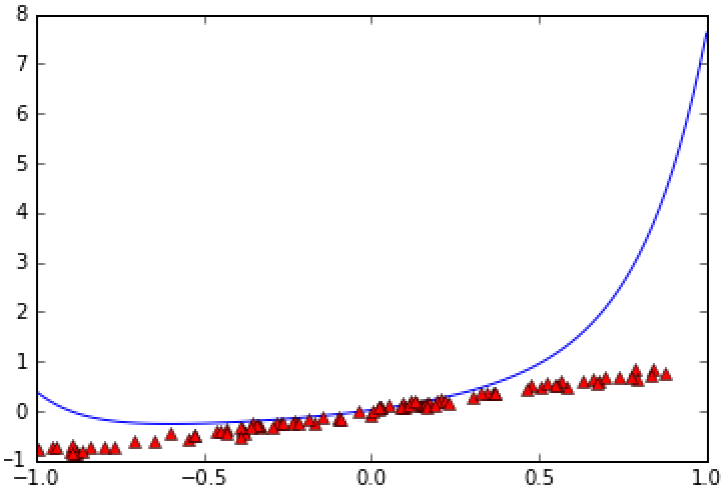
此时系数为：[1.0, 0.990126574459581]

- **将节点设置为 4 时：**



此时系数为：[1.0, 0.7511915732806681, 0.9697545463306919, 0.6620929046006641, 0.9710031651864832]

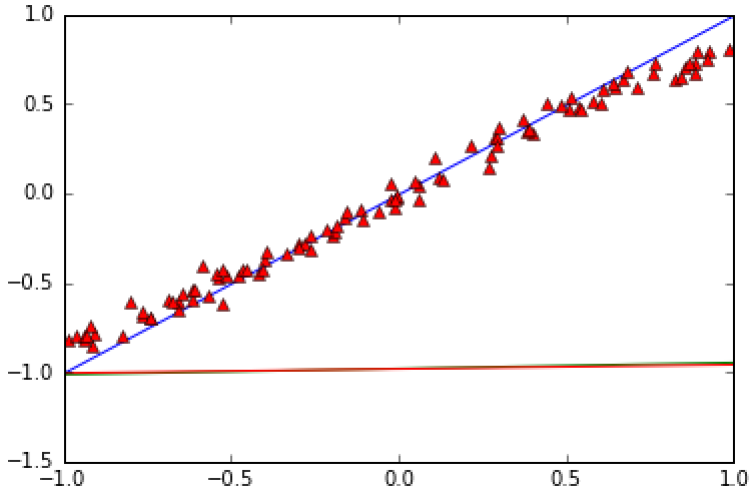
- **将函数次数设置为 9 时：**



此时系数为：[1.0, 0.904589685136276, 0.9879038867279343, 0.9010292197656132, 0.9949418003314818, 0.9003381362680228, 1.0065455422247616, 0.9029459389783547, 1.0178471065536556]

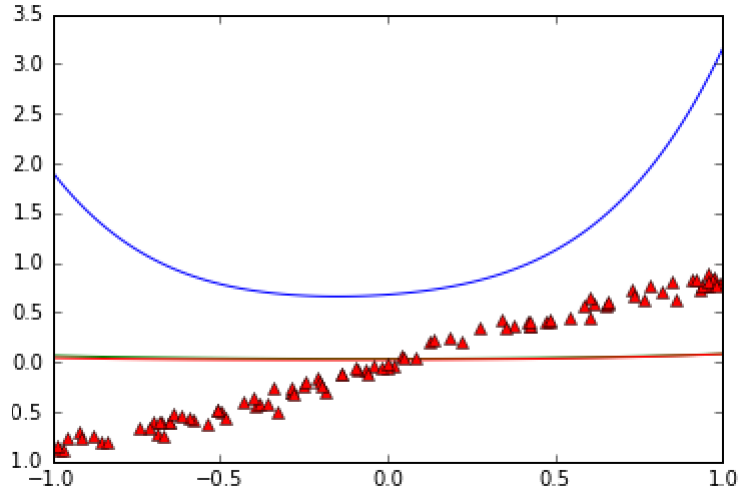
加入惩罚项的共轭梯度法来求多项式拟合曲线

- **将函数次数设置为 1 时：**



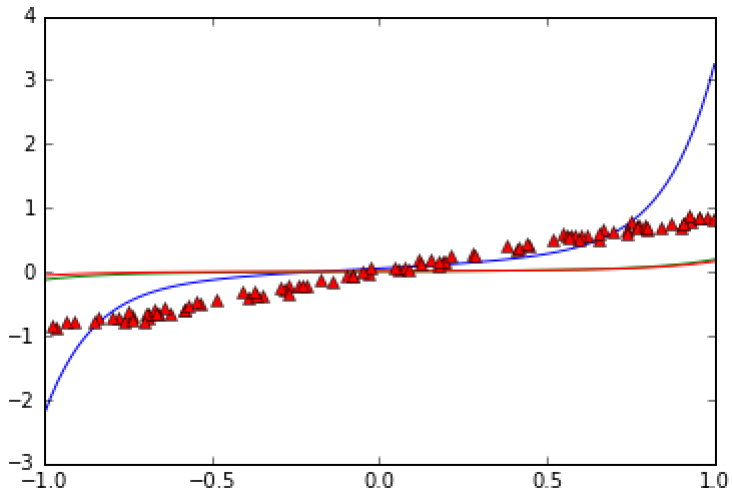
此时系数为：[0.023385865072536247, 0.0176884789919103]

- **将节点设置为 4 时：**



此时系数为：[0.015547686321115852, 0.010336213251112693, 0.023195052702504923, 0.009153617787935263, 0.023791613408833474]

- **将函数次数设置为 9 时：**



此时系数为：[0.02069031672081543, 0.013447678885083582, 0.025340826543411816, 0.010845412141006693, 0.02334825671211621, 0.009889830111635984, 0.021366013012824304, 0.009438316210881755, 0.019931886492944705, 0.009175885445150965]

程序结果分析

算法的空间复杂性

- 最小二乘法：在这个程序中，由于只使用了一维数组（大概有十几个），所以其空间复杂性为：O(n);
- 加惩罚项的最小二乘法：分析同上，空间复杂度 O(n);
- 梯度下降法 :该程序数据结构仍为数组，但由于迭代需要，所以空间复杂性为：

$O(n^2)$;

- 加惩罚项的梯度下降法：分析同上，空间复杂性为：O(n2);
- 共轭梯度法：该算法与梯度下降法在逻辑上极为相似，由于迭代的存在，其空间复杂性依然为 $O(n^2)$;
- 加惩罚项的共轭梯度法：分析同上，空间复杂性为 $O(n^2)$。

算法的时间复杂性

- 最小二乘法：在这个程序中，由于整个程序即为一个函数，所以其时间复杂性复杂性只与程序中的循环结构来定，最深的有四重循环，但是前两重深度为所要求的拟合函数的项数，数目较小，所以该算法时间复杂度为：$O(n^2)$;
- 加惩罚项的最小二乘法：分析同上，时间复杂度 $O(n^2)$;
- 梯度下降法:：由于该程序存在太多迭代，而迭代深度一般不超过两层，当然还有很多循环结构，于是我们可以断定其时间复杂度为：$O(n^3)$;
- 加惩罚项的梯度下降法：分析同上，时间复杂度 $O(n^3)$;
- 共轭梯度法：此算法与梯度下降法同样需要很多次迭代，所以时间复杂度也相同为：$O(n^3)$;
- 加惩罚项的共轭梯度法：分析同上，时间复杂度 $O(n^3)$;

编程语言与开发环境

- Python 版本：3.5.2
- Spyder IDE 版本：4.2.0
- 系统版本：windows 10

对过拟合的思考

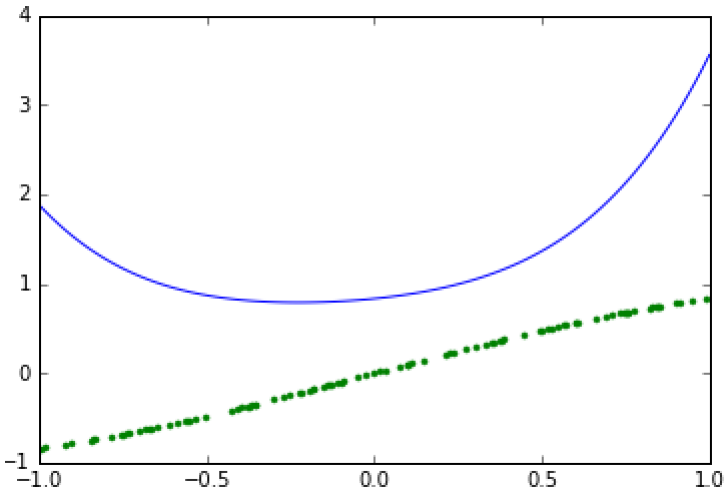
过拟合现象

为了得到一致假设而使假设变得过度复杂称为过拟合。[3]

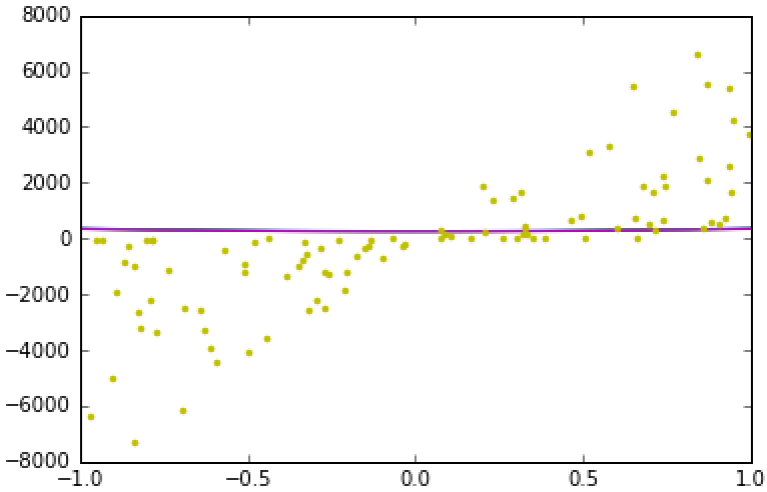
通过这一只有一行字的定义，我们可以很明确地知道在多项式拟合曲线这一实验中，过拟合的意思就是本来目标方程的次数较低，而你却用次数较高的方程去拟合，自然就会产生“过度”！经过查阅资料得知，“过拟合现象”在机器学习中时常出现，因为在我们的研究中，假设数据满足独立同分布（IID，independently and identically distributed），即当前已产生的数据可以对未来的数据进行推测与模拟，因此都是使用历史数据建立模型，就是说用已经存在的数据去训练，然后使用该模型去拟合未知的数据。但是一般独立同分布的假设往往不成立，即数据的分布可能会发生变化，而且可能当前的数据量过少，不足以对整个数据集进行估计，因此往往需要防止模型过拟合，提高模型泛化能力。

[3] 选自 360 百科：http://www.e1v.cn/98d

在我的这个实验中，同样存在过拟合过拟合现象，例如在梯度下降法中，由于我的目标曲线基本接近直线，所以当拟合函数次数相对较高时，就出现了比较明显的过拟合现象。



当然产生过拟合的原因还有很多，例如数据分布过于稀疏，如下图，虽然算法效率较高，但是仍然没法实现合理拟合。这是一种没法避免的结果，别说是计算机，就算我们人，如果拿到如下的数据集，也没法很好地将其符合的曲线完美地得出。此即该算法出现过拟合之必然性。



克服过拟合方法

经过查阅资料，我大致总结出以下几个防止过拟合的方法：

- ✧ 减小拟合函数次数
- ✧ 提前终止
- ✧ 数据集扩增
- ✧ 正则化（加惩罚项）
- ✧ 合理丢弃部分数据
- ✧ k-fold 交叉方法

接下来我想分别谈谈我的理解：

- 减小拟合函数次数

正像我在之前的拟合过程中所犯的错误一样，由于我明知测试数据阶数较低。仍然使用比较高的拟合函数去拟合，自然会出错。而解决这一问题的最简单的方法，我想就是这个减小拟合函数次数的方法了吧。

在实际的编程中，可以加入判断，将设定的拟合函数次数作为最高次数，然后从一次至高次函数来拟合，看最后得到的一系列拟合函数中那个最好，就输出哪个。

- 提前终止

主要使用在像梯度下降法等以迭代为主的算法中，由于计算机的盲目计算的属性，即使已经得到了正确的答案，其仍然在按照既定的思路不停地运算。

为了能防止过拟合，我在程序中对程序生成的前后两次系数做差，当结果小于某一个常数时，就终止迭代，虽然有些时候，并不能得到比较比较合理的结果，但是在防止过拟合时确实十分有用，例如将该差值设定过大会出现“欠拟合”，至少说明这种方法在防止过拟合时有比较好的作用。

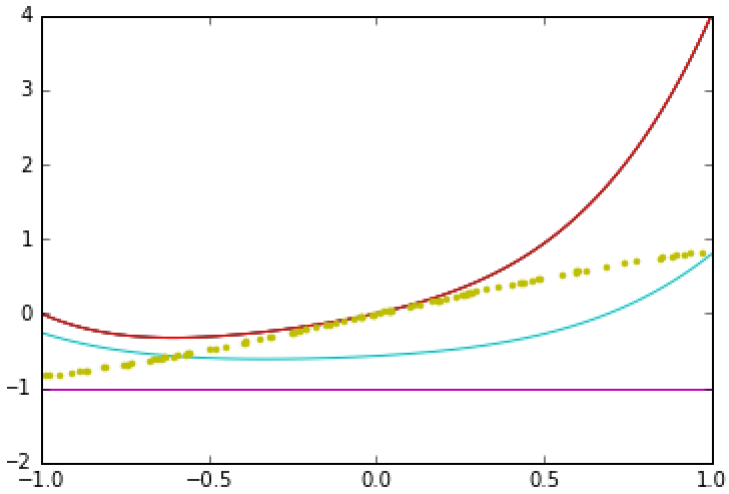
- 数据集增扩

有句话叫“磨刀不误砍柴工”，在机器学习多项式拟合时，我觉得也是十分实用的，对于此问题，“刀”指的是能够 出好的模型的原始数据，当训练数据量较小时，当然没法很好地训练出“好刀”。当然我们的这把刀完全可以通过较大的数据量来磨得锋利些。

在这种防止过拟合的方法中，又有几种比较常用的策略：

- ◆ 从数据源头采集更多数据；
- ◆ 复制原有数据并加上随机噪声；
- ◆ 重采样；
- ◆ 根据当前数据集估计数据分布参数，使用该分布产生更多数据。

还是之前的梯度下降法的例子，在将训练集扩大之后，同样的拟合函数次数，却得到完全不同的结果：（下图为增大数据集后的梯度下降法，拟合函数次数 4）



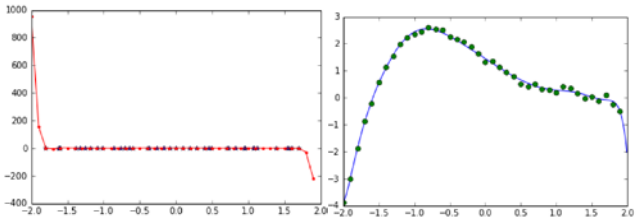
- 正则化

像本次实验所广泛使用的一种防止过拟合方法，通过加入惩罚项来实现，本次实验主要采用基于 2 范数，即在目标函数后面加上参数的 2 范数和项，即参数的平方和与参数的积项的方式来实现。例如：

$$r^2 \equiv \sum_{i=1}^{n}[y_i - (a_0 + a_1x_i + \cdots + a_nx_i^n)]^2 + \lambda||w||_2$$

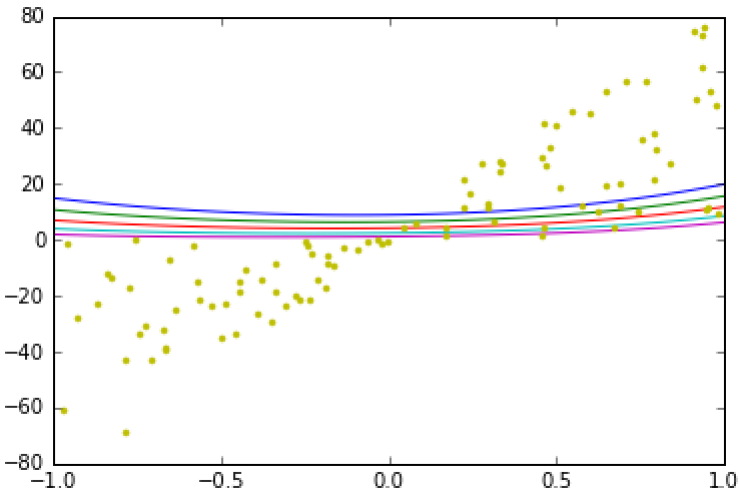
通过加上 2 范数使 w（$[a_0\ a_1 \dots a_n]^T$）有更小的参数值，意味着模型的复杂度更低，对训练数据的拟合更好（奥卡姆剃刀原理），不会过分拟合训练数据，从而减少过拟合几率，以提高模型的泛化能力。

在拟合曲线函数次数比较大（我使用的是 15 次），未加惩罚项时，明显有过拟合的情况，而加入惩罚项时，明显减弱了这种现象。



- 合理丢弃部分数据

有些情况下，由于数据的分散，造成即使添加了惩罚项、增大 训练集都无济于事，例如在上个议题“过拟合现象”中的那个例子，即使加入惩罚项，也只能得到下图：



可见要想实现完全防止过拟合，还需要其他的方法，既然以上方法都没有什么效果，那么唯一有效的方法也就是现在要讨论的，合理去除部分不需要的训练/测试数据。

- k-fold 交叉方法

把训练样例分成 k 份，然后进行 k 次交叉验证过程，每次使用不同的一份作为验证集合,其余 k-1 份合并作为训练集合。每个样例会在一次实验中被用作验证样例，在 k-1 次实验中被用作训练样例。每次实验中，使用上面讨论的交叉验证过程来决定在验证集合上取得最佳性能的迭代次 n^*，然后计算这些迭代次数的均值。最后，运行一次 BP 算法，训练所有 m 个实例并迭代 n^*次。

小结

完成了这次实验只有两个字能形容这种感觉——“艰难”！

在此之前，觉得机器学习没有什么实质性的东西，可是经过了这次实验，我算明白了，

机器学习的实践性确实特别强！这次实验只是小小的多项式拟合，本来看似比较简单的问题，但是为了保证优化性能，不仅仅需要理解最小二乘法，同时需要掌握梯度下降法与共轭梯度法。在进行了这些实验之后不仅锻炼了我 Python 编程能力，而且加深了我对这几种算法的理解，同时也对机器学习产生了浓厚的兴趣与学习的欲望！

总结这次实验，我觉得我有如下这些优点：

- 思路清晰
- 圆满完成
- 程序可拓展（可求高阶）

当然免不了有过多的缺点：

- 由于对题目的误解，之前费了好大劲在研究直线回归
- 平时知识掌握不好，关键时刻需要大量查阅资料
- 即使是将程序 地整出来，还是有很多地方自己并不是太懂
- 程序冗杂
- 拟合过程有些瑕疵，效果没有想象中的好

附录

```
# -*- coding: utf-8 -*-
"""
Spyder Editor
Created on Wed Oct 15 16:05:28 2016
@author: 匡盟盟
1：最小二乘法
2：加惩罚项最小二乘法
3：梯度下降法求正弦函数
4：梯度下降法（可加惩罚项）
5：共轭梯度法（可加惩罚项）
"""
print(__doc__)
import matplotlib.pyplot as plt
import numpy as np
import random
import scipy as sp
from scipy.optimize import leastsq
f = plt.figure()
draw = f.add_subplot(111)
#常量定义区
xlabel=[]
ylabel=[]
W=[]
A=[]
B=[]
X=[]
```

```
Y=[]
tempA=[]
tempB=[]
tail=0
step=0.001
#生成数据
def createdata(trainnum,times):
    X = np.arange(-1,1,2.0/trainnum)
    Y = [((x-1)*(x*x-1)+0.5)*np.sin(x) for x in X]
    plt.plot(X,Y,'r^')
    tail=times+1
#产生噪点
def producenoise():
    i = 0
    for x in X:
        r=float(random.randint(80,100))/100
        xlabel.append(x*r)
        ylabel.append(Y[i]*r)
        i+=1
#最小二乘法
def lsm(trainnum,times):
    createdata(trainnum,times)
    producemoise()
    for i in range(tail):
        tempA=[]
        for j in range(tail):
            temp=0.0
            for k in range(trainnum):
                d=1.0
                for m in range(0,j+i):
                    d*=xlabel[k]
                temp+=d
            tempA.append(temp)
        A.append(tempA)
    for i in range(tail):
        temp=0.0
        for k in range(trainnum):
            d=1.0
            for j in range(i):
                d=d*xlabel[k]
                temp+=ylabel[k]*d
        B.append(temp)
    #求解 AXY=B,即在文档中的最后一部分
    XY=np.linalg.solve(A,B)
```

```
        print('系数的矩阵表示为：[a0,---,a%d]'%(times))
        print(XY)
        for i in range(0,length):
            temp=0.0
            for j in range(0,tail):
                d=1.0
                for k in range(0,j):
                    d*=XY[i]
                d*=XY[j]
                temp+=d
            Y.append(temp)
        draw.plot(X,Y,color='r',linestyle='-',marker='.')
#加惩罚项的最小二乘法
def rlsm(trainnum,times,lamata=1):
    createdata(trainnum,times)
    producenoise()
    def fit(p, xlabel):
        f = numpy.poly1d(p)
        return f(xlabel)
    #残差函数
    def residuals(p, ylabel, xlabel):
        f=numpy.poly1d(p)
        ret=f(xlabel)-ylabel
        ret = np.append(ret, np.sqrt(lamata/10.0) * p)
        return ret
    xlabel = np.linspace(-2, 2, 1000)
    p= np.random.randn(tail)
    #调用第三方库
    plsq = leastsq(residuals, p, args=(ylabel, xlabel))
    print ('系数的矩阵表示: ', plsq[0])
    pl.plot(xlabel, fit(plsq[0], xlabel))
    pl.plot(xlabel, ylabel, 'go')
#梯度下降拟合正弦曲线
def gds(trainnum,times):
    createdata(trainnum,times)
    producenoise()
#因为是正弦曲线，需要重新生成 Y
    Y = [np.sin(x) for x in X]
    seita = np.array([0.,0.,0.,0.])
    store_seita = []
    store_seita.append(seita)
    k = 1
#控制循环次数
    while k < 20:
```

```
        temp_s = np.array([0.,0.,0.,0.])
        for j in range(trainnum):
            temp_h = 0
            for i in range(len(seita)):
                temp_h += seita[i]*np.power(X[j], i)
            tempA = Y[j]
            tempB= X[j]
            for i in range(len(temp_s)):
                temp_s[i] += (temp_h - tempA)*np.power(tempB, i)
#只要前后两次代价查不满足要求，就一直迭代
        while True:
            temp = seita
            cost_first = 0
            for i in range(trainnum):
                temp_h = 0
                for j in range(len(seita)):
                    temp_h += seita[j]*np.power(X[i], j)
                tempA = Y[i]
                cost_first += (temp_h - tempA)*(temp_h - tempA)/2.0
            seita-=temp_s*step/trainnum
            cost_second = 0
            for i in range(trainnum):
                temp_h = 0
                for j in range(len(seita)):
                    temp_h += seita[j]*np.power(X[i], j)
                tempA = Y[i]
                cost_second += (temp_h - tempA)*(temp_h - tempA)/2.0
            if cost_second - cost_first > 0:
                seita = temp
                step -= 0.0001
            else:
                break
        store_seita.append(seita)
        k += 1
        delta = 0
        for i in range(len(seita)):
            delta += (store_seita[0][i]-seita[i])*(store_seita[0][i]-seita[i])
        delta=np.sqrt(delta)
#用栈来保存新生成的 seita
        for i in range(10):
            store_seita[i-1] = store_seita[i]
        store_seita[9] = seita
        delta = 0
        for i in range(len(seita)):
```

```
                delta += (store_seita[0][i]-seita[i])*(store_seita[0][i]-seita[i])
            delta=np.sqrt(delta)
        ylabel = []
        for i in range(trainnum):
                temp=0
                for j in range(len(seita)):
                    temp += seita[j]*np.power(X[i], j)
                ylabel.append(temp)
        plt.plot(X, ylabel, 'r')
        plt.show()
#梯度下降法（lamata 为||w||2 的系数）
def gd(trainnum,times,lamata=1):
        createdata(trainnum,times)
        producenoise()
        select={}
#初始设定一个 w
        for i in range(tail):
            W.append(0.1);
        for l in range(lamata):
#前后代价差小于设定时，才结束
                while True:
                    cost_first=0
                        for i in range(tail):
                                cost_first+=W[i]**2
                     cost_first=l*math.sqrt(cost_first)
                     for j in range(trainnum):
                            temp=0
                            for q in range(tail):
                                temp+=W[q]*(X[j]**q)
                                if q==0:
                                    temp-=Y[j]
                    cost_first+=temp**2/(2*trainnum)
                    for i in range(tail):
                                temp=0
                            for q in range(tail):
                                temp+=W[q]**2
                            temp=math.sqrt(temp)
                        #求 w 导数
                            t=l*W[i]/temp
                            for j in range(trainnum):
                                temp=0
                                for q in range(tail):
                                    temp+=W[q]*(X[j]**q)
                                    if q==0:
```

```
                                    temp-=Y[j]
                                t+=temp*i*W[i]*X[j]**(i-1)/trainnum
                        W[i]-=step*t
                cost_second=0
                    for i in range(tail):
                        cost_second+=W[i]**2
                  cost_second=l*math.sqrt(cost_second)
                  for j in range(trainnum):
                        temp=0
                        for q in range(tail):
                            temp+=W[q]*(X[j]**q)
                            if q==0:
                                temp-=Y[j]
            cost_second+=temp**2/(2*trainnum)
            if cost_first-cost_second<10e-5:
                    break
            f = sp.poly1d(W)
            fx=sp.linspace(-1,1,100)
            er=sp.sum((f(xlabel)-ylabel)**2)
            plt.plot(fx, f(fx)+0.5)
            select[l]=[]
            select[l].append(er)
            select[l].append(list(W))
            for i in range(tail):
                W[i]=0.1
    for l in range(1):
        r=0
        if select[l][0] < select[r][0]:
            r=l
    plt.show()
#共轭梯度法
def cg(trainnum,times,lamata=1):
    createdata(trainnum,times)
    producenoise()
#初始生成 w
    for i in range(tail):
        W.append(0.1);
    for l in range(lamata):
#迭代代价差小于设定时退出
        while True:
                cost_first=0
                for i in range(tail):
                    cost_first+=l*(W[i]**2)
                for j in range(trainnum):
```

```
                    temp=0
                    for q in range(tail):
                        temp+=W[q]*(X[j]**q)
                        if q==0:
                            temp-=Y[j]
                    cost_first+=temp**2/(2*trainnum)
                for i in range(tail):
                    #求导数
                    t=2*l*W[i]
                    for j in range(trainnum):
                        temp=0
                        for q in range(tail):
                            temp+=W[q]*(X[j]**q)
                            if q==0:
                                temp-=Y[j]
                        t+=temp*i*W[i]*X[j]**(i-1)/trainnum
                        W[i]=W[i]-step*t
                cost_second=0
                for i in range(tail):
                    cost_second+=l*(W[i]**2)
                for j in range(trainnum):
                    temp=0
                    for q in range(tail):
                        temp+=W[q]*(X[j]**q)
                        if q==0:
                            temp-=Y[j]
                    cost_second+=temp**2/(2*trainnum)
                if cost_first-cost_second<10e-5:
                    break
            f = sp.poly1d(W)
            fx=sp.linspace(-1,1,100)
              e=sp.sum(f(x)-y)**2)
            plt.plot(fx, f1(fx)-1)
            select[l]=[]
            select[l].append(er)
            select[l].append(list(W))
            for i in range(tail):
                W[i]=1
        for l in range(lamata):
            r=0
            if select[l][0] < select[r][0]:
                r=l
        plt.show()
#实际测试
```

```
select = input("请输入所要使用的拟合函数:")
trainnum=input("请输入拟合节点数量：")
times=input("请输入拟合曲线次数：")
lamata=input("请输入加惩罚项时的 lamata（整数，默认为 1）：")
if(select==1):
    lsm(trainnum,times)
elif(select==2):
    rlsm(trainnum,times,lamata)
elif(select==3):
    gds(trainnum,times)
elif(select==4):
    gd(trainnum,times,lamata)
elif(select==5):
    cg(trainnum,times,lamata)
else:
    print("暂未收录此算法！\n")
```