哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称：　机器学习

课程类型：　选修

实验题目：　多项式拟合正弦函数

学号：　1170300520

姓名：　郭子阳

一、实验目的

掌握最小二乘法求解（无惩罚项的损失函数）、掌握加惩罚项（2范数）的损失函数优化、梯度下降法、共轭梯度法、理解过拟合、克服过拟合的方法（如加惩罚项、增加样本）

二、实验要求及实验环境

实验要求

1. 生成数据，加入噪声；
2. 用高阶多项式函数拟合曲线；
3. 用解析解求解两种loss的最优解（无正则项和有正则项）；
4. 优化方法求解最优解（梯度下降，共轭梯度）；
5. 用你得到的实验数据，解释过拟合。
6. 用不同数据量，不同超参数，不同的多项式阶数，比较实验效果；
7. 语言不限，可以用matlab，python。求解解析解时可以利用现成的矩阵求逆。梯度下降，共轭梯度要求自己求梯度，迭代优化自己写。不许用现成的平台，例如pytorch，tensorflow的自动微分工具。

实验环境

- Python 3.7.0
- JupyterLab 1.1.3

三、设计思想（本程序中的用到的主要算法及数据结构）

算法原理

1. 最小二乘法原理（以线性回归为例）：

   假设给定一系列散列值（数据集）记为$D=\{(x_1,y_1),(x_2,y_2),(x_3,y_3)\ldots(x_n,y_n)\}$，找到一个函数$y=ax+b$（也可记得$f(x)=ax+b$），使得$f(x)$函数尽可能拟合D。求解函数$f(x)$的方法很多种。最小二乘法寻找拟合函数$f(x)$的原理和思想关键：平方差之和最小，即使得

   $$Q=(ax_1+b-y_1)^2+(ax_2+b-y_2)^2+\ldots+(ax_n+b-y_n)^2$$

   最小，即求解

   $$Q=\sum_{i=1}^{n}(\tilde{y_i}-y_i)^2=\sum_{i=1}^{n}(ax+b-y_i)^2$$

   的最小值。

   因为$(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)$均是已知变量，于是问题转化为求解$Q=f(a,b)$的最小值，即求解$(a,b)$点，使得$f(a,b)$值极小。

   使用偏导数解$f(a,b)$极小值：

   $$\frac{\partial f(a.b)}{\partial a}=\frac{\partial\sum_{i=1}^{n}(ax_i+b-y_i)^2}{\partial a}=0$$
   $$\frac{\partial f(a.b)}{\partial b}=\frac{\partial\sum_{i=1}^{n}(ax_i+b-y_i)^2}{\partial b}=0$$

2. 梯度下降法原理：

   梯度下降是一种迭代算法。要使用梯度下降法找到一个函数的局部极小值，必须向函数上当前点对应梯度（或者是近似梯度）的**反方向**的规定步长距离点进行迭代搜索。位置更新公式如下：

$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta)$$

   其中，α为步长（学习率），$\frac{\partial J(\theta)}{\partial \theta_i}$为在$\theta_i$位置处的梯度向量，方向指向上升最快的方向

3. 共轭梯度法原理：

   共轭梯度法是属于最小化类的迭代方法。为了求解$Ax = b$这样的一次函数，可以先构造一个二次齐函数

$$f(x) = \frac{1}{2} x^T A x - b^T x$$

   这样求解Ax = b的值可以转换为求解f(x)的最小值。

   初始化时$x_{(k)}$表示第k次迭代的解向量，$d_{(k)}$表示第k次迭代的方向向量，$r_{(k)}$表示第k次迭代的残差向量。这样，在进行第k次迭代时主要分为四个步骤：

$$r_{(k)} = Ax_{(k-1)}$$

$$d_{(k)} = -r_{(k)} + \frac{r_{(k)}^T r_{(k)}}{r_{(k-1)}^T r_{(k-1)}} d_{(k-1)}$$

$$\alpha_{(k)} = -\frac{d_{(k)}^T r_{(k)}}{d_{(k)}^T A d_{(k)}}$$

$$x_{(k)} = x_{(k-1)} + \alpha_{(k)} d_{(k)}$$

算法实现

1. 生成加噪声的数据

   正弦函数，周期为2，取样步长为0.2，共取10个点，为其加上均值为0，方差为0.2点高斯噪声

```python
T = 1
n = 1
step = (T / n) * 0.2
x_raw = np.arange(0, 2*T, step, float)
y_raw = np.sin(math.pi * x_raw)
plt.plot(x_raw, y_raw, color='m', linestyle='', marker='.')
plt.show()

mu = 0
sigma = 0.2
x = x_raw + random.gauss(mu, sigma)
y = y_raw + random.gauss(mu, sigma)
x = np.transpose(np.mat(x))
y = np.transpose(np.mat(y))

plt.plot(x_raw, y_raw, color='m', linestyle='', marker='.')
plt.show()
```

2. 使用最小二乘法拟合数据

   最小二乘法使用如下方程：

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \cdots & \sum_{i=1}^{n} x_i^k \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \cdots & \sum_{i=1}^{n} x_i^{k+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} x_i^k & \sum_{i=1}^{n} x_i^{k+1} & \cdots & \sum_{i=1}^{n} x_i^{2k} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \\ \vdots \\ \sum_{i=1}^{n} x_i^k y_i \end{bmatrix}.$$

```python
def least_square(x, y, order):
    matrix_left = np.empty([order + 1, order + 1], dtype = float)
    matrix_right = np.empty([order + 1, 1], dtype = float)
    for i in range(0, order + 1):
        row = matrix_left[i]
        for j in range(i, order + 1 + i):
            sum = 0
            for xx in x:
                sum = sum + xx**j
            row[j - i] = sum
    for i in range(0, order + 1):
        sum = 0
        j = 0
        for xx in x:
            sum = sum + y[j] * xx**i
            j = j + 1
        matrix_right[i][0] = sum
    return np.linalg.solve(matrix_left, matrix_right)

def func_solve(x, a):
    res=0
    for i in range(len(a)):
        res+=a[i]*x**i
    return res

# 拟合1-20阶的方程
for i in range(20):
    ax = plt.subplot(4, 5, 1+i)
    ax.set_title('order=' + str(i+1))
    plt.xticks(())
    plt.yticks(())
    a = least_square(x, y, i+1)
    after_x = np.arange(-0.2, 2*T+0.1, 0.01)
    after_y = func_solve(after_x, a)
    plt.ylim([-1.3, 1.3])
    plt.plot(x, y, color='m', linestyle='', marker='.')
    plt.plot(after_x,after_y,color='g',linestyle='-',marker='')

plt.tight_layout()
plt.show()
```

3. 最小二乘法的解析解

   化简上一个中最小二乘法的方程，可得

$$\begin{bmatrix} 1 & x_1 & \cdots & x_1^k \\ 1 & x_2 & \cdots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^k \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

   于是，其解析解为：

$$(A^T A)\beta = A^T Y$$
$$\Leftrightarrow \beta = (A^T A)^{-1} A^T Y$$

   实现如下：

```
# 注意此处的x与y是样本的x和y
def analytical_solution_without_regularizer(x, y, order):
    matrix_left = np.zeros((len(x), order+1))
    for i in range(len(x)):
        for j in range(order+1):
            if j == 0:
                matrix_left[i][j] = 1
            else:
                matrix_left[i][j] = matrix_left[i][j-1] * x[i][0]
    m1 = matrix_left
    m2 = np.transpose(m1)
    # 注意此处为广义逆
    return np.dot(np.dot(np.linalg.pinv(np.dot(m2, m1)), m2),  y)
```

4. 使用梯度下降法拟合函数时，需要首先定义代价函数

   定义代价函数为：

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

   化简可解得梯度为：

$$J(\Theta) = \frac{1}{2m}(X\Theta - \vec{y})^T(X\Theta - \vec{y})$$

$$\nabla J(\Theta) = \frac{1}{m}X^T(X\Theta - \vec{y})$$

```python
def gradient_function(theta, X, y):
    temp = np.dot(X, theta) - y
    return (1.0/m) * np.dot(np.transpose(X), temp)

def gradient_decent(theta, alpha, X, y):
    # theta为初始位置（列向量）
    num = 0
    gradient = gradient_function(theta, X, y)
    while not np.all((np.absolute(gradient) <= 1e-5) | num>60000):
        num += 1
        theta = theta - alpha * gradient
        gradient = gradient_function(theta, X, y)
    return theta

order = 6
theta = np.ones((order + 1, 1))
alpha = 0.01
X = np.zeros((m, order + 1))
for i in range(m):
    for j in range(order + 1):
        if j == 0:
            X[i][j] = 1
        else:
            X[i][j] = X[i][j-1] * x[i]
res = gradient_decent(theta, alpha, X, y)
```

5. 共轭梯度法可用于求解函数，在拟合函数问题中，需要求解的函数即为：

$$\begin{bmatrix} n & \sum_{i=1}^n x_i & \cdots & \sum_{i=1}^n x_i^k \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 & \cdots & \sum_{i=1}^n x_i^{k+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^n x_i^k & \sum_{i=1}^n x_i^{k+1} & \cdots & \sum_{i=1}^n x_i^{2k} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n y_i \\ \sum_{i=1}^n x_i y_i \\ \vdots \\ \sum_{i=1}^n x_i^k y_i \end{bmatrix}.$$

该方法伪代码如下：

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

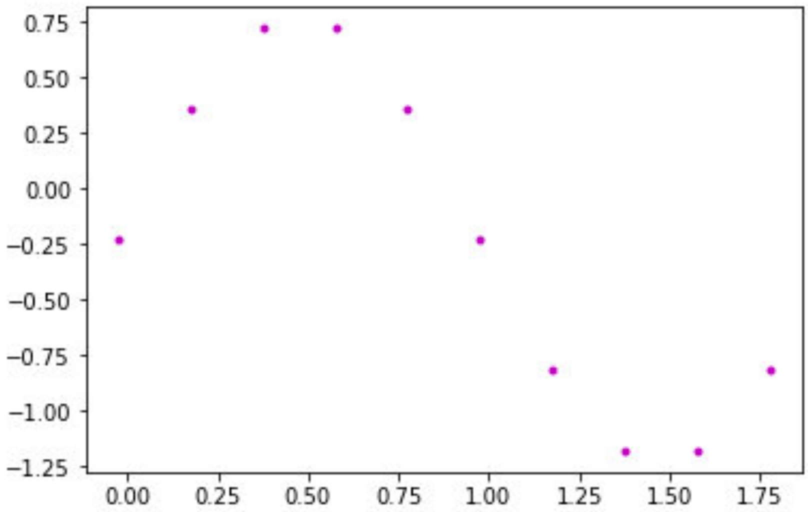
实现如下：

```
def conjugate_gradient(A, b, order):
    res = np.ones((order+1, 1))
    r = b - np.dot(A, res)
    p = r
    k = 0
    while True:
        alpha = np.dot(np.transpose(r), r) / np.dot(np.dot(np.transpose(p), A), p)
        res = res + alpha * p
        r1 = r - alpha * np.array(np.dot(A, p))
        if np.all(np.absolute(r1) < 1e-5):
            return res
        beta = np.array(np.dot(np.transpose(r1), r1) / np.dot(np.transpose(r), r))[0][0]
        p = r1 + beta * p
        k = k + 1
        r = r1

order = 9
matrix_left = np.empty([order + 1, order + 1], dtype = float)
matrix_right = np.empty([order + 1, 1], dtype = float)
for i in range(0, order + 1):
    row = matrix_left[i]
    for j in range(i, order + 1 + i):
        sum = 0
        for xx in x:
            sum = sum + xx**j
```

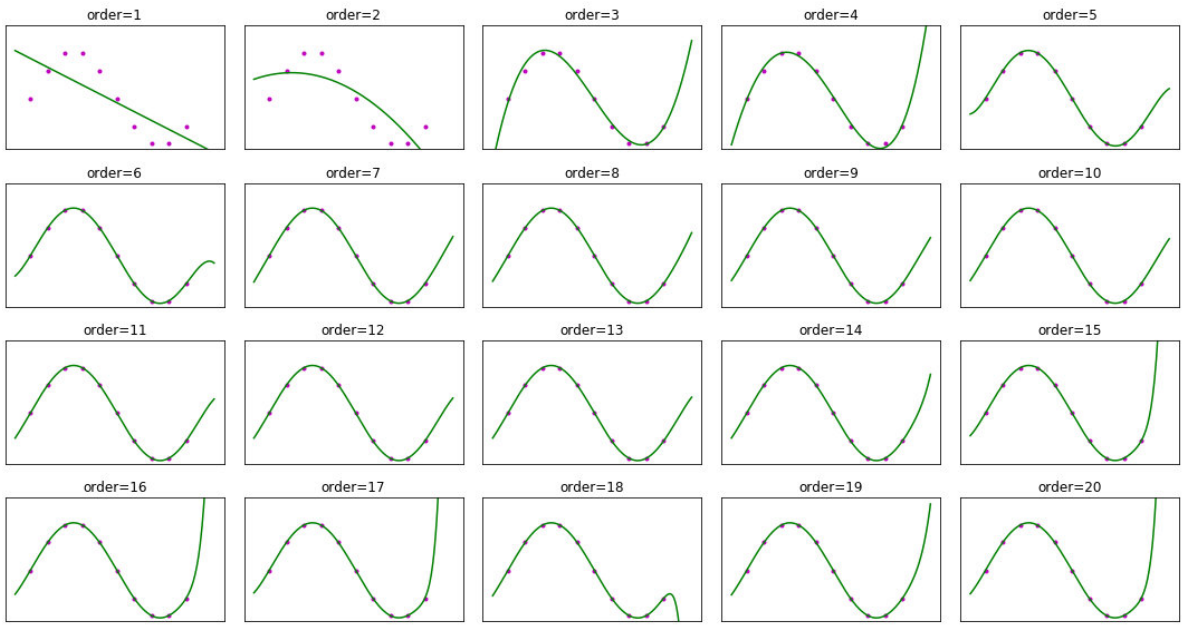
```
        row[j - i] = sum

for i in range(0, order + 1):
    sum = 0
    j = 0
    for xx in x:
        sum = sum + y[j] * xx**i
        j = j + 1
    matrix_right[i][0] = sum

res = conjugate_gradient(matrix_left, matrix_right, order)
```

四、实验结果与分析

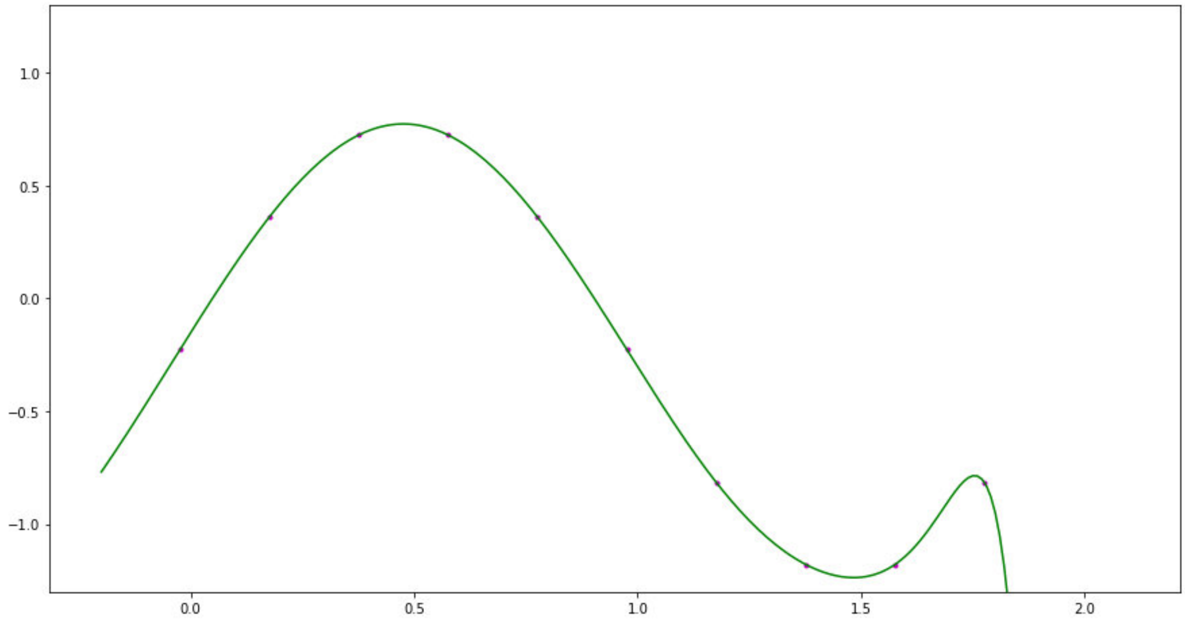
1. 生成加入高斯噪声的散点：



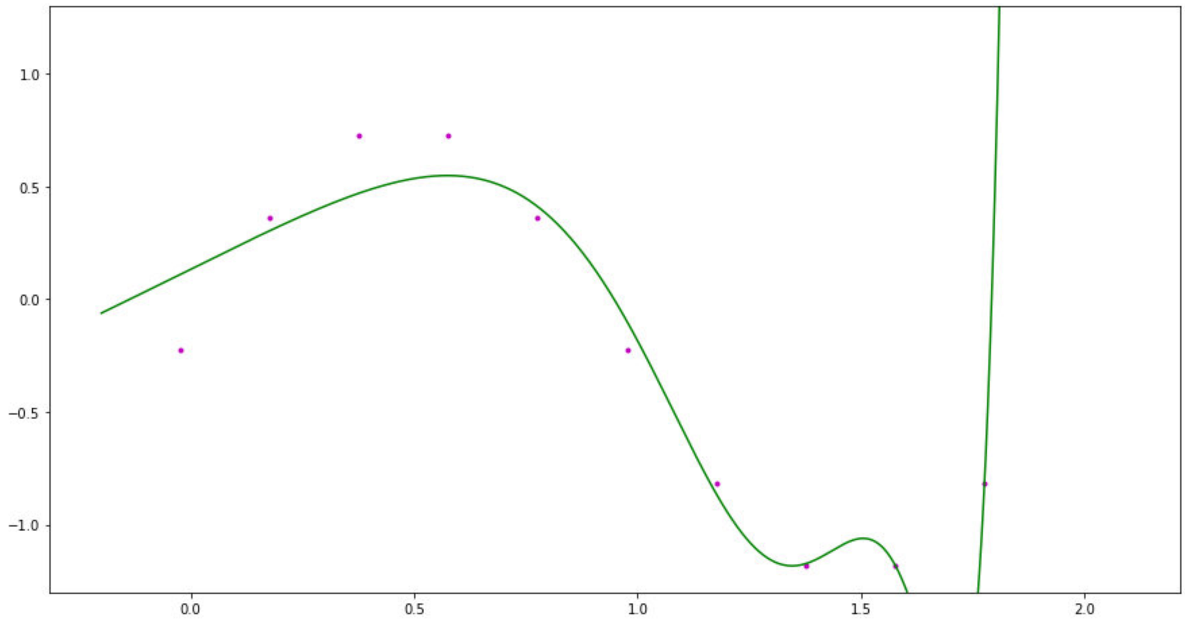
2. 使用最小二乘法拟合1-20阶多项式：



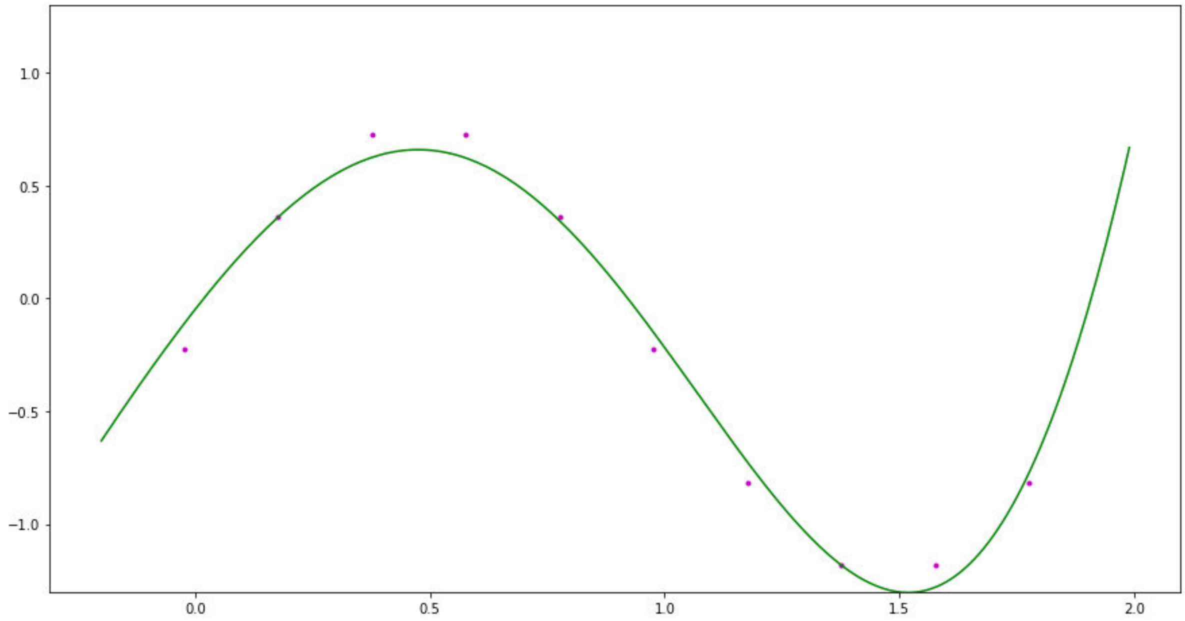
3. 使用无正则项的解析解拟合16阶多项式：



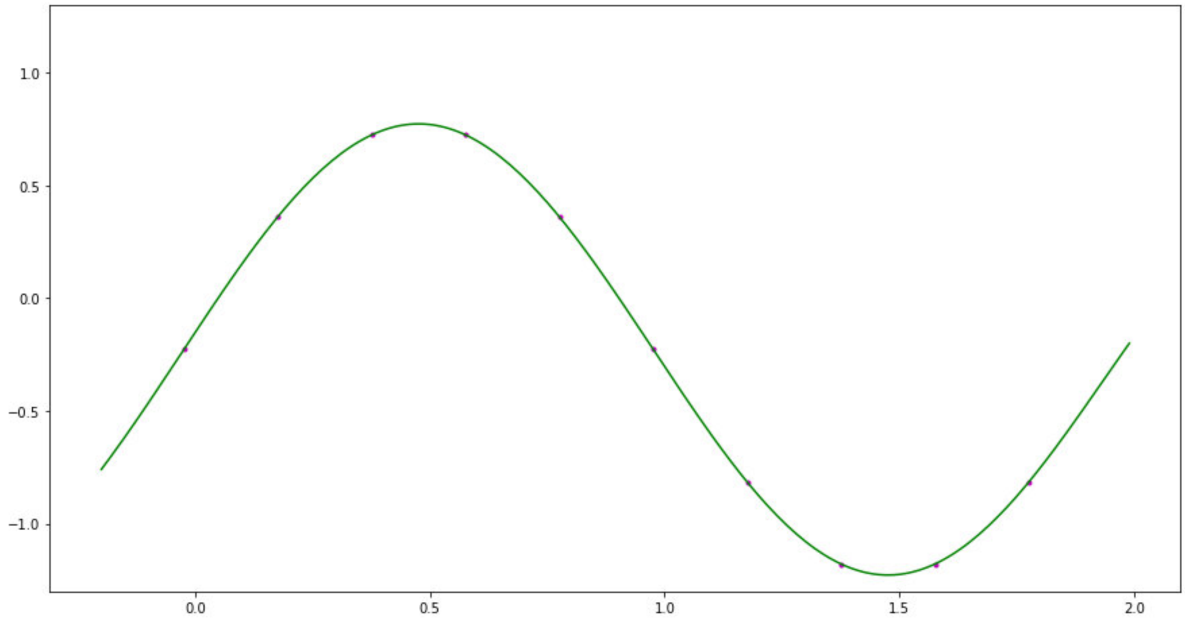
4. 使用有正则项的解析解拟合16阶多项式（惩罚项过大出现退化）：



5. 使用梯度下降拟合6阶多项式：



6. 共轭梯度法拟合9阶多项式：



五、结论

1. 使用多项式拟合函数时，阶数越高函数的能力越强。
2. 在散点个数低于阶数的情况下，可能会出现过拟合的情况，这时需要通过增加数据点或者使用惩罚项来防止过拟合。
3. 惩罚项过大的情况下可能出现退化的现象。
4. 梯度下降法的步长（学习率）需要不停地调整才能获得较好的效果。
5. 梯度下降法可能会陷入局部最低点，可以通过设置多个起始点来解决。
6. 多项式拟合中梯度下降法和共轭梯度法使用的是均方误差代价函数，其本质依旧是解决最小二乘问题

六、参考文献

[1]. 深入浅出--梯度下降法及其实现, https://www.jianshu.com/p/c7e642877b0e

[2]. Conjugate gradient method, https://en.wikipedia.org/wiki/Conjugate_gradient_method