最优化大作业实验报告

20302021 陈泽轩

1　问题描述

我们要用**邻近点梯度法、交替方向乘子法和次梯度法**来解决一个10节点分布式系统的一范数正则化最小二乘问题：

$$\min_x \quad \frac{1}{2}\|A_1x-b_1\|_2^2+\cdots+\frac{1}{2}\|A_{10}x-b_{10}\|_2^2+\lambda\|x\|_1$$

其中，

- A_i是一个5×200维的测量矩阵，元素服从均值为0方差为1的高斯分布，可以随机设定。
- b_i是一个5维测量向量。通过$b_i=A_ix+e_i$得到，这里的x是真值，也就是说b_i是原先就有的值。
- e_i为5维的测量噪声，服从均值为0方差为0.1的高斯分布，可以随机生成。
- x是我们要求解的变量。x的**真值**是200维稀疏向量且稀疏度为5，即只有5个非零元素，非零元素服从均值为0方差为1的高斯分布，x的真值可以随机生成。我们的目标是让该优化问题的最优解尽可能接近生成的真值。
- λ是可调参数，控制正则化强度。

对于每种算法，我们需要给出每步计算结果与真值的距离以及每步计算结果与最优解的距离。此外，还要讨论正则化参数λ对计算结果的影响。

大致的步骤流程如下：

- 先生成已知值：A_i、真值x、b_i；
- 然后设计**邻近点梯度法、交替方向乘子法和次梯度法**三个优化算法；
- 分别进行迭代求解，记录过程解，得到最优解。计算每步计算结果与真值的距离以及每步计算结果与最优解的距离，并绘制图形；
- 分析三个算法的结果；
- 设计算法，分析正则化参数λ对计算结果的影响。

2 算法设计

2.1 生成数据

我们先随机生成已知值，包括A_i、真值x、b_i。代码如下：

```
np.random.seed(0)  # 保证每次产生相同的随机生成变量
num = 10
d1 = 5
d2 = 200
lamda = 0.01  # 可以调整

# 随机生成矩阵A、x的真值、和向量b
A = np.array([np.random.normal(0, 1, (d1, d2)) for _ in range(num)])
x_true = np.zeros(d2)
# 随机选择5个位置非0，其他位置为0
nonzero_indices = np.random.choice(d2, 5, replace=False)
x_true[nonzero_indices] = np.random.normal(0, 1, d1)
b = np.array([A[i].dot(x_true) + np.random.normal(0, 0.1, d1) for i in
range(num)])
```

2.2 邻近点梯度法

邻近点梯度法用于有结构不可微的目标函数，优化问题描述为：

$$\min f_0(x) = s(x) + r(x)$$

其中$s(x)$是一个光滑易求导函数，$r(x)$是一个非光滑易求邻近点梯度函数。在我们的问题中，

$$s(x) = \frac{1}{2}\sum_{i=1}^{10}||A_i x - b_i||^2，\ r(x) = \lambda||x||_1$$

迭代过程分为两步：

$$x^{k+\frac{1}{2}} = x^k - \alpha^k \nabla s(x^k)$$
$$x^{k+1} = \underset{x}{argmin}\ r(x) + \frac{1}{2\alpha}||x - x^{k+\frac{1}{2}}||^2$$

第一步是对可微部分求梯度，第二步是对不可微部分求邻近点投影。

第二步的argmin实际上也是一个优化问题，通过求次梯度=0代入可以得到：

$$x^{k+1} = x^{k+\frac{1}{2}} - \alpha g_r(x^k)$$

其中，$g_r(x^k)$是r(x)在x^k处的次梯度，根据$g_r(x^k)$分类讨论就可以求出x^{k+1}，对于一范数正则化，这就是**软门限算法**，软门限表达式如下：

$$\text{soft\_threshold}(x, s) = \begin{cases} x - s, & \text{if } x > s \\ 0, & \text{if } |x| \le s \\ x + s, & \text{if } x < -s \end{cases} = sign(x) * \max(abs(x) - s, 0)$$

我们可以发现，两步迭代结合起来就是：

$$x^{k+1} = x^k - \alpha^k \nabla s(x^k) - \alpha g_r(x^k)$$

这和次梯度法的形式是一样的。

对于我们的问题，

$$x^{k+\frac{1}{2}} = x^k - \alpha^k \sum_{i=1}^{10} A_i^T (A_i x - b_i)$$

$$x^{k+1} = \text{soft\_threshold}(x^{k+\frac{1}{2}}, \alpha\lambda)$$

也就是说，先用梯度下降算$x^{k+\frac{1}{2}}$，然后用软门限算x^{k+1}。我们使用**固定步长**。

根据迭代过程就可以设计出邻近点梯度的算法了，记录每步解，然后得到最优解，计算每步解和真值以及最优解的距离，然后绘制图形。

算法如下：

```python
def soft_thresholding(x, threshold):
    '''软门限法算邻近点投影，x是邻近点'''
    return np.sign(x) * np.maximum(np.abs(x) - threshold, 0)

def proximal_gradient_method(A, b, lamda, alpha=0.0001, max_iter=5000, tol=1e-5, if_draw=True):
    '''邻近点梯度法求解'''
    start_time = time.time()

    n, d1, d2 = A.shape
    x = np.zeros(d2)  # 初始解

    iterates = []  # 记录每步的解

    # 迭代求解
    for _ in range(max_iter):
        x_old = x.copy()

        gradient = np.zeros(d2)
        # 算梯度
        for i in range(n):
            gradient += A[i].T.dot(A[i].dot(x) - b[i])

        # 软门限法算argmin
        x = soft_thresholding(x - alpha * gradient, lamda * alpha)

        iterates.append(x)  # 记录解

        # 判断收敛
        if np.linalg.norm(x - x_old, ord=2) < tol:
            break

```

```
    end_time = time.time()
    diff_time = end_time - start_time

    # 计算每步解与真值之间以及最优解之间的距离
    distances_to_true = [np.linalg.norm(iterate - x_true, ord=2) for iterate in iterates]
    distances_to_opt = [np.linalg.norm(iterate - x, ord=2) for iterate in iterates]

    if if_draw:
        # 绘制距离变化图
        plt.plot(distances_to_true, label='distance to true')
        plt.plot(distances_to_opt, label='distance to optimal')
        plt.title('proximal gradient method')
        plt.xlabel('iteration')
        plt.ylabel('distance')
        plt.grid()
        plt.legend()
        plt.show()

    print(f'proximal gradient using time(alpha={alpha}, lambda={lamda}): {diff_time}')  # 打印时间
    print(f'distance of proximal gradient x_opt and x_true(alpha={alpha}, lambda={lamda}):'
          f'{np.linalg.norm(x - x_true)}', end='\n\n')  # 打印二范数误差
    return x, distances_to_true, distances_to_opt
```

2.3 交替方向乘子法

交替方向乘子法是有约束算法，是增广拉格朗日乘子法的变形，一般形式为：

$$\begin{aligned} &\min f_1(x) + f_2(y) \\ &s.t.\ Ax + By = d \end{aligned}$$

增广拉格朗日函数为：

$$L_c(x, y, v) = f_1(x) + f_2(y) + v^T(Ax + By) + \frac{C}{2}||Ax + By||^2$$

C是一个大于0的常数，一般取1。

迭代过程为：

$$\begin{aligned} x^{k+1} &= \underset{x}{argmin}\ L_c\left(x, y^k, v^k\right) \\ y^{k+1} &= \underset{y}{argmin} L_c(x^{k+1}, y, v^k) \\ v^{k+1} &= v^k + C(Ax^{k+1} + By^{k+1}) \end{aligned}$$

对于我们的问题，首先要引入约束$x - y = 0$将问题变为有约束优化问题：

$$\begin{aligned} \min \quad & \frac{1}{2}\sum_{i=1}^{10}||A_i x - b_i||^2 + \lambda||y||_1 \\ s.t. \quad & x - y = 0 \end{aligned}$$

然后写出增广拉格朗日函数：

$$L_c(x, y, v) = \frac{1}{2}\sum_{i=1}^{10}||A_i x - b_i||^2 + \lambda||y||_1 + v^T(x - y) + \frac{C}{2}||x - y||^2$$

迭代过程（解出argmin）：

$$\begin{aligned} x^{k+1} &= \underset{x}{argmin}\ \frac{1}{2}\sum_{i=1}^{10}||A_i x - b_i||^2 + (v^k)^T x + \frac{C}{2}||x - y^k||^2 \\ &= (\sum_{i=1}^{10} A_i^T A_i + CI)^{-1}(Cy^k + \sum_{i=1}^{10} A_i^T b_i - v^k) \\ y^{k+1} &= \underset{y}{argmin}\ \lambda||y||_1 - (v^k)^T y + \frac{C}{2}||x^{k+1} - y||^2 \\ &= \underset{y}{argmin}\ \lambda||y||_1 + \frac{C}{2}||y - (x^{k+1} + \frac{1}{C}v^k)||^2 \\ &= \text{soft\_threshold}(x^{k+1} + \frac{1}{C}v^k, \frac{\lambda}{C}) \\ v^{k+1} &= v^k + C(x^{k+1} - y^{k+1}) \end{aligned}$$

同样，根据迭代过程就可以设计出交替方向乘子法的算法了，记录每步解，然后得到最优解，计算每步解和真值以及最优解的距离，然后绘制图形。

算法如下：

```
def admm(A, b, lamda, C=1, max_iter=1000, tol=1e-5, if_draw=True):
    '''交替方向乘子法求解'''
    start_time = time.time()

    n, d1, d2 = A.shape
    x = np.zeros(d2)  # 初始解
    y = np.zeros(d2)
    v = np.zeros(d2)

    iterates = []  # 记录每步的解

    for _ in range(max_iter):
        x_old = x.copy()

        # 更新x
        x = np.linalg.inv(np.sum([A[i].T.dot(A[i]) for i in range(n)],
axis=0) + C*np.eye((d2)))
        x = x.dot(np.sum([A[i].T.dot(b[i]) for i in range(n)], axis=0) +
C*y - v)
        # 更新y
```

```
        y = soft_thresholding(x + v/C, lamda/C)
        # 更新v
        v += C*(x-y)

        iterates.append(x)

        # 判断收敛
        if np.linalg.norm(x - x_old, ord=2) < tol:
            break

    # 计算每步解与真值之间以及最优解之间的距离
    distances_to_true = [np.linalg.norm(iterate - x_true, ord=2) for iterate in iterates]
    distances_to_opt = [np.linalg.norm(iterate - x, ord=2) for iterate in iterates]

    end_time = time.time()
    diff_time = end_time - start_time

    if if_draw:
        # 绘制距离变化图
        plt.plot(distances_to_true, label='distance to true')
        plt.plot(distances_to_opt, label='distance to optimal')
        plt.title('alternating direction method of multipliers')
        plt.xlabel('iteration')
        plt.ylabel('distance')
        plt.grid()
        plt.legend()
        plt.show()

    print(f'admm using time(C={C}, lambda={lamda}): {diff_time}')  # 打印所用时间
    print(f'distance of admm x_opt and x_true(C={C}, lambda={lamda}):'
          f'{np.linalg.norm(x - x_true)}', end='\n\n')  # 打印二范数误差
    return x, distances_to_true, distances_to_opt
```

2.4 次梯度法

次梯度法用于不可微目标函数，用次梯度代替梯度，迭代过程如下：

$$x^{k+1} = x^k - \alpha^k g_0\left(x^k\right)$$

其中$g_0(x^k)$是$f_0(x^k)$的次梯度，$g_0\left(x\right) \in \partial f_0\left(x\right)$。

对于我们的问题，我们还是把$f_0(x)$分成$s(x) = \frac{1}{2}\sum_{i=1}^{10}||A_i x - b_i||^2$和$r(x) = \lambda||x||_1$两个部分，$s(x)$直接求梯度，$r(x)$求次梯度，则

$$g_0(x^k) = \nabla s(x^k) + g_r(x^k)$$

$g_r(x^k)$是$\lambda||x||_1$的次梯度，如下：

$$g_r(x^k) = \begin{cases} \lambda & x^k > 0 \\ [-\lambda, \lambda] & x^k = 0 \\ -\lambda & x^k < 0 \end{cases}$$

迭代公式：

$$\begin{aligned} x^{k+1} &= x^k - \alpha^k(\nabla s(x^k) + g_r(x^k)) \\ &= x^k - \alpha^k(\sum_{i=1}^{10} A_i^T(A_i x - b_i) + g_r(x^k)) \end{aligned}$$

我们使用**固定步长**。

通过观察邻近点梯度法的迭代公式，其实可以发现邻近点梯度法和次梯度法本质是一样的。

算法如下：

```python
def subgradient(A, b, lamda, alpha=0.0001, max_iter=5000, tol=1e-5, if_draw=True):
    '''次梯度法求解'''
    start_time = time.time()

    n, d1, d2 = A.shape
    x = np.zeros(d2)  # 初始解

    iterates = []  # 记录每步的解

    for _ in range(max_iter):
        x_old = x.copy()

        # 次梯度
        g = np.empty_like(x)
        for i, data in enumerate(x):
            if data == 0:
                g[i] = 2 * np.random.random() - 1  # [-1, 1]
            else:
                g[i] = np.sign(x[i])
        g *= lamda
        g += np.sum([A[i].T.dot(A[i].dot(x) - b[i]) for i in range(n)], axis=0)
        # 更新x
        x = x - alpha*g

        iterates.append(x)

        # 判断收敛
        if np.linalg.norm(x - x_old, ord=2) < tol:
            break
```

```
30
31      end_time = time.time()
32      diff_time = end_time - start_time
33
34      # 计算每步解与真值之间以及最优解之间的距离
35      distances_to_true = [np.linalg.norm(iterate - x_true, ord=2) for
    iterate in iterates]
36      distances_to_opt = [np.linalg.norm(iterate - x, ord=2) for iterate
    in iterates]
37
38      if if_draw:
39          # 绘制距离变化图
40          plt.figure()
41          plt.plot(distances_to_true, label='distance to true')
42          plt.plot(distances_to_opt, label='distance to optimal')
43          plt.title('subgradient')
44          plt.xlabel('iteration')
45          plt.ylabel('distance')
46          plt.grid()
47          plt.legend()
48          plt.show()
49
50      print(f'subgradient using time(alpha={alpha}, lambda={lamda}):
    {diff_time}')  # 打印时间
51      print(f'distance of subgradient x_opt and x_true(alpha={alpha},
    lambda={lamda}):'
52            f'{np.linalg.norm(x - x_true)}', end='\n\n')  # 打印二范数误差
53      return x, distances_to_true, distances_to_opt
```

2.5 正则化参数调整

最后我们还要调整正则化参数，代码如下：

```
1  def adjust_lamda(A, b, lamdas, method):
2      '''
3      调整正则化参数，lamdas是参数列表，method决定用哪个优化算法，同时作为绘制图
    形的suptitle，
4      method只能取值'proximal gradient', 'admm' 或'subgradient'
5      '''
6      fig, axes = plt.subplots(int(sqrt(len(lamdas))),
    ceil(len(lamdas)/2), figsize=(12, 8))  # 创建多个子图
7      # 画每一个参数值对应的子图
8      for i, lamda in enumerate(lamdas):
9          if method == 'proximal gradient': r1 =
    proximal_gradient_method(A, b, lamda, if_draw=False)
10         elif method == 'admm': r1 = admm(A, b, lamda, if_draw=False)
11         elif method == 'subgradient': r1 = subgradient(A, b, lamda,
    if_draw=False)
```

```
12
13          row, col = divmod(i, ceil(len(lamdas)/2))  # 计算子图位置
14          axes[row, col].plot(r1[1], label='distance to true')
15          axes[row, col].plot(r1[2], label='distance to opt')
16          axes[row, col].set_title(r"$\lambda = $" + f"{lamda}")
17          axes[row, col].set_xlabel('iteration')
18          axes[row, col].set_ylabel('distance')
19          axes[row, col].grid()
20          axes[row, col].legend()
21
22      plt.suptitle(method)
23      plt.tight_layout()
24      plt.show()
```

指定lamdas作为参数列表，比如：

```
1  lamdas = [0.01, 0.1, 1, 5, 50, 100]
```

然后调参。

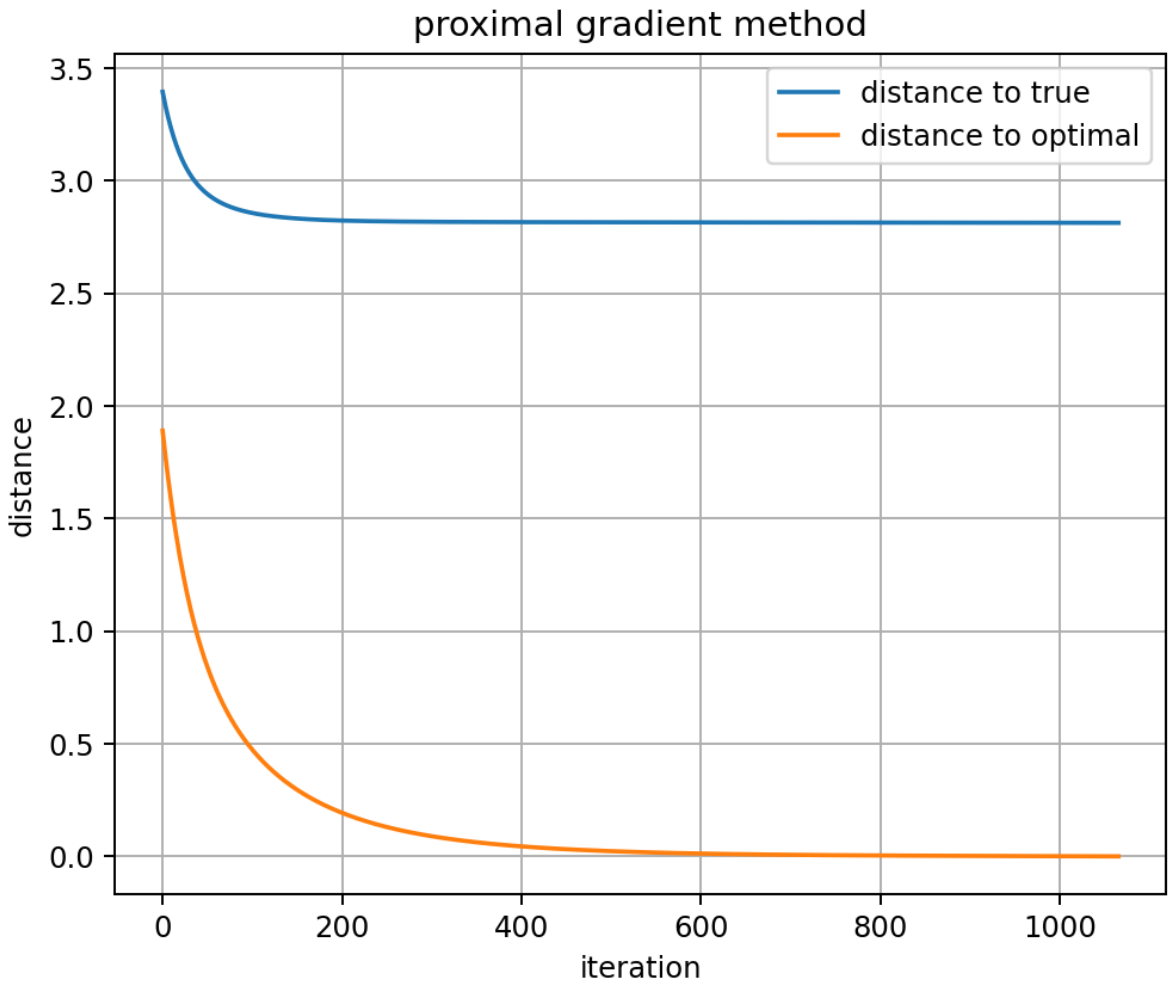
3 结果分析

先运行多次源代码文件产生结果。

3.1 三种算法的结果

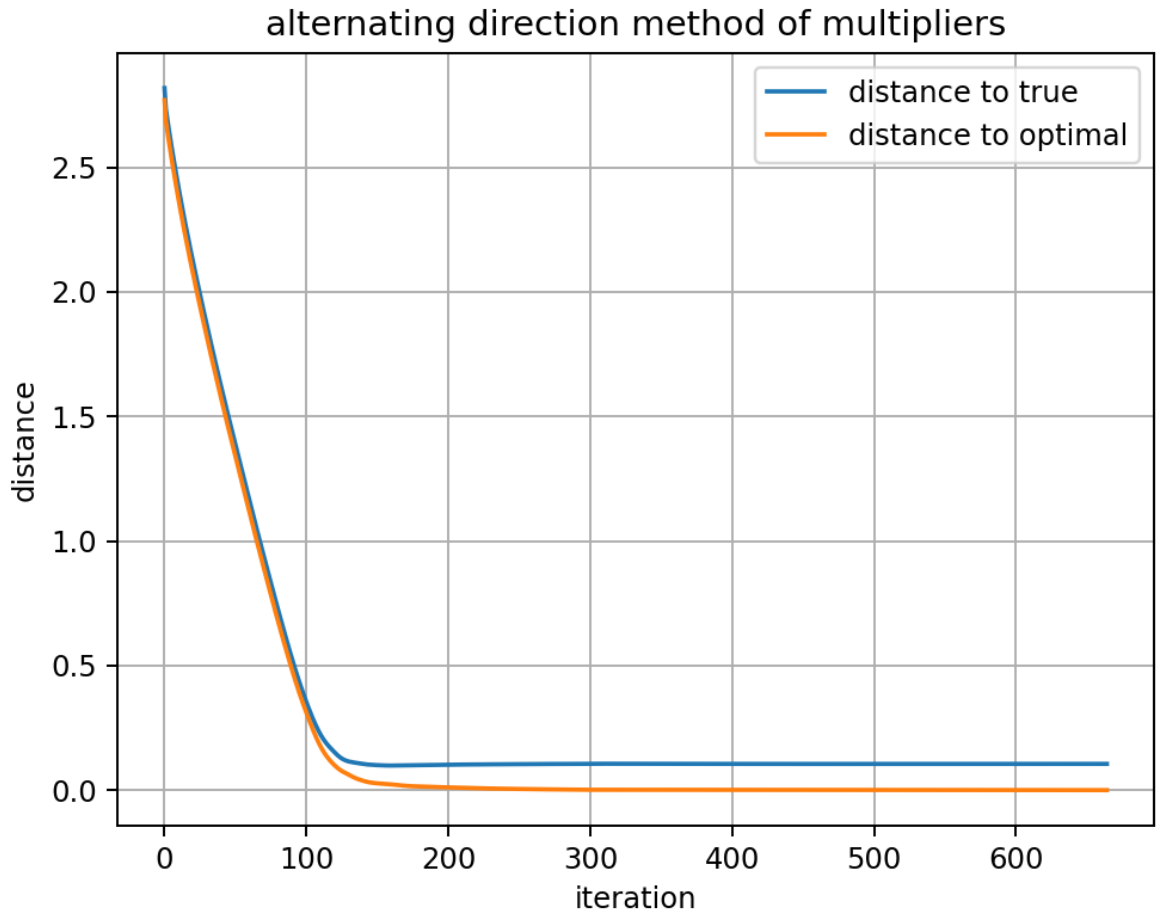
我们令$\lambda = 0.01$来进行求解。绘制每步解与真值之间以及最优解之间的距离曲线。

3.1.1 邻近点梯度法结果



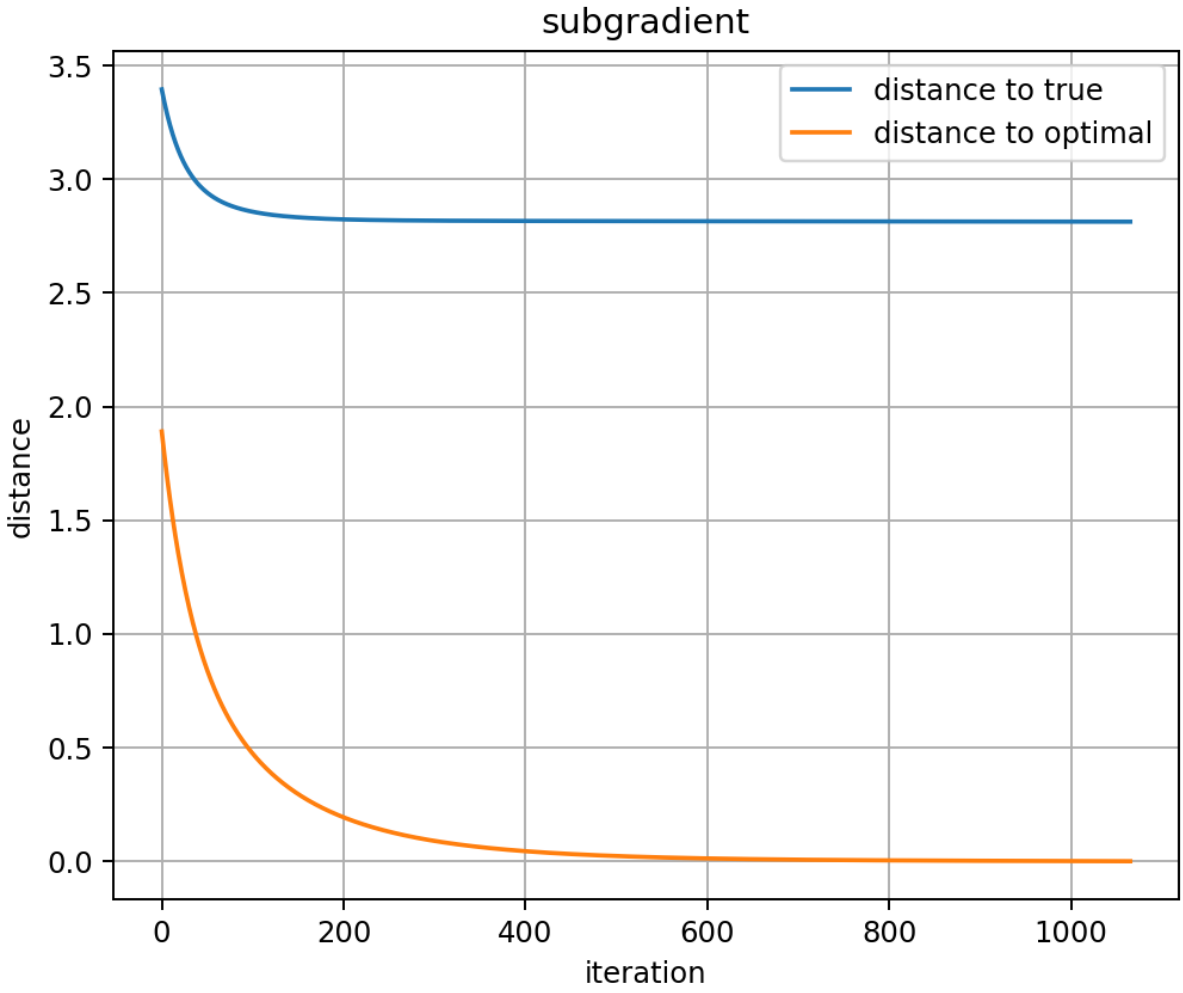
可以看到解是单调的，也就是说解会越来越接近最优解以及真值，但是最优解和真值之间存在一定的距离，是正则化强度较小的原因。

3.1.2 交替方向乘子法结果



解是单调的，且在正则化强度比较小的情况下也能很好的解出真值，效果很好。而且我们能看到收敛速度很快。

3.1.3 次梯度法结果



和邻近点梯度法几乎一致，算法设计部分也说到了，次梯度法和邻近点梯度法本质是一样的，尤其是固定步长的时候。区别在于邻近点梯度法使用的是软门限算法，次梯度法随机选取一个次梯度进行梯度下降。

3.1.4 对比分析

以上三种算法所用时间和二范数误差（最优解和真实解之间的二范数距离）：

```
proximal gradient using time(alpha=0.0001, lambda=0.01): 0.041871070861816406
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=0.01):2.0108056602625766

admm using time(C=1, lambda=0.01): 8.300342321395874
distance of admm x_opt and x_true(C=1, lambda=0.01):0.10709337591099677

subgradient using time(alpha=0.0001, lambda=0.01): 0.22507405281066895
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=0.01):2.010787277462632
```

可以看到，在正则化强度比较小的时候，交替方向乘子法的解是最好的，但是交替方向乘子法速度特别慢，足足比其他两个方法慢了20-100倍，速度慢的原因是迭代x的时候需要矩阵求逆，逆的开销是很大的。

综上，交替方向乘子法效果最好，但是速度最慢，邻近点梯度法和次梯度法效果相同。

3.2 正则化参数对结果的影响

正则化项就是 $\lambda||x||_1$，目的是让**解变得稀疏**，从而提高泛化能力，减少过拟合和复杂性。理论上，λ 越大，正则化强度越大，解越稀疏。

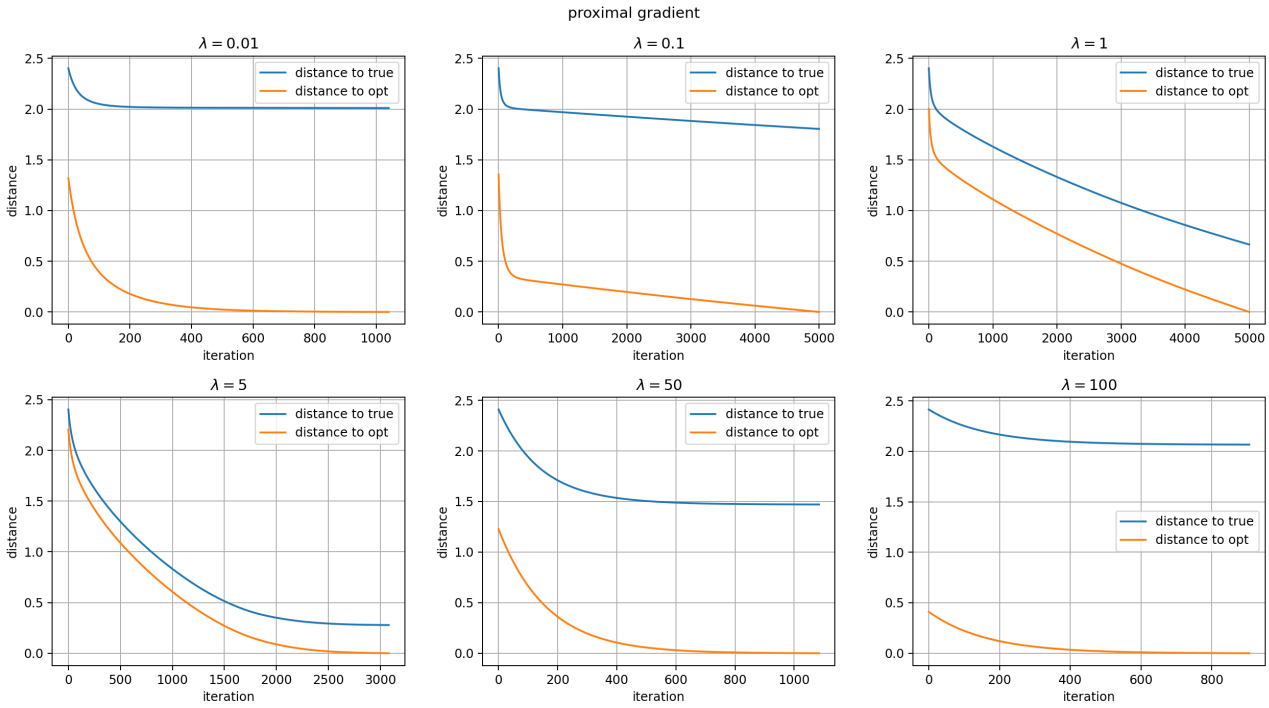
我们调整参数 λ，分析 λ 的影响，参数值选择如下：

```
lamdas = [0.01, 0.1, 1, 5, 50, 100]
```

这是调了很多次才确定的大致范围。

3.2.1 对邻近点梯度法的影响

距离曲线：



所用时间和最优解的二范数误差：

```
proximal gradient using time(alpha=0.0001, lambda=0.01): 0.04883456230163574
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=0.01):2.0108056602625766

proximal gradient using time(alpha=0.0001, lambda=0.1): 0.2466413974761963
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=0.1):1.8047782670044044

proximal gradient using time(alpha=0.0001, lambda=1): 0.19358062744140625
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=1):0.6651121236676977

proximal gradient using time(alpha=0.0001, lambda=5): 0.11955428123474121
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=5):0.277660645046701

proximal gradient using time(alpha=0.0001, lambda=50): 0.04453921318054199
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=50):1.4704665116543747

proximal gradient using time(alpha=0.0001, lambda=100): 0.03696107864379883
distance of proximal gradient x_opt and x_true(alpha=0.0001, lambda=100):2.0661212455528224
```

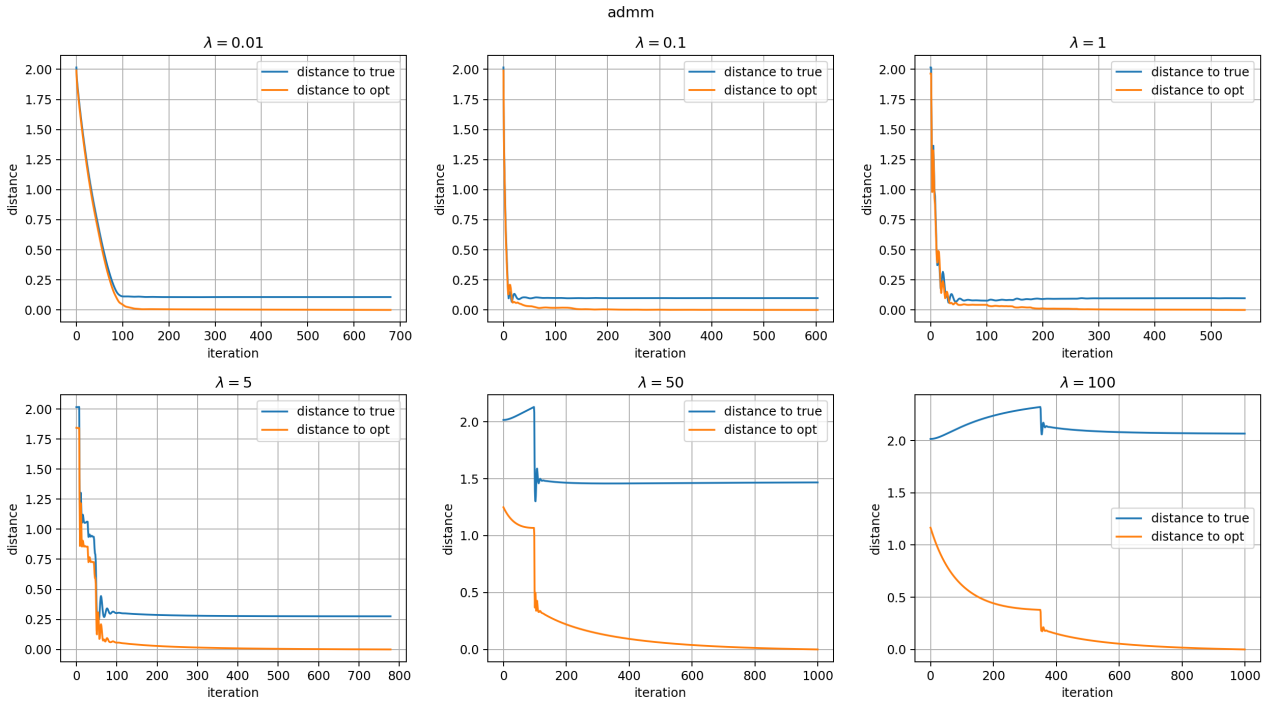
可以看到，和理论有些出入，并不是 λ 越大正则化效果越好，有一个转折点，也就是**随着 λ 增大，最优解的正确性先变好再变差，即最优解和真值先接近后远离**。

λ 过小，解不够稀疏，而真值是一个稀疏向量，故和真值有一些距离；λ 取值合适，解的稀疏性适中，和真值比较接近；λ 过大，解过于稀疏，都很接近于0，拟合程度很低，故和真值有所偏离。

对于我们的问题，邻近点梯度法的最优 λ 大概在5左右。

3.2.2 对交替方向乘子法的影响

距离曲线：



所用时间和最优解的二范数误差：

```
admm using time(C=1, lambda=0.01): 5.953181982040405
distance of admm x_opt and x_true(C=1, lambda=0.01):0.10709337591099677

admm using time(C=1, lambda=0.1): 5.82041072845459
distance of admm x_opt and x_true(C=1, lambda=0.1):0.09828343643155092

admm using time(C=1, lambda=1): 4.4336159229278564
distance of admm x_opt and x_true(C=1, lambda=1):0.09709518002339822

admm using time(C=1, lambda=5): 6.626552104949951
distance of admm x_opt and x_true(C=1, lambda=5):0.2757059670418071

admm using time(C=1, lambda=50): 8.792205333709717
distance of admm x_opt and x_true(C=1, lambda=50):1.4668795932186385

admm using time(C=1, lambda=100): 8.382733821868896
distance of admm x_opt and x_true(C=1, lambda=100):2.066969783510911
```

影响和邻近点梯度法类似，但是我们会发现 λ 取值比较小的时候，交替方向乘子法效果还是很好，这说明交替方向乘子法的解本身就具有稀疏性。实际上，在 $\lambda = 0.01$ 时， 交替方向乘子法的最优解的部分维度如下：

```
[-7.24867153e-07 -5.20489262e-07 -1.02001511e-02 -1.00261814e-02
  6.60863382e-03  4.96220828e-07  1.65357424e-02  2.16035361e-02
 -4.40448075e-07  3.18550324e-08 -2.40024869e-07  6.78918373e-07
  4.04626554e-03  1.11853320e-06  1.45624759e-07  9.15894773e-08
 -5.38032624e-03  5.04572217e-07 -7.41330826e-03  8.61389751e-07
```

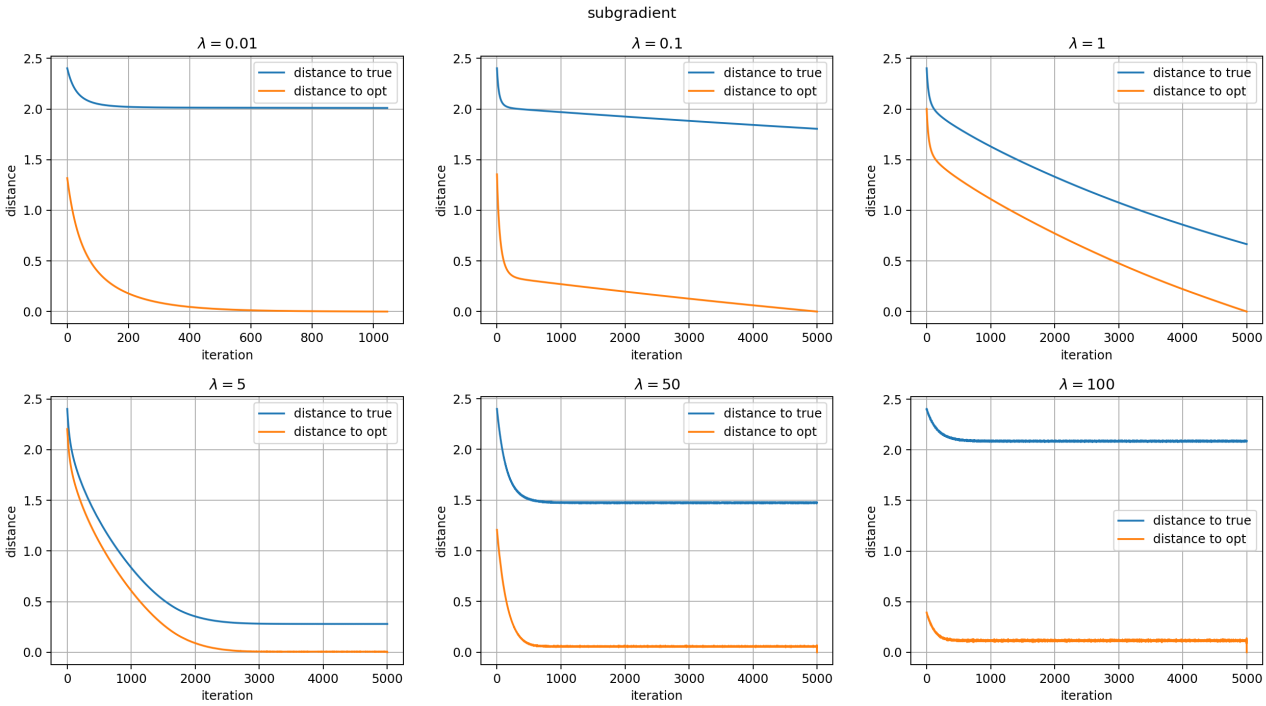
可以看到确实是十分稀疏的。

而且当 λ 取值适中时，收敛速度会非常快，大概十几步就收敛，我的理解是快速稀疏导致的快速收敛。当 λ 变大时会出现断层现象，然后会偏离真值，我的理解是解快速过于稀疏，然后就不会怎么变化了。

对于我们的问题，交替方向乘子法的最优 λ 大概在1左右。

3.2.3 对次梯度法的影响

距离曲线：



所用时间和最优解的二范数误差：

```
subgradient using time(alpha=0.0001, lambda=0.01): 0.24123454093933105
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=0.01):2.010787277462632

subgradient using time(alpha=0.0001, lambda=0.1): 1.0805079936981201
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=0.1):1.8048202626663588

subgradient using time(alpha=0.0001, lambda=1): 0.9999935626983643
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=1):0.665479583310972

subgradient using time(alpha=0.0001, lambda=5): 0.9640612602233887
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=5):0.27927697267970697

subgradient using time(alpha=0.0001, lambda=50): 0.9869132041931152
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=50):1.476391851054622

subgradient using time(alpha=0.0001, lambda=100): 1.0020630359649658
distance of subgradient x_opt and x_true(alpha=0.0001, lambda=100):2.089066572779819
```

前面提到，次梯度法和邻近点梯度法本质是一样的，所以结果几乎一致，λ 对其影响也和邻近点梯度法一致。

对于我们的问题，次梯度法的最优 λ 大概在5左右。

综上，正则化参数会让解变得稀疏，从而更接近真值，但不能太稀疏，否则没有拟合能力。另外，交替方向乘子法对正则化参数相对最不敏感。