

# 最优化理论-期末报告

21307035 邓栩瀛

## 问题描述

用 MATLAB 或 Python 实现：

考虑一个10节点的分布式系统。结点*i*有线性测量 $b_i = A_i x + e_i$ ，其中 $b_i$ 为5维的测量值， $A_i$ 为5×200维的测量矩阵， $x$ 为200维的未知稀疏向量且稀疏度为5， $e_i$ 为5维的测量噪声。从所有 $b_i$ 与 $A_i$ 中恢复 $x$ 的一范数正则化最小二乘模型如下：

$$\min_x \frac{1}{2} \|A_1 x - b_1\|_2^2 + \cdots + \frac{1}{2} \|A_{10} x - b_{10}\|_2^2 + \lambda \|x\|_1 \quad (1)$$

其中 $\lambda > 0$ 为正则化参数，请分别设计下述算法求解该问题：

- 1、邻近点梯度法
- 2、交替方向乘子法
- 3、次梯度法

在实验中，设 $x$ 的真值中的非零元素服从均值为0方差为1的高斯分布， $A_i$ 中的元素服从均值为0方差为1的高斯分布， $e_i$ 中的元素服从均值为0方差为0.1的高斯分布。对于每种算法，请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，并考虑正则化参数 $\lambda$ 对计算结果的影响

## 算法设计

### 邻近点梯度算法

邻近点梯度法常用于求解以下形式的优化问题：

$$\min f_0(x) = s(x) + r(x) \quad (2)$$

其中 $s(x)$ 光滑可微， $r(x)$ 非光滑不可微。

迭代过程：首先对光滑项做一次梯度下降，得到中间结果 $x^{k+\frac{1}{2}}$ ，然后将这个中间结果代入非光滑项中求其邻近点投影。

$$\begin{aligned} x^{k+\frac{1}{2}} &= x^k - \alpha \times \nabla s(x^k) \\ x^{k+1} &= \operatorname{argmin}_x \{r(x) + \frac{1}{2\alpha} \|x - x^{k+\frac{1}{2}}\|^2\} \end{aligned} \quad (3)$$

求解得到

$$x^{k+1} = \begin{cases} x^{k+\frac{1}{2}} + p, & x^{k+\frac{1}{2}} \\ 0, & |x^{k+\frac{1}{2}}| \leq \alpha \times p \\ x^{k+\frac{1}{2}} - p, & x^{k+\frac{1}{2}} > \alpha \times p \end{cases} \quad (4)$$

1. 初始化变量：

- $x$  和  $x_{\text{prev}}$ ：当前迭代和上一次迭代的优化变量。

- `distance_to_true` 和 `distance_to_optimal`：记录每次迭代后当前解与真实解之间的距离以及当前解与上一次迭代解之间的距离。

## 2. 迭代更新：

- `num_iterations`：迭代的次数。
- 在每次迭代中，首先计算梯度 `gradient`。对于每个节点 `i`，计算 `A[i]` 与 `(A[i]xx-b[i])` 的乘积并累加到 `gradient` 中。
- 将正则化项的梯度 `lambda_val * sign(x)` 加到 `gradient` 中，其中 `sign(x)` 表示 `x` 的逐元素符号函数。
- 更新 `x`：使用步长为 0.001，通过减去步长乘以梯度来更新 `x` 的值。
- 更新 `x_prev`：将 `x` 的当前值复制给 `x_prev`。

## 3. 记录距离：

- 在每次迭代后，计算当前解 `x` 与真实解 `x_true` 之间的距离，并将其记录到 `distance_to_true` 中。
- 计算当前解 `x` 与上一次迭代解 `x_prev` 之间的距离，并记录到 `distance_to_optimal` 中。

```
def proximal_gradient_method():
    x = np.zeros(num_features)
    x_prev = np.zeros(num_features)
    distance_to_true = []
    distance_to_optimal = []

    for _ in range(num_iterations):
        gradient = np.zeros(num_features)
        for i in range(num_nodes):
            gradient += np.dot(A[i].T, np.dot(A[i], x) - b[i])
        gradient += lambda_val * np.sign(x)
        x_prev = x.copy()
        x = x_prev - 0.001 * gradient # 步长为0.001

        distance_to_true.append(np.linalg.norm(x - x_true))
        distance_to_optimal.append(np.linalg.norm(x - x_prev))

    return x, distance_to_true, distance_to_optimal
```

## 交替方向乘子法

交替方向乘子法常用于求解以下形式的优化问题：

$$\begin{aligned} \min & f_1(x) + f_2(y) \\ \text{s.t.} & Ax + By = 0 \end{aligned} \quad (5)$$

算法如下：

$$\begin{cases} x^{k+1} = \operatorname{argmin}_x L_c(x, y^k, v^k) \\ y^{k+1} = \operatorname{argmin}_y L_c(x^{k+1}, y, v^k) \\ v^{k+1} = v^k + c(Ax^{k+1} + By^{k+1}) \end{cases} \quad (6)$$

其中  $L_c(x, y, v)$  为原目标函数的增广拉格朗日函数， $c$  为惩罚项系数。

以上算法可以等价变形为如下形式

$$\begin{cases} x^{k+1} = \operatorname{argmin}_x f_1(x) + \frac{c}{2} \|Ax + By^k + \frac{v^k}{c}\|^2 \\ y^{k+1} = \operatorname{argmin}_y f_2(x) + \frac{c}{2} \|Ax^{k+1} + By + \frac{v^k}{c}\|^2 \\ v^{k+1} = v^k + c(Ax^{k+1} + By^{k+1}) \end{cases} \quad (7)$$

1. 初始化变量：

- `x`、`z` 和 `u`：优化变量和乘子。
- 通过使用 `np.random.randn` 来生成随机数来初始化 `x`、`z` 和 `u`。

2. 迭代更新：

- `num_iterations` 迭代的次数。
- 在每次迭代中，首先更新 `x`：通过求解线性方程组，使用矩阵求逆的方式求解。其中，左侧矩阵是所有节点的数据项的加权和以及正则化项，右侧矩阵是所有节点数据项与目标值的加权和以及正则化项与乘子的差。
- 更新 `z`：通过将 `x` 与乘子 `u` 相加，然后取逐元素的最大值得到 `z`。
- 更新 `u`：通过将 `x` 与 `z` 相减，然后将结果累加到乘子 `u` 上。

3. 记录距离：

- 在每次迭代后，计算当前解 `x` 与真实解 `x_true` 之间的距离，并将其记录到 `distance_to_true` 中。
- 计算当前解 `x` 与上一次迭代解 `z` 之间的距离，并记录到 `distance_to_optimal` 中。

```
def alternating_direction_method():
    x = np.random.randn(num_features)
    z = np.random.randn(num_features)
    u = np.random.randn(num_features)
    distance_to_true = []
    distance_to_optimal = []
    for _ in range(num_iterations):
        x = np.linalg.solve(sum([np.dot(A[i].T, A[i]) for i in range(num_nodes)]) + lambda_val *
np.eye(num_features), sum([np.dot(A[i].T, b[i]) for i in range(num_nodes)]) + lambda_val * (z - u))
        z = np.maximum(0, x + u)
        u = u + x - z
        distance_to_true.append(np.linalg.norm(x - x_true))
        distance_to_optimal.append(np.linalg.norm(x - z))
    return x, distance_to_true, distance_to_optimal
```

## 次梯度法

用  $g_0(x) \in \partial f_0(x)$  表示  $f_0(x)$  在  $x$  处的次梯度。所谓次梯度法，就是在不可微点处用次梯度代替梯度，然后使用梯度下降法。当函数可微时，与梯度下降法无异。

$$x^{k+1} = x^k - \alpha \times g_0(x^k) \quad (8)$$

次梯度法的步长选取：

- 固定步长： $\alpha^k = \alpha$
- 不可加，但平方可加： $\alpha^k = \frac{1}{k}$
- 不可加、平方不可加，但能收敛至 0： $\alpha^k = \frac{1}{\sqrt{k}}$

1. 初始化变量：

- `x`：表示优化变量。
- `distance_to_true` 和 `distance_to_optimal`：用于记录每次迭代后当前解与真实解之间的距离以及当前解的次梯度的范数。

## 2. 迭代更新：

- `num_iterations` 决定了迭代的次数。
- 在每次迭代中，首先计算次梯度 `subgradient`。对于每个节点 `i`，计算 `A[i]` 与 `(A[i] * x - b[i])` 的乘积并累加到 `subgradient` 中。
- 将正则化项的次梯度 `lambda_val * sign(x)` 加到 `subgradient` 中，其中 `sign(x)` 表示 `x` 的逐元素符号函数。

## 3. 计算步长：

- 根据迭代次数 `_`，计算当前的步长 `step_size`。步长的选择是通过除以迭代次数加一的平方根来动态调整。

## 4. 更新变量：

- 使用步长乘以次梯度来更新变量 `x` 的值。
- 对更新后的 `x` 进行非负投影，即将小于零的元素设为零。

## 5. 记录距离：

- 在每次迭代后，计算当前解 `x` 与真实解 `x_true` 之间的距离，并将其记录到 `distance_to_true` 中。
- 计算当前解的次梯度 `subgradient` 的范数，并记录到 `distance_to_optimal` 中。

```
def subgradient_method():
    x = np.zeros(num_features)
    distance_to_true = []
    distance_to_optimal = []
    for _ in range(num_iterations):
        subgradient = np.zeros(num_features)
        for i in range(num_nodes):
            subgradient += np.dot(A[i].T, np.dot(A[i], x) - b[i])
        subgradient += lambda_val * np.sign(x)
        step_size = 0.01 / np.sqrt(_ + 1) # 步长为0.01 / sqrt(t+1)
        x = x - step_size * subgradient
        x = np.maximum(0, x) # 非负投影
        distance_to_true.append(np.linalg.norm(x - x_true))
        distance_to_optimal.append(np.linalg.norm(subgradient))
    return x, distance_to_true, distance_to_optimal
```

## 数值实验

### 设置初始系数

```
num_nodes = 10 # 10节点分布式系统
num_features = 200 # 200维未知稀疏向量
num_measurements = 5 # 5维测量噪声
sparsity = 5 # 稀疏度为5
lambda_val = 1 # 正则化参数
num_iterations = 1000 # 迭代次数
```

## 初始数据的生成

生成稀疏向量x的真值

```
np.random.seed(0)
x_true = np.zeros(num_features)
nonzero_indices = np.random.choice(num_features, sparsity, replace=False)
x_true[nonzero_indices] = np.random.normal(0, 1, sparsity)
```

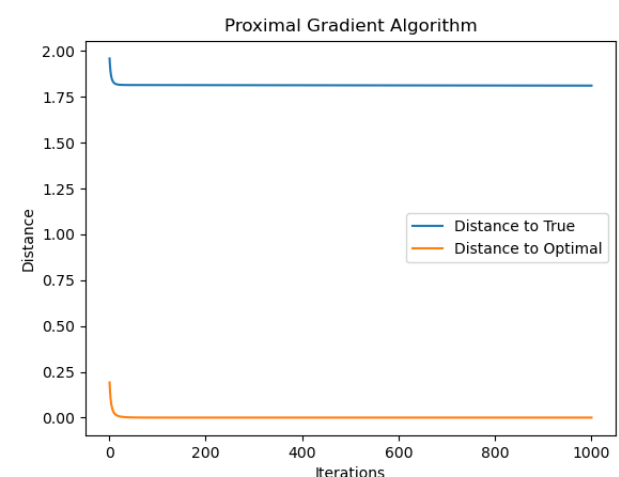
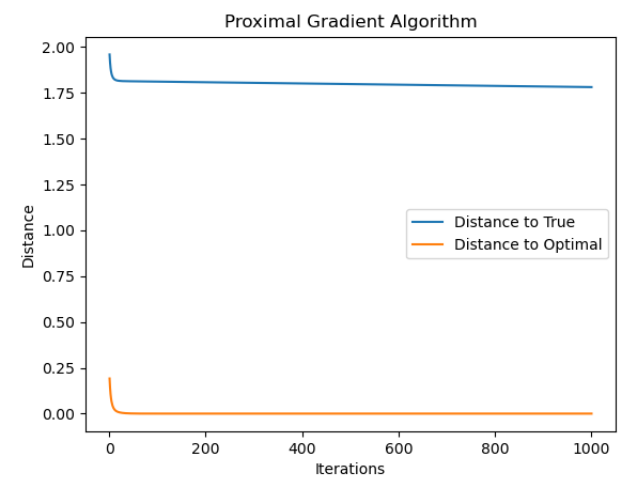
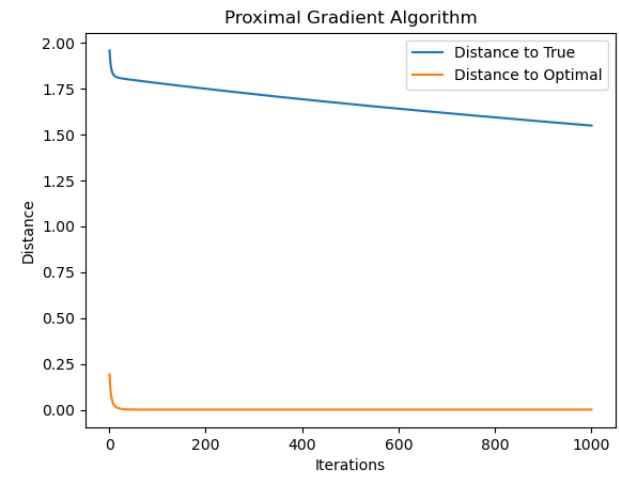
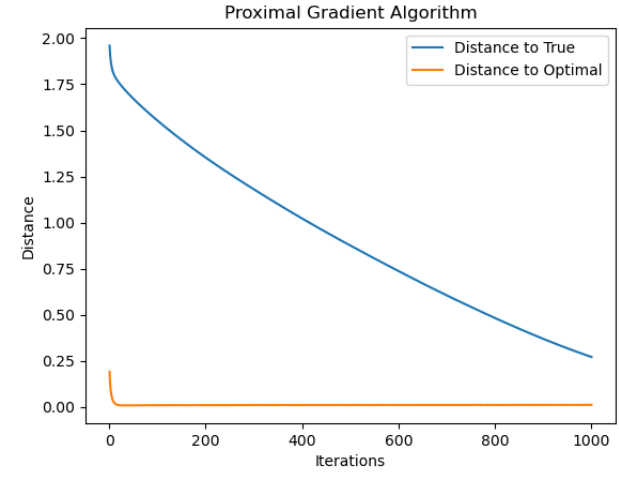
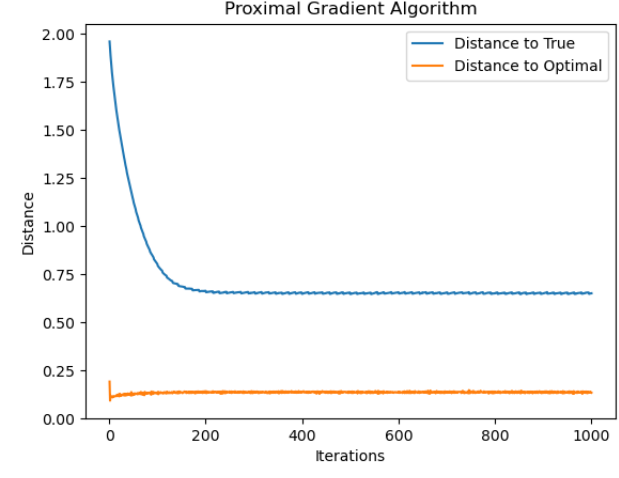
生成测量矩阵A和测量噪声e

```
A = np.random.normal(0, 1, (num_nodes, num_measurements, num_features))
e = np.random.normal(0, 0.1, (num_nodes, num_measurements))
```

生成测量值b

```
b = np.zeros((num_nodes, num_measurements))
for i in range(num_nodes):
    b[i] = np.dot(A[i], x_true) + e[i]
```

## 邻近点梯度算法

正则化参数p	图像
0.001	
0.01	
0.1	
1	
10	

交替方向乘子法

正则化参数p	图像
0.001	
0.01	
0.1	
1	
10	

次梯度法

正则化参数p	图像
0.001	
0.01	
0.1	
1	
10	



### 邻近点梯度算法

随着正则化参数 $p$ 的增大，收敛速度变快，收敛的迭代次数变少。在上面的实验中，当 $p=1$ 时，收敛速度足够快，回归效果也足够好。

由于数据是随机生成的，因此以上结果并不是固定的。

### 交替方向乘子法

在交替方向乘子法中，正则化参数 $p$ 对结果的影响似乎并不明显。将交替方向乘子法的图像与邻近点梯度法、次梯度法相比较，可以看出交替方向乘子法的回归效果并不是很理想，尽管最优解能够收敛，然而最优解与真值的差距较大。

### 次梯度法

在次梯度法中，正则化参数 $p$ 对结果的影响比较明显， $p$ 越小，收敛速度越快，回归效果也更好。