Final Project:
LASSO Optimization

Convex Optimization (Fall 2024)

22336226 王泓沣
Lectured by: Lei Yang
Sun Yat-sen University

1 问题描述

用**邻近梯度法**、**交替方向乘子法**、**次梯度法**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:

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

其中

- b_i 为 5 维的测量值
- A_i 为 5×200 维的测量矩阵，其中的元素服从均值为 0 方差为 1 的高斯分布
- x 为 200 维的未知稀疏向量且稀疏度为 5，真值中的非零元素服从均值为 0 方差为 1 的高斯分布
- e_i 为 5 维的测量噪声，其中的元素服从均值为 0 方差为 0.1 的高斯分布
- $\lambda > 0$ 为正则化参数，控制正则化强度

2 邻近梯度法

2.1 Algorithm

目标函数可被划分为光滑部分和非光滑部分。

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

其中光滑部分 $s(x) = \frac{1}{2}\sum_{i=10}^{10}\|A_ix - b_i\|^2$，不光滑部分 $r(x) = \lambda\|x\|_1$。
迭代步为

$$\begin{cases} x^{k+\frac{1}{2}} = x^k - \alpha^k \nabla s(x^k) \\ x^{k+1} = \arg\min_x \left(r(x) + \frac{1}{2\alpha}\|x - x^{k+\frac{1}{2}}\|^2\right) \end{cases}$$

求次梯度等于 0 代入得

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

其中 $g_r(x^k)$ 为 $r(x)$ 在 x^k 的次梯度，可分类讨论得 x^{k+1}（即软门限算法）

$$soft\_threshold(x, s) = \begin{cases} x - s, & x > s \\ 0, & |x| \le s \\ x + s, & x < -s \end{cases}$$

即

$$x^{k+\frac{1}{2}} = x^k - \alpha^k \sum_{i=1}^{10} A_i^\top (A_ix - b_i)$$
$$x^{k+1} = sign(x) * \max\left(|x| - \alpha\lambda, 0\right)$$

使用固定迭代步长。

2.2 Code

```
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

    # 计算每步解与真实解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.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 Result

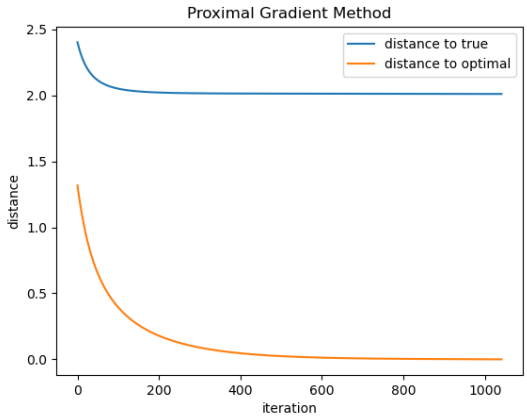

Figure 1: Proximal Gradient Method

解单调，且最优值和真值间存在一定的距离。

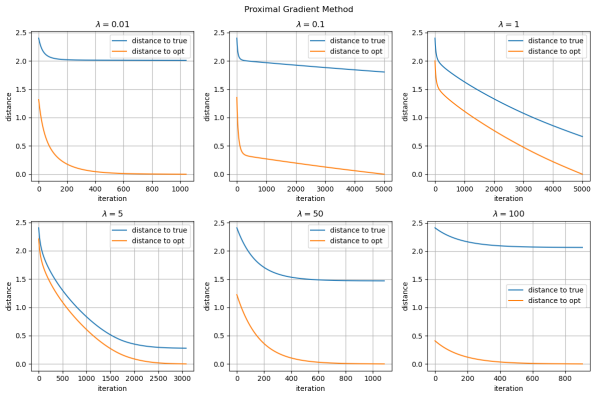

Figure 2: PGM v.s. λ

最优解和真值随 λ 增大先接近后远离。

3 交替方向乘子法

3.1 Algorithm

ADMM 的一般形式为

$$\min\left(f_1(x)+f_2(y)\right) s.t. Ax+By-c=0$$

其增广拉格朗日函数为

$$L_\sigma(x,y,\nu)=f_1(x)+f_2(y)+\nu^t op(Ax+By)+\frac{\sigma}{2}\|Ax+By\|^2$$

其中 σ 为罚因子。迭代步为

$$\begin{cases} x^{k+1}=\arg\min_x L_\sigma(x,y^k,\lambda^k) \\ y^{k+1}=\arg\min_y L_\sigma(x^{k+1},y,\nu^k) \\ \nu^{k+1}=\nu^k+\sigma(Ax^{k+1}+By^{k+1}) \end{cases}$$

对于一范数正则化，引入约束 $x=y$，则原问题转化为

$$\min\frac{1}{2}\sum_{i=1}^{10}\|A_ix-b_i\|^2+\lambda\|y\|_1 s.t. x-y=0$$

其增广拉格朗日函数为

$$L_\sigma(x,y,\nu)=\frac{1}{2}\sum_{i=1}^{10}\|A_ix-b_i\|^2+\lambda\|y\|_1+\nu^\top(x-y)+\frac{\sigma}{2}\|x-y\|^2$$

解得迭代步为

$$\begin{cases}x^{k+1}=(\sum_{i=1}^{10}A_i^\top A_i+\sigma I)^{-1}(\sigma y^k+\sum_{i=1}^{10}A_i^topb_i-\nu^k)\\ y^{k+1}=soft\_threshold(x^{k+1}+\frac{1}{\sigma}\nu^k,\frac{lambda}{\sigma})\\ \nu^{k+1}=\nu^k+\sigma(x^{k+1}-y^{k+1})\end{cases}$$

3.2 Code

```
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
    r = []

    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)
        r.append(f0(A, b, x, lamda))

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

    # 计算每步解与真实解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.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
```

3.3 Result

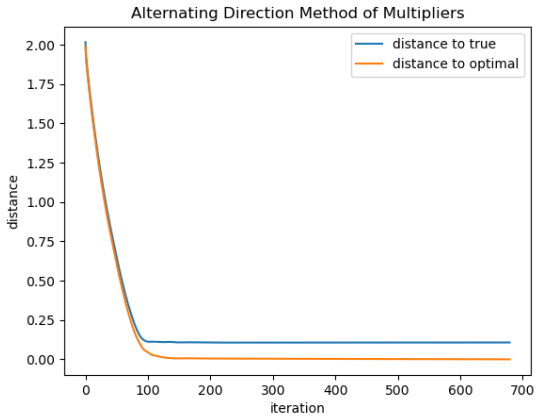

Figure 3: ADMM

解单调且收敛速度较快

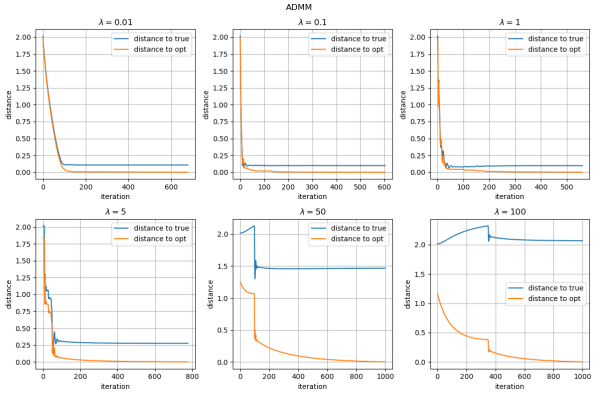

Figure 4: ADMM v.s. λ

当 λ 取值合适时，ADMM 的快速稀疏会导致快速收敛，但当 λ 过大时，由于解过于稀疏，结果会发生较大的偏差。

4 次梯度法

4.1 Algorithm

对一范数正则化部分求次梯度 $g_r(x^k)$

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

迭代公式为

$$x^{k+1} = x^k - \alpha^k (\sum_{i=1}^{10} A_i^\top (A_i x - b_i) + g_r(x^k))$$

4.2 Code

```
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

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

    # 计算每步解与真实解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.figure()
        plt.plot(distances_to_true, label='distance to true')
        plt.plot(distances_to_opt, label='distance to optimal')
        plt.title('Subgradient')
        plt.xlabel('iteration')
        plt.ylabel('distance')
        plt.legend()
        plt.show()

    print(f'subgradient using time(alpha={alpha}, lambda={lamda}): {diff_time}')  #
        打印时间
    print(f'distance of subgradient 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
```

4.3 Result

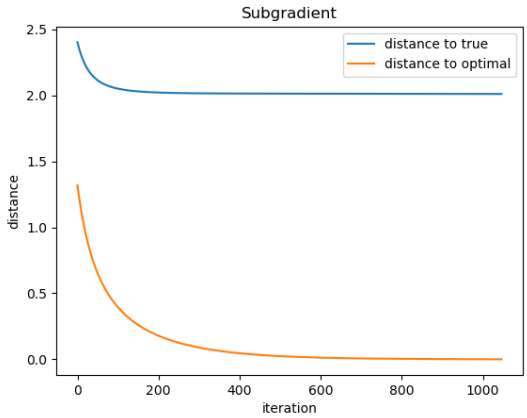

Figure 5: Subgradient

次梯度法和邻近梯度法本质相同，区别在于邻近梯度法使用软门限算法，而次梯度法随机选取一个负次梯度方向进行下降。

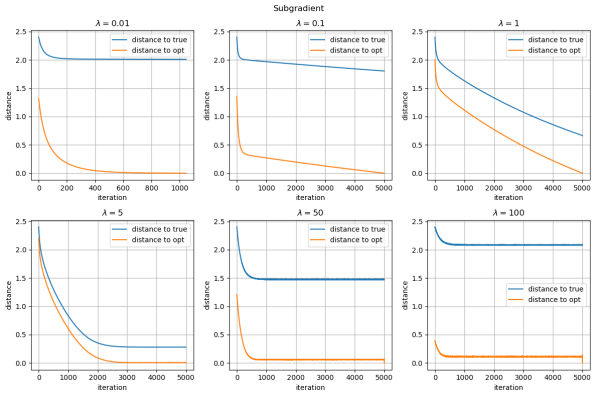

Figure 6: Subgradient v.s. λ

观察到次梯度法的下降速度在各 λ 处较快，整体曲线随 *lambda* 变化的趋势与邻近梯度法类似。

5 Conclusion

正则化参数 λ 的作用是让解变得稀疏，当 λ 过小时收敛速度慢，当 λ 过大时拟合真值的效果不佳。邻近梯度法和次梯度法整体曲线相似，交替方向乘子法对 λ 相对较敏感。