2020 年 春 季学期研究生课程考核

实验报告（二）

考 核 科 目：组合优化与凸优化

学生所在院（系）：计算机科学与技术学院

学 生 所 在 学 科：计算机科学与技术

学 生 姓 名：于晟健

学 号：19S003037

学 生 类 别：全日制学术型硕士研究生

考 核 结 果　　阅 卷 人

实验 2　有约束最优化方法

2.1 实验要求

体会优化当中：目标函数+正则项的基本优化方式，实际上绝大部分的学习机器模型就在这个基本的模型上不断变化目标函数（往往对应损失/代价函数）：

- 如果是平方损失，就是最小二乘；
- 如果是 Hinge 损失，就是 SVM；
- 如果是指数损失，类似 Boosting；
- 如果是对数损失，就是逻辑斯蒂回归等等；

和正则项（往往对应先验信息，一般常用范数形式表达）：

- 表示的稀疏性，则用$\|\cdot\|_1 \ldots \|\cdot\|_0$，$\|\cdot\|_p$等表示；
- 表示的平滑性，则其梯度之和值会比较小；
- 局部特性，则局部算子的相关性大，非局部算子相关性小等等；

产生各种类似的模型，当然都利用了一些不同的先验信息。其中，关于 0 范数和 1 范数可参考 Elad 的书 *Michael Elad, sparse and redundant representations: from theory to applications, Springer* 2010。

2.2 实验任务

2.2.1 任务一

阅读增广拉格朗日方法 ALM（分布式增广拉格朗日方法）和交替方向乘子法 ADMM，实现或熟悉算法后调用库或源代码，实现基本测试问题并进行比较。

2.2.2 任务二

阅读 General PCA 或 Robust PCA，理解低秩与稀疏约束（*Panoramic Robust PCA for foreground-background separation on noisy, free-motion camera video, IEEE Transactions on Computational Imaging*）；基于此，实现视频中的前背景分离，可以采用 ADMM（Alternating direction method of multipliers）等方法求解。

或者根据自己的研究方向，将问题形式化为带正则项，或可分离的形式，然后用 ADMM 等方法求解，如有源代码，尝试改进。

2.3 实验步骤和对比结果

2.3.1 实现等式约束下的增广拉格朗日方法和交替方向乘子法并比较

设有如下优化问题：

$$\min f(x) + g(y), s.t. Ax + By = c$$

故其增广拉格朗日形式为（其中惩罚因子 $\rho>0$）：

$$L_\rho(x,y,\lambda)=f(x)+g(y)+\lambda(Ax+By-c)+\frac{\rho}{2}\|Ax+By-c\|_2^2$$

（1）实现增广拉格朗日方法

若用增广拉格朗日法解决该优化问题，则迭代形式为：

$$(x^{k+1},y^{k+1})\leftarrow arg\min_{x,y}L_\rho(x,y,\lambda^k)$$

$$\lambda^{k+1}\leftarrow\lambda^k+\rho(Ax^{k+1}+By^{k+1}-c)$$

其中，(x^{k+1},y^{k+1})可通过坐标轮换法进行求解，算法代码如下：

```
def ALM(loss_function: example, start: point, lama: float, epsilon=1e-1, iteration_max=1000)
-> list:
    """

    :param loss_function:
    :param start:
    :param lama:
    :param epsilon:
    :param iteration_max:
    :return:
    """
    points, M, k = [start], len(start), 0

    while True:
        # find the new point by cyclic coordinate method
        p = points[k]
        p_old = p
        while True:
            direction = [0] * M
            direction[np.mod(k, M)] = 1
            step = golden_search(loss_function, lama, p, direction)
            p = p + point(direction[0] * step, direction[1] * step)
            points.append(p)
            k += 1
            if k > iteration_max or (points[k] - points[k - 1]).L2() < epsilon: break
        # update the lama
        lama = lama + loss_function.rho * loss_function.subject_to(p)
        # if meet the termination condition then break
        if k > iteration_max or (p - p_old).L2() < epsilon: break

    return points
```

注意，在此处我们将坐标轮换法求解的迭代次数计入总迭代次数，显示算法的迭代复杂度。

（2）实现交替方向乘子法

若用交替方向乘子法解决该优化问题，则迭代形式为：

$$x^{k+1}\leftarrow arg\min_{x}L_\rho(x,y^k,\lambda^k)$$

$$y^{k+1}\leftarrow arg\min_{y}L_\rho(x^k,y,\lambda^k)$$

$$\lambda^{k+1}\leftarrow\lambda^k+\rho(Ax^{k+1}+By^{k+1}-c)$$

算法代码如下：

```
def ADMM(loss_function: example, start: point, lama: float, epsilon=1e-1, iteration_max=1000)
-> list:
    """
    Alternating Direction Method of Multipliers
    :param loss_function:
    :param start:
    :param lama:
```

```
    :param epsilon:
    :param iteration_max:
    :return:
    """
    points, M, k = [start], len(start), 0

    while True:
        # update the point
        p = points[k]
        direction = [1, 0]
        step = golden_search(loss_function, lama, p, direction)
        p = p + point(direction[0] * step, direction[1] * step)
        direction = [0, 1]
        step = golden_search(loss_function, lama, p, direction)
        p = p + point(direction[0] * step, direction[1] * step)
        points.append(p)
        k += 1
        # update the lama
        lama = lama + loss_function.rho * loss_function.subject_to(p)
        # if meet the termination condition then break
        if k > iteration_max or (points[k] - points[k - 1]).L2() < epsilon: break

    return points
```

（3）比较增广拉格朗日方法和交替方向乘子法

增广拉格朗日方法和交替方向乘子法相比，前者要求的是对两个原始变量的联合最小化，后者是将对偶上升法的可分解性和乘子法的上界收敛属性融合在一起的算法。

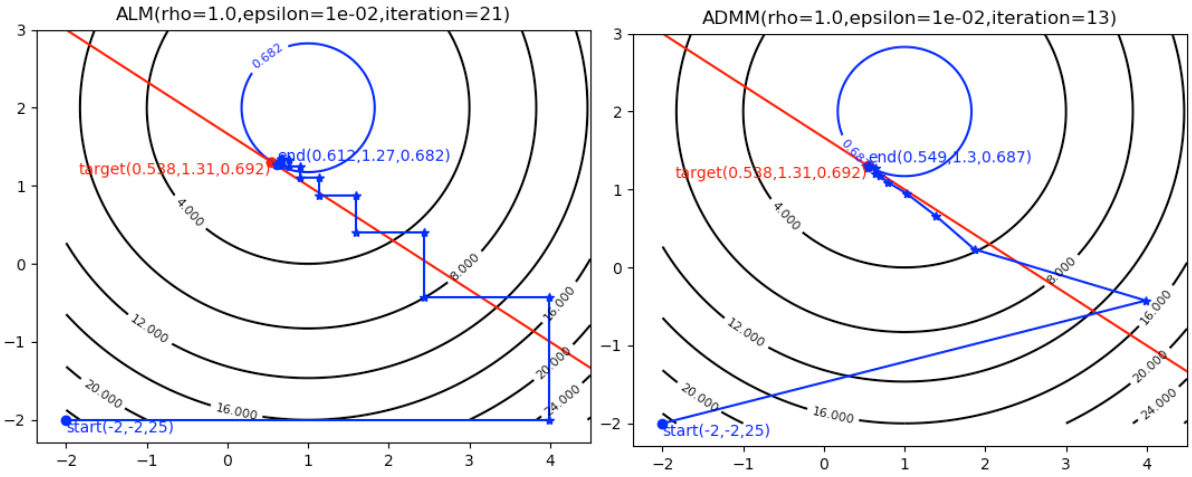
因此，我们以下列优化问题作为样例进行测试：

$$\min_{x,y}(x-1)^2+(y-2)^2 \quad \text{s.t. } 2x+3y=5$$

即有

```
class point:
    x, y = 0, 0

    def __init__(self, x: float, y: float):
        self.x, self.y = x, y

    def __add__(self, other):
        return point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return point(self.x - other.x, self.y - other.y)

    def __len__(self):
        return 2

    def __str__(self):
        return str(self.x) + "," + str(self.y)

    def L2(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5


class example:
    rho = 1.0

    def __init__(self, rho: float):
        self.rho = rho

    def F(self, p: point):
        return (p.x - 1) ** 2 + (p.y - 2) ** 2

    def subject_to(self, p: point):
        return 2 * p.x + 3 * p.y - 5

    def L(self, p: point, lam: float):
        return self.F(p) + lam * self.subject_to(p) + 0.5 * self.rho *(self.subject_to(p)**2)
```

我们以 rho=1，(-2, -2)为出发点得到以下结果：



由图可知，ADMM 的迭代次数明显少于 ALM 的迭代次数，同时结果更为精确。

2.3.2 实现基于 Robust PC 的视频前背景分离

（1）实现基于非精确 ALM 的 Robust PCA

主成分分析（PCA）可以有效的找出数据中最重要的元素和结构，去除噪音和冗余，能将原有的复杂数据进行降维。最简单的主成分分析方法就是 PCA，从线性代数的角度看，PCA 的目标就是使用另一组基去重新描述得到的新的数据空间，通过这组新的基，能揭示与原有的数据间的关系，即这个维度最重要的“主元”。PCA 的目标就是找到这样的“主元”，最大程度的去除冗余和噪音的干扰。

与经典 PCA 一样，Robust PCA（鲁棒主成分分析）本质上也是寻找数据在低维空间上的最佳投影问题。当观测数据较大时，PCA 无法给出理想的结果，而 Robust PCA 能够从较大的且稀疏噪声污染的观测数据中恢复出本质上低秩的数据。Robust PCA 考虑的是这样一个问题：一般的数据矩阵 D 包含结构信息，也包含噪声。那么可以将这个矩阵分解为两个矩阵相加：D=A+E，A 是低秩的（由于内部有一定的结构信息造成各行或列间是线性相关的），E 是稀疏的（含有噪声则是稀疏的），则 Robust PCA 可以写成以下的优化问题：

$$\min_{A,E} \|A\|_* + \lambda \|E\|_{1,1} \quad \text{s.t.}\, A + E = D \qquad (1)$$

其中，(1, 1)范数为$||A||_{1,1} = \sum_{i=1}^{m}\sum_{j=1}^{m}|a_{ij}|$。

对于某类观测的视频图像，将每一帧图像表示为 m 维矢量 V_i，若该视频共包含 n 帧图像序列，那么该观测视频就可以用 n 个矢量组成的数据矩阵 $D=[v_1,v_2,...,v_n] \in R^{m*n}(i=1,2,...,n)$来表示。视频背景建模的 Robust PCA 可描述为：对数据矩阵 D 进行分解，恢复出具有极大相似性的背景部分（低秩矩阵 A）和分布范围很小的运动目标或前景部分（稀疏矩阵 E）。这样就形成了(1)所示的凸优化问题，计算出 A 和 E 就可以分离出背景和前景（运动目标）了。

算法代码如下：

```python
def PCA_IALM(X, lam=0.01, tol=1e-7, maxIter=1000):
    Y = X
    norm_two = norm(Y.ravel(), 2)
    norm_inf = norm(Y.ravel(), np.inf) / lam
    dual_norm = np.max([norm_two, norm_inf])
    Y = Y / dual_norm
    A = np.zeros(Y.shape)
    dnorm = norm(X, 'fro')
    mu = 1.25 / norm_two
    rho = 1.5
    sv = 10.
    n = Y.shape[1]
    itr = 0
    while True:
        Eraw = X - A + (1 / mu) * Y
        Eupdate = np.maximum(Eraw - lam / mu, 0) + np.minimum(Eraw + lam / mu, 0)
        U, S, V = svd(X - Eupdate + (1 / mu) * Y, full_matrices=False)
        svp = (S > 1 / mu).shape[0]
        if svp < sv:
            sv = np.min([svp + 1, n])
        else:
            sv = np.min([svp + round(0.05 * n), n])

        Aupdate = np.dot(np.dot(U[:, :svp], np.diag(S[:svp] - 1 / mu)), V[:svp, :])
        A = Aupdate
        E = Eupdate
        # print itr
        Z = X - A - E
        Y = Y + mu * Z
        mu = np.min([mu * rho, mu * 1e7])
        itr += 1
        if ((norm(Z, 'fro') / dnorm) < tol) or (itr >= maxIter):
            break
    print("iteration:%d" % (itr))
    return A, E
```

（2）实现视频前背景分离的结果

我们使用 ShoppingMall 数据集，含有 100 个商场监控的 BMP 位图。基于此，我们实现了前背景分离，展示部分结果如下（其中从左到右分别为 A、E 和 D）：