



最优化方法程序作业

姓　　名　　 江银

班　　级　　 计算机实验班01

学　　号　　 202103151211

提交日期　　 2023.5.30

目录

[一、程序运行环境 1](#_Toc136544545)

[二、调用库简介 1](#_Toc136544546)

[三、代码功能分析 1](#_Toc136544547)

[3.1、算法介绍与功能阐述 1](#_Toc136544548)

[3.2、主程序介绍 2](#_Toc136544549)

[3.3、LASSO问题的连续化策略 4](#_Toc136544550)

[3.4、LASSO问题的Huber光滑化梯度法 6](#_Toc136544551)

[四、实验结果分析 9](#_Toc136544552)

[五、附录 11](#_Toc136544553)

# 一、程序运行环境

电脑操作系统：Windows11

电脑配置参数：

处理器 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

显卡 NVIDIA GeForce RTX 3060 Laptop GPU

编程语言：Python3.10

# 二、调用库简介

用到的Python工具包：numpy库、scipy库、matplotlib库。

Numpy和Scipy用于矩阵乘法、求逆等一系列运算，matplotlab用于生成可视化的图表。

import numpy as np   
import matplotlib.pyplot as plt   
import scipy

# 三、代码功能分析

## 3.1、算法介绍与功能阐述

LASSO问题如下：

其中,且。

由于目标函数不光滑，在某些点处无法求出梯度，因此不能直接对原始问题使用梯度法求解。因此我们利用Huber光滑化方法，将不光滑项以光滑函数逼近ℓ1-范数，得到一个可微函数。然后我们利用梯度下降法，对原问题进行近似求解。

为了加快算法的收敛速度，我们采用连续化策略来调整正则化系数μ。

程序大致分为主程序部分，和LASSO\_grad\_huber()函数完成的Huber光滑化和梯度法部分，以及LASSO\_con()函数完成的连续化策略部分。

## 3.2、主程序介绍

主函数分为三个部分，分别为构建LASSO问题，求解LASSO问题，和结果可视化。

构建LASSO问题即对问题进行初始化。

seed = 114514  
np.random.seed(seed)

该代码用于初始化随机种子。

m = 512  
n = 1024  
r = 0.1  
A = np.random.randn(m, n)  
u = scipy.sparse.random(n, 1, density=r, data\_rvs=scipy.randn)  
b = A \* u  
mu = 1e-3

生成随机矩阵A和向量b。A为m行n列的矩阵，u为n行的列向量。这里定义实数r=0.1，作为向量u的稀疏度。令b=Au。此外，首先给定正则化系数μ=1e-3。

Lambda = np.linalg.eig(np.matmul(A.T, A))  
index = np.argmax(Lambda[0])  
L = np.real(Lambda[0][index])  
x0 = np.random.randn(n, 1)

求解的最大特征值L，用于步长的选取。

求解LASSO问题的过程如下。

class opts:  
 def \_\_init\_\_(self):   
 opts.maxit = 4000  
 opts.ftol = 1e-8  
 opts.alpha0 = 1 / L

定义类opts，在类中定义一些属性。maxit表示最大迭代次数，ftol表示相邻两次迭代函数值所允许的差值，alpha0表示步长的初始值。

根据梯度法在凸函数上的收敛性定理，固定步长需不超过才能保证算法收敛。因此步长的初始值设置为。

x, out = LASSO\_con(x0, A, b, mu, opts)  
f\_star = min(out.fvec)   
opts.maxit = 4000  
x, out = LASSO\_con(x0, A, b, mu, opts)  
data1 = (out.fvec - f\_star) / f\_star  
k1 = min(len(data1), 4000)  
data1 = data1[1:k1]

将收敛时得到的函数值作为真实的最优值的参考f\*。在μ=1e-3的情况下，运行Huber光滑化的梯度法，并将迭代得到的信息储存在data1中。

mu = 1e-2  
opts.ftol = 1e-8  
opts.alpha0 = 1 / L  
[x, out] = LASSO\_con(x0, A, b, mu, opts)  
f\_star = min(out.fvec)

x, out = LASSO\_con(x0, A, b, mu, opts)  
data2 = (out.fvec - f\_star) / f\_star  
k2 = min(len(data2), 4000)  
data2 = data2[1:k2]

这里将μ改为1e-2，再进行一次Huber光滑化的梯度法。

求解LASSO问题得到随着迭代次数变化的目标函数值后，需要将它们进行可视化。这里采用python的matplotlib库进行可视化。

fig1 = plt.figure()  
plt.semilogy(range(k1 - 1), data1, '-', color=[0.2, 0.1, 0.99], linewidth=2)  
plt.semilogy(range(k2 - 1), data2, '-.', color=[0.99, 0.1, 0.2], linewidth=1.5)  
plt.legend(['\u03BC = 10^-3', '\u03BC = 10^-2'])  
plt.ylabel('$(f(x^k) - f^\*)/f^\*$', fontsize=14)  
plt.xlabel('Iterations')  
plt.savefig('1.png')  
fig2 = plt.figure()  
u = u.todense()  
plt.subplot(2, 1, 1)  
plt.plot(u, color=[0.2, 0.1, 0.99], marker='x', linestyle='none')  
plt.xlim([1, 1024])  
plt.title('Exact Solution')  
plt.subplot(2, 1, 2)  
plt.plot(x, color=[0.2, 0.1, 0.99], marker='x', linestyle='none')  
plt.xlim([1, 1024])  
plt.title('Gradient Descent Solution')  
plt.savefig('2.png')  
plt.show()

如此就得到了LASSO问题的结果并进行了可视化，具体算法实现如下。

## 3.3、LASSO问题的连续化策略

连续化策略即正则化参数μ的运算策略。我们选择从较大的正则化参数逐渐减小到，并求解相应的LASSO问题。这样可以使得求解对应的优化问题时，可以利用对应的解作为逼近解加速求解过程。

接下来研究其与BP问题罚函数的关系。对于BP问题，

利用二次罚函数法，令为罚因子，则有

令，可以看出罚函数法的增大罚因子与之的相关性。

由于越大，对应的LASSO问题越容易解得答案，因此，连续化策略起到了加速运算的作用。

此外在调用迭代算法求解时，，η为0到1的实数，代表缩小因子。

def LASSO\_con(x0, A, b, mu0, opts):  
 opts.maxit = 30  
 opts.maxit\_inn = 200  
 opts.ftol = 1e-8  
 opts.gtol = 1e-6  
 opts.factor = 0.1  
 opts.mu1 = 100  
 opts.gtol\_init\_ratio = 1 / opts.gtol  
 opts.ftol\_init\_ratio = 1e5  
 opts.etaf = 1e-1  
 opts.etag = 1e-1  
  
 out = outs()  
 out.fvec = []  
  
 k = 0  
 x = x0  
 mu\_t = opts.mu1  
 f = func(A, b, mu\_t, x)  
 opts1 = opts()  
  
 opts1.ftol = opts.ftol \* opts.ftol\_init\_ratio  
 opts1.gtol = opts.gtol \* opts.gtol\_init\_ratio  
 out.itr\_inn = 0

迭代前需要将变量初始化，迭代过程中需要将信息存放在out类中。Out.fvec提供了每一步迭代的函数值，out.itr\_inn记录了内层迭代次数，out.fval记录了迭代终止时的目标函数值，out.itr则记录了外层的迭代次数。

Opts.maxit记录最大外层迭代次数，opts.maxit\_inn记录最大内层迭代次数，opts.ftol记录函数值判断结束的差值条件，opts.gtol记录梯度判断结束的差值条件，opts.factor记录正则化系数的衰减率，opts.mul是初始的正则化系数，opt.alpha0是初始步长，opts.ftol\_init\_ratio是初始时opts.ftol的放大倍数，opts.gtol\_init\_ratio则是初始时opts.gtol的放大倍数，opts.etaf时每步外层循环opts.ftol的缩减，opts.etag同理。Opts.opts1则是用来传参的类，用于向内层算法提供具体参数。

接下来开始循环迭代。

while k < opts.maxit:  
 opts1.maxit = opts.maxit\_inn  
 opts1.gtol = max(opts1.gtol \* opts.etag, opts.gtol)  
 opts1.ftol = max(opts1.ftol \* opts.etaf, opts.ftol)  
 opts1.alpha0 = opts.alpha0  
 opts1.sigma = 1e-3 \* mu\_t  
 fp = f  
 x, out1 = LASSO\_grad\_huber(x, A, b, mu\_t, mu0, opts1)  
 f = out1.fvec[-1]  
 out.fvec = np.append(out.fvec, out1.fvec)  
 k = k + 1  
 nrmG = np.linalg.norm(x - prox(x - np.matmul(A.T, (np.matmul(A, x) - b)), mu0), 2)  
 if ~out1.flag:  
 mu\_t = max(mu\_t \* opts.factor, mu0)  
 if mu\_t == mu0 and (nrmG < opts.gtol or abs(f - fp) < opts.ftol):  
 break  
 out.itr\_inn = out.itr\_inn + out1.itr  
out.fval = f  
out.itr = k  
return x, out

opts1对象的参数定义已在前面陈述。调用内层循环函数，out.fvec记录每一步x对应的原始函数值。

当内层循环因达到收敛条件而退出时，缩减当前正则化系数，并判断收敛。退出后更新总迭代次数。退出外层循环后，记录当前函数值和外层迭代次数。

def func(A, b, mu0, x):  
 w = np.matmul(A, x) - b  
 f = 0.5 \* np.matmul(w.T, w) + mu0 \* np.linalg.norm(x, 1)  
 return f  
  
  
def prox(x, mu):  
 y = np.max(np.abs(x) - mu, 0)  
 y = np.sign(x) \* y  
 return y

这两个函数分别用来计算LASSO问题的目标函数和的临近算子。

## 3.4、LASSO问题的Huber光滑化梯度法

在连续化策略函数迭代的内层循环中，调用了Huber光滑化梯度法的代码，具体流程如下。

def LASSO\_grad\_huber(x, A, b, mu, mu0, opts):  
 opts.maxit = 200  
 opts.sigma = 0.1  
 opts.alpha0 = 0.01  
 opts.gtol = 1e-6  
 opts.ftol = 1e-8  
 r = np.matmul(A, x) - b  
 g = np.matmul(A.T, r)  
  
 huber\_g = np.sign(x)  
 idx = abs(x) < opts.sigma  
 huber\_g[idx] = x[idx] / opts.sigma  
  
 g = g + mu \* huber\_g  
 nrmG = np.linalg.norm(g, 2)  
 f = .5 \* np.linalg.norm(r, 2) \*\* 2 + mu \* (  
 np.sum(x[idx] \*\* 2 / (2 \* opts.sigma)) + sum(abs(x[abs(x) >= opts.sigma]) - opts.sigma / 2))

out.flag标记是否收敛。huber\_g 通过二次函数近似，求近似后的梯度来实现梯度法，进而得到光滑化函数f的梯度。

out = outs()  
out.fvec = .5 \* np.linalg.norm(r, 2) \*\* 2 + mu0 \* np.linalg.norm(x, 1)  
  
alpha = opts.alpha0  
eta = 0.2  
rhols = 1e-6  
gamma = 0.85  
Q = 1  
Cval = f

初始化初始步长、步长衰减率以及线搜索参数之后，进行迭代主循环。

for k in range(1, opts.maxit + 1):  
 fp = f  
 gp = g  
 xp = x  
 nls = 1  
 while True:  
 x = xp - alpha \* gp  
 r = np.matmul(A, x) - b  
 g = np.matmul(A.T, r)  
 huber\_g = np.sign(x)  
 idx = abs(x) < opts.sigma  
 huber\_g[idx] = x[idx] / opts.sigma  
 f = .5 \* np.linalg.norm(r, 2) \*\* 2 + mu \* (  
 np.sum(x[idx] \*\* 2 / (2 \* opts.sigma)) + sum(abs(x[abs(x) >= opts.sigma]) - opts.sigma / 2))  
 g = g + mu \* huber\_g  
 if f <= Cval - alpha \* rhols \* nrmG \*\* 2 or nls >= 10:  
 break  
 alpha = eta \* alpha  
 nls = nls + 1  
 nrmG = np.linalg.norm(g, 2)  
 forg = .5 \* np.linalg.norm(r, 2) \*\* 2 + mu0 \* np.linalg.norm(x, 1)  
 out.fvec = np.append(out.fvec, forg)  
 if nrmG < opts.gtol or abs(fp - f) < opts.ftol:  
 break  
 dx = x - xp  
 dg = g - gp  
 dxg = abs(np.matmul(dx.T, dg))  
 if dxg > 0:  
 if k % 2 == 0:  
 alpha = np.matmul(dx.T, dx) / dxg  
 else:  
 alpha = dxg / (np.matmul(dg.T, dg))  
 alpha = max(min(alpha, 1e12), 1e-12)  
 Qp = Q  
 Q = gamma \* Qp + 1  
 Cval = (gamma \* Qp \* Cval + f) / Q

对于连续化策略下的每一个正则化系数μ，使用梯度下降法求解f的最小值。使用线搜索选取合适步长更新迭代点。

满足线搜索准则或超过10次步长衰减后退出线搜索。线搜索结束后得到更新的x和g，计算梯度范数和原始目标函数值,fvec记录每一步的目标函数值，并进行内层循环的收敛判断。

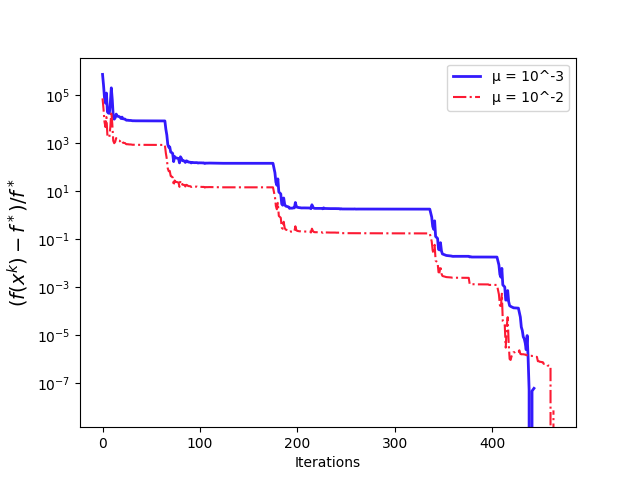
采用BB步长作为下一步迭代的初始步长，令，，偶数步对应BB步长，奇数步对应BB步长。

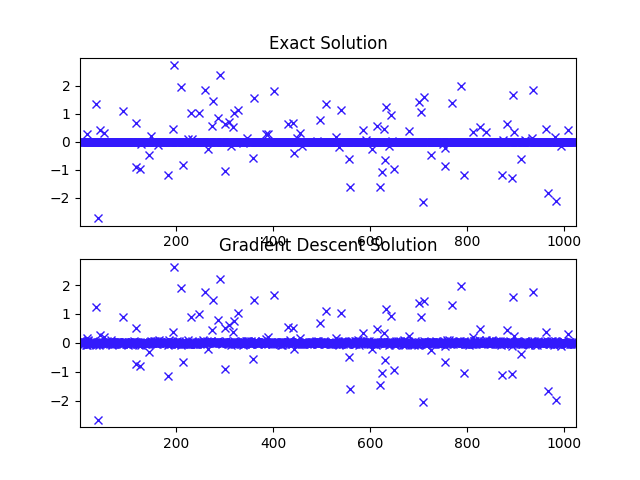
Q则是用于计算线搜索准则中的递推常数。递推常数满足，。序列Q满足，。

if k == opts.maxit:  
 out.flag = 1  
else:  
 out.flag = 0  
out.fval = f  
out.itr = k  
out.nrmG = nrmG  
return x, out

若因超过最大迭代次数而退出，out.flag标记为1，否则标记为0，用来判断是否进行正则化系数的衰减。

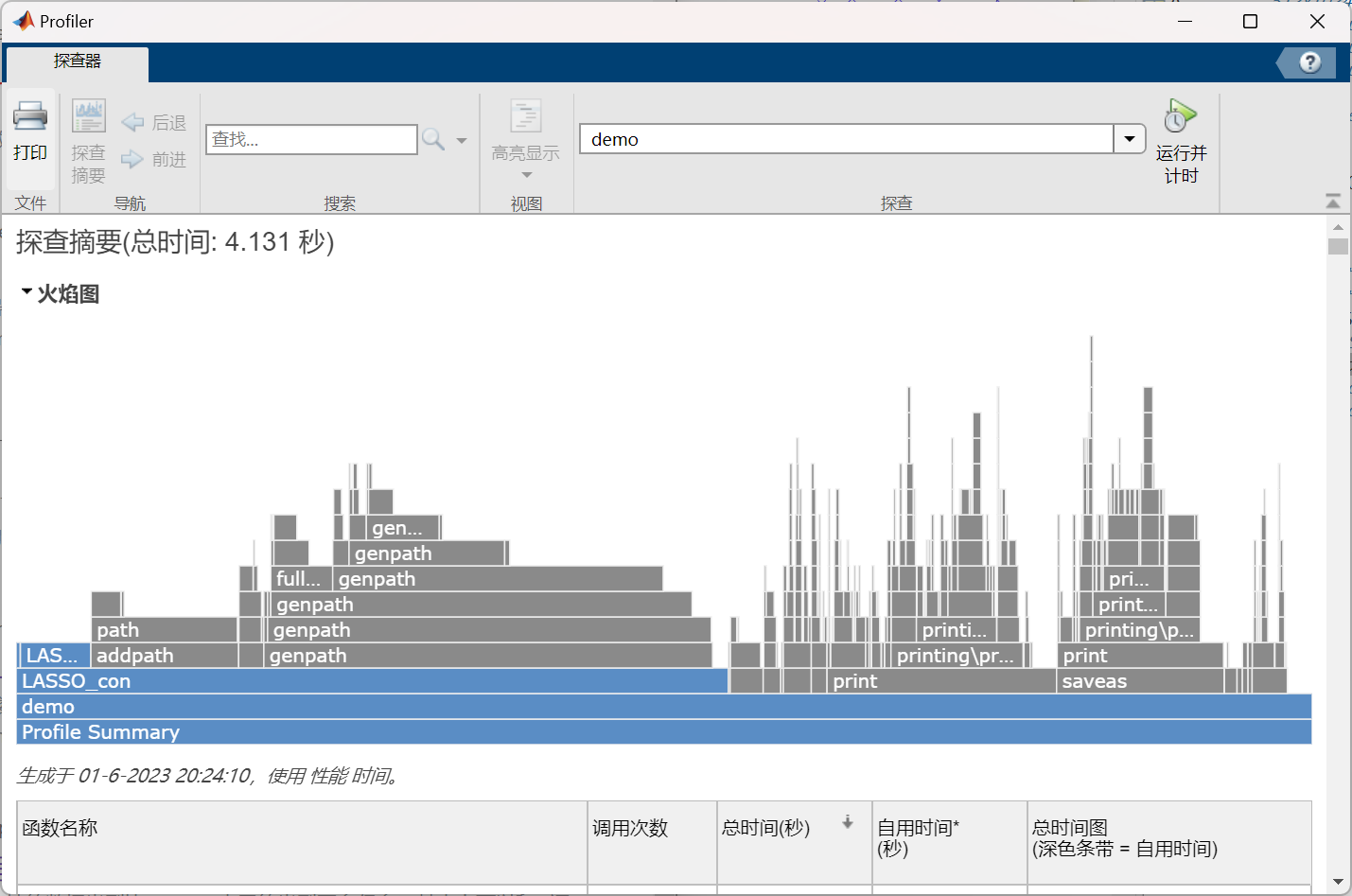
# 四、实验结果分析





我们通过连续化策略加速了求解过程，在450步左右收敛到了最小值。在μ分别为1e-2和1e-3时的迭代过程图像如图。

根据梯度法得到的解的分量大小如图，和精确解相近，大部分分量集中在0附近。因此可以认为算法得到了稀疏性良好的解。



编写MATLAB代码后，在MATLAB中使用profile观察代码的用时情况。可以发现在主程序中，LASSO的连续化策略部分花了50%的时间，Huber光滑化的梯度法总共占用了5%左右的时间，因此认为算法效率较高。

# 五、附录

代码见文件