Optimization Theory and Algorithm II **Homework 4 - 09/11/2021**

Homework 4

Lecturer: Xiangyu Chang *Scribe: 吴博成*

Student Number: 2193211134

1 HW 1

1.1 问题重述

在 LASSO 问题中，我们要解决具有 l_1 范数约束的最小二乘问题。在本工作中，我们使用以下设置：

$$x^* = \arg\min_x \frac{1}{2}\|Ax - b\|^2 + \lambda\|x\|_1$$

在此问题中，A 和 B 设置为：

```
import numpy as np
np.random.seed(2021) # set a constant seed to get samerandom matrixs
A = np.random.rand(500, 100)
x_ = np.zeros([100, 1])
x_[:5, 0] += np.array([i+1 for i in range(5)]) # x_denotes expected x
b = np.matmul(A, x_) +np.random.randn(500, 1) *0.1 #add a noise to b
lam = 10 # try some different values in {0.1, 1, 10}
```

通过以下算法解决问题：

(1) Proximal Gradient Descent.

(2) *BCD*.

(3) ADMM.

1.2 问题求解

$$\min_x f(x) = \min_x(g(x) + h(x)) = \min_x \left(\left\{ \frac{1}{2}\|Ax - b\|^2 \right\} + \{\lambda\|x\|_1\} \right)$$

$$\nabla g(x) = A^\top(Ax - b)$$

$$\operatorname{prox}_{\lambda r\|\cdot\|}(z) = \arg\min \left(\frac{1}{2}\|x - z\|^2 + \lambda\|x\|_1 \right) = S_r(z) = \begin{cases} z - \lambda r & , z > \lambda r \\ 0 & , \quad |z| \leqslant \lambda r \\ z + \lambda r & , z < \lambda r \end{cases}$$

迭代步:

$$x^{k+1} = \operatorname{prox}_{s^{k+1}h}\left(x^k - s^{k+1}\nabla g\left(x^k\right)\right)$$

自定义函数准备：

```
# 软阈值函数 s_t(x)
def Soft_Thresholding(r,x):
    result = np.zeros([len(x), 1])
    for i in range(len(x)):
        if x[i]>r:
            result[i] = x[i]-r
        elif -r<=x[i]<=r:
            result[i] = 0
        elif x[i]<-r:
            result[i] = x[i]+r
        else:
            np.ERR_WARN("输入变量有非数值变量")
    return result
def target_function(x):
    return 1/2*(np.linalg.norm(A@x - b, ord=2) ** 2) + lam*np.linalg.norm(x,ord=1)
import matplotlib.pyplot as plt
def make_plot(result_matrix,colnames,lam,name):
    plt.figure(num=1, figsize=(16, 7))
    x = [0.]+list(np.log(result_matrix[1:,0])) # 对x轴进行log采样
    for i in range(1,6): #令i依次为x_1,x_2,....,x_5的列标
        y = result_matrix[:,i]
        y_lable = colnames[i]
        plt.subplot(121)
        plt.plot(x,y,label = y_lable, linewidth = 0.5,color = "blue")
    plt.title("Converage of X\nlambda = {}".format(lam))
    plt.legend()
    plt.subplot(122)
    plt.plot(x,result_matrix[:,-1],label = "Target Function",linewidth = 2, color = "red")
    plt.xlabel("Log Iteration Times")
    plt.title("Converage of Target function\nlambda = {}".format(lam))
    plt.savefig(name + " Converage of lambda {}.png".format(lam),dpi = 500)
    plt.show()
```

1.2.1 Proximal Gradient Descent

```
# 近端梯度下降法 min f(x) = min g(x)+ h(x)，其中h不可微
def Proximal_GD_for_Lasso(A,b,eps,step,lam):
    t = 0 #计数器
    s = step
    x = np.zeros([A.shape[1], 1])# 初始值全0矩阵
    err = np.inf
    result_matrix = np.c_[t,x.T,target_function(x)]
```

```
    while (err > eps and t < 1e7):
        origin_x = x
        dgx = A.T@(A@x-b)
        x = Soft_Thresholding(s*lam,x-s*dgx)
        fx = target_function(x)
        err = np.linalg.norm(origin_x - x,ord=1)
        t += 1
        result_matrix = np.r_[result_matrix,np.c_[t,x.T,fx]] #结果存入矩阵方便画图
        #print(np.count_nonzero(x)) #调试用代码
        #print (1/2*(np.linalg.norm(A@x - b, ord=2) ** 2))
    print("*"*100 + "\nlambda为：{lam}\n迭代次数为：{t}\n目标函数最优值为：{fx} \n最优解中非零元素的个数：{x_is0}\n最优解为：\n{x}\n".format(lam = lam,t = t, fx = fx,x_is0 = np.count_nonzero(x),x = list(x.T)))
    return result_matrix
colnames = ["iteration "] + ["x_{}".format(i) for i in range(1,A.shape[1]+1)] + ["target_function"] #创建result_matrix的列名列表，形如：["iteration","x_1","x_2",...,"x_100","target_function"]
result1 = Proximal_GD_for_Lasso(A,b,eps = 1e-10,step = 1e-4,lam = 0.1)
make_plot(result1,colnames,0.1,"PGD")
result2 = Proximal_GD_for_Lasso(A,b,eps = 1e-10,step = 1e-4,lam = 1)
make_plot(result2,colnames,1,"PGD")
result3 = Proximal_GD_for_Lasso(A,b,eps = 1e-10,step = 1e-4,lam = 10)
make_plot(result3,colnames,10,"PGD")
```

****

lambda 为：0.1

迭代次数为：11583

目标函数最优值为：161.76162041300677

最优解中非零元素的个数：83

最优解为: [1.02699279e+00, 1.97548064e+00, 3.00898077e+00, 3.96911781e+00, 5.00888533e+00, 6.14080817e-03, 8.78371697e-03, 0.00000000e+00,

....

-1.59887852e-02, 0.00000000e+00, -2.61074396e-03, 0.00000000e+00]

****

lambda 为：1

迭代次数为：7072

目标函数最优值为：152.75977442209626

最优解中非零元素的个数：22

最优解为：[1.00528902e+00, 1.96140737e+00, 2.98836366e+00, 3.95517755e+00, 4.98537467e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,

....

0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00]

****

lambda 为：10

迭代次数为：5836

目标函数最优值为：151.93227105788

最优解中非零元素的个数：11

最优解为：[9.91578620e-01, 1.96261371e+00, 2.97499677e+00, 3.95205915e+00, 4.97976441e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,

....

0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00]

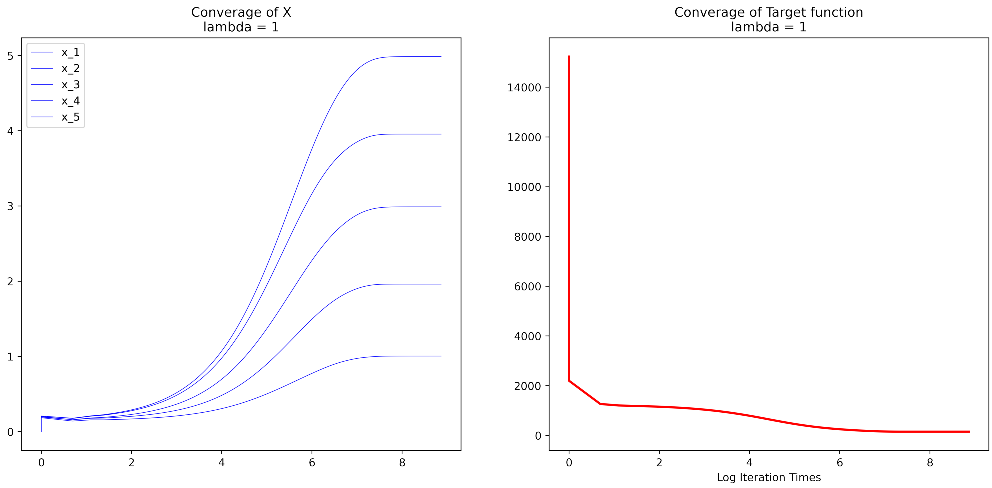

Figure 1: Converage of LASSO with PGD

1.2.2 B C D

Algorithm 1 Block Coordinate Descent

1: **Input:** Given a initial starting point $\mathbf{x}^0 = (\mathbf{x}_1^0, \dots, \mathbf{x}_K^0) \in \mathbb{R}^n$, and $t = 0$

2: **for** $t = 0, 1, \dots, T$ **do**

3: **for** $k = 0, 1, \dots, K$ **do**

4: $x_k^t = \dfrac{A_i(b - \hat{A}\tilde{x}) - \lambda \operatorname{sign}(x)}{\|A_i\|^2} = \begin{cases} \dfrac{A_i(b - \hat{A}\hat{x}) - \lambda}{\|A_i\|^2}, & A_i(b - \hat{A}\hat{x}) > \lambda \\ \dfrac{A_i(b - \hat{A}\hat{x}) + \lambda}{\|A_i\|^2}, & A_i(b - \hat{A}\tilde{x}) < -\lambda \\ 0, & \text{otherwise} \end{cases}$

5: **end for**

6: **end for**

7: **Output:** $\mathbf{x}^T$.

Figure 2: BCD Algorithm

```
# BCD方法
def BCD_for_Lasso(A,b,eps,step,lam):
    t = 0 #计数器
    s = step
    x = np.zeros([A.shape[1], 1])# 初始值全0矩阵
    err = np.inf
    result_matrix = np.c_[t,x.T,target_function(x)]
    while (err > eps and t < 1e7):
        origin_x = np.array(x, copy=True)
        for k in range(A.shape[1]):
            Ak = A[:,k]
            A_tilde = np.delete(A, k, axis=1)
            x_tilde = np.mat(np.delete(x, k)).T
            x[k] = Soft_Thresholding(s*lam,Ak@(b − A_tilde@x_tilde))/(np.linalg.norm(Ak,ord=2)**2)
        fx = target_function(x)
        f_star = min(result_matrix[:,−1])
        err = abs(f_star−fx)
        t += 1
        result_matrix = np.r_[result_matrix,np.c_[t,x.T,fx]] #结果存入矩阵方便画图
        # print (fx)
    print("*"*100 + "\nlambda为: {lam}\n迭代次数为: {t}\n目标函数最优值为: {fx} \n最优解为: \n{x}\n".
        format(lam = lam,t = t, fx = fx,x = list(x.T)))
    return result_matrix
colnames = ["iteration"] + ["x_{}".format(i) for i in range(1,A.shape[1]+1)] + ["target_function"] #创建
    result_matrix的列名列表, 形如: ["iteration","x_1","x_2",...,"x_100","target_function"]
result4 = BCD_for_Lasso(A,b,eps = 1e−3,step = 1e−4,lam = 0.1)
make_plot(result4,colnames,0.1,"BCD")
```

```
result5 = BCD_for_Lasso(A,b,eps = 1e−3,step = 1e−4,lam = 1)
make_plot(result5,colnames,1,"BCD")
result6 = BCD_for_Lasso(A,b,eps = 1e−3,step = 1e−4,lam = 10)
make_plot(result6,colnames,10,"BCD")
```

****

lambda 为：0.1

迭代次数为：677

目标函数最优值为：169.3241335658165

最优解为：[1.01497573e+00, 1.95699565e+00, 3.00454193e+00, 3.96483701e+00, 4.99779927e+00, -2.24232158e-03, 4.00167743e-04, -1.60964557e-02,

....

-4.49720292e-02, -2.51318399e-02, -3.42465099e-02, -1.42105193e-02]

****

lambda 为：1

迭代次数为：677

目标函数最优值为：169.32281176358563

最优解为：[[1.01496582e+00, 1.95698009e+00, 3.00452326e+00, 3.96483203e+00, 4.99778881e+00, -2.25093538e-03, 3.86068206e-04, -1.61020137e-02,

....

-4.49661376e-02, -2.51380666e-02, -3.42498318e-02, -1.42230067e-02]]

****

lambda 为：10

迭代次数为：565

目标函数最优值为：171.67190606638644

最优解为：[1.00476658e+00, 1.98489239e+00, 3.02427713e+00, 3.96679065e+00, 4.98465856e+00, 7.47658134e-03, -5.55173047e-03, -1.53355773e-02,

....

-2.98916088e-02, -7.64854605e-03, -1.95648910e-02, -9.59369455e-03]

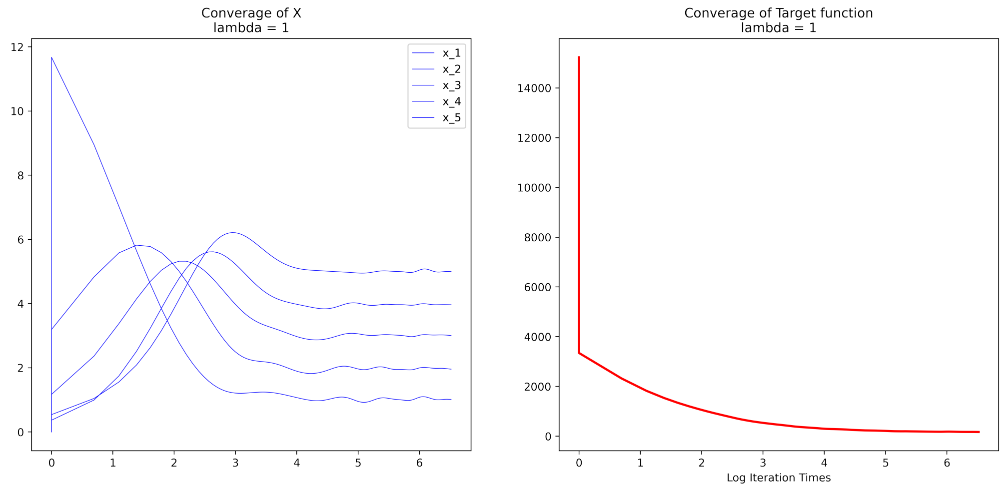

Figure 3: Converage of LASSO with BCD

1.2.3 ADMM

由 ADMM 算法可知：

$$
\begin{aligned}
\mathbf{x}^{t+1} &= \arg\min_{\mathbf{x}} \left\{ \frac{1}{2}\|A\mathbf{x}-\mathbf{b}\|^2 + \frac{\rho}{2}\left\|\mathbf{x}-\mathbf{z}^t+\mathbf{u}^t\right\|^2 \right\} \\
&= \left(A^\top A+\rho I\right)^{-1}\left(A^\top \mathbf{b}+\rho\left(\mathbf{z}^t-\mathbf{u}^t\right)\right) \\
\mathbf{z}^{t+1} &= \arg\min_{\mathbf{z}} \left\{ \lambda\|\mathbf{z}\|_1 + \frac{\rho}{2}\left\|\mathbf{x}^{t+1}-\mathbf{z}-\mathbf{b}+\mathbf{u}^t\right\|^2 \right\} \\
&= S_{\lambda/\rho}\left(\mathbf{x}^{t+1}+\mathbf{u}^t\right)
\end{aligned}
$$

其中 $S_{\lambda/\rho}$ 是软阈值函数。对于 $\mathbf{u}^{t+1}$，

$$
\mathbf{u}^{t+1} = \mathbf{u}^t + \mathbf{x}^{t+1} - \mathbf{z}^{t+1}
$$

```
def ADMM_for_Lasso(A,b,eps,rou,lam):
    t = 0 #计数器
    s = rou
    n = A.shape[1] # dim
    x = np.zeros([n, 1])# 初始值全0矩阵
    z = np.zeros([n, 1])
    u = np.zeros([n, 1])
    err = np.inf
    result_matrix = np.c_[t,x.T,target_function(x)]
    while (err > eps and t < 1e7):
        x = np.mat((A.T@A+s*np.identity(n))).I@(A.T@b+s*(z-u))
        z = Soft_Thresholding(lam/s,x+u)
        u = u+x-z
        f_star = min(result_matrix[:,-1])
```

```python
            fx = target_function(x)
            err = abs(f_star-fx)
            t += 1
            result_matrix = np.r_[result_matrix,np.c_[t,x.T,fx]] #结果存入矩阵方便画图
            # print (fx)
    print("*"*100 + "\nlambda为：{lam}\n迭代次数为：{t}\n目标函数最优值为：{fx} \n最优解为：\n{x}\n".
        format(lam = lam,t = t, fx = fx,x = list(x.T)))
    return result_matrix
colnames = ["iteration"] + ["x_{}".format(i) for i in range(1,A.shape[1]+1)] + ["target_function"] #创建
    result_matrix的列名列表，形如：["iteration","x_1","x_2",...,"x_100","target_function"]
result4 = ADMM_for_Lasso(A,b,eps = 1e-4,rou = 1e-4,lam = 0.1)
make_plot(result4,colnames,0.1,"ADMM")
result5 = ADMM_for_Lasso(A,b,eps = 1e-4,rou = 1e-4,lam = 1)
make_plot(result5,colnames,1,"ADMM")
result6 = ADMM_for_Lasso(A,b,eps = 1e-4,rou = 1e-4,lam = 10)
make_plot(result6,colnames,10,"ADMM")
```

****

lambda 为：0.1

迭代次数为：334

目标函数最优值为：164.79324365355603

最优解为：[1.02970622e+00, 1.97642761e+00, 3.01118604e+00,

3.96765916e+00, 5.00800434e+00, 1.06087319e-02,

....

-1.44814664e-03, -6.31966242e-03, 1.17685534e-03]

****

lambda 为：1

迭代次数为：3337

目标函数最优值为：163.8387330692975

最优解为：[1.02158966e+00, 1.95916135e+00, 2.98374476e+00,

3.94307433e+00, 4.97981373e+00, 1.27413185e-02,

....

1.46256456e-03, -8.81095715e-03, 2.51487330e-03]

****

lambda 为：10

迭代次数为：4590

目标函数最优值为：163.6493323136842

最优解为：[1.01813511e+00, 1.95179719e+00, 2.97173512e+00,

3.92133246e+00, 4.94000000e+00, 1.53538226e-02,

....

1.97849963e-03, -1.32613149e-02, 3.30760049e-03]

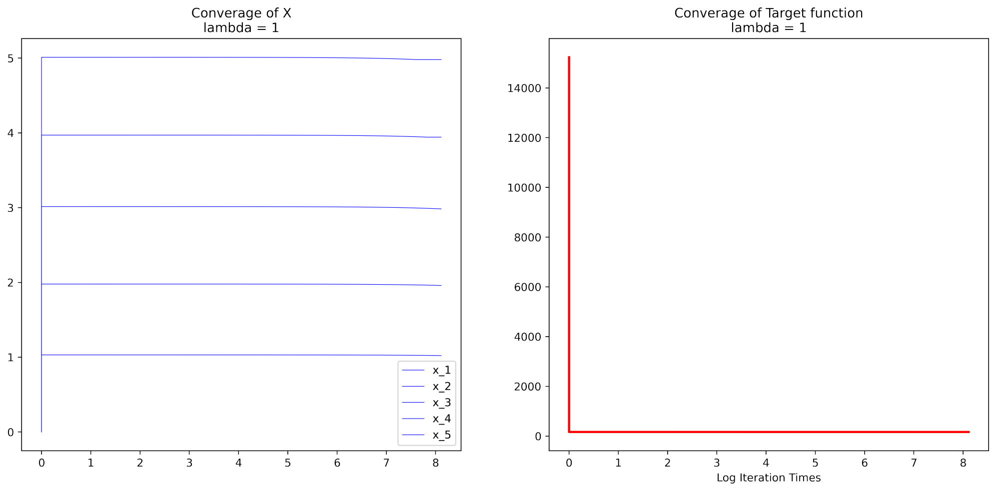

Figure 4: Converage of LASSO with ADMM

1.2.4 结论

对 LASSO 问题而言:ADMM 速度 >BCD 速度 >PGD 速度

随着 λ 逐渐增大，LASSO 最优解中零解的个数会增加

2 HW 2

2.1 问题重述

阅读教科书 P470，并使用数据集 a9a 实现 Logistic 回归：

(1) 固定学习率的 SGD

(2) 学习速率递减的 SGD

(3)SVRG

2.2 问题求解

2.2.1 数据预处理

原始数据集每条数据有 14 个特征，分别为 age,workclass,fnlwgt(final weight),education,education-num,marital-status,occupation,relationship,race,sex,captital-gain,captital-loss,hours-per-week 和 native-country。其中有 6 个特征是连续值，包括 age,fnlwgt.education-num,captital-gain,captital-loss,hours-per-week; 其它 8 个特征是离散的。本数据首先要做的处理是：将连续特征离散化，将有 M 个类别的离散特征转换为 M 个二进制特征。

本数据集共有 48842 条数据，每条数据从原始特征的 14 个转换成 123 个，并以 2：1 的比例分为训练集和测试集，其中 a9a 为训练集，用来训练分类器模型；a9a-t 是测试集，用来预测模型的分类效果。它共有两个类别，标签分别用-1 和 1 表示，标签的含义是一个人一年的薪资是否超过 50K，1 表示超过 50K，-1 表示不超过 50K。

变换后的数据下载地址：https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.htmla9a

每个特征转换方式如下：

（1）age：连续值，拓展为 5 位，即第 1-5 维，采用 one-hot 方式，划分标准如下

1.age<=25, 第 1 维为 1；

2.26<=age<=32, 第 2 维为 1；

3.33<=age<=40, 第 3 维为 1；

4.41<=age<=49, 第 4 维为 1；

5.age>=50, 第 5 维为 1；

（2）workclass：离散值，取值为 Private,Self-emp-not-inc,Self-emp-inc,Federal-gov,Local-gov,State-gov,Without-pay,Never-worked, 共 8 个取值，扩展为 8 位，即 6-13 维

（3）fnlwgt：连续值，扩展为 5 位，即 14-18 维，划分标准如下

1.fnlwgt<=110000, 第 14 维为 1；

2.110000<=fnlwgt<=159999, 第 15 维为 1；

2.160000<=fnlwgt<=196335, 第 16 维为 1；

2.196336<=fnlwgt<=259865, 第 17 维为 1；

2.fnlwgt>=259866, 第 18 维为 1；

（4）education：离散值，取值有：Bachelors，Some-college，11th，HS-grad，Prof-school，Assoc-acdm，Assoc-voc，9th，7th-8th，12th，Masters，1st-4th，10th，5-6th，Preschool 共 16 个，扩展为 16 位，即 19-34 维。

（5）education-num：连续值，扩展为 5 位，即 35-39 维，划分标准如下

1.11th，9th，7-8th，12th，1st-4th，10th，5th-6th，Preschool：第 35 维为 1；

2.HS-grad：第 36 维为 1；

3.Some-college：第 37 维为 1；

4.Assoc-acdm，Assoc-voc：第 38 维为 1；

5.Bachelors，Prof-school，Masters，Doctorate：第 39 维为 1。

（6）marital-status：离散值，取值有：Married-civ-spouse，Divorced，Never-married，Separated，Wideowed，Married-spouse-absent，Married-AF-spouse，扩展为 7 位，即 40-46 维。

（7）occupation：离散值，取值有：Tech-support，Craft-repair，Other-service，Sales，Exec-managerial，Prof-specialty，Handlers-cleaners，Machine-op-inspct，Adm-clerical，Farming-fishing，Transport-moving，Priv-house-serv，Protective-serv，Armed-Forces 共 14 个，扩展为 14 位，即 47-60 维。

（8）relationship：离散值，取值为 Wife，Own-Child，Husband，Not-in-family，Other-relative，Unmarrie 共 6 个，扩展为 6 位，即 61-66 维。

（9）race：离散值，取值有：White，Asian-Pac-Islander，Amer-Indian-Eskimo，Other，Black 共 5 个，扩展为 5 位，即 67-71 维。

（10）sex：离散值，取值有 Female，Male 共 2 个，扩展为 2 位，即 72-73 维。

（11）captital-gain：连续值，扩展为 2 位，即 74-75 维，划分标准如下

1.captital-gain=0：第 74 维为 1；

2.captital-gain 0：第 75 维为 1.

（12）captital-loss：连续值，扩展为两位，即 76-77 维，划分标准如下

1.captital-loss=0：第 76 维为 1；

2.captital-loss 0：第 77 维为 1

（13）hours-per-week：连续值，扩展为 5 位，即 78-82 维，划分标准如下

1.hours-per-week<=34：第 78 维为 1；

2.35<=hours-per-week<=39：第 79 维为 1；

3.hours-per-week=40：第 80 维为 1；

4.41<=hours-per-week<=47：第 81 维为 1；

5.hours-per-week>=48：第 82 维为 1；

（14）native-country：离散值，取值有：United-States，Cambodia，England，Puerto-Rico，Canada，Germany，Outlying-US(Guam-USVI-etc)，India，Japan，Greece，South，China，Cuba，Iran，Honduras，Philippines，Italy，Poland，Jamaica，Vietnam，Mexico，Portugal，Ireland，France，Dominican-Republic，Laos，Ecuador，Taiwan，Haiti，Columbia，Hungary，Guatemala，Nicaragua，Scotland，Thailand，Yugoslavia，EI-Salvador，TrinidadTobago，Peru，Hong，Holand-Netherlands 共 41 个，扩展为 41 位，即 83-123 维。

```python
import pandas as pd
import numpy as np
np.random.seed(2021)
#########数据预处理############
def make_data(dataset): #将数据处理成123维和
    m = dataset.shape[0]
```

```
    A = np.zeros([m,123])
    b = list(dataset [:,0]. T)
    for i in range(m):
        for dics in dataset[i ,1:]:
            if dics is not np.NaN:
                [n,value] = [int(dic) for dic in dics.split(":")] #将字符串分割，例如6:1表示下标为6的字符串的
                    值为1
                A[i,n−1] = value
    return (A,b)
data = pd.read_table("a9a", header=None, delimiter=" ").iloc[:,:−1].values
(A,b) = make_data(data)

##########逻辑回归损失函数#########
def loss_function(A,b,x,lam):
    m = A.shape[1]
    return np.average([np.log(1 + np.exp(−b[i]∗(A[i,:]@x))) for i in range(m)]) + lam∗np.linalg.norm(x,ord=2)∗∗2
def dfi(A,b,x,lam,i): #第i个分量梯度
    return 2∗lam∗x − np.mat(((np.exp(−b[i]∗(A[i,:]@x))∗b[i]∗A[i,:])/(1 + np.exp(−b[i]∗(A[i,:]@x))))).T
```

2.2.2 有固定步长的随机梯度下降法

```
def SGD_Fixed_Step(A,b,eps,step,lam):
    t = 0 #计数器
    s = step
    m = A.shape[0]
    x = np.zeros([A.shape[1], 1])# 初始值全0矩阵
    err = np.inf
    result_matrix = np.c_[t,x.T,loss_function(A,b,x,lam)]
    while (err > eps and t < 1e7):
        origin_x = x
        i = np.random.randint(0,m−1)
        x = x − s∗dfi(A,b,x,lam,i)
        fx = loss_function(A,b,x,lam)
        f_star = min(result_matrix[:,−1])
        err = abs(f_star−fx)
        t += 1
        result_matrix = np.r_[result_matrix,np.c_[t,x.T,fx]] #结果存入矩阵方便画图
        print (fx) #调试用代码
    f_star = min(result_matrix[:,−1])
    print("∗"∗100 + "\nlambda为：{lam}\n迭代次数为：{t}\n目标函数最优值为：{fx} \n最优解为：\n{x}\n".
        format(lam = lam,t = t, fx = f_star,x = list(x.T)))
    return result_matrix
colnames = ["iteration "] + ["x_{}".format(i) for i in range(1,A.shape[1]+1)] + ["target_function"] #创建
    result_matrix的列名列表，形如：["iteration","x_1","x_2",...,"x_100","target_function"]
```

```python
result4 = SGD_Fixed_Step(A,b,eps = 1e-6,step = 1e-2,lam = 1e-2/A.shape[0]) #lam = 1e-2/N
pd.DataFrame(columns=colnames,data=result4).to_csv('result4.csv')
```

****

lambda 为：3.0711587481956940e-07

迭代次数为：630

目标函数最优值为：0.44054572

最优解为：[[-2.30207672e-01, -1.35742131e-01, -5.56272147e-03,

1.33903253e-01, -4.38445961e-04, -1.50242037e-01,

....

0.00000000e+00, 0.00000000e+00, 0.00000000e+00]]

2.2.3 有固定步长的随机梯度下降法

```python
def SGD_Diminishing_Step(A,b,eps,step,lam):
    t = 0 #计数器
    s = step
    m = A.shape[0]
    x = np.zeros([A.shape[1], 1])# 初始值全0矩阵
    err = np.inf
    result_matrix = np.c_[t,x.T,loss_function(A,b,x,lam)]
    while (err > eps and t < 1e7):
        origin_x = x
        i = np.random.randint(0,m-1)
        s = s*0.995
        x = x - s*dfi(A,b,x,lam,i)
        fx = loss_function(A,b,x,lam)
        f_star = min(result_matrix[:,-1])
        err = np.linalg.norm(origin_x - x,ord=1)
        t += 1
        result_matrix = np.r_[result_matrix,np.c_[t,x.T,fx]] #结果存入矩阵方便画图
        print (fx) #调试用代码
    f_star = min(result_matrix[:,-1])
    print("*"*100 + "\nlambda为：{lam}\n迭代次数为：{t}\n目标函数最优值为：{fx} \n最优解为：\n{x}\n".format(lam = lam,t = t, fx = f_star,x = list(x.T)))
    return result_matrix
colnames = ["iteration"] + ["x_{}".format(i) for i in range(1,A.shape[1]+1)] + ["target_function"] #创建result_matrix的列名列表，形如：["iteration","x_1","x_2",...,"x_100","target_function"]
result5 = SGD_Diminishing_Step(A,b,eps = 1e-6,step = 1e-2,lam = 1e-2/A.shape[0]) #lam = 1e-2/N
pd.DataFrame(columns=colnames,data=result5).to_csv('result5.csv')
```

****
lambda 为：3.0711587481956946e-07
迭代次数为：2047
目标函数最优值为：0.50152046
最优解为：[[-9.55658017e-02, -4.53523955e-02, -1.85955433e-02,
-1.84636670e-02, 1.71802303e-02, -1.42950471e-01,
....
2.35564562e-03, 4.85605777e-05, 0.00000000e+00]]

2.2.4 有固定步长的随机梯度下降法

```python
def SVRG(A,b,eps,learning_rate,lam,T):
    s = 0 #计数器
    step = learning_rate
    m = A.shape[0]
    x_tilde = np.zeros([A.shape[1], 1])# 初始值全0矩阵
    err = np.inf
    result_matrix = np.c_[s,x_tilde.T,loss_function(A,b,x_tilde,lam)]
    while ((err > eps or err == 0) and s < 1e7): #防止因为t选择0导致err = 0，直接弹出循环
        origin_x = x_tilde
        z_tilde = np.average([dfi(A,b,x_tilde,lam,i) for i in range(m)])
        x = {0:x_tilde}
        for t in range(1,T+1): #进行T步迭代后计算一次全梯度
            i = np.random.randint(0,m-1)
            x[t] = x[t-1] - step*(dfi(A,b,x[t-1],lam,i) - dfi(A,b,x_tilde,lam,i) + z_tilde)
        t = np.random.randint(0,T-1)
        x_tilde = x[t]
        fx = loss_function(A,b,x_tilde,lam)
        f_star = min(result_matrix[:,-1])
        err = abs(f_star-fx)
        s += 1
        result_matrix = np.r_[result_matrix,np.c_[s,x_tilde.T,fx]] #结果存入矩阵方便画图
        print (fx) #调试用代码
    f_star = min(result_matrix[:,-1])
    print("*"*100 + "\nlambda为：{lam}\n迭代次数为：{t}\n目标函数最优值为：{fx} \n最优解为：\n{x}\n".
        format(lam = lam,t = t, fx = f_star,x = list(x_tilde.T)))
    return result_matrix
colnames = ["iteration"] + ["x_{}".format(i) for i in range(1,A.shape[1]+1)] + ["target_function"] #创建
    result_matrix的列名列表，形如：["iteration","x_1","x_2",...,"x_100","target_function"]
result6 = SVRG(A,b,eps = 1e-6,learning_rate = 1e-2,lam = 1e-2/A.shape[0], T = 10) #lam = 1e-2/N
pd.DataFrame(columns=colnames,data=result6).to_csv('result6.csv')
```

****
lambda 为：3.0711587481956 94e-07
迭代次数为：1047
目标函数最优值为：0.56475524
最优解为：[[-0.08353022, -0.08407306, -0.08393807, -0.08384721, -0.08365582,
-0.08061148, -0.08463632, -0.08473917, -0.08479116, -0.08462586,
....
-0.08496314, -0.08502265,-0.08502189, -0.08502324, -0.08502339]]

2.2.5 有固定步长的随机梯度下降法

```
import matplotlib.pyplot as plt
def make_plot(result_matrix,label,color):
    x = [0.]+list(np.log(result_matrix[1:,0])) # 对x轴进行log采样
    plt.plot(x,result_matrix[:,−1],label = label,linewidth = 2, color = color)
plt.xlabel(”Log Iteration Times”)
plt.title(”Converage of Target function and X\nlambda = {}”.format(”1e−2/N”))
make_plot(result4,”SGD with Fixed Learning Rate”,”black”)
make_plot(result5,”SGD with Diminishing Learning Rate”,”blue”)
make_plot(result6,”SVRG”,”red”)
plt.legend()
plt.savefig(”Converage of Logistic Regression.png”,dpi=500)
plt.show()
```

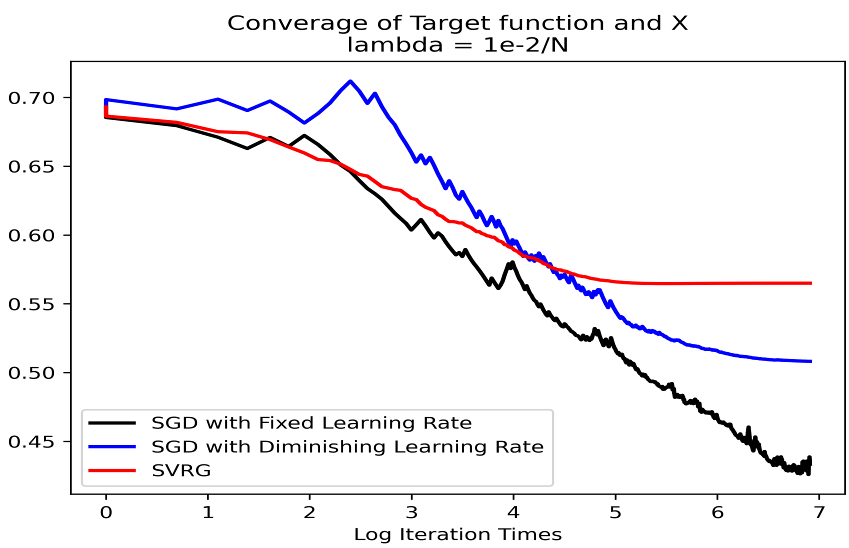

Figure 5: Converage of Logistic Regression