声明:

1. python的运行环境为3.9.5
2. 我在本地电脑将Python\Python39\Lib\site-packages\matplotlib\mpl-data\fonts\ttf的DejaVuSans.ttf改为了微软雅黑,所以助教您如果在您电脑上运行,可能会出现图像的label和title乱码的现象
3. 本次实验独立完成,绝不存在抄袭现象,如有雷同,可以当面对质

第一题

作业：SVM的前世今生

$$\min_{w,b,\xi_k(\forall k)} \frac{1}{2}||w||_2^2 + C\sum_{k=1}^{n}\xi_k$$
$$s.t. \quad y_k\left(w^{\top}x_k + b\right) \geq 1 - \xi_k \ (k = 1,2,\cdots,n)$$
$$\xi_k \geq 0 \ (k = 1,2,\cdots,n)$$

SVM

SVM---Support Vector Machine

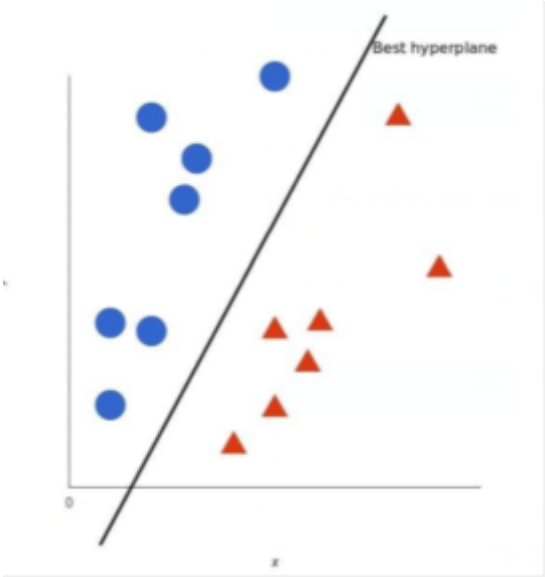
什么是支撑向量机

支撑向量机是一类按监督学习方式对数据进行二元类的广义线性分类器，其决策边界是对学习样本求解的最大边距超平面。SVM的的学习策略就是间隔最大化，可形式化为一个求解凸二次规划的问题，也等价于正则化的合页损失函数的最小化问题。SVM的的学习算法就是求解凸二次规划的最优化算法。

超平面是分割输入变量空间的线。在SVM中，选择超平面以最佳地将输入变量空间中的点与它们的类（0级或1级）分开。在二维中，我们可以将其视为一条线，并假设我们的所有输入点都可以被这条线完全分开。

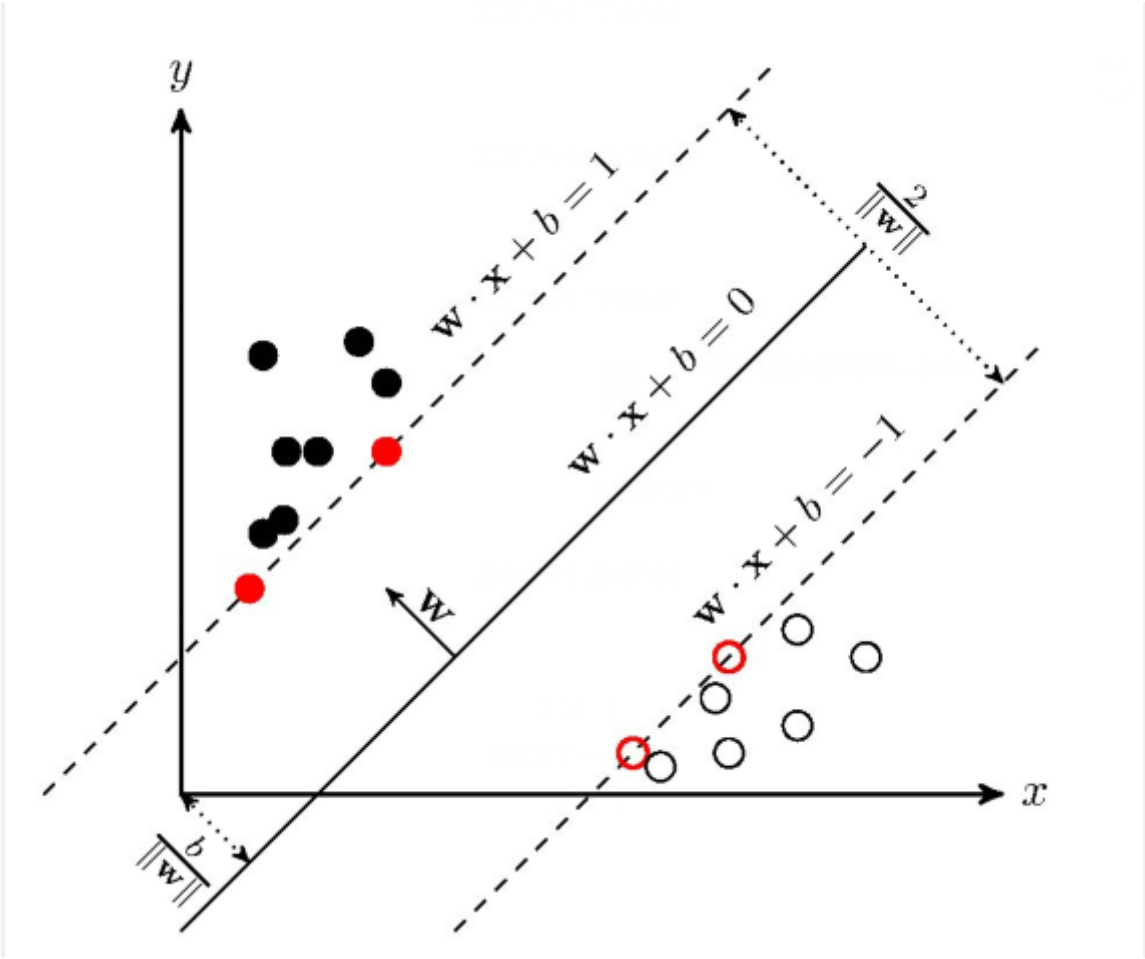
举例

我们假定有两个类别：红色和蓝色，所给的数据有两个特征：x 和 y。我们需要求出一条直线将这些红色和蓝色的点分开。



这样的直线不止一条，那如何定义最优呢，对于SVM来说，超平面（在本例中是一条线）对每个类别最近的元素距离最远。

算法原理



SVM学习的基本想法是求解能够正确划分训练数据集并且几何间隔最大的分离超平面。如上图所示，$\omega \cdot x + b = 0$ 即为分离超平面，对于线性可分的数据集来说，这样的超平面有无穷多个（即感知机），但是几何间隔最大的分离超平面却是唯一的。

给定训练集$T = \{ (x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n) \}$

其中$x_i \in R^n, y_i \in \{ +1, -1 \}$, x_i 为第i个特征向量， y_i 为类标记，当它等于$+1$时为正例；为-1时为负例。再假设训练数据集是**线性可分**的.

几何间隔：对于给定的数据集 T 和超平面 $\omega \cdot x + b = 0$ ，定义超平面关于样本点 (x_i, y_i) 的几何间隔为

$$\gamma_i = y_i(\frac{w}{\parallel w \parallel} \cdot x_i + \frac{b}{\parallel w \parallel})(i = 1, 2, 3, \ldots, n)$$

超平面关于所有样本点的几何间隔的最小值为

$$\gamma = \underset{i=1,2,3,\ldots,n}{min} \gamma_i$$

上述距离就是我们所谓的支持向量到超平面的距离。

根据以上定义，SVM模型的求解最大分割超平面问题可以表示为以下约束最优化问题

$$\underset{w,b}{max} \gamma$$

$$s.t. y_i(\frac{w}{\parallel w \parallel} \cdot x_i + \frac{b}{\parallel w \parallel}) \geq \gamma$$

将约束两边同时除以γ，得到

$$y_i(\frac{w}{\parallel w \parallel \gamma} \cdot x_i + \frac{b}{\parallel w \parallel \gamma}) \geq 1$$

由于$\parallel w \parallel$和γ都是标量,为了方便，我们令

$$\omega = \frac{w}{\parallel w \parallel \gamma}$$

$$b = \frac{b}{\parallel w \parallel \gamma}$$

得到

$$y_i(\omega \cdot x_i + b) \geq 1$$

由于最大化γ，等于最大化$\frac{1}{\|w\|}$，也就等于最小化$\frac{1}{2} \parallel w \parallel^2$

因此SVM模型的求解最大分割超平面问题又可以表示为以下约束最优化问题

$$\underset{w,b}{min}\frac{1}{2} \parallel w \parallel^2$$

$$s.t.y_i(\omega \cdot x_i + b) \geq 1(i = 1, 2, 3, \ldots, n)$$

这是一个含有不等式约束的凸二次规划问题，可以对其使用拉格朗日乘子法得到其对偶问题。

$$\text{拉格朗日函数：}\ L(w, b, \alpha) = \frac{1}{2} \parallel w \parallel^2 - \sum_{i=1}^{n} \alpha_i (y_i(w \cdot x_i + b) - 1)$$

$$\alpha_i\text{为拉格朗日乘子，}\ \alpha_i \geq 0$$

令$\theta(w) = \underset{\alpha_i \geq 0}{max} L(w, b, \alpha)$

故有

$$\theta(w) = \begin{cases} \frac{1}{2} \parallel w \parallel^2, x \in \text{可行域内} \\ \infty, x \in \text{不可行域内} \end{cases}$$

于是原约束问题等价于

$$\underset{w,b}{min}\theta(w) = \underset{w,b}{min}\underset{\alpha_i \geq 0}{max} L(w, b, \alpha) = p^*$$

我们还知道

$$\underset{\alpha_i \geq 0}{max}\underset{w,b}{min} L(w, b, \alpha) = d^*$$

当满足KKT条件时$p^* = d^*$

KKT条件为

$$\begin{cases} \alpha_i \geq 0 \\ y_i(w_i \cdot x_i + b) - 1 \geq 0 \\ \alpha_i(y_i(w_i \cdot x_i + b) - 1) = 0 \end{cases}$$

化简得到

$$\min_{\alpha}\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)-\sum_{i=1}^{n}\alpha_i$$

$$s.t.\sum_{i=1}^{n}\alpha_i y_i=0$$

$$\alpha_i\geq 0(i=1,2,3,\ldots,n)$$

到这里都是基于训练集数据线性可分的假设下进行的，但是实际情况下几乎不存在完全线性可分的数据，为了解决这个问题，引入了“软间隔”的概念，即允许某些点不满足约束 $y_j(w\cdot x_j+b)\geq 1$

为此，我们采用$hinge$损失，将原优化问题改写为

$$\min_{w,b,\xi_i}\frac{1}{2}\parallel w\parallel^2+C\sum_{i=1}^{n}\xi_i$$

$$s.t.y_j(w\cdot x_j+b)\geq 1-\xi_i$$

$$\xi_i\geq 0$$

我们把ξ_i称为松弛变量

选择惩罚参数C，构造并求解凸二次规划问题

$$\min_{\alpha}\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j(x_i\cdot x_j)-\sum_{i=1}^{n}\alpha_i$$

$$s.t.\sum_{i=1}^{n}\alpha_i y_i=0$$

$$0\leq\alpha_i\leq C(i=1,2,3,\ldots,n)$$

得到$\alpha^*=(\alpha_1^*,\alpha_2^*,\alpha_3^*,\ldots,\alpha_n^*,)$

然后计算$w^*=\sum_{i=1}^{n}\alpha_i^* y_i x_i$

选择α^*的一个分量α_j^*满足条件$0\leq\alpha_j^*\leq C$，计算$b^*=y_j-\sum_{i=1}^{n}\alpha_i^* y_i(x_i\cdot x_j)$

求分离超平面 $w^*\cdot x+b^*=0$

第二题

对于$\mathbb{R}^2$空间非二次规划问题 $\min f(x) = e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} + e^{-x_1-0.1}$，分析回溯直线搜索采用不同的 α, β 值时，误差随迭代次数改变的情况。注：初始值相同。

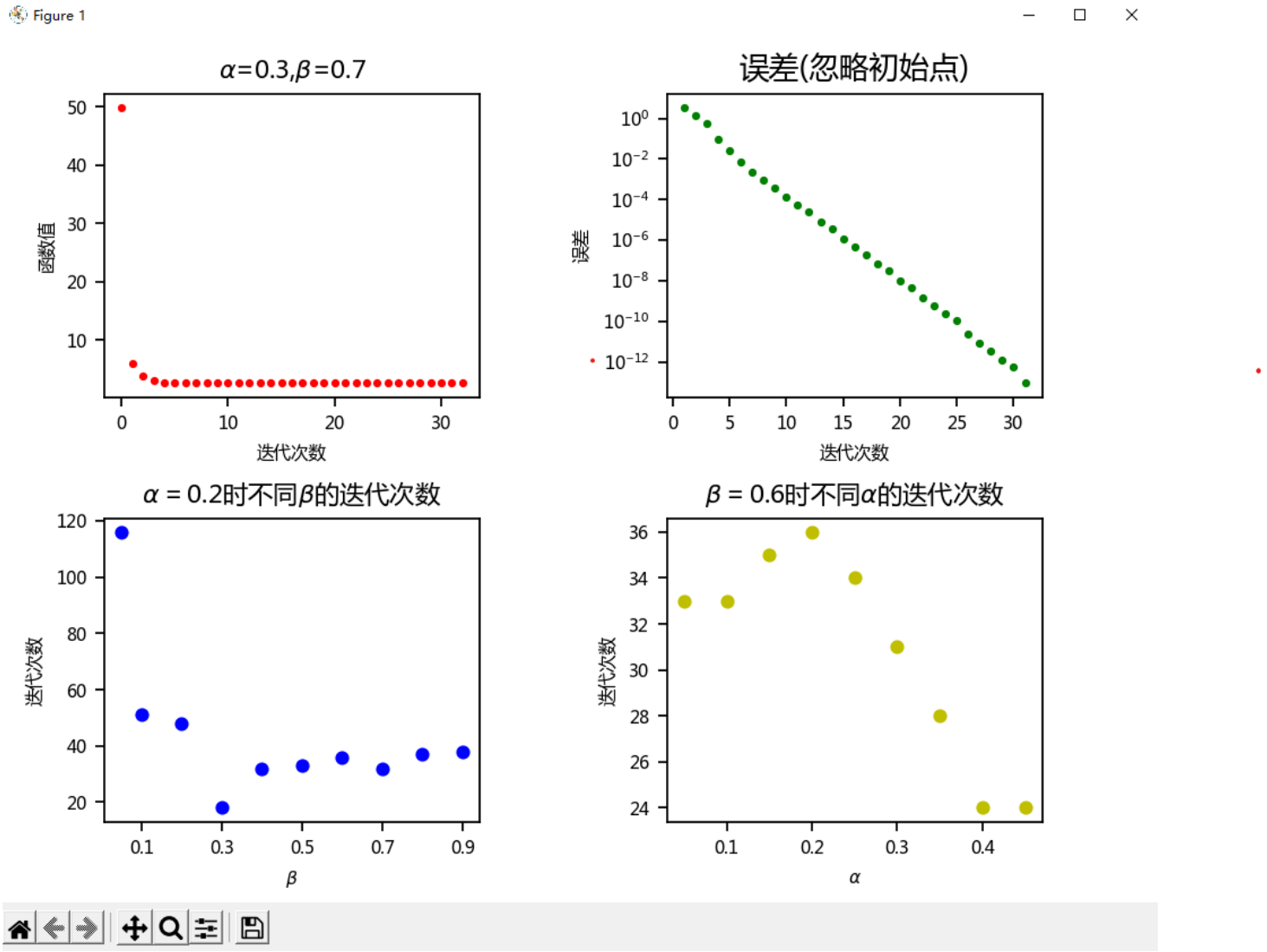
代码

```
import math
import sys
import numpy
import matplotlib.pyplot as plt
best_value = 0
a_array = [0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45]
b_array = [0.05,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
plot_array_k = []       #记录a=0.3,b=0.7时的迭代
plot_array_value = []   #记录a=0.3,b=0.7时的函数值
plot_array_Totalk = []  #记录迭代次数
plot_array_error = []
delta = 10e-7
#函数值
def f(tmp):
    x1=tmp[0]
    x2=tmp[1]
    value = math.exp(x1+3*x2-0.1) + math.exp(x1-3*x2-0.1) + math.exp(-x1-0.1)
    return value
#一阶导数
def f1(tmp):
    x1=tmp[0]
    x2=tmp[1]
    value1 = math.exp(x1+3*x2-0.1) + math.exp(x1-3*x2-0.1) - math.exp(-x1-0.1)
    value2 = 3*math.exp(x1+3*x2-0.1) - 3*math.exp(x1-3*x2-0.1)
    value=[value1,value2]
    return value
#二范数
def norm(tmp):
    array = f1(tmp)
    value = math.sqrt( pow(array[0],2) + pow(array[1],2) )
    return value
#得到步长
def get_dk(tmp):
    array = f1(tmp)
    value = [ -array[0], -array[1] ]
    return value
#内积
def inner_product(tmp1,tmp2):
    array = f1(tmp1)
    value = array[0]*tmp2[0]+array[1]*tmp2[1]
    return value
#更新点坐标
def update_point(tmp1,int,tmp2):
    value1 = tmp1[0]+int*tmp2[0]
    value2 = tmp1[1]+int*tmp2[1]
    value = [value1,value2]
    return value
#main函数
def get_plot_error():
    for i in range(0,len(plot_array_value),1):
```

```
        plot_array_error.append( plot_array_value[i] - best_value )
def main(a,b):
    global plot_array_k,plot_array_value,best_value
    x=[]
    y=[]
    point=[1.0,1.0]
    k = 0
    x.append(k)
    y.append(f(point))
    while( norm(point) > delta):
        tk = 1 #步长
        dk = get_dk(point) #方向
        while( f( [ point[0]+tk*dk[0], point[1]+tk*dk[1] ] ) > f(point) + a*tk*inner_product(poi
            tk *= b
        point = update_point(point,tk,dk)
        k += 1
        x.append(k)
        y.append(f(point))
    plot_array_Totalk.append(k)
    if a==0.3 and b==0.7:
        best_value = f(point)
        plot_array_k = x
        plot_array_value = y
def plot_figure():
    fig = plt.figure(num=1,dpi=160)
    plt.subplots_adjust(left=0.09, bottom=0.093, right=0.9, top=0.93,hspace=0.4, wspace=0.5)#防止
    #第一个图
    plt.subplot(221)
    plt.tick_params(labelsize=7)#刻度减小
    plt.scatter(x = plot_array_k,y = plot_array_value,color='r',s=5)
    plt.xlabel('迭代次数',fontsize=7)
    plt.ylabel('函数值',fontsize=7)
    plt.title('a=0.3,b=0.7',fontsize=10)
    #第二个图
    get_plot_error()
    plt.subplot(222)
    plt.tick_params(labelsize=7)
    plt.scatter(x=plot_array_k[1:32],y=plot_array_error[1:32],color='g',s=5)
    plt.xlabel('迭代次数',fontsize=7)
    plt.ylabel('误差',fontsize=7)
    plt.yscale('log')#设置纵坐标的缩放，写成m³格式
    plt.title('误差(忽略初始点)')
    #第三个图
    plt.subplot(223)
    plt.tick_params(labelsize=7)#刻度减小
    plt.xticks([0.1,0.3,0.5,0.7,0.9,1.0])#自设定横坐标
    plt.scatter(x=b_array,y=plot_array_Totalk[0:10],color='b',s=20)
    plt.title(r'$ \alpha =0.2 $'+'时不同'+r'$ \beta $'+'的迭代次数',fontsize=10)
    plt.xlabel(r'$ \beta $',fontsize=7)
    plt.ylabel('迭代次数',fontsize=7)
    #第四个图
```

```
    plt.subplot(224)
    plt.tick_params(labelsize=7)#刻度减小
    plt.xticks([0.1,0.2,0.3,0.4,0.5])#自设定横坐标
    plt.scatter(x=a_array,y=plot_array_Totalk[10:19],color='y',s=20)
    plt.title(r'$ \beta =0.6 $'+'时不同'+r'$ \alpha $'+'的迭代次数',fontsize=10)
    plt.xlabel(r'$ \alpha $',fontsize=7)
    plt.ylabel('迭代次数',fontsize=7)
    plt.show()
#程序入口
if __name__=="__main__":
    for i in range(0,len(b_array),1):
        main(0.2,b_array[i])
    for i in range(0,len(a_array),1):
        main(a_array[i],0.6)
    main(0.3,0.7)
    plot_figure()
```

实验结果



实验分析

1. 当给定$\alpha = 0.3, \beta = 0.7, \delta = 10^{-7}$时,经过32次迭代,算法停止,得到最优值约为2.559
2. 显然误差随着迭代次数增加而减小
3. 当给定$\alpha = 0.2$时,β从0.05到0.9变化,可以看出迭代次数并没有量级的改变,多数集中在20~50之前
4. 当给定$\beta = 0.6$时,α从0.05到0.45变化,可以看出迭代次数并没有量级的改变,集中在20~40之间
5. 由以上分析可知,α和β的值对迭代次数并无较大影响

心得

这道题非常简单,只需要对照伪代码翻译即可,没有遇到任何问题

第三题

10.15 等式约束熵极大化。 考虑等式约束熵极大化问题

$$\begin{array}{ll} \text{minimize} & f(x) = \sum_{i=1}^{n} x_i \log x_i \\ \text{subject to} & Ax = b, \end{array}$$

其中 $\mathbf{dom}\, f = \mathbf{R}^n_{++}$，$A \in \mathbf{R}^{p\times n}$，$p < n$。（一些相关分析见习题 10.9。）

生成一个 $n = 100$，$p = 30$ 的问题实例，随机选择 A（验证其为满秩阵），随机选择一个正向量作为 $\hat{x}$（例如，其分量在区间 $[0, 1]$ 上均匀分布），然后令 $b = A\hat{x}$。（于是，$\hat{x}$ 可行。）

采用以下方法计算该问题的解。

(a) **标准 Newton 方法。** 可以选用初始点 $x^{(0)} = \hat{x}$。

(b) **不可行初始点 Newton 方法。** 可以选用初始点 $x^{(0)} = \hat{x}$（和标准 Newton 方法比较），也可以选用初始点 $x^{(0)} = \mathbf{1}$。

(c) **对偶 Newton 方法**，即将标准 Newton 方法应用于对偶问题。

证实三种方法求得相同的最优点（和 Lagrange 乘子）。比较三种方法每步迭代的计算量，假设利用了相应的结构。（但在你的实现中不需要利用结构计算 Newton 步径。）

(a)

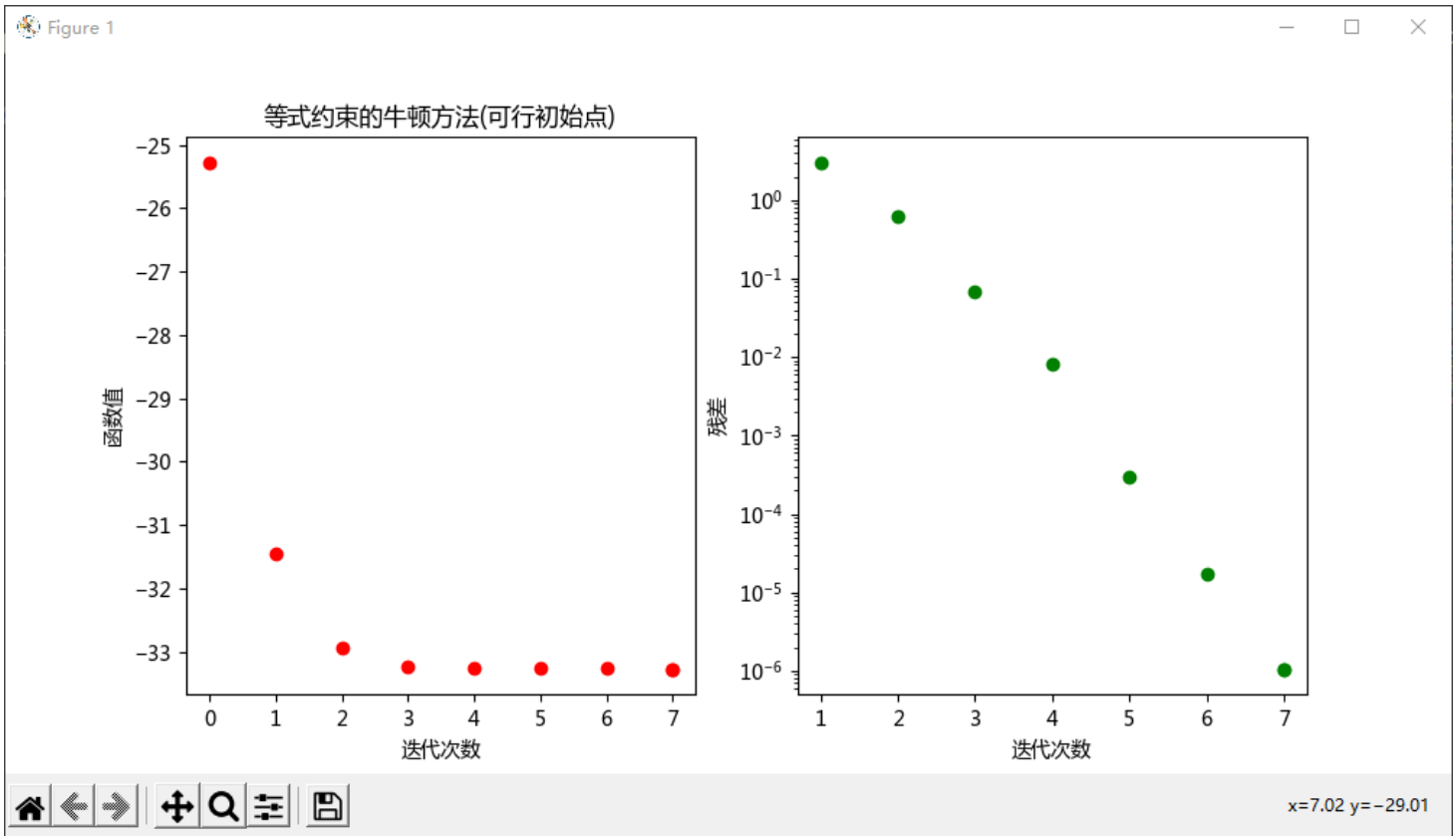
代码

```
import numpy as np
import matplotlib.pyplot as plt
import sys
aerfa = 0.3
beta = 0.7
delta = 10e-7
n = 100
p = 30
#随机生成满秩矩阵(p*n)
def matrix():
    value = np.random.randint(0,100,(p,n))#随机生成p*n的A矩阵
    while( np.linalg.matrix_rank(value)<p ):#保证满秩
        value = np.random.randint(0,2,(p,n))
    return value
#随机生成一维向量(n*1)
def vector():
    value = np.random.rand(n,1) # n*1矩阵
    return value
#生成b向量(n*1)
def b_vector(tmp1,tmp2):
    value = np.dot(tmp1,tmp2)
    return value
#函数值
def f(tmp):
    value = 0
    for i in range(0,n,1):
        value += tmp[i][0]*np.log(tmp[i][0])
    return value
#一阶导数(n*1)
def f1(tmp):
    value = np.zeros( (n,1) )
    for i in range(0,n):
        value[i][0] = 1+np.log(tmp[i][0])
    return value
#二阶导数矩阵(n*n)
def f2(tmp):
    value = np.random.randint(0,1,(n,n))#生成0矩阵(n*n)
    for i in range(0,n):
        value[i][i]=1/tmp[i][0]
    return value
#求二阶导数的逆矩阵(n*n)
def inverse_f2(tmp):
    value = f2(tmp)#二阶导数
    value = np.linalg.inv(value)#求逆
    return value
#求方向d_nt^k(n*1)
def d_nt(tmp,tmpA):
    temp1 = np.hstack( (f2(tmp),np.transpose(tmpA)) )#横向合并矩阵
    temp2 = np.hstack( (tmpA,np.zeros((p,p))) )
    temp3 = np.vstack( (temp1,temp2) )#纵向合并矩阵
```

```
    temp4 = np.vstack( ( -f1(tmp),np.zeros( (p,1) ) ) )
    temp_value = np.dot(np.linalg.inv(temp3),temp4)
    value = np.zeros((n,1))
    for i in range(0,n,1):
        value[i][0] = temp_value[i][0]
    return value
#求lamba
def lamba2(tmp,tmpA):
    inverse_d_nt = np.transpose(d_nt(tmp,tmpA))
    value1 = np.dot(inverse_d_nt,f2(tmp))#d^T*f_2
    value = np.dot(value1,d_nt(tmp,tmpA))#d^T*f_2*d
    return value
#main函数
def main(script,*argv):
    k = 0
    xk = vector()#随机生成初始点(n*1)
    A = matrix()#随机生成矩阵(n*n)
    b = b_vector(A,xk)#随机生成b向量(n*1)
    x=list()#记录迭代次数
    y=list()#记录函数值
    t=list()#记录tk
    residual = list()
    x.append(k)
    y.append(f(xk))
    while( 1/2*lamba2(xk,A) > delta ):
        tk = 1 #步长
        dk = d_nt(xk,A)#方向
        #print("times:",k,np.dot(A,dk))#验证一下Ad=0
        while( f(xk+tk*dk) > f(xk)-aerfa*tk*lamba2(xk,A) ):
            tk *= beta
        xk += tk*dk
        k += 1
        x.append(k)
        y.append(f(xk))
        t.append(tk)
        residual.append(lamba2(xk,A))
    x.append(k)
    y.append(f(xk))
    residual.append(lamba2(xk,A))
    #绘图
    fig = plt.figure(num=1,figsize=(10,5))
    #第一个
    plt.subplot(121)
    plt.scatter(x=x,y=y,color='r')
    plt.xlabel('迭代次数')
    plt.ylabel('函数值')
    plt.title('等式约束的牛顿方法(可行初始点)')
    #第二个
    plt.subplot(122)
    plt.scatter(x=x[1:len(x)],y=residual,color='g')
    plt.xlabel('迭代次数')
```

```
    plt.ylabel('残差')
    plt.yscale('log')
    plt.show()
#入口
if __name__=="__main__":
    main(*sys.argv)
```

实验结果



实验分析

1. 经过多次实验,对于随机的矩阵A, x, b,可行初始点的等式约束的牛顿方法仅需要少于10次的迭代,即可得到最优值.
2. 迭代到最后,对偶残差和原残差均小于δ

心得

对照伪代码翻译即可，需要注意矩阵的行列关系

(b)

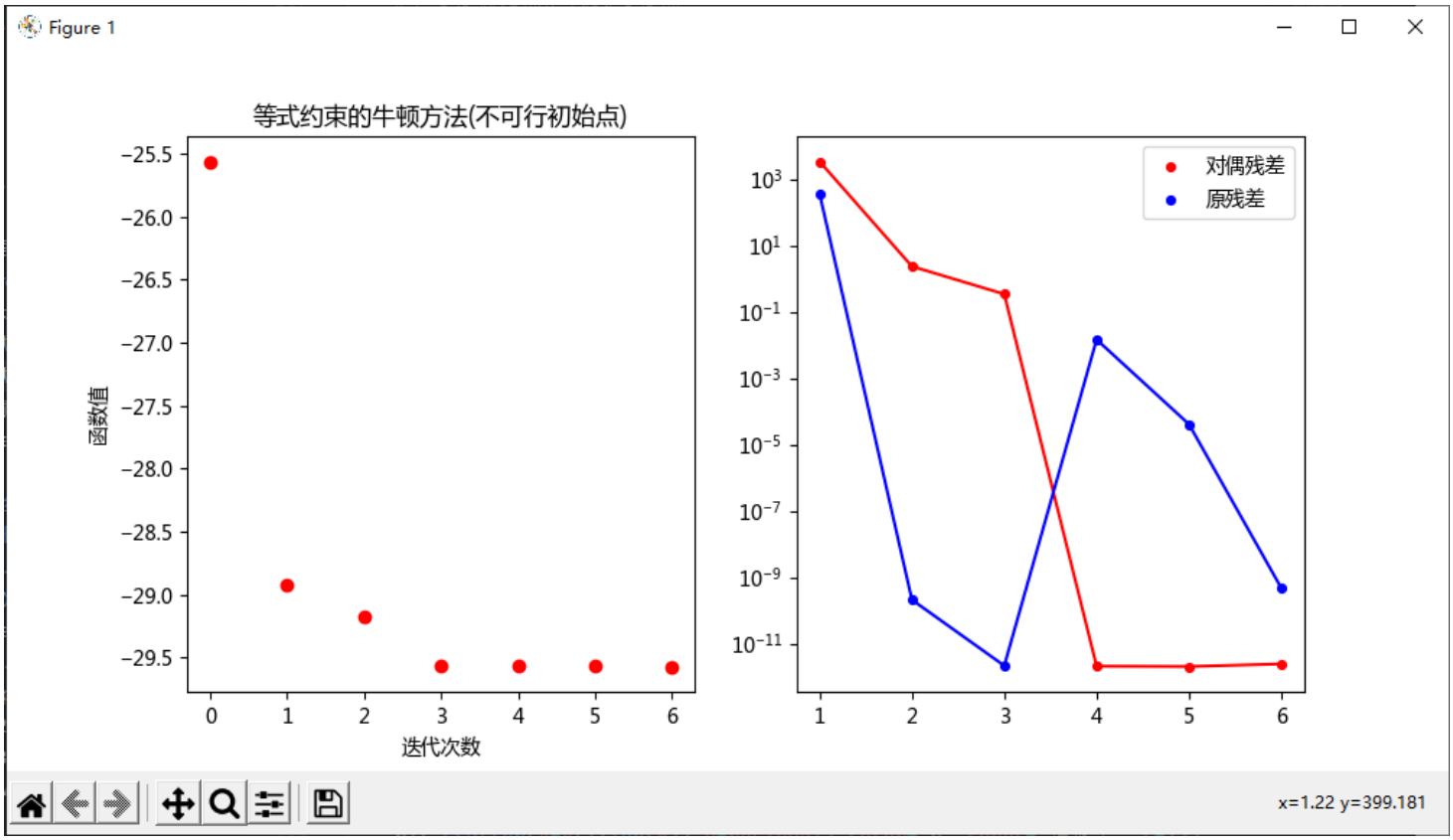
代码

```
import math
import numpy as np
import matplotlib.pyplot as plt
import sys
import scipy
aerfa = 0.3
beta = 0.7
delta = 10e-7
n = 100
p = 50
#随机生成满秩矩阵(p*n)
def matrix():
    value = np.random.randint(0,100,(p,n))#随机生成p*n的A矩阵
    while( np.linalg.matrix_rank(value)<p ):#保证满秩
        value = np.random.randint(0,100,(p,n))
    return value
#函数值
def f(tmp):
    value = 0
    for i in range(0,n,1):
        value += tmp[i][0]*np.log(tmp[i][0])
    return value
#一阶导数(n*1)
def f1(tmp):
    value = np.zeros( (n,1) )
    for i in range(0,n,1):
        value[i][0] = 1+np.log(tmp[i][0])
    return value
#二阶导数矩阵(n*n)
def f2(tmp):
    value = np.zeros( (n,n) )#生成0矩阵(n*n)
    for i in range(0,n,1):
        value[i][i]=1/tmp[i][0]
    return value
#r的二范数
def norm_r2(tmpx,tmpv,tmpA,tmpb):
    value = 0
    value1 = f1(tmpx)+np.dot(np.transpose(tmpA),tmpv)
    value2 = np.dot(tmpA,tmpx) - tmpb
    for i in range(0,n,1):
        value += value1[i][0]*value1[i][0]
    for i in range(0,p,1):
        value += value2[i][0]*value2[i][0]
    return math.sqrt(value)
def get_norm(tmpx,tmpv,tmpA,tmpb):
    value_1 = 0
    value_2 = 0
    value1 = f1(tmpx)+np.dot(np.transpose(tmpA),tmpv)
    value2 = np.dot(tmpA,tmpx) - tmpb
    for i in range(0,n,1):
```

```
        value_1 += value1[i][0]*value1[i][0]
    for i in range(0,p,1):
        value_2 += value2[i][0]*value2[i][0]
    return {math.sqrt(value_1),math.sqrt(value_2)}
#xk和vk的方向
def one_d_nt(tmpx,tmpA,tmpv,tmpb,flag):
    temp1 = np.hstack( (f2(tmpx),np.transpose(tmpA)) )#横向合并矩阵
    temp2 = np.hstack( (tmpA,np.zeros((p,p))) )#横向合并矩阵
    temp3 = np.vstack( ( temp1,temp2 ) )#纵向合并矩阵
    temp4 = -np.vstack( (f1(tmpx)+np.dot(np.transpose(tmpA),tmpv),np.dot(tmpA,tmpx)-tmpb) )#纵向
    if np.linalg.matrix_rank(temp3) == n+p:
        temp_value = np.dot(np.linalg.inv(temp3),temp4)
    else:#伪逆矩阵
        temp_value = np.dot(scipy.linalg.pinv(temp3),temp4)
    value_dxk = np.zeros((n,1))
    value_dvk = np.zeros((p,1))
    if flag == True:
        for i in range(0,n,1):
            value_dxk[i][0] = temp_value[i][0]
        return value_dxk
    else:
        for i in range(n,n+p,1):
            value_dvk[i-n][0] = temp_value[i][0]
        return value_dvk
#检查Ax=b
def check(tmp1,tmp2):
    for i in range(0,p,1):
        if abs(tmp1[i][0]-tmp2[i][0]) > delta:
            return False
        return True
#获得合法的tk值
def Get_Valid_tk(tmpx,tmpdx):
    tk = 1
    while(1):
        for i in range(0,n,1):
            if ( tmpx[i][0] + tk*tmpdx[i][0] < 0 ) :
                break
        if (i == n-1):
            return tk
        else:
            tk *= beta
#main函数
def main(script,*argv):
    k = 0
    xk = np.random.rand(n,1)#随机生成初始点(n*1)(不一定满足Axk=b)
    vk = np.random.rand(p,1)#随机生成(p*1)列向量
    A = matrix()#随机生成矩阵(n*n)
    b = np.dot(A,np.random.rand(n,1))#生成b向量(p*1)=(p*n)*(n*1)
    x=list()#记录迭代次数
    y=list()#记录函数值
    t=list()
```

```
    residual1=list()
    residual2=list()
    x.append(k)
    y.append(f(xk))
    #不可行初始点迭代到可行点
    while( not( check( np.dot(A,xk),b)==True and norm_r2(xk,vk,A,b)<= delta) ):
        dxk = one_d_nt(xk,A,vk,b,True)#d方向
        dvk = one_d_nt(xk,A,vk,b,False)#v方向
        tk = Get_Valid_tk(xk,dxk)#步长为1时可能会使得x跳出定义域，先确定一个有效的tk值
        while( norm_r2(xk+tk*dxk,vk+tk*dvk,A,b) > (1-aerfa*tk)*norm_r2(xk,vk,A,b) ):
            tk *= beta
        xk += tk*dxk
        vk += tk*dvk
        k += 1
        x.append(k)
        y.append(f(xk))
        t.append(tk)
        v1,v2=get_norm(xk,vk,A,b)
        residual1.append(v1)
        residual2.append(v2)
    #绘图
    fig = plt.figure(num=1,figsize=(10,5))
    #第一个图
    plt.subplot(121)
    plt.scatter(x=x,y=y,color='r')
    plt.xlabel('迭代次数')
    plt.ylabel('函数值')
    plt.title('等式约束的牛顿方法(不可行初始点)')
    #第二个图
    plt.subplot(122)
    plt.scatter(x=x[1:len(x)],y=residual1,color='r',label='对偶残差',s=17)
    plt.scatter(x=x[1:len(x)],y=residual2,color='b',label='原残差',s=17)
    plt.plot(x[1:len(x)],residual1,color='r')
    plt.plot(x[1:len(x)],residual2,color='b')
    plt.yscale('log')#设置纵坐标的缩放，写成m³格式
    plt.legend()
    plt.show()
    print("end")
#入口
if __name__=="__main__":
    main(*sys.argv)
```

实验结果



实验分析

1. 经过多次实验,对于随机的矩阵A, x, b,不可行初始点的等式约束的牛顿方法仅需要少于10次的迭代,即可得到最优值.
2. 迭代到最后,对偶残差和原残差均小于δ

心得

本次写代码的过程中,出现了困扰我一段时间的bug,当迭代的时候,会出现类似于log中的参数有负数的情况,由于理论不扎实,第四章的时候上课已经跟不上了就没有听,甚至在写这次代码的时候,完全不清楚这章的算法什么意思,只是对着伪代码翻译,所以碰见log中参数存在复数的bug我感到很疑惑,明明是完全把伪代码翻译过来了,竟然出现了错误,后来在复习的时候看回放我才发现罗老师说过,在迭代的过程中要保证x^k在定义域内,看到这里我才知道为什么会有之前的bug,为此我增加了一个Get_Valid_tk()函数,在给定x^k和dx^k时,求出使得x^k在定义域内的尽量大的t^k,得到有效的t^k再开始进行回溯

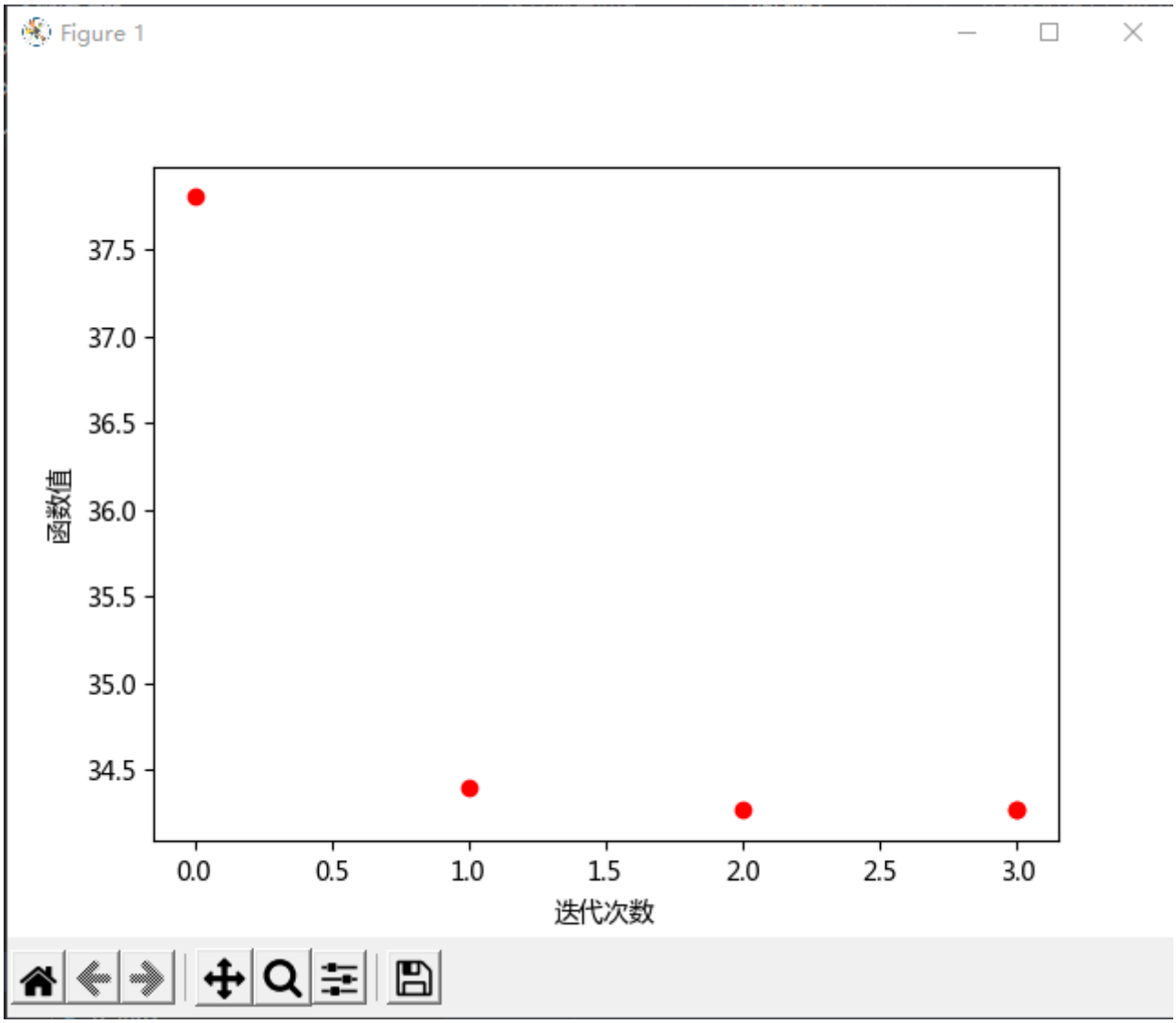
(c)

代码

```
import numpy as np
import matplotlib.pyplot as plt
import sys
import math
p = 30
n = 100
aerfa = 0.3
beta = 0.7
delta = 10e-5
xk = np.random.rand(n,1) #生成xk
A= np.random.uniform(-0.1,0.1,(p,n))
b = np.dot(A,xk) #生成b向量
vk = np.random.rand(p,1) #生成列矩阵
def g(tmp):
    global A,b
    temp_M = np.dot(np.transpose(A),tmp)
    value = 0
    for i in range(0,p,1):
        value += b[i][0]*tmp[i][0]
    for i in range(0,n,1):
        value += math.exp(-1-temp_M[i])
    return value
def g1(tmp):
    global A,b
    temp_M = np.dot(np.transpose(A),tmp)
    value = np.zeros((p,1))
    for i in range(0,p,1):
        value[i][0] = b[i][0]
        for j in range(0,n,1):
            value[i][0] +=  -A[i][j] * math.exp(-1- temp_M[j]  )
    return value
def g2(tmp):
    global A,b
    temp_M = np.dot(np.transpose(A),tmp)
    value = np.zeros( ( p,p) )
    for i in range(0,p,1):
        for k in range(0,p,1):
            value[i][k] = 0
            for j in range(0,n,1):
                value[i][k] += A[i][j]*A[k][j]*math.exp(-1-temp_M[j])
    return value
def d_nt(tmp):
    value = -np.dot( np.linalg.inv(g2(tmp)) , g1(tmp))
    return value
def lamba2(tmp):
    value1 = np.dot( np.transpose(d_nt(tmp)), g2(tmp))
    value = np.dot(value1,d_nt(tmp))
    return value
def main(script,*argv):
    global vk
```

```
    k = 0
    x=list()#记录迭代次数
    y=list()#记录函数值
    x.append(k)
    y.append(g(vk))
    while( 1/2*lamba2(vk) > delta ):
        tk = 1 #步长
        dk = d_nt(vk)#方向
        #print(vk+tk*dk)
        #print(vk)
        print(g(vk+tk*dk))
        while( g(vk+tk*dk) > g(vk)-aerfa*tk*lamba2(vk) ):
            tk *= beta
        vk += tk*dk
        k += 1
        x.append(k)
        y.append(g(vk))
    x.append(k)
    y.append(g(vk))
    #绘图
    print("times:",k)
    print("value:",g(vk))
    plt.scatter(x=x,y=y,color='r')
    plt.xlabel('迭代次数')
    plt.ylabel('函数值')
    plt.show()
#入口
if __name__=="__main__":
        main(*sys.argv)
```

实验结果



实验分析

$$拉格朗日函数：L(x,\lambda)=f(x)+\lambda^T(Ax-b)$$

$$拉格朗日对偶函数：g(\lambda)=inf L(x,\lambda)$$

$$化简得：g(\lambda)=-b^Tv-\sum_{i=1}^{n}e^{-1-(A^T\lambda)_i}$$

$$对偶问题：maxg(\lambda)$$

$$对偶问题是凹函数,因此求：min-g(\lambda)$$

1. 只需将求得的结果取相反数即使原优化问题的最小值
2. 这次代码的编写其实是有些失败的,虽然可以得到正确的结果,但是速度很慢,每一次迭代回溯的过程均需要调用$g2()$函数,而这个函数的时间复杂度是$O(p^2n)$,反复调用这个函数,时间开销特别大,此外,由于对偶问题需要用到e的指数次幂,因此当把v的初始值设定过大或者过小时会出现math.exp()函数

超时的情况,所以我在程序中设定v的初始值在-0.1到0.1之间.由于是手动求导,因此我猜想可能是求导方法有误因而浪费了大量时间,但由于时间限制没办法优化,所以只能提交这个版本.

课程收获

之所以选这门课,是因为听学长说这门课虽然难但是可以学到东西,上了这门课发现确实是这样,这门课我认为更像是一门纯数学课,课程中充斥着大量数学符号和严谨的数学推导,上课经常会出现听不懂或者是虽然可以听懂,但是不知道在讲什么的情况.后期讲到四五章的时候,已经完全跟不上了,所幸放弃了,直到期末复习的时候才开始从头梳理,对这门课才算有了片面的认识。

这门课不仅仅让我学到了一些优化算法,我也是时隔将近一年才开始用Python,发现Python真的很好用,不仅仅了解了Python语法,也学了markdown的使用.