CVX Homework

> 计算机85 张子诚

第一次大作业

SVM综述

SMO 算法

```
"""
参考了SMO算法的论文
[Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector
Machines]
(https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-
14.pdf)
复现了该算法
"""
from __future__ import division, print_function
import os
import numpy as np
import random as rnd
filepath = os.path.dirname(os.path.abspath(__file__))

class SVM():
    """
        Simple implementation of a Support Vector Machine using the
        Sequential Minimal Optimization (SMO) algorithm for training.
    """
    def __init__(self, max_iter=10000, kernel_type='linear', C=1.0,
epsilon=0.001):
        self.kernels = {
            'linear' : self.kernel_linear,
            'quadratic' : self.kernel_quadratic
        }
        self.max_iter = max_iter
        self.kernel_type = kernel_type
        self.C = C
        self.epsilon = epsilon
    def fit(self, X, y):
        # Initialization
        n, d = X.shape[0], X.shape[1]
        alpha = np.zeros((n))
        kernel = self.kernels[self.kernel_type]
        count = 0
        while True:
            count += 1
            alpha_prev = np.copy(alpha)
            for j in range(0, n):
                i = self.get_rnd_int(0, n-1, j) # Get random int i~=j(i!=j)
                x_i, x_j, y_i, y_j = X[i,:], X[j,:], y[i], y[j]
                k_ij = kernel(x_i, x_i) + kernel(x_j, x_j) - 2 * kernel(x_i,
x_j)
```

```
                if k_ij == 0:
                    continue
                alpha_prime_j, alpha_prime_i = alpha[j], alpha[i]
                (L, H) = self.compute_L_H(self.C, alpha_prime_j, alpha_prime_i,
y_j, y_i)

                # Compute model parameters
                self.w = self.calc_w(alpha, y, X)
                self.b = self.calc_b(X, y, self.w)

                # Compute E_i, E_j
                E_i = self.E(x_i, y_i, self.w, self.b)
                E_j = self.E(x_j, y_j, self.w, self.b)

                # Set new alpha values
                alpha[j] = alpha_prime_j + float(y_j * (E_i - E_j))/k_ij
                alpha[j] = max(alpha[j], L)
                alpha[j] = min(alpha[j], H)

                alpha[i] = alpha_prime_i + y_i*y_j * (alpha_prime_j - alpha[j])

            # Check convergence
            diff = np.linalg.norm(alpha - alpha_prev)
            if diff < self.epsilon:
                break

            if count >= self.max_iter:
                print("Iteration number exceeded the max of %d iterations" %
(self.max_iter))
                return
        # Compute final model parameters
        self.b = self.calc_b(X, y, self.w)
        if self.kernel_type == 'linear':
            self.w = self.calc_w(alpha, y, X)
        # Get support vectors
        alpha_idx = np.where(alpha > 0)[0]
        support_vectors = X[alpha_idx, :]
        return support_vectors, count
    def predict(self, X):
        return self.h(X, self.w, self.b)
    def calc_b(self, X, y, w):
        b_tmp = y - np.dot(w.T, X.T)
        return np.mean(b_tmp)
    def calc_w(self, alpha, y, X):
        return np.dot(X.T, np.multiply(alpha,y))
    # Prediction
    def h(self, X, w, b):
        return np.sign(np.dot(w.T, X.T) + b).astype(int)
    # Prediction error
    # Ei=ui-yi is the error on the ith traning example
    def E(self, x_k, y_k, w, b):
        return self.h(x_k, w, b) - y_k
    def compute_L_H(self, C, alpha_prime_j, alpha_prime_i, y_j, y_i):
        if(y_i != y_j):
            return (max(0, alpha_prime_j - alpha_prime_i), min(C, C -
alpha_prime_i + alpha_prime_j))
        else:
            return (max(0, alpha_prime_i + alpha_prime_j - C), min(C,
alpha_prime_i + alpha_prime_j))
    def get_rnd_int(self, a,b,z):
        i = z
        cnt=0
        while i == z and cnt<1000:
            i = rnd.randint(a,b)
            cnt=cnt+1
        return i
    # Define kernels
    def kernel_linear(self, x1, x2):
        return np.dot(x1, x2.T)
    def kernel_quadratic(self, x1, x2):
        return (np.dot(x1, x2.T) ** 2)
```

第二次大作业

回溯直线搜索下的梯度下降法

> 源代码见`LineSearch_GD.ipynb`

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation
from IPython.display import HTML, Image
import seaborn as sns
import time
# sns.set()
%matplotlib inline
plt.style.use('seaborn-white')
```

$$
\begin{aligned}
L(x) &= \exp(x_1+3x_2-0.1)+\exp(x_1-3x_2-0.1)+\exp(-x_1-0.1)\\
\frac{\partial L}{\partial x_1} &= \exp(x_1+3x_2-0.1)+\exp(x_1-3x_2-0.1)-\exp(-x_1-0.1)\\
\frac{\partial L}{\partial x_2} &= 3\exp(x_1+3x_2-0.1)-3\exp(x_1-3x_2-0.1)
\end{aligned}
$$

```
L=lambda x1,x2: np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)+np.exp(-x1-0.1)
```

```
def compute_grad(x1,x2):
    g1=np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)-np.exp(-x1-0.1)
    g2=3*np.exp(x1+3*x2-0.1)-3*np.exp(x1-3*x2-0.1)
    return g1,g2
```

```
#初始点
a,b=1.5,-0.8
```

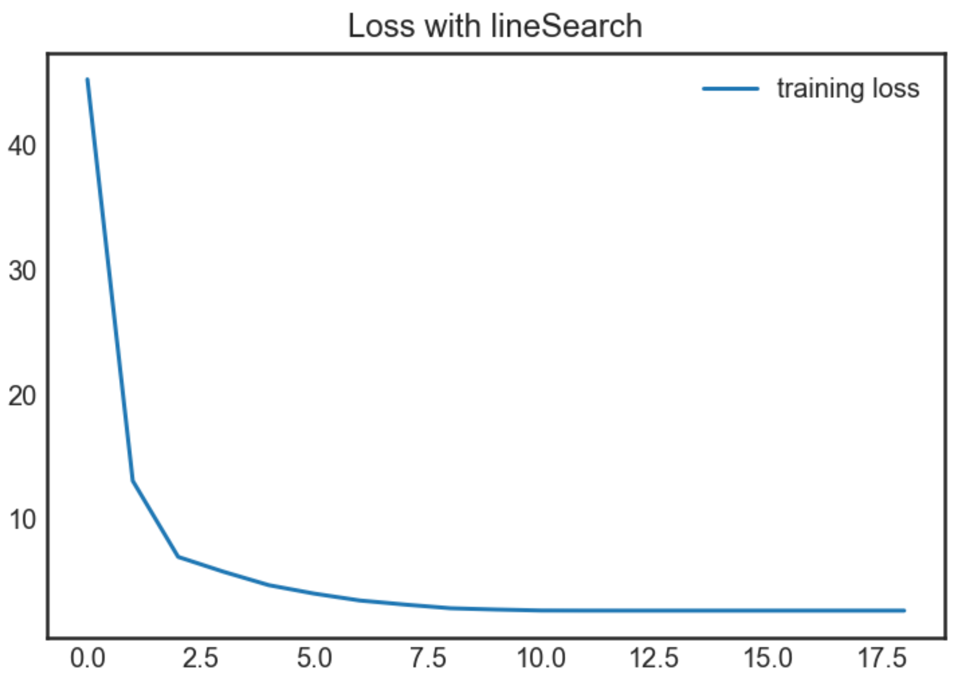
```
# 回溯直线搜索
def condtition(x1,x2,g1,g2,t,alpha):
    left=L(x1-t*g1,x2-t*g2)
    right=L(x1,x2)-alpha*t*(g1**2+g2**2)
    return left>right
def linesearch(x1,x2,g1,g2):
    t=1
    alpha,beta=0.5,0.8
    while condtition(x1,x2,g1,g2,t,alpha):
        t=beta*t
    return t
```

```
a0,b0=a,b
learning_rate=0.001
Loss_0=[]
a0_history,b0_history=[a0],[b0]
startTime=time.time()
iteration=0
p=2.559266696757757
while True:
    g1,g2=compute_grad(a0,b0)
    loss=L(a0,b0)
    Loss_0.append(loss)
    if loss-p<1e-8:
        break
    print("第{}轮loss:{}".format(iteration,loss))
    learning_rate=linesearch(a0,b0,g1,g2)
    a0-=learning_rate*g1
    b0-=learning_rate*g2
    a0_history.append(a0)
    b0_history.append(b0)
    iteration+=1

Loss_0.append(L(a0,b0))
print('{} sec(s)'.format(time.time()-startTime))
```

```
第0轮loss:45.27096045246693
第1轮loss:12.991251368635966
第2轮loss:6.880166553211321
第3轮loss:5.686298108953727
第4轮loss:4.606273040583032
第5轮loss:3.92784136334933
第6轮loss:3.376936408493199
第7轮loss:3.0460138836458395
第8轮loss:2.7525324263427704
第9轮loss:2.6487151256908437
第10轮loss:2.5687344406666313
第11轮loss:2.5604484380112584
第12轮loss:2.5593557491093453
第13轮loss:2.5592852188744315
第14轮loss:2.55926719456417
第15轮loss:2.559266812545181
第16轮loss:2.559266728457101
0.005035400390625 sec(s)
```

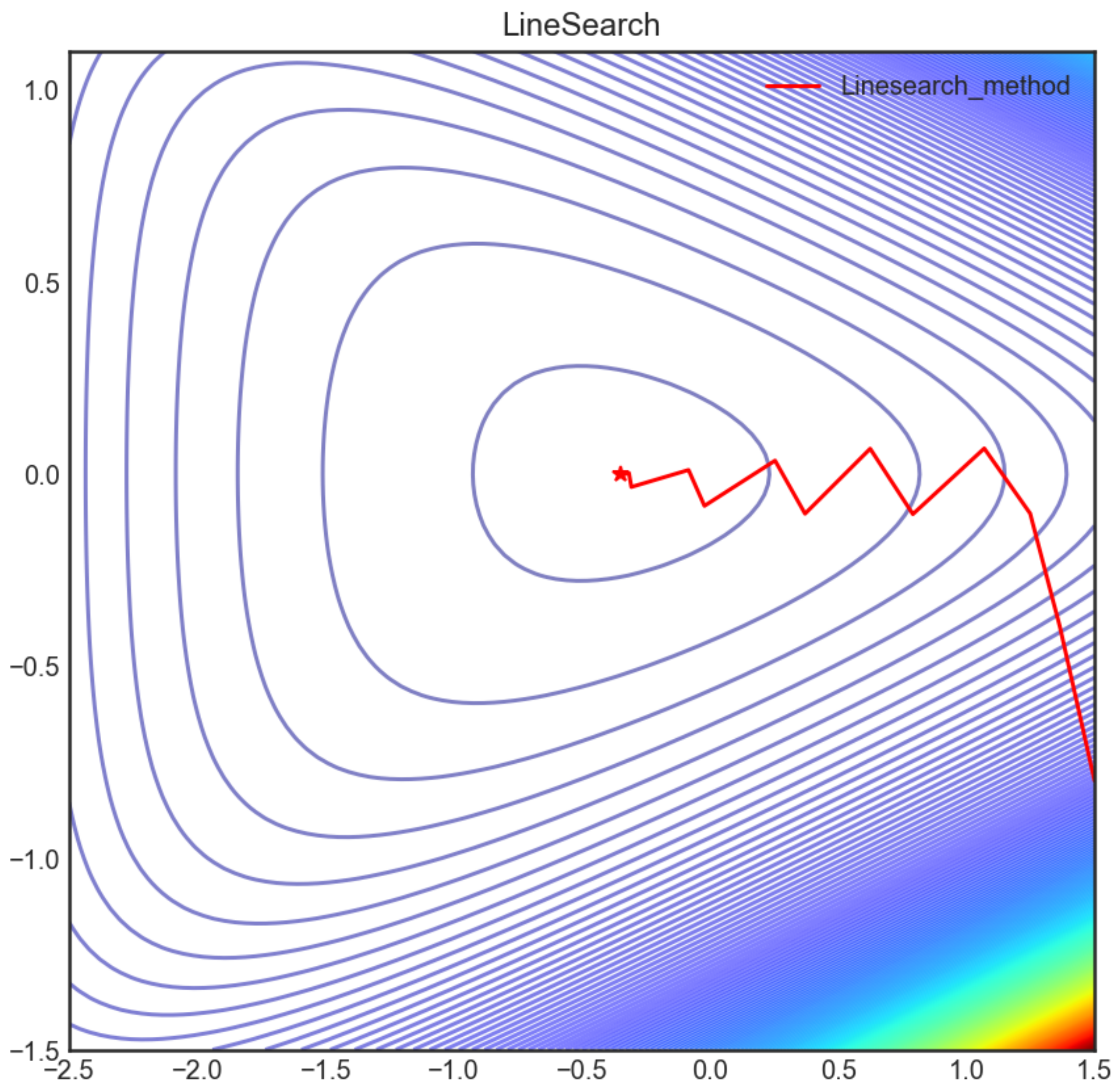
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
plt.plot(Loss_0,label='training loss')
plt.title("Loss with lineSearch")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d50731a6c8>
```



```
x,y=np.linspace(-2.5,1.5,100),np.linspace(-1.5,1.1,100)
X,Y=np.meshgrid(x,y)
l=L(X,Y)

fig1,ax1=plt.subplots(figsize=(7,7))
ax1.contour(X,Y,l,250,cmap='jet',alpha=0.5)
ax1.plot(a0_history,b0_history,'red',label='Linesearch_method',lw=1.5)
ax1.plot(a0_history[-1],b0_history[-1],'*',color='red',markersize=6)
plt.title("LineSearch")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d5073d8988>
```

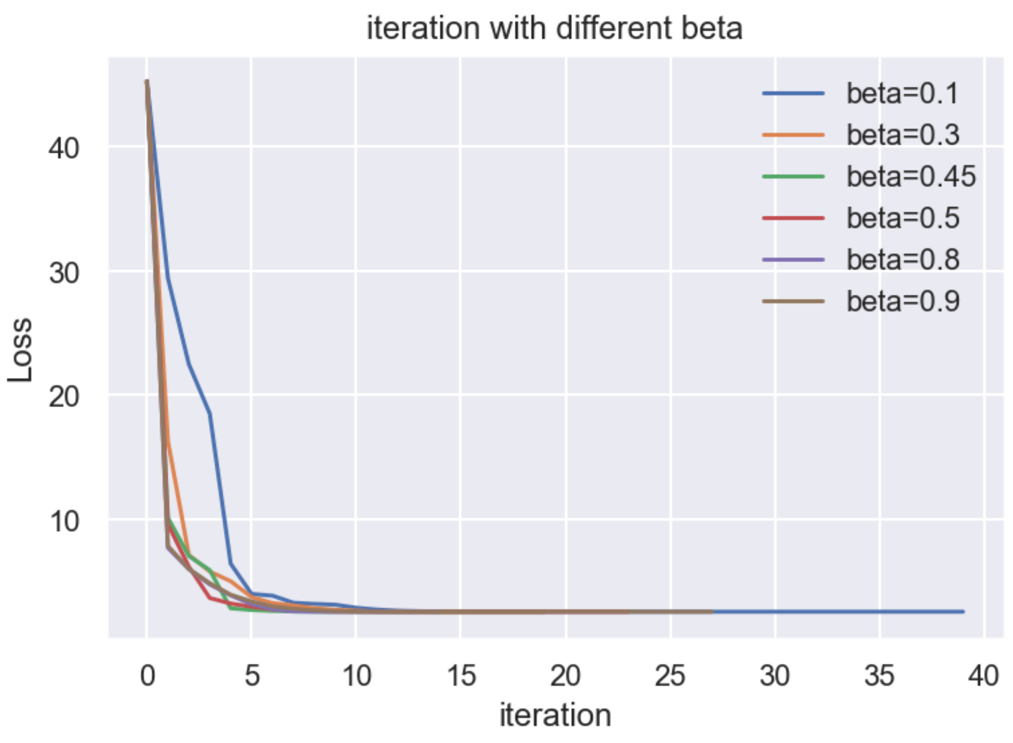


> LineSearch 的方法在条件数比较大时震荡，主要原因个人认为在于每个方向分配了相同的步长

```
# 固定alpha 为0.2,变化beta
sns.set()
beta_arr=[0.1,0.3,0.45,0.5,0.8,0.9]
for beta in beta_arr:
    a0,b0=1.5,-0.8
    alpha=0.4
    Loss_0=[]
    iteration=0

    while True:
        g1,g2=compute_grad(a0,b0)
        loss=L(a0,b0)
        Loss_0.append(loss)
        if loss-p<1e-8:
            print("alpha={},beta={}:iteration={}".format(alpha,beta,iteration))
            break
        t=1
        while L(a0-t*g1,b0-t*g2)>L(a0,b0)-alpha*t*(g1**2+g2**2):
            t=t*beta
        a0-=t*g1
        b0-=t*g2
        iteration+=1
    name='beta='+str(beta)
    plt.plot(Loss_0,label=name)

plt.legend()
plt.ylabel("Loss")
plt.xlabel("iteration")
plt.title("iteration with different beta")
```

```
alpha=0.4,beta=0.1:iteration=39
alpha=0.4,beta=0.3:iteration=19
alpha=0.4,beta=0.45:iteration=19
alpha=0.4,beta=0.5:iteration=23
alpha=0.4,beta=0.8:iteration=14
alpha=0.4,beta=0.9:iteration=27
```

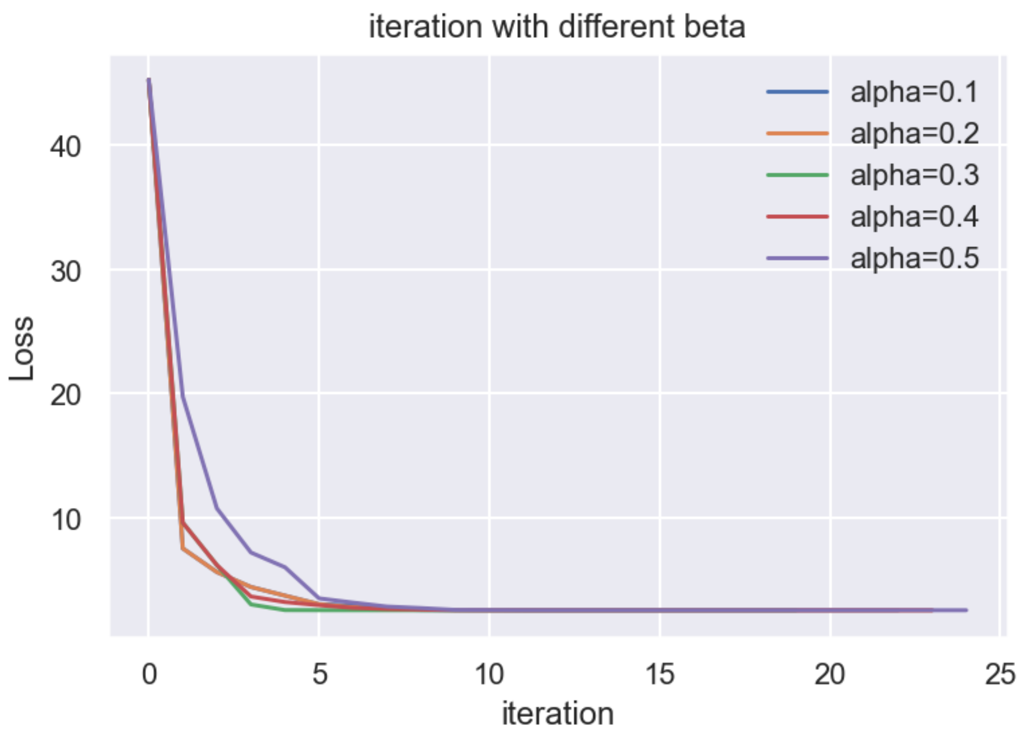
```
Text(0.5, 1.0, 'iteration with different beta')
```



```
alpha_arr=[0.1,0.2,0.3,0.4,0.5]
for alpha in alpha_arr:
    a0,b0=1.5,-0.8
    beta=0.5
    Loss_0=[]
    iteration=0

    while True:
        g1,g2=compute_grad(a0,b0)
        loss=L(a0,b0)
        Loss_0.append(loss)
        if loss-p<1e-8:
            print("alpha={},beta={}:iteration={}".format(alpha,beta,iteration))
            break
        t=1
        while L(a0-t*g1,b0-t*g2)>L(a0,b0)-alpha*t*(g1**2+g2**2):
            t=t*beta
        a0-=t*g1
        b0-=t*g2
        iteration+=1
    name='alpha='+str(alpha)
    plt.plot(Loss_0,label=name)

plt.legend()
plt.ylabel("Loss")
plt.xlabel("iteration")
plt.title("iteration with different beta")
```

```
alpha=0.1,beta=0.5:iteration=22
alpha=0.2,beta=0.5:iteration=23
alpha=0.3,beta=0.5:iteration=19
alpha=0.4,beta=0.5:iteration=23
alpha=0.5,beta=0.5:iteration=24
```

```
Text(0.5, 1.0, 'iteration with different beta')
```



> 在此问题中α, β 的参数组合我并没有发现很好的规律只要两个超参数的选择不是很离谱，在该问题中迭代的性能差异并不是很大

最速下降法

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation
from IPython.display import HTML,Image
import seaborn as sns
import time

%matplotlib inline
plt.style.use('seaborn-white')
```

算法流程

目标函数(object function)

$$L(x) = \exp(x_1 + 3x_2 - 0.1) + \exp(x_1 - 3x_2 - 0.1) + \exp(-x_1 - 0.1)$$

梯度计算 (compute gradient)

$$\frac{\partial L}{\partial x_1} = \exp(x_1 + 3x_2 - 0.1) + \exp(x_1 - 3x_2 - 0.1) - \exp(-x_1 - 0.1)$$
$$\frac{\partial L}{\partial x_2} = 3\exp(x_1 + 3x_2 - 0.1) - 3\exp(x_1 - 3x_2 - 0.1)$$

最速下降（quadratic norm)

- given a start point $x \in domL$,choose matrix P
- repeat
  - compute gradient $\nabla L(x)$
  - compute steepest descent direction $\Delta x_{sd} = -P^{-1}\nabla L(x)$
  - line search: choose learning rate t
  - update: $x := x + t\Delta x_{sd}$
- until convergence

回溯直线搜索(line search)

- initialize $\alpha \in (0, 0.5]$,$\beta \in [0, 1]$,$t = 1$
- while $L(x + t\Delta x_{sd}) > L(x) + \alpha t \nabla L(x)^T \Delta x_{sd}$
  - $t := \beta t$
- end

代码实现

函数的定义

```python
# object function
def L(x1,x2):
    return np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)+np.exp(-x1-0.1)
```

```python
# compute gradient
def compute_grad(x1,x2):
    g1=np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)-np.exp(-x1-0.1)
    g2=3.0*np.exp(x1+3*x2-0.1)-3.0*np.exp(x1-3.0*x2-0.1)
    return np.array([g1,g2])
```

```python
# line search
def condition(x,delta,g,t,alpha):
    temp=x+t*delta
    left=L(temp[0],temp[1])
    right=L(x[0],x[1])+alpha*t*np.dot(g,delta)
    return left>right

def linesearch(x,g,delta,beta,alpha):
    t=1
    while condition(x,delta,g,t,alpha):
        t=beta*t
    return t
```

```python
# steepest descent
def steepest(x,P,optimal_val,alpha,beta,Loss_history,x_history,y_history):
    loss=0.0
    step=0
    while np.abs(loss-optimal_val)>=1e-8: # 收敛条件
        g=compute_grad(x[0],x[1])
        delta=-np.linalg.inv(P).dot(g)
        loss=L(x[0],x[1])
        Loss_history.append(loss)
        print("第{}轮:Loss:{}".format(step,loss))

        lr=linesearch(x,g,delta,beta,alpha)
        step+=1
        x+=lr*delta

        x_history.append(x[0])
        y_history.append(x[1])
```

```python
# 初始化参数
x_0=np.array([1.5,-1.1])
```

最优值p^*

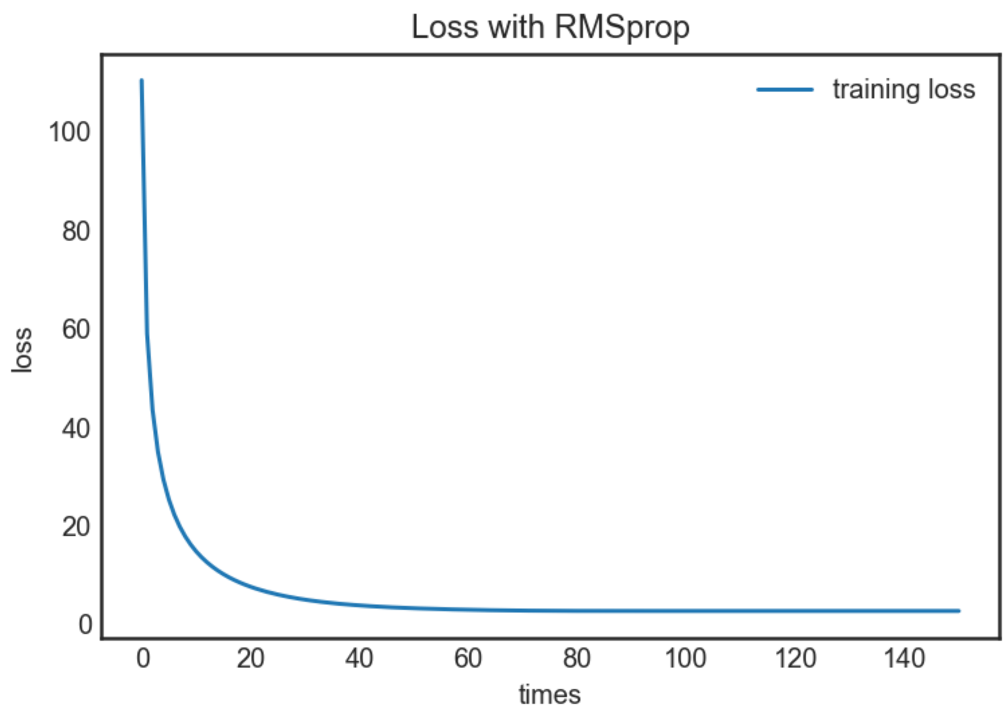
```python
# 先用RMSProp寻找到最优值p*,为后面的验证实验所用
# start point
x_1=x_0.copy()
# 超参数 [0.9,0.99,0.999]
decay_rate=0.9
cache=np.zeros(2)
```

```
# 数值上防止分母为0
eps=1e-8
learning_rate=0.05
# 记录历史数值，为以后的可视化用
Loss_1,a1_history,b1_history=[],[x_1[0]],[x_1[1]]
# 迭代次数
iteration=150
# 记录运行时间
startTime=time.time()

for t in range(iteration):
    g=compute_grad(x_1[0],x_1[1])
    loss=L(x_1[0],x_1[1])
    Loss_1.append(loss)
    if t%10==0:
        print("第{}轮,Loss:{}".format(t,loss))

    cache=decay_rate*cache+(1-decay_rate)*np.square(g)
    x_1-=learning_rate*g /(np.sqrt(cache)+eps)

    a1_history.append(x_1[0])
    b1_history.append(x_1[1])

Loss_1.append(L(x_1[0],x_1[1]))
print('运行时长{} sec(s)'.format(time.time()-startTime))
```

```
第0轮,Loss:110.2986375893409
第10轮,Loss:14.657838241267232
第20轮,Loss:7.471903727682384
第30轮,Loss:4.854062749418031
第40轮,Loss:3.684256004110696
第50轮,Loss:3.1115100867923675
第60轮,Loss:2.801231783934415
第70轮,Loss:2.6356693196903223
第80轮,Loss:2.571738129366431
第90轮,Loss:2.5598461467083244
第100轮,Loss:2.5592690846126462
第110轮,Loss:2.559266696724308
第120轮,Loss:2.5592666966582156
第130轮,Loss:2.5592666966582156
第140轮,Loss:2.5592666966582156
运行时长0.005002260208129883 sec(s)
```

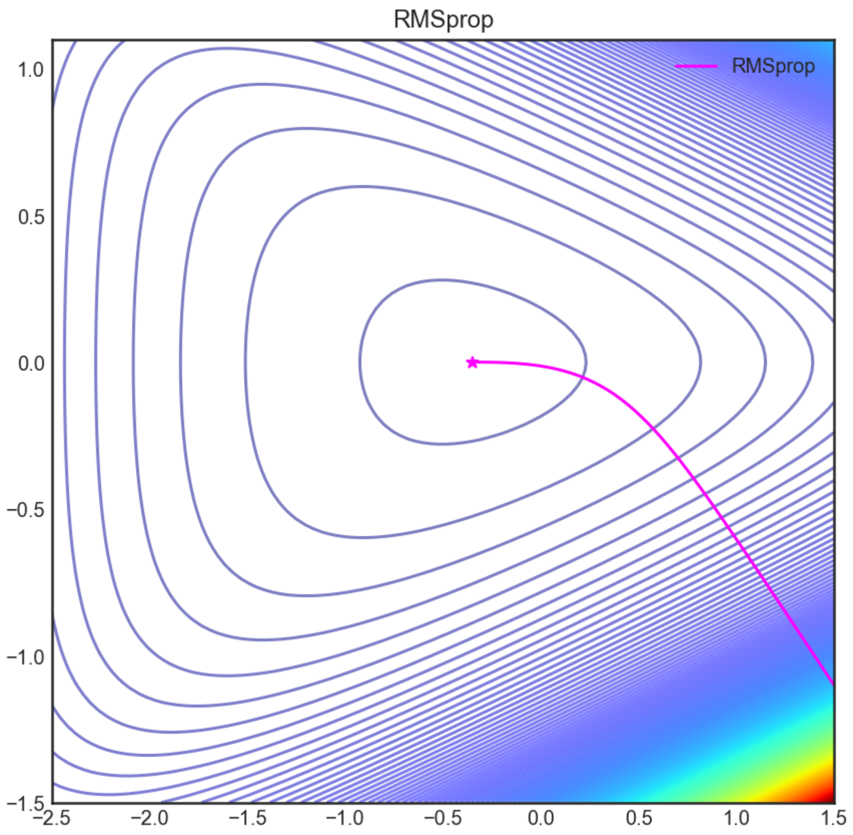
```
%config InlineBackend.figure_format='retina'
plt.plot(Loss_1,label='training loss')
plt.title('Loss with RMSprop')
plt.xlabel('times')
plt.ylabel('loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x204e8005888>
```



```
# 可视化
x,y=np.linspace(-2.5,1.5,100),np.linspace(-1.5,1.1,100)
X,Y=np.meshgrid(x,y)
l=L(X,Y)
fig1,ax1=plt.subplots(figsize=(7,7))
ax1.contour(X,Y,l,250,cmap='jet',alpha=0.5)
ax1.plot(a1_history,b1_history,'magenta',label='RMSprop',lw=1.5)
ax1.plot(a1_history[-1],b1_history[-1],'*',color='magenta',markersize=6)
plt.title('RMSprop')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x204e9299dc8>
```



$p^* = 2.5592666966582156$

```
optimal_val=Loss_1[-1]
optimal_val
```

```
2.5592666966582156
```

验证部分

(1) P的选择

$$P = \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix} or \begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix}$$

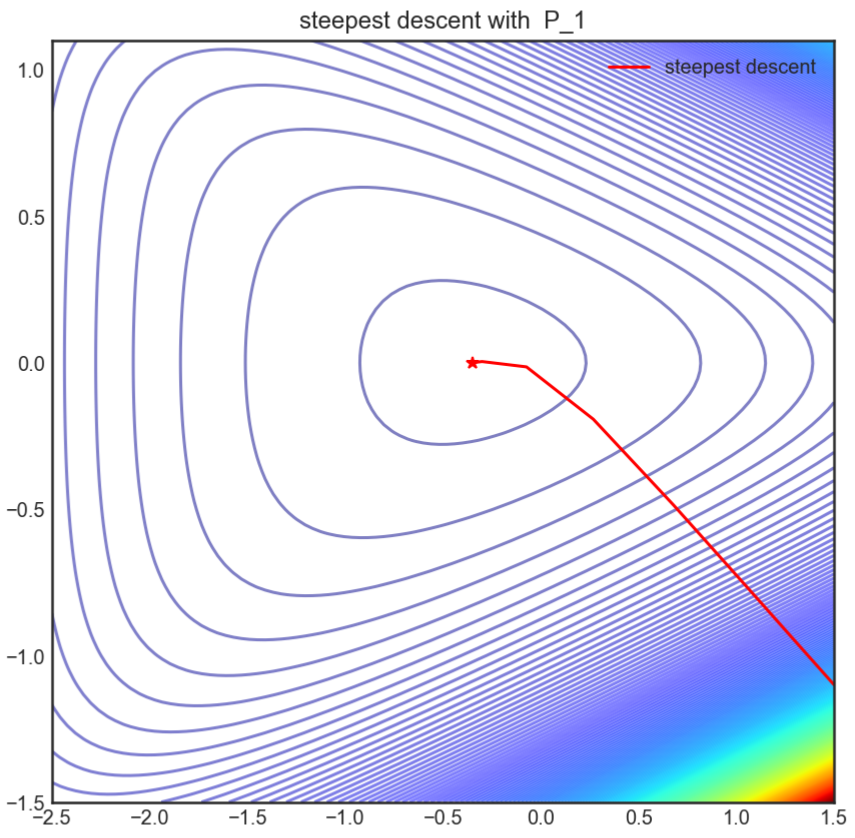
```
alpha,beta=0.5,0.8
```

```
#运行时间
startTime=time.time()

P=np.array([[2,0],[0,8]])
x_2=x_0.copy()
loss_2,a2_history,b2_history=[],[x_2[0]],[x_2[1]]
steepest(x_2,P,optimal_val,alpha,beta,loss_2,a2_history,b2_history)
loss_2.append(L(x_2[0],x_2[1]))
print("运行时间:{}sec(s)",format(time.time()-startTime))
```

```
fig2,ax2=plt.subplots(figsize=(7,7))
ax2.contour(X,Y,l,250,cmap='jet',alpha=0.5)
ax2.plot(a2_history,b2_history,'red',label='steepest descent',lw=1.5)
ax2.plot(a2_history[-1],b2_history[-1],'*',color='red',markersize=6)
plt.title('steepest descent with  P_1')
plt.legend()
```

```
第0轮:Loss:110.2986375893409
第1轮:Loss:30.013640056379245
第2轮:Loss:8.59421439206315
第3轮:Loss:3.4690257601691976
第4轮:Loss:2.6571941838349162
第5轮:Loss:2.5620525518793507
第6轮:Loss:2.5593553639676285
第7轮:Loss:2.559269600186121
第8轮:Loss:2.5592667918070147
第9轮:Loss:2.559266699776521
运行时间:{}sec(s) 0.006972789764404297
```

```
<matplotlib.legend.Legend at 0x204e9999c08>
```



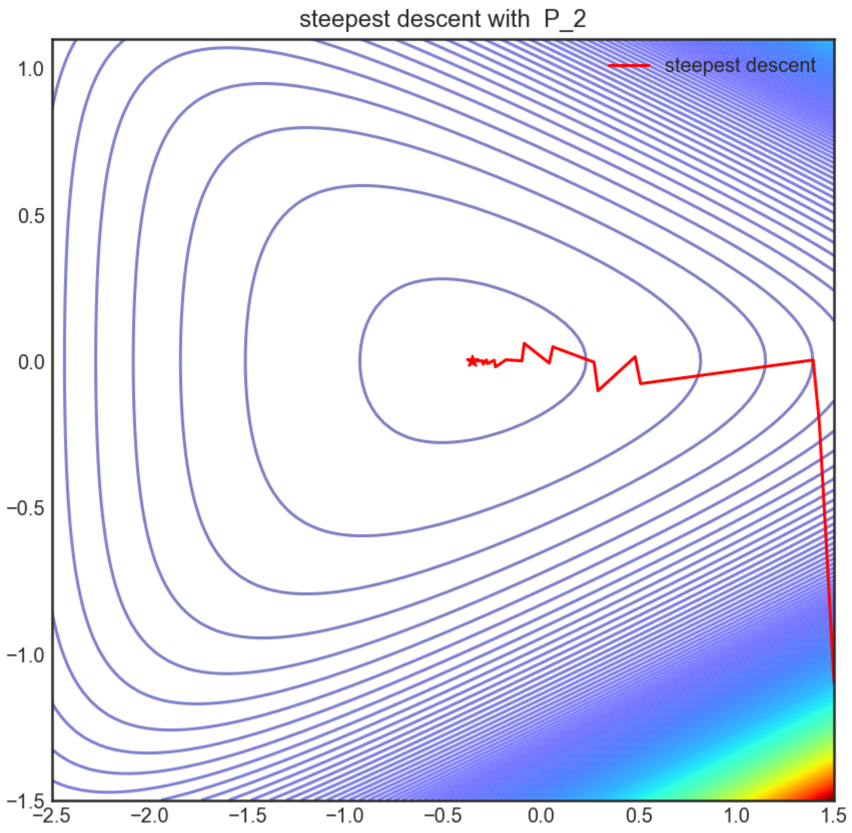
```
#
startTime=time.time()
```

```
P=np.array([[8,0],[0,2]])
x_3=x_0.copy()
loss_3,a3_history,b3_history=[],[x_3[0]],[x_3[1]]
steepest(x_3,P,optimal_val,alpha,beta,loss_3,a3_history,b3_history)
loss_3.append(L(x_3[0],x_3[1]))
print("运行时间{}sec(s)".format(time.time()-startTime))


fig3,ax3=plt.subplots(figsize=(7,7))
ax3.contour(X,Y,l,250,cmap='jet',alpha=0.5)
ax3.plot(a3_history,b3_history,'red',label='steepest descent',lw=1.5)
ax3.plot(a3_history[-1],b3_history[-1],'*',color='red',markersize=6)
plt.title('steepest descent with  P_2')
plt.legend()
```

```
第0轮:Loss:110.2986375893409
第1轮:Loss:24.542458934837875
第2轮:Loss:9.177617624920112
第3轮:Loss:7.51203469973362
第4轮:Loss:3.6409366725852705
第5轮:Loss:3.493385866784929
第6轮:Loss:3.2157923683581724
第7轮:Loss:3.0658969066332706
第8轮:Loss:2.7950469964730016
第9轮:Loss:2.7566932391681065
第10轮:Loss:2.6739894373553907
第11轮:Loss:2.639680505219266
第12轮:Loss:2.596079828281874
第13轮:Loss:2.5790244482624676
第14轮:Loss:2.5743851896365837
第15轮:Loss:2.566808840829111
第16轮:Loss:2.565377390738795
第17轮:Loss:2.5634040695499563
第18轮:Loss:2.5622660277795353
第19轮:Loss:2.5606588798240226
第20轮:Loss:2.5600456097973927
第21轮:Loss:2.559832064473344
第22轮:Loss:2.559677168451981
第23轮:Loss:2.5595766003285223
第24轮:Loss:2.5594769722309816
第25轮:Loss:2.5594188479926525
第26轮:Loss:2.559337540786589
第27轮:Loss:2.5593103183685995
第28轮:Loss:2.559295846779657
第29轮:Loss:2.5592803882680006
第30轮:Loss:2.5592767718579275
第31轮:Loss:2.559274183921147
第32轮:Loss:2.5592717046789106
第33轮:Loss:2.559270310713498
第34轮:Loss:2.5592691932480385
第35轮:Loss:2.5592684693965477
第36轮:Loss:2.559267528905025
第37轮:Loss:2.559267305864849
第38轮:Loss:2.5592671517664973
第39轮:Loss:2.5592669978075535
第40轮:Loss:2.5592669162752975
第41轮:Loss:2.55926684599056
```

```
第42轮:Loss:2.5592668026609333
第43轮:Loss:2.559266771949961
第44轮:Loss:2.5592667547053876
第45轮:Loss:2.5592667321797515
第46轮:Loss:2.5592667205499344
第47轮:Loss:2.5592667079327365
第48轮:Loss:2.5592667052501854
运行时间0.027323246002197266sec(s)
```

```
<matplotlib.legend.Legend at 0x204e9872c48>
```



> 在二次范数下，我们通过后面的学习可以知道P 好的选择是$\nabla^2 f(x)$，但实际中$\nabla^2 f(x)$ 的计算量十分大，所以找到一个特征值与$\nabla^2 f(x)$ 相近的矩阵P 代替一个不错的选择

牛顿法

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation
from IPython.display import HTML,Image
import seaborn as sns
import time

%matplotlib inline
plt.style.use('seaborn-white')
```

算法流程

目标函数(object function)

$$L(x) = \exp(x_1 + 3x_2 - 0.1) + \exp(x_1 - 3x_2 - 0.1) + \exp(-x_1 - 0.1)$$

梯度计算 (compute gradient)

$$\frac{\partial L}{\partial x_1} = \exp(x_1 + 3x_2 - 0.1) + \exp(x_1 - 3x_2 - 0.1) - \exp(-x_1 - 0.1)$$
$$\frac{\partial L}{\partial x_2} = 3\exp(x_1 + 3x_2 - 0.1) - 3\exp(x_1 - 3x_2 - 0.1)$$

Hession matrix ($\nabla^2 f(x)$)

$$\frac{\partial^2 L}{\partial x_1 \partial x_1} = \exp(x_1 + 3x_2 - 0.1) + \exp(x_1 - 3x_2 - 0.1) + \exp(-x_1 - 0.1)$$
$$\frac{\partial^2 L}{\partial x_1 \partial x_2} = 3\exp(x_1 + 3x_2 - 0.1) - 3\exp(x_1 - 3x_2 - 0.1)$$
$$\frac{\partial^2 L}{\partial x_2 \partial x_1} = 3\exp(x_1 + 3x_2 - 0.1) - 3\exp(x_1 - 3x_2 - 0.1)$$
$$\frac{\partial^2 L}{\partial x_2 \partial x_2} = 9\exp(x_1 + 3x_2 - 0.1) + 9\exp(x_1 - 3x_2 - 0.1)$$

牛顿法 (Newton Method)

- given a start point $x \in domL$
- repeat
    - compute gradient $\nabla L(x)$
    - compute newton direction $\Delta x_{nt} = -\nabla^2 L(x)^{-1} \nabla L(x)$
    - compute newton decrement $\lambda(x) = (\Delta x_{nt}^T \nabla^2 L(x) \Delta x_{nt})^{1/2}$
    - line search: choose learning rate t
    - update: $x := x + t\Delta x_{nt}$

回溯直线搜索(line search)

- initialize $\alpha \in (0, 0.5]$,$\beta \in [0, 1]$,$t = 1$
- while $L(x + t\Delta x_{sd}) > L(x) + \alpha t \nabla L(x)^T \Delta x_{sd}$
    - $t := \beta t$
- end

```python
# object function
def L(x1,x2):
    return np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)+np.exp(-x1-0.1)
```

```python
# compute gradient
def compute_grad(x1,x2):
    g1=np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)-np.exp(-x1-0.1)
    g2=3.0*np.exp(x1+3*x2-0.1)-3.0*np.exp(x1-3.0*x2-0.1)
    return np.array([g1,g2])
```

```python
def Hessian(x1,x2):
    h_11=np.exp(x1+3*x2-0.1)+np.exp(x1-3*x2-0.1)+np.exp(-x1-0.1)
    h_12=3*np.exp(x1+3*x2-0.1)-3*np.exp(x1-3*x2-0.1)
    h_21=h_12
    h_22=9*np.exp(x1+3*x2-0.1)+9*np.exp(x1-3*x2-0.1)
    return np.array([[h_11,h_12],[h_21,h_22]])
```

```python
# line search
def condition(x,delta,g,t,alpha):
    temp=x+t*delta
    left=L(temp[0],temp[1])
    right=L(x[0],x[1])+alpha*t*np.dot(g,delta)
    return left>right

def linesearch(x,g,delta,beta,alpha):
    t=1
    while condition(x,delta,g,t,alpha):
        t=beta*t
    return t
```

```python
# newton method with different parameter
def newton_method(x,beta,alpha,Loss_history,x_history,y_history):
    loss,step=0.0,0
    decrement=1
    eps=1e-8
    while (decrement/2>eps):
        # compute delta and decrement
        g=compute_grad(x[0],x[1])
        H=Hessian(x[0],x[1])
        H_1=np.linalg.inv(H)
        delta=-H_1.dot(g)
        decrement=(g.dot(H_1)).dot(g)
        lr=linesearch(x,g,delta,beta,alpha)
        # comupte loss
        loss=L(x[0],x[1])
        Loss_history.append(loss)
        print("第{}轮:Loss:{}".format(step,loss))
        step+=1
        # update
        x+=lr*delta

        x_history.append(x[0])
        y_history.append(x[1])
```

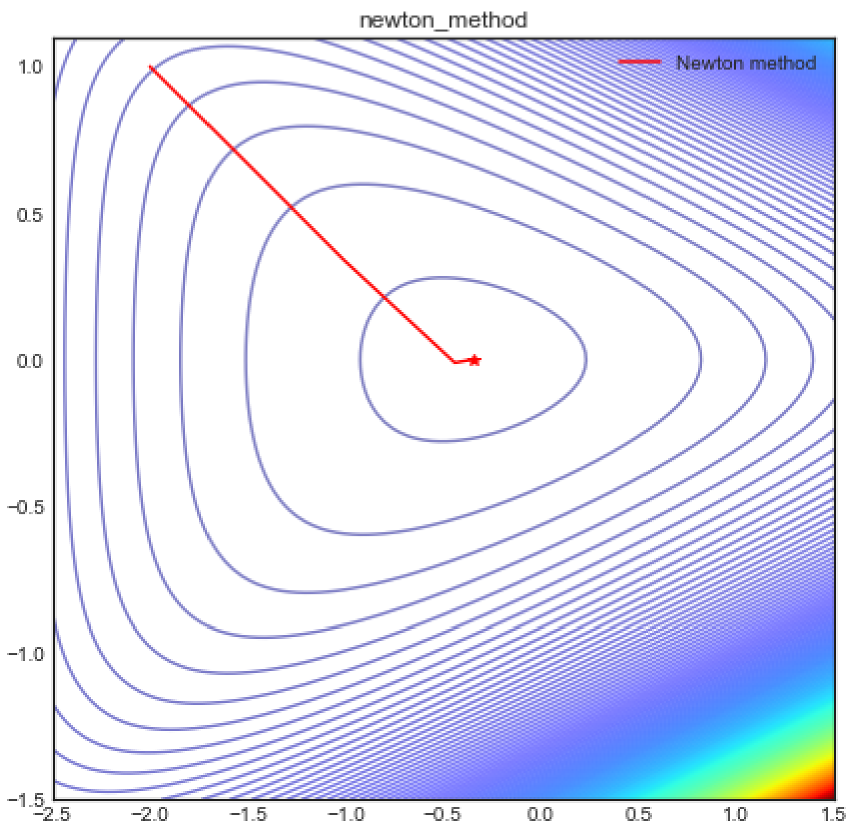
```
# 初始值
x_0=np.array([-2.0,1.0])
```

```
# 初始化参数
alpha,beta=0.5,0.8
```

```
startTime=time.time()

x,y=np.linspace(-2.5,1.5,100),np.linspace(-1.5,1.1,100)
X,Y=np.meshgrid(x,y)
l=L(X,Y)

x_1=x_0.copy()
loss_1,a1_history,b1_history=[],[x_1[0]],[x_1[1]]
newton_method(x_1,beta,alpha,loss_1,a1_history,b1_history)
loss_1.append(L(x_1[0],x_1[1]))
print("运行时间:{}sec(s)".format(time.time()-startTime))
fig1,ax1=plt.subplots(figsize=(7,7))
ax1.contour(X,Y,l,250,cmap='jet',alpha=0.5)
ax1.plot(a1_history,b1_history,'red',label='Newton method',lw=1.5)
ax1.plot(a1_history[-1],b1_history[-1],'*',color='red',markersize=6)
plt.title("newton_method")
plt.legend()
```

```
第0轮:Loss:9.151594300001733
第1轮:Loss:3.5108098694624488
第2轮:Loss:2.571211829697262
第3轮:Loss:2.559728937954584
第4轮:Loss:2.559285110722278
第5轮:Loss:2.559267432602447
第6轮:Loss:2.5592667260910007
第7轮:Loss:2.559266697835487
运行时间:0.005003929138183594sec(s)
```

```
<matplotlib.legend.Legend at 0x1a30db8d288>
```



> 牛顿法的优点就是收敛速度很快,而且在如此低维小规模的情况下其速度也不慢

第三次大作业

数据生成

```
import numpy

# 创造矩阵A
def creat_A(p, n):
    return numpy.random.random(size=(p, n))

# 创造可行解x
def creat_x(n):
    return numpy.random.random(size=(n, 1))
if __name__ == '__main__':
    # 保证A是行满秩
    numpy.random.seed(0)
    A = creat_A(30, 100)
    while numpy.linalg.matrix_rank(A) != 30:
        A = creat_A(30, 100)
    x_can = creat_x(100)
    b = numpy.dot(A, x_can)
    numpy.savetxt('A.txt', A)
    numpy.savetxt('x_can.txt', x_can)
    numpy.savetxt('b.txt', b)
```

牛顿法

```python
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation
from IPython.display import HTML,Image
import seaborn as sns
import time

%matplotlib inline
plt.style.use('seaborn-white')
```

等式约束熵极大化

$$\begin{aligned} \min \quad & f(x)=\sum_{i=1}^{n} x_i \log x_i \\ s.t. \quad & Ax=b \end{aligned}$$

其中$domf=\mathbb{R}^n_{++}, A\in \mathbb{R}^{p\times n}, p<n$

假设$n=100, p=30$

分别采用下列方法计算

- 标准Newton
- 不可行初始点的Newton 方法
- 对偶问题的Newton 方法

```python
# 全局变量 维数
n,p=100,30
```

```python
# 生成矩阵A和x,求出b
# 其中x作为初值
# 固定随机种子
np.random.seed(0)
x_0=np.random.uniform(size=(n,1))
# 寻找满秩矩阵
while True:
    A=np.random.random(size=(p,n)) # A的分布不要有负数
    if np.linalg.matrix_rank(A)==p:
        print('Rank(A)={}'.format(np.linalg.matrix_rank(A)))
        break
# 生成b
b=np.dot(A,x_0)
A_T=A.T
print(b.shape)
```

```
Rank(A)=30
(30, 1)
```

```python
# 定义f(x)

f=lambda x:np.sum(x*np.log(x))
```

standard Newton Method with equation constraint

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} x_i \log x_i \\ s.t. \quad & Ax = b \end{aligned}$$

$\nabla f(x)$

$$\frac{\partial f}{\partial x_i} = \log x_i + 1$$

```
# 梯度

grad=lambda x:np.log(x)+1
```

$\nabla^2 f(x)$

$$\frac{\partial f}{\partial x_i \partial x_j} = 0$$
$$\frac{\partial f}{\partial x_i \partial x_i} = \frac{1}{x_i}$$

```
# Hessian matrix
def H(x): # input(100,1)=>(100,100)
    temp=np.reshape(1.0/x,n)
    return np.diag(temp)
```

$\lambda^2(x) = v^T \nabla^2 f(x) v$

```
# Newton decrement
# 计算的为lambda的平方
# def dec(x,v):
#     return np.dot(np.dot(v.T,H(x)),v)
dec=lambda x,v:np.dot(np.dot(v.T,H(x)),v)
```

```
# 回溯直线搜索
def lineSearch(x,d,Decr): # x:迭代点,d:方向, dec:Newton decrement
    alpha,beta,t=0.5,0.8,1
    while (f(x+t*d)>f(x)-alpha*t*Decr):
        t=t*beta
    return t
```

算法流程

给定起始点x, ϵ
Repeat

- 解KKT 方程=> $\Delta x_{nt}, \lambda(x)$
- quit if $\lambda^2/2 \leq \epsilon$
- line search 确定步长t
- Update: $x := x + t\Delta x_{nt}$

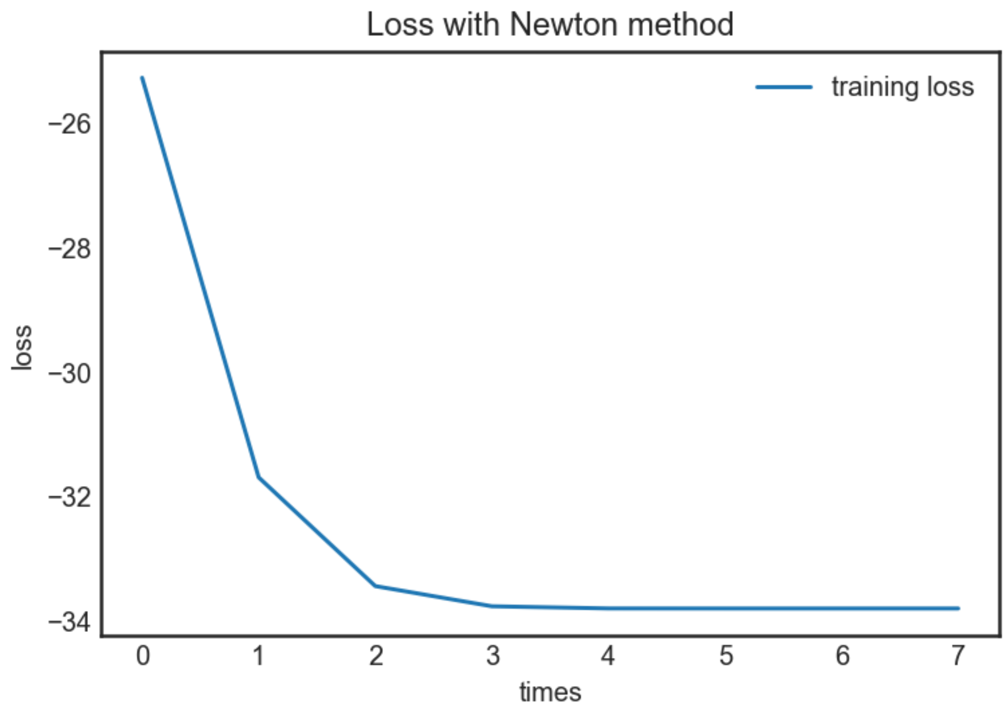
```
# given x,epsilon
# 这一部分全部使用np.narray
x_1=x_0.copy()
epsilon=1e-20
# 迭代次数
step=0
#
Loss_1=[]
z1,z2=np.zeros((30,30)),np.zeros((30,1))
start_time=time.time()
while True:
    # 计算loss
    loss=f(x_1)
    Loss_1.append(loss)
    print("第{}轮:{}".format(step,loss))
    # 计算梯度
    g,Hessian=grad(x_1),H(x_1)
    # solve KKT equation
    W=np.bmat([[Hessian,A_T],[A,z1]]).A # ndim=(130,130)
    y=-np.bmat([[g],[z2]]).A
    ans=np.linalg.solve(W,y)
    # 原方向，对偶方向
    d,w=ans[:n],ans[n:]
    # Newton decrement
    decrment=dec(x_1,d)[0][0] # 此处应转换(1,1)=>num
    # stop criteria
    if 0.5*decrment<=epsilon:
        break
    t=lineSearch(x_1,d,decrment)
    # update
    x_1=x_1+t*d
    step+=1

print("运行时间:{}sec".format(time.time()-start_time))
print('最优解x:{}'.format(x_1.T))
```

```
第0轮:-25.28682676146397
第1轮:-31.697888382730685
第2轮:-33.441493955324795
第3轮:-33.76524461665182
第4轮:-33.79999421704181
第5轮:-33.8005789968245
第6轮:-33.80057915377152
第7轮:-33.80057915377154
运行时间:0.03899884223937988sec
最优解x:[[0.45166225 0.84568564 0.62153312 0.32612493 0.2820677  0.3639384
  0.6348756  0.41255929 0.52089115 0.3350072  0.34637185 0.60604723
  0.47780668 0.66914287 0.51982616 0.31537145 0.27842364 0.6054929
```

```
 0.46693882 0.5353154  0.54863546 0.56366831 0.50945812 0.46803076
 0.35828658 0.50910637 0.2741074  0.34986025 0.33905688 0.45523651
 0.36334229 0.46494663 0.34730977 0.42964977 0.44376248 0.74709176
 0.36196932 0.52321455 0.78111212 0.33303606 0.60925773 0.75305684
 0.53771784 0.29948453 0.56923248 0.36495248 0.35873151 0.45689283
 0.29889381 0.33187752 0.66805924 0.5996819  0.60377927 0.38187966
 0.58511276 0.4775522  0.49479036 0.51264297 0.40829874 0.47382847
 0.32141838 0.55714756 0.42809245 0.4909001  0.3921601  0.4333641
 0.49079635 0.2929885  0.59853906 0.48687892 0.63869254 0.36033579
 0.53328371 0.44875958 0.3741046  0.35570344 0.36126836 0.60781606
 0.45106908 0.60035148 0.46111003 0.50843291 0.30237889 0.44347874
 0.49607919 0.3426815  0.40525825 0.35568381 0.44051307 0.46155512
 0.40318006 0.83399458 0.31007935 0.44541947 0.32324288 0.38854002
 0.76768556 0.33710145 0.60170978 0.41566571]]
```

```
%config InlineBackend.figure_format='retina'
plt.plot(Loss_1,label='training loss')
plt.title('Loss with Newton method')
plt.xlabel('times')
plt.ylabel('loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x20ff5180a88>
```



Starting from Infeasible Newton Method

```
# 读取数据
A=np.loadtxt('A.txt')
b=np.loadtxt('b.txt')
```

```
# 梯度
grad=lambda x:np.log(x)+1
# Hessian
def H(x): # input(100,1)=>(100,100)
    temp=np.reshape(1.0/x,n)
    return np.diag(temp)
# 原函数
f=lambda x:np.sum(x*np.log(x))
# 对偶残差与原残差
def r(x,v):
    rol_1=grad(x)+np.dot(A.T,v)
    rol_2=np.dot(A,x)-b
    return np.concatenate((rol_1,rol_2),axis=0)
#计算二范数
Norm=lambda x:np.linalg.norm(x)
```

```
# Newton direction
def d_nt(grad,hessian,v):
    zero1=np.zeros((30,30))
    # 拼接矩阵
    W_c1=np.concatenate((hessian,A),axis=0) # 纵向
    W_c2=np.concatenate((A.T,zero1),axis=0)
    W=np.concatenate((W_c1,W_c2),axis=1) # 横向

    b_new=-r(x,v)
    # d,w
    res=np.linalg.solve(W,b_new)
    return res[:n],res[n:]
```

算法(Infeasible start Newton method)

给定起始点$x \in dom f$,v,tolerance $\epsilon > 0, \alpha \in (0, 1/2)$

Repeat

- 计算原对偶牛顿方向$\Delta x_{nt}, \Delta v_{nt}$
- 对$\|r\|_2$ 直线回溯搜索
  - $t := 1$
  - **while** $\|r(x + t\Delta x_{nt}, v + t\Delta v_{nt})\|_2 > (1 - \alpha t)\|r(x, v)\|_2$
- 更新:$x := x + t\Delta x_{nt}, v := v + t\Delta v_{nt}$

Until $Ax = b$ and $\|r(x, v)\|_2 \leq \epsilon$

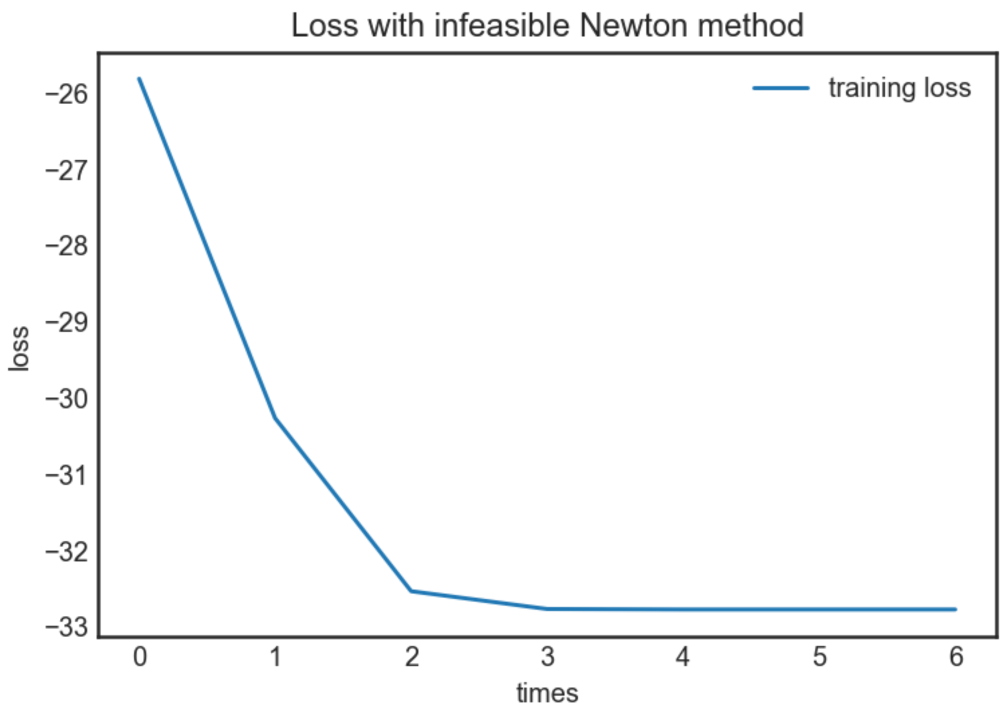
```
# 随机初始化参数
x=np.random.uniform(size=(n,))
v=np.random.random(p)
# 超参数
```

```python
eps=1e-12
alpha,beta=0.2,0.5

# 原方向以及对偶方向
Hessian=H(x)
g=grad(x)
dk,dv=d_nt(g,Hessian,v)
step=0
start_time=time.time()
loss_hist=[]
while (Norm(r(x,v))>eps) and ((np.dot(A,x)-b).any()>1e-4):
    loss=f(x)
    print("第{}轮r(x,v)={},loss={}".format(step,Norm(r(x,v)),loss))
    loss_hist.append(loss)
    # line Search
    t=1.0
    while Norm(r(x+t*dk,v+t*dv))>(1-alpha*t)*Norm(r(x,v)):
        t=beta*t
    x+=t*dk
    v+=t*dv
    Hessian=H(x)
    g=grad(x)
    dk,dv=d_nt(g,Hessian,v)
    step+=1
print("最优解x={}".format(x))
print("最优值p={}".format(f(x)))
print("计算时间={}sec".format(time.time()-start_time))
```

```
第0轮r(x,v)=69.57356345900297,loss=-25.82140130626489
第1轮r(x,v)=5.476387539373369,loss=-30.27382509702816
第2轮r(x,v)=1.3467250832035884,loss=-32.54316575001707
第3轮r(x,v)=0.19819709552704912,loss=-32.77664323634768
第4轮r(x,v)=0.007322873236739119,loss=-32.78268432720578
第5轮r(x,v)=1.2252356910121065e-05,loss=-32.782692895293614
第6轮r(x,v)=3.8308966240138544e-11,loss=-32.782692895318014
最优解x=[0.61240132 0.5149147  0.60478261 0.68467012 0.65637014 0.46460761
 0.55790405 0.51210362 0.41623328 0.82888925 0.34576797 0.48288991
 0.30172109 0.39117254 0.29953613 0.41954109 0.6530249  0.60804933
 0.40469509 0.52827795 0.77095639 0.62488936 0.54991489 0.38838572
 1.08907977 0.41769904 0.4056318  0.48302407 0.75967002 0.4091075
 0.49851168 0.63867099 0.36740862 0.41175233 0.47032237 0.30919096
 0.60444848 0.42247143 0.44322015 0.47864496 0.36577649 0.38463484
 0.47889732 0.57331565 0.4679424  0.90465925 0.40695724 0.40546196
 0.75938205 0.49866763 0.36734444 0.3583664  0.5426326  0.49064586
 0.4481237  0.34757991 0.5333863  0.65137441 0.42117122 0.37181017
 0.42410897 0.46093533 0.60132985 0.39720543 0.543493   0.61850285
 0.73593926 0.43983627 0.63974375 0.35688797 0.4462899  0.40709589
 0.47427926 0.43790682 0.54249449 0.4389736  0.54416886 0.41345152
 0.58472043 0.42030161 0.49273256 0.34177592 0.58823503 0.38233969
 0.45626731 0.43123299 0.36820406 0.51290433 0.6717     0.44097957
 0.68196377 0.53515258 0.61523883 0.52864123 0.65772349 0.5190426
 0.52485979 0.53073706 0.53973349 0.39359532]
最优值p=-32.782692895318014
计算时间=0.03200030326843262sec
```

```
%config InlineBackend.figure_format='retina'
plt.plot(loss_hist,label='training loss')
plt.title('Loss with infeasible Newton method')
plt.xlabel('times')
plt.ylabel('loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x20ff6402a48>
```



```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
# 对偶函数可行解，由v计算x
def back(v):
    x = np.zeros(100)
    for i in range(100):
        x[i] = np.exp(-np.dot(v, A[:, i]) - 1)
    return x


# 代入可行解，求得对偶函数
def f(v, b, A):
    result = -np.dot(b, v)
    for i in range(100):
        result = result - np.exp(-np.dot(v, A[:, i])-1)
    return -result


# 计算梯度
def grad(v, b, A):
    result = np.zeros(30)
    for i in range(30):
        s = 0
        for k in range(100):
            s = s + A[i][k]*np.exp(-np.dot(v, A[:, k])-1)
        result[i] = -b[i] + s
    return -result


# 计算Hessian矩阵
def hessian(v, A):
    result = np.zeros((30, 30))
    for i in range(30):
        for j in range(30):
            h = 0
            for k in range(100):
                h = h + A[i][k]*A[j][k]*np.exp(-np.dot(v, A[:, k])-1)
            result[i][j] = -h
    return -result
```

```
#牛顿方向
dnt=lambda hessian,grad:-np.dot(np.linalg.inv(hessian),grad)
# 牛顿减少量
lambda_2=lambda dk,hessian:np.dot(np.dot(dk,hessian),dk.T)
```

牛顿法 (Newton Method)

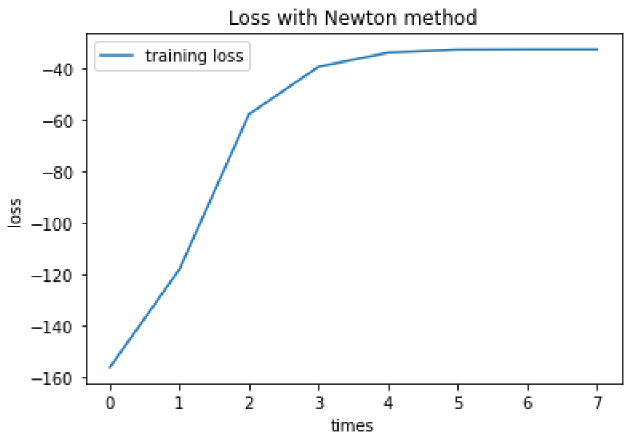
- given a start point $x \in domL$
- repeat
  - compute gradient $\nabla L(x)$
  - compute newton direction $\Delta x_{nt} = -\nabla^2 L(x)^{-1} \nabla L(x)$
  - compute newton decrement $\lambda(x) = (\Delta x_{nt}^T \nabla^2 L(x) \Delta x_{nt})^{1/2}$
  - line search: choose learning rate t
  - update: $x := x + t\Delta x_{nt}$

回溯直线搜索(line search)

- initialize $\alpha \in (0, 0.5]$,$\beta \in [0, 1]$,$t = 1$
- while $L(x + t\Delta x_{sd}) > L(x) + \alpha t \nabla L(x)^T \Delta x_{sd}$
  - $t := \beta t$
- end

```
import time
if __name__ == '__main__':
    A = np.loadtxt('A.txt')
    b = np.loadtxt('b.txt')
    # 1.给定初始v和误差值
    v = np.ones(30)*0.2
    e = 1e-16
    # 2.确定牛顿方向和牛顿减少量
    hessian_v = hessian(v, A)
    grad_v = grad(v, b, A)
    dk = dnt(hessian_v, grad_v)
    lambda_k2 = lambda_2(dk, hessian_v)

    step=0
    Loss_1=[]
    # 3.判断是否停止
    start_time=time.time()
    while lambda_k2/2 > e:
        #
        loss = -f(v,b,A)
        Loss_1.append(loss)
        print("第{}轮:loss={}".format(step,loss))
        step+=1
        # 4.回溯直线搜索
        alpha = 0.2
        beta = 0.5
        t = 1
        while f(v+t*dk, b, A) > f(v, b, A)-alpha*t*lambda_k2:
            t = beta*t
        # 5. 更新
        v = v + t*dk
        hessian_v = hessian(v, A)
        grad_v = grad(v, b, A)
        dk = dnt(hessian_v, grad_v)
        lambda_k2 = lambda_2(dk, hessian_v)
    x = back(v)
    print("最优值:x={}".format(x))
    print("最优解:".format(-f(v, b, A)))
    print("计算时间:{}sec".format(time.time()-start_time))

    plt.plot(Loss_1,label='training loss')
    plt.title('Loss with Newton method')
    plt.xlabel('times')
    plt.ylabel('loss')
    plt.legend()
```

```
第0轮:loss=-156.10648835869313
第1轮:loss=-118.13581097077937
第2轮:loss=-57.94775939186362
第3轮:loss=-39.52813669621822
第4轮:loss=-34.06792473493707
第5轮:loss=-32.87195746922535
第6轮:loss=-32.7833023058429
第7轮:loss=-32.78269292711094
最优值:x=[0.61240132 0.5149147  0.60478261 0.68467012 0.65637014 0.46460761
 0.55790405 0.51210362 0.41623328 0.82888925 0.34576797 0.48288991
 0.30172109 0.39117254 0.29953613 0.41954109 0.6530249  0.60804934
 0.40469509 0.52827795 0.77095639 0.62488936 0.54991489 0.38838572
 1.08907978 0.41769904 0.4056318  0.48302407 0.75967002 0.4091075
 0.49851168 0.63867099 0.36740862 0.41175234 0.47032237 0.30919096
 0.60444848 0.42247143 0.44322015 0.47864496 0.36577649 0.38463484
 0.47889732 0.57331565 0.4679424  0.90465925 0.40695724 0.40546196
 0.75938206 0.49866763 0.36734444 0.3583664  0.5426326  0.49064586
 0.4481237  0.34757991 0.5333863  0.65137441 0.42117122 0.37181017
 0.42410897 0.46093533 0.60132985 0.39720543 0.543493   0.61850285
 0.73593926 0.43983627 0.63974375 0.35688797 0.4462899  0.40709589
 0.47427926 0.43790682 0.54249449 0.43897361 0.54416886 0.41345152
 0.58472043 0.42030161 0.49273256 0.34177592 0.58823503 0.38233969
 0.45626731 0.431233   0.36820406 0.51290433 0.6717     0.44097957
 0.68196377 0.53515258 0.61523883 0.52864123 0.65772349 0.5190426
 0.52485979 0.53073706 0.53973349 0.39359533]
最优解:
计算时间:3.2437829971313477sec
```



> 在numpy使用中尽量少用for循环，多用矩阵运算，这个比第一种方法慢好多
> 不过也有可能很大原因时间用在Hessian矩阵上的，这个比前一个的Hessian复杂好多

总结

这个应该是我这学期最上心的一次作业前前后后搞了一个多星期，害不说了，完结撒花，感谢陪伴