In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from math import log2, sqrt
```

• Implement the functions phi(x,t,Q,p,A,b), grad(x,t,Q,p,A,b) and hess(x,t,Q,p,A,b) which return respectively the function value, gradient and hessian of φt(x) at point x.

In [2]:
```python
def phi(x,t,Q,p,A,b):
    self_concordant=-np.sum([log2(b[i]-np.dot(A[i][:],x)) for i in range(len(x))])
    return t*(0.5*np.dot(x,np.dot(Q,x))+np.dot(p,x))+self_concordant
```

In [3]:
```python
def grad(x,t,Q,p,A,b):
    self_concordant_gradient=[np.sum([A[i][k]/(b[i]-np.dot(A[i][:],x)) for i in range(len(x))])
 for k in range(len(x))]
    return [t*(np.dot(Q[i][:],x)+p[i])+self_concordant_gradient[i] for i in range(len(x))]
```

In [4]:
```python
def hess(x,t,Q,p,A,b):
    self_concordant_hess=[[np.sum([(A[p][j]*A[p][i])/(b[p]-np.dot(A[p][:],x))**2 for p in range
(len(x))]) for i in range(len(x))] for j in range(len(x))]
    return [[t*Q[i][j]+self_concordant_hess[i][j] for i in range(len(x))] for j in range(len(x
))]
```

Please make sur that when calling those functions:
- x,p and b are lists of elements (i.e x=[1,2,3] or np.array([1,2,3]))
- Q and A are matrices (arrays) (i.e A=np.zeros((1,1)) or A=[[1]] when A is a scalar)).

• Implement the function [xnew,gap] = dampedNewtonStep(x,f,g,h), which compute the damped Newton step at point x.

In [7]:
```python
def dampedNewtonStep(x,f,g,h):
    gradient=g(x)
    hes=h(x)
    square_newton_decrement=np.dot(gradient,np.dot(np.linalg.inv(hes),gradient))
    x_new=list(x-(1/(1+sqrt(square_newton_decrement)))*np.dot(np.linalg.inv(hes),gradient))
    gap=square_newton_decrement/2
    return [x_new,gap]
```

• Implement the function [xstar,xhist] = dampedNewton(x0,f,g,h,tol) which min- imizes the function f starting at x0 using the damped Newton algorithm.

In [54]:
```python
def dampedNewton(x0,f,g,h,tol):
    xhist=[x0]
    [x_new,gap]=dampedNewtonStep(x0,f,g,h)
    xhist.append(x_new)
    while(gap>tol):
        [x_new,gap]=dampedNewtonStep(x_new,f,g,h)
        xhist.append(x_new)
    return [x_new,xhist]
```

• Implement a function [xstar,xhist] = newtonLS(x0,f,g,h,tol) which minimizes the function f starting at x0 using the Newton algorithm with backtracking line-search.

In [9]:
```python
def backtracking(x,f,g,direction,alpha,beta):
    t=1    # 0<alpha<0.5 and 0<beta<1
    f_x=f(x)
    gradient=g(x)
    while(f(np.array(x)+t*np.array(direction))>=f_x+alpha*t*np.dot(gradient,direction)):
        t=beta*t
    return t
```

In [10]:
```python
def newtonStep(x,f,g,h):
    gradient=g(x)
    hes=h(x)
    square_newton_decrement=np.dot(gradient,np.dot(np.linalg.inv(hes),gradient))
    step=list(-np.dot(np.linalg.inv(hes),gradient))
    gap=square_newton_decrement/2
    return [step,gap]
```

In [53]:
```python
def newtonLS(x0,f,g,h,tol):
    xhist=[x0]
    x=x0
    [step,gap]=newtonStep(x,f,g,h)
    xhist.append(x)
    while(gap>=tol):
        print(gap)
        t=backtracking(x,f,g,step,0.1,0.2)
        x=list(np.array(x)+t*np.array(step))
        [step,gap]=dampedNewtonStep(x,f,g,h)
        xhist.append(x)
    return [x,xhist]
```

In [282]:
```python
Q=[[10]]; p=[1]; A=[[2]]; b=[-1];t=0.001 # Declare some parameters t=0.001; # Set the barrier
 parameter
# Declare f as an anonymous function which takes one # input x, and returns phi(x,t,Q,p,A,b)
f = lambda x: phi(x,t,Q,p,A,b);
g = lambda x: grad(x,t,Q,p,A,b); # same
h = lambda x: hess(x,t,Q,p,A,b); # same
# Perform a Damped Newton Step at x=-1.
x0 =[-8];
print('For x0=[-8] the damped newton method returns:\n' ,dampedNewton(x0,f,g,h,0.01))
print('For x0=[-8] the newton method using backtracking line search returns:\n' ,newtonLS(x0,f
,g,h,0.01))
x0 =[-2];
print('For x0=[-2] the damped newton method returns:\n' ,dampedNewton(x0,f,g,h,0.01))
print('For x0=[-2] the newton method using backtracking line search returns:\n' ,newtonLS(x0,f
,g,h,0.01))
```

```
For x0=[-8] the damped newton method returns:
 [[-10.180161128926617], [[-8], [-9.4751131221719458], [-10.180161128926617]]]
For x0=[-8] the newton method using backtracking line search returns:
 [[-9.9559999999999995], [[-8], [-8], [-9.9559999999999995]]]
For x0=[-2] the damped newton method returns:
 [[-10.26449943792071], [[-2], [-2.7268556228748091], [-3.767785291471514], [-5.182692260430
4571], [-6.9142506123286118], [-8.6500000213849013], [-9.8521766500884169], [-10.26449943792
071]]]
For x0=[-2] the newton method using backtracking line search returns:
 [[-10.068226413660899], [[-2], [-2], [-3.4251833740831295], [-8.1536687132469226], [-10.068
226413660899]]]
```

• coding the generic functions [Q,p,A,b] = transform svm primal(tau,X,y) and [Q,p,A,b] = transform svm dual(tau,X,y), which writes the primal and the dual SVM problem as particular instances of the quadratic problem (QP).

In [12]:
```python
def transform_svm_primal(tau,X,y):
    n,d=np.shape(X)[0],np.shape(X)[1]
    Q=np.zeros((n+d,n+d))
    Q[:d,:d]=np.eye(d)
    p1,p2=list(np.zeros(d)),list((1/(tau*n))*np.ones(n))
    p=np.array(p1+p2)
    b1,b2=list(-np.ones(n)),list(np.zeros(n))
    b=np.array(b1+b2)
    A=np.zeros((2*n,n+d))
    for i in range(n):
        A[i,:d]=-y[i]*X[i]
        A[i,d:]=list(-np.eye(n)[i])
        A[n+i,:d]=0
        A[n+i,d:]=list(-np.eye(n)[i])
    return [Q,p,A,b]
```

In [13]:
```python
def transform_svm_dual(tau,X,y):
    n,d=np.shape(X)[0],np.shape(X)[1]
    Q=np.array([[y[i]*y[j]*np.dot(X[i],X[j].T) for i in range(n)] for j in range(n)])
    p=-np.ones(n)
    A=np.zeros((2*n,n))
    b1,b2=list((1/(tau*n))*np.ones(n)),list(np.zeros(n))
    b=np.array(b1+b2)
    for i in range(n):
        A[i,:]=list(np.eye(n)[i])
        A[n+i,:]=list(-np.eye(n)[i])
    return [Q,p,A,b]
```

See the handwriting for the justifications.

• coding a function [x sol,xhist] = barr method(Q,p,A,b,x 0,mu,tol) which implements the barrier method to solve QP given the inputs (Q, p, A, b) and the initial point x_0 (which should be strictly feasible).

In [14]:
```python
def barr_method(Q,p,A,b,x0,mu,tol):
    t=30
    f = lambda z: phi(z,t,Q,p,A,b)
    g = lambda z: grad(z,t,Q,p,A,b)
    h = lambda z: hess(z,t,Q,p,A,b)
    xhist=[x0]
    x_opt=dampedNewton(x0,f,g,h,tol)[0]
    xhist.append(x_opt)
    m=np.shape(A)[0]
    while(m>=t*tol):
        x_opt=dampedNewton(x_opt,f,g,h,tol)[0]
        xhist.append(x_opt)
        t=mu*t
    return[x_opt,xhist]
```

initializing with t=30 for fast convergence, we will use another small value as example but it takes much more time due to the expensive calculations

In [21]:
```python
names=['sepal_length','sepal_width','petal_length','petal_width','classes','offset']
df=pd.read_table('/Users/adilrhiulam/Downloads/iris.data.txt',sep=',',names=names)
offset=np.mean(df[['sepal_length','sepal_width','petal_length','petal_width']].values,axis=1)
df[['offset']]=offset
```

loading the data and adding the offset column

In [22]:
```python
data=df[(df.classes=='Iris-virginica') | (df.classes=='Iris-versicolor')]
X=data[['sepal_length','sepal_width','petal_length','petal_width','offset']].values
classes=data[['classes']].values
Y=[]
for i in range(len(classes)):
    if classes[i][0]=='Iris-versicolor':
        Y.append(1)
    else:
        Y.append(-1)
X_train=X[:int(80*len(X)/100)]
Y_train=Y[:int(80*len(X)/100)]
X_test=X[int(80*len(X)/100):]
Y_test=Y[int(80*len(X)/100):]
```

Train data represent 80% of data and test data 20%

In [15]:
```python
def accuracy(true_labels,predicted_labels):
    count=0
    for i in range(len(true_labels)):
        if true_labels[i]==predicted_labels[i]:
            count+=1
    return count/len(true_labels)
```

Accuracy function on predicted labels (for train or test data).

tau=80:

In [453]:
```python
[Q,p,A,b]=transform_svm_primal(80,X_train,Y_train)
```

In [454]:
```python
(n,d)=np.shape(X_train)
w=np.zeros(d)
w[0]=1
z=np.zeros(n)
for i in range(n):
    r=2-Y_train[i]*X_train[i][0]
    z[i]=np.max([0.5,r])
x0=np.array(list(w)+list(z))
```

Initializing with the strict feasible point. (see the handwritting answers for demo)

In [497]:
```python
[x_opt,xhist]=barr_method(Q,p,A,b,x0,20,0.1)
print('The optimal abscisse is for tau=80:',x_opt)
```

```
The optimal abscisse is for tau=80: [0.013041908427347391, 0.0067724632430134048, 0.00527713
63341868891, 0.00040171413413219061, 0.0063733042643190525, 21.799693997824495, 21.804867358
368078, 21.800071436404188, 21.817626606212297, 21.805488877588704, 11.546487653807036, 11.5
31551019934554, 11.570385213050157, 11.531792326118843, 11.558564617026182, 11.5708939144693
93, 11.543549581902404, 11.551150303500895, 11.538223388753869, 11.553298013324353, 11.52983
0234042928, 11.545879010193161, 11.549207407420143, 11.543790507220846, 11.554982198367952,
11.537155634540364, 11.544068795384986, 11.5370690629218, 11.539458975724818, 11.53678051009
7426, 11.532130335647965, 11.528128263173596, 11.525946023473614, 11.54086149968429, 11.5556
29786129186, 11.557969345773524, 11.55885589603357, 11.550182491860154, 11.538212876085929,
11.548806054605421, 11.536479101009721, 11.527569591302825, 11.542576460023792, 11.549026203
193309, 11.555359666322518, 11.551974406451524, 11.538073856062983, 11.550332024551036, 11.5
69758270156717, 11.550848893746011, 11.547075138685321, 11.547712213596208, 11.5397075552245
76, 11.568483225089357, 11.549235838767144, 11.852862402882087, 11.833144871769552, 11.86057
579136399, 11.845371374058194, 11.851307108447928, 11.87270273229089, 11.813778717383444, 1
1.864815932231808, 11.849253244611983, 11.869230896958927, 11.847771934589517, 11.8433001049
45659, 11.853437036055855, 11.829520648238383, 11.834978970800565, 11.848281015680291, 11.84
8447953647764, 11.881745437447556, 11.873280086776717, 11.830403961693577, 11.85834681788711
2, 11.829879815341743, 11.872980640098419, 11.83888889251619, 11.855857341879293, 11.8638010
06855114, 11.837574901757728, 11.838471582410232, 11.846596830636692, 11.860354748473201]
```

In [569]:
```python
w_estimated=np.array(x_opt)[:d]
Y_est_test=[int(np.sign(np.dot(w_estimated,X_test[i]))) for i in range(len(X_test))]
Y_est_train=[int(np.sign(np.dot(w_estimated,X_train[i]))) for i in range(len(X_train))]
print('The accuracy on train_data predicted for tau=80 is:',accuracy(Y_train,Y_est_train))
print('The accuracy on test_data predicted for tau=80 is:',accuracy(Y_test,Y_est_test))
```

```
The accuracy on train_data predicted for tau=80 is: 0.625
The accuracy on test_data predicted for tau=80 is: 0.0
```

Poor classification for that choice of tau, all predicted labels are incorrect for test data.

tau=50:

In [498]:
```python
[Q,p,A,b]=transform_svm_primal(50,X_train,Y_train)
(n,d)=np.shape(X_train)
w=np.zeros(d)
w[0]=1
z=np.zeros(n)
for i in range(n):
    r=2-Y_train[i]*X_train[i][0]
    z[i]=np.max([0.5,r])
x0=np.array(list(w)+list(z))
```

In [596]:
```python
[x_opt,xhist]=barr_method(Q,p,A,b,x0,20,0.1)
```

In [594]:
```python
print('The optimal abscisse is for tau=50, mu=20, t0=30:',m)
```

```
The optimal abscisse is for tau=50, mu=20, t0=30: [0.020900755016298272, 0.01085115233699657
6, 0.008467126030203635, 0.0006502723876276507, 0.010217324383746347, 13.737019070090195, 1
3.745441277333883, 13.737631697940293, 13.766238506326763, 13.746452434156986, 7.45429034738
6235, 7.430348885148202, 7.492603732624514, 7.430738080952353, 7.473650848879338, 7.49341612
6938129, 7.449583751225801, 7.461769223141055, 7.441043015333243, 7.465214618369771, 7.42759
5307197062, 7.45331383791624, 7.458655351681674, 7.449965176766656, 7.467911295301816, 7.439
3283992184625, 7.4504190440131906, 7.4391894408176755, 7.44302448437325, 7.438735534327371,
7.431281363685931, 7.424862861214176, 7.421361600219601, 7.4452723544331505, 7.4689542822440
31, 7.472699497238931, 7.474122103299902, 7.460218722410189, 7.441020116495696, 7.4580048536
44356, 7.438249152368327, 7.423968409710862, 7.448022186764046, 7.458363341463408, 7.4685137
36773697, 7.463085065928304, 7.440804612702562, 7.460457125040868, 7.491598773333478, 7.4612
82840780674, 7.455236148185847, 7.456256236180783, 7.443426550485544, 7.48955814575315, 7.45
8698930236691, 7.945427389177231, 7.91381448255769, 7.957785825931101, 7.933413390515108, 7.
942931088899503, 7.977228388975515, 7.882770013397695, 7.964583496615642, 7.939637523012057,
7.971660757420737, 7.937256244901077, 7.930091825650259, 7.946340717246974, 7.90800619189562
2, 7.916757332642765, 7.938076408760151, 7.938342810368234, 7.991720890394056, 7.97815906204
8121, 7.909418770957631, 7.954212534666724, 7.908580182601664, 7.97767448626441, 7.923017273
840223, 7.9502211450908575, 7.962953195896455, 7.920910167859679, 7.9223479016476555, 7.9353
79731828107, 7.957426887087927]
```

In [572]:
```python
w_estimated=np.array(x_opt)[:d]
Y_est_test=[int(np.sign(np.dot(w_estimated,X_test[i]))) for i in range(len(X_test))]
Y_est_train=[int(np.sign(np.dot(w_estimated,X_train[i]))) for i in range(len(X_train))]
print('The accuracy on train_data predicted for tau=50 is:',accuracy(Y_train,Y_est_train))
print('The accuracy on test_data predicted for tau=50 is:',accuracy(Y_test,Y_est_test))
```

```
The accuracy on train_data predicted for tau=50 is: 0.625
The accuracy on test_data predicted for tau=50 is: 0.0
```

Poor classification, we should decrease tau.

tau=5 with the use of the dual problem

In [518]:
```python
tau=5
[Q,p,A,b]=transform_svm_dual(tau,X_train,Y_train)
(n,d)=np.shape(X_train)
lamda_feasible=[1/(2*tau*n) for i in range(n)]
```

Strict feasible point for initialization for dual problem, see the handwriting for demo

In [521]:
```python
[x_opt,xhist]=barr_method(Q,p,A,b,lamda_feasible,20,0.1)
print('The optimal value for the dual problem is for tau=5:',x_opt)
```

```
The optimal value for the dual problem is for tau=5: [0.00063971717410782824, 0.000660330181
18423373, 0.00065435871884055496, 0.00070104755561441515, 0.00067011927624610588, 0.000693394
09265536456, 0.00066935448491317 01, 0.00069556727967519155, 0.00065858639855438553, 0.000701
46114701329857, 0.00070752856385115556, 0.00067710232765363671, 0.00067755235863393927, 0.00
06831547649050545, 0.00067064490967972139, 0.00064621022783225895, 0.00069667547891381085,
0.00067438107909259244, 0.00069533174125933667, 0.00068449720692348621, 0.00069525033762336 7
86, 0.00066467761874226252, 0.00069424718207708696, 0.00068099560625262546, 0.00065831894774
556049, 0.00065288350436301559, 0.00066139383398459289, 0.00067268868866656546, 0.0006840350
0620895397, 0.00066519133077343163, 0.00068833685153781412, 0.00068335163807998675, 0.000673
8767879669815, 0.00070833030363931515, 0.00070430389360498 6, 0.00067285016895863962, 0.00065
676644819775206, 0.00068121670272926129, 0.00068113332035734016, 0.00069570716697256658, 0.0
0070128941155789943, 0.00067772066493835 16, 0.00067921106896788781, 0.00069445800896056776,
0.00069186588855793703, 0.00067755960470936201, 0.00068266248448784 59, 0.0006660803312502445
7, 0.00068004666428637408, 0.00068269671946752061, 0.0009142001551058754, 0.0009156544070888
3097, 0.00093934938491979734, 0.00092591829468757151, 0.00092279962745211812, 0.000940057862
34533544, 0.00090099858828794339, 0.00094053729573785117, 0.00092545424382623614, 0.00094317
69591008235, 0.00094356959268133457, 0.00092878981553374119, 0.00093859886119255533, 0.00090
896880241312554, 0.00090890825604397296, 0.00093170520754516596, 0.00093541985556341563, 0.0
0095459570623471018, 0.00092622776670564446, 0.00092033153648525285, 0.00093804851550837343,
0.00091406854408693575, 0.00093878317613342035, 0.00093546309852298153, 0.000937880630549346
23, 0.00094925059656944413, 0.00093657831141155537, 0.00093577838748276892, 0.0009215608035 1
686002, 0.00095261411192304869]
```

In [583]:
```python
w_estimated=np.sum([x_opt[i]*Y_train[i]*X_train[i] for i in range(len(x_opt))],axis=0)
Y_est_test=[int(np.sign(np.dot(w_estimated,X_test[i]))) for i in range(len(X_test))]
Y_est_train=[int(np.sign(np.dot(w_estimated,X_train[i]))) for i in range(len(X_train))]
print('The accuracy on train_data predicted for tau=5 is:',accuracy(Y_train,Y_est_train))
print('The accuracy on test_data predicted for tau=5 is:',accuracy(Y_test,Y_est_test))
```

```
The accuracy on train_data predicted for tau=5 is: 0.625
The accuracy on test_data predicted for tau=5 is: 0.0
```

Poor classification, all predicted classes are incorrect.

tau=0.1

In [545]:
```python
tau=0.1
[Q,p,A,b]=transform_svm_dual(tau,X_train,Y_train)
(n,d)=np.shape(X_train)
lamda_feasible=[1/(2*tau*n) for i in range(n)]
```

In [591]:
```python
[x_opt,xhist]=barr_method(Q,p,A,b,lamda_feasible,20,0.1)
print('The optimal value for the dual problem is for tau=0.1, t0=30 and mu=20:',x_opt)
```

```
The optimal value for the dual problem is for tau=0.1, t0=30 and mu=20: [0.1117676349695147
6, 0.12041935457605729, 0.1208687854834351, 0.12272650049077397, 0.12197138119192312, 0.1226
1219454177777, 0.12198698391893892, 0.1200717473529444, 0.11952718667731341, 0.1226685779939
8208, 0.12212486863108546, 0.12174205509648069, 0.11974859681970855, 0.12248474764912441, 0.
11680839477908, 0.11307118028943468, 0.12294618822542447, 0.11810890303225778, 0.12313695212
175443, 0.12070651086895029, 0.1233144055754078, 0.11814250003606201, 0.12322233820701503,
0.121969504001195, 0.11773080704203531, 0.1177136715668027, 0.12127320346231887, 0.122775344
30714454, 0.12253396318855725, -2.5469951975894345, 0.12105208575873212, 0.1192279515018007
8, 0.11922331496205832, 0.1235787160382089, 0.12315095931034892, 0.12187766212835499, 0.1206
5085910908438, 0.12219311177416368, 0.1211649092750355, 0.12241565894125365, 0.1227556865123
5841, 0.12204226997706408, 0.12070053365864904, 0.12005936623853927, 0.12231997734716743, 0.
12045023888306094, 0.12159013211100546, 0.11977774613748986, 0.11081382205324355, 0.12151194
982820107, 0.04853726985146633, 0.12165476555267003, 0.12230113781075583, 0.1222866045822705
2, 0.11986556764462422, 0.12135242950105883, 0.12161686324813281, 0.12254283192516165, 0.121
71617787673322, 0.12168930491922028, 0.12325447385597783, 0.12246608211421256, 0.12262889996
498208, 0.12012356688037426, 0.11392198311477413, 0.12204847952557668, 0.1229250497325741,
0.12268885655067446, -0.3328908583903212, 0.12273610797744035, 0.12207183391808724, 0.121557
52851323466, 0.12118628304284071, 0.12317121261592776, 0.12254957059322003, 0.12320930821479
568, 0.12327385978863255, 0.12325250088459296, 0.12061351896198315, 0.12349768315771971]
```

In [582]:
```python
w_estimated=np.sum([x_opt[i]*Y_train[i]*X_train[i] for i in range(len(x_opt))],axis=0)
Y_est_test=[int(np.sign(np.dot(w_estimated,X_test[i]))) for i in range(len(X_test))]
Y_est_train=[int(np.sign(np.dot(w_estimated,X_train[i]))) for i in range(len(X_train))]
print('The accuracy on train_data predicted for tau=0.1 is:',accuracy(Y_train,Y_est_train))
print('The accuracy on test_data predicted for tau=0.1 is:',accuracy(Y_test,Y_est_test))
```

```
The accuracy on train_data predicted for tau=0.1 is: 0.975
The accuracy on test_data predicted for tau=0.1 is: 0.9
```

Good performance, then we deduce that tau should be choosen small for good accuracy.

In [16]:
```python
def barr_method_bis(Q,p,A,b,x0,mu,tol):
    t=1
    f = lambda z: phi(z,t,Q,p,A,b)
    g = lambda z: grad(z,t,Q,p,A,b)
    h = lambda z: hess(z,t,Q,p,A,b)
    xhist=[x0]
    x_opt=dampedNewton(x0,f,g,h,tol)[0]
    xhist.append(x_opt)
    m=np.shape(A)[0]
    while(m>=t*tol):
        x_opt=dampedNewton(x_opt,f,g,h,tol)[0]
        xhist.append(x_opt)
        t=mu*t
    return [x_opt,xhist]
```

Using t=1 instead of 30 in this function

In [586]:
```python
tau=0.1
[Q,p,A,b]=transform_svm_dual(tau,X_train,Y_train)
(n,d)=np.shape(X_train)
lamda_feasible=[1/(2*tau*n) for i in range(n)]
```

In [589]:
```python
[x_opt,xhist]=barr_method_bis(Q,p,A,b,lamda_feasible,15,0.01)
print('The optimal value for the dual problem is for tau=0.1, t0=1 and mu=14:',x_opt)
```

```
The optimal value for the dual problem is for tau=0.1, t0=1 and mu=14: [0.12267070677020703,
0.12419875051778635, 0.12427022980985092, 0.12459753336850518, 0.1244648257821923, 0.1245779
6294084364, 0.12447224841812642, 0.12413036678312782, 0.12402760527931571, 0.124590264311786
16, 0.12448869230442362, 0.12442937552787701, 0.12405283174146445, 0.12455554472917268, 0.12
358633582800146, 0.12291715392313789, 0.12463877039628271, 0.12376871230993144, 0.1246694833
4392386, 0.12423783804950025, 0.12470415331699007, 0.12379463647962621, 0.12468478590091429,
0.12446156728977073, 0.12371601898444473, 0.1237187224864712, 0.12433756914070489, 0.1246077
0077450939, 0.12456566796697367, -2.6682124794421553, 0.12429925669581006, 0.123970976379794
72, 0.12398088229704764, 0.12474865817420255, 0.12467477987285555, 0.12445557156601855, 0.12
423435768693658, 0.12450001122641413, 0.12432754152313936, 0.12454347875629616, 0.1246023724
964964, 0.12447848372454924, 0.12423879144515208, 0.12412568365995107, 0.12452690482993892,
0.12419953900668135, 0.12439986117514175, 0.12407863463240992, 0.12256165636985608, 0.124385
75734256985, 0.056514333815858936, 0.12439516847306613, 0.12451558336233937, 0.1245138433454
3304, 0.12405681996446757, 0.12434773341913857, 0.12438717363076052, 0.12456370402278802, 0.
12441550127985992, 0.1243938920845019, 0.12468669357295317, 0.12454590607429784, 0.124573742
65976072, 0.12410822008138719, 0.12278244089598489, 0.12446181029793647, 0.1246290701641889
1, 0.12458470231542819, -0.35256522656845318, 0.12459828669075 35, 0.12446891477009089, 0.124
37294358134073, 0.12432282201669474, 0.12467323133375693, 0.12455817143993789, 0.12468113683
525044, 0.12469130747268631, 0.1246870158623069, 0.12420217349508017, 0.12473350845218489]
```

In [592]:
```python
w_estimated=np.sum([x_opt[i]*Y_train[i]*X_train[i] for i in range(len(x_opt))],axis=0)
Y_est_test=[int(np.sign(np.dot(w_estimated,X_test[i]))) for i in range(len(X_test))]
Y_est_train=[int(np.sign(np.dot(w_estimated,X_train[i]))) for i in range(len(X_train))]
print('The accuracy on train_data predicted for tau=0.1 is:',accuracy(Y_train,Y_est_train))
print('The accuracy on test_data predicted for tau=0.1 is:',accuracy(Y_test,Y_est_test))
```

```
The accuracy on train_data predicted for tau=0.1 is: 0.9625
The accuracy on test_data predicted for tau=0.1 is: 0.9
```

Very good performance

4. Plot the duality gap versus iterations (using the damped Newton Method) for the primal and dual problem in semilog-scale for different values of the barrier method parameter μ = 2, 15, 50, 100 and comment the results.

In [18]:
```python
def duality_gap(Q,p,A,b,x0,mu,tol):
    duality_gap=[]
    t=1
    f = lambda z: phi(z,t,Q,p,A,b)
    g = lambda z: grad(z,t,Q,p,A,b)
    h = lambda z: hess(z,t,Q,p,A,b)
    xhist=[x0]
    x_opt=dampedNewton(x0,f,g,h,tol)[0]
    xhist.append(x_opt)
    m=np.shape(A)[0]
    duality_gap.append(m/t)
    itera=0
    while(m>=t*tol):
        x_opt=dampedNewton(x_opt,f,g,h,tol)[0]
        xhist.append(x_opt)
        t=mu*t
        duality_gap.append(m/t)
        itera+=1
    return [itera,duality_gap]
```

The duality_gap function returns the number of inner iterations until the while criteria is attained and all the duality gap values over those iterations in a list.

In [23]:
```python
tau=0.1
[Q,p,A,b]=transform_svm_dual(tau,X_train,Y_train)
(n,d)=np.shape(X_train)
lambda_feasible=[1/(2*tau*n) for i in range(n)]
```

In [35]:
```python
dual_gap_dual_1=duality_gap(Q,p,A,b,lambda_feasible,2,0.01)
print('The evolution of the duality gap for the dual problem and mu=2 is :\n',dual_gap_dual_1[1
])
```

```
The evolution of the duality gap for the dual problem and mu=2 is :
 [160.0, 80.0, 40.0, 20.0, 10.0, 5.0, 2.5, 1.25, 0.625, 0.3125, 0.15625, 0.078125, 0.039062
5, 0.01953125, 0.009765625]
```

In [36]:
```python
dual_gap_dual_2=duality_gap(Q,p,A,b,lambda_feasible,15,0.01)
print('The evolution of the duality gap for the dual problem and mu=15 is :\n',dual_gap_dual_2[
1])
```

```
The evolution of the duality gap for the dual problem and mu=15 is :
 [160.0, 10.666666666666666, 0.7111111111111111, 0.047407407407407405, 0.003160493827160493
6]
```

In [37]:
```python
dual_gap_dual_3=duality_gap(Q,p,A,b,lambda_feasible,50,0.01)
print('The evolution of the duality gap for the dual problem and mu=50 is :\n',dual_gap_dual_3[
1])
```

```
The evolution of the duality gap for the dual problem and mu=50 is :
 [160.0, 3.2, 0.064, 0.00128]
```

In [38]:
```python
dual_gap_dual_4=duality_gap(Q,p,A,b,lambda_feasible,100,0.01)
print('The evolution of the duality gap for the dual problem and mu=100 is :\n',dual_gap_dual_4
[1])
```

```
The evolution of the duality gap for the dual problem and mu=100 is :
 [160.0, 1.6, 0.016, 0.00016]
```

In [28]:
```python
[Q,p,A,b]=transform_svm_primal(0.1,X_train,Y_train)
(n,d)=np.shape(X_train)
w=np.zeros(d)
w[0]=1
z=np.zeros(n)
for i in range(n):
    r=2-Y_train[i]*X_train[i][0]
    z[i]=np.max([0.5,r])
x0=np.array(list(w)+list(z))
```

In [39]:
```python
dual_gap_prim_1=duality_gap(Q,p,A,b,x0,2,0.01)
print('The evolution of the duality gap for the primal problem and mu=2 is :\n',dual_gap_prim_1
[1])
```

```
The evolution of the duality gap for the primal problem and mu=2 is :
 [160.0, 80.0, 40.0, 20.0, 10.0, 5.0, 2.5, 1.25, 0.625, 0.3125, 0.15625, 0.078125, 0.039062
5, 0.01953125, 0.009765625]
```

In [40]:
```python
dual_gap_prim_2=duality_gap(Q,p,A,b,x0,15,0.01)
print('The evolution of the duality gap for the primal problem and mu=15 is :\n',dual_gap_prim_
2[1])
```

```
The evolution of the duality gap for the primal problem and mu=15 is :
 [160.0, 10.666666666666666, 0.7111111111111111, 0.047407407407407405, 0.003160493827160493
6]
```

In [41]:
```python
dual_gap_prim_3=duality_gap(Q,p,A,b,x0,50,0.01)
print('The evolution of the duality gap for the primal problem and mu=50 is :\n',dual_gap_prim_
3[1])
```

```
The evolution of the duality gap for the primal problem and mu=50 is :
 [160.0, 3.2, 0.064, 0.00128]
```

In [42]:
```python
dual_gap_prim_4=duality_gap(Q,p,A,b,x0,100,0.01)
print('The evolution of the duality gap for the primal problem and mu=100 is :\n',dual_gap_prim
_4[1])
```

```
The evolution of the duality gap for the primal problem and mu=100 is :
 [160.0, 1.6, 0.016, 0.00016]
```

In [47]:
```python
dual_gap_prim_1==dual_gap_dual_1
```

Out[47]: True

In [50]:
```python
dual_gap_prim_2==dual_gap_dual_2
```

Out[50]: True

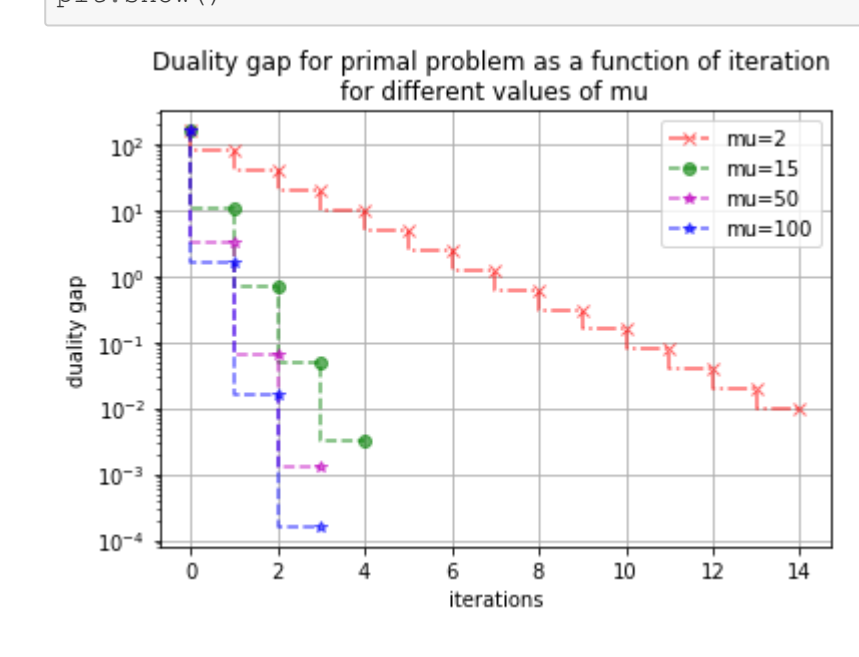
In [49]:
```python
dual_gap_prim_3==dual_gap_dual_3
```

Out[49]: True

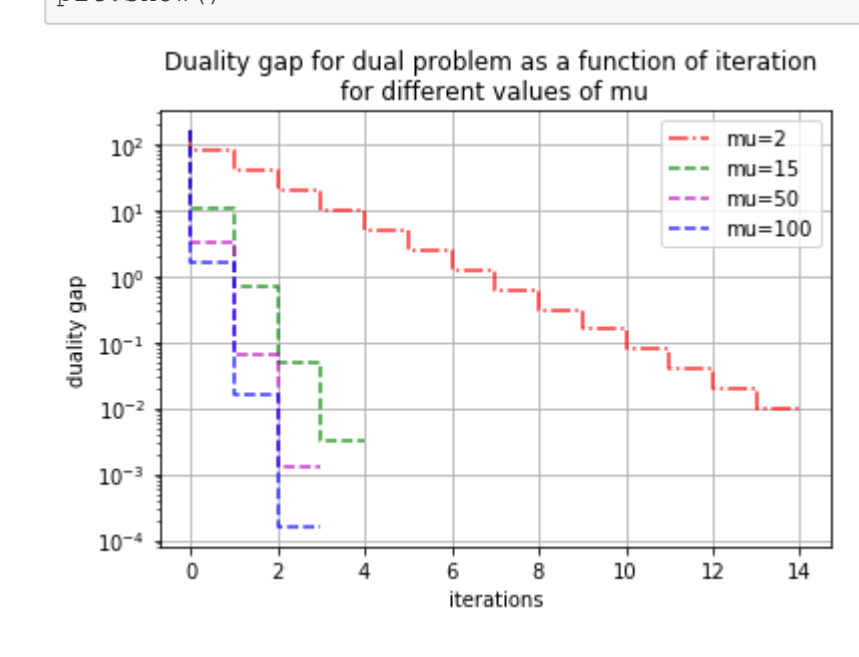
In [48]:
```python
dual_gap_prim_4==dual_gap_dual_4
```

Out[48]: True

In [46]:
```python
plt.step([x for x in range(dual_gap_prim_1[0]+1)],dual_gap_prim_1[1],c='r',linestyle='-.',marke
r='x',alpha=0.6,label="mu=2")
plt.step([x for x in range(dual_gap_prim_2[0]+1)],dual_gap_prim_2[1],c='g',linestyle='dashed',m
arker='o',alpha=0.6,label="mu=15")
plt.step([x for x in range(dual_gap_prim_3[0]+1)],dual_gap_prim_3[1],c='m',linestyle='dashed',m
arker='*',alpha=0.6,label="mu=50")
plt.step([x for x in range(dual_gap_prim_4[0]+1)],dual_gap_prim_4[1],c='b',linestyle='dashed',m
arker='*',alpha=0.6,label="mu=100")
plt.xlabel('iterations')
plt.ylabel('duality gap')
plt.legend()
plt.title('Duality gap for primal problem as a function of iteration \nfor different values of
 mu')
plt.yscale('log')
plt.grid()
plt.show()
```



In [44]:
```python
plt.step([x for x in range(dual_gap_dual_1[0]+1)],dual_gap_dual_1[1],c='r',linestyle='-.',alpha
=0.8,label="mu=2")
plt.step([x for x in range(dual_gap_dual_2[0]+1)],dual_gap_dual_2[1],c='g',linestyle='dashed',a
lpha=0.8,label="mu=15")
plt.step([x for x in range(dual_gap_dual_3[0]+1)],dual_gap_dual_3[1],c='m',linestyle='dashed',a
lpha=0.8,label="mu=50")
plt.step([x for x in range(dual_gap_dual_4[0]+1)],dual_gap_dual_4[1],c='b',linestyle='dashed',a
lpha=0.8,label="mu=100")
plt.xlabel('iterations')
plt.ylabel('duality gap')
plt.legend()
plt.title('Duality gap for dual problem as a function of iteration \nfor different values of m
u')
plt.yscale('log')
plt.grid()
plt.show()
```



The figure below show the evolution of the duality gap evolution with the number of inner iterations of the barrier method using as parameters: t0=1, tol=0.01 and tau=0.1 and diffrent values of mu. We can notice that the primal and dual problems give the same duality gap evolution, For mu=2 the condition of optimality approximation to respect is attained at the 14-th iteration while for mu=15 it's for 4-th iteration and 3-th iteration for mu=50 and mu=100, The width of each horizental step is the number of damped Newton steps calculated for that outer iteration. The height of each vertical step is equal to a multiple of mu because the duality gap is reduced by the factor mu at the end of each outer iteration.