Logistic Regression

To help you better understand the pros and cons of first- and second-order methods, we will look at Logistic Regression as an example.

Step 1: You already have the gradient descent algorithm (use the line search version from PA1). Now, code up the Newton's method algorithm as well. Write a function using the format below:

```
def nt(f,fp,fpp,y,A,xinit,maxit,tol):
  '''
  Note that you do no need to input a step size parameter
  fpp is the function  handle  of  the  Hessian
  '''
  x = xinit
  it = 0
  cur_tol = abs(f(y,A,x))
  prev_tol = 0.0
  while it < maxit and cur_tol >= tol:
    x -= np.dot(np.linalg.inv(fpp(y,A,x)),fp(y,A,x))
    prev_tol = cur_tol
    cur_tol = abs(f(y,A,x))
    cur_tol = abs((cur_tol-prev_tol)/prev_tol)
    it += 1
  return x, it
```

Note: we will implement the basic version of Newton's method (not BFGS or L-BFGS). We will also use the basic method to implement the Newton step by inverting the Hessian using np.linalg.inv. There are more efficient ways to do this inversion, but for the purpose of this assignment, do **NOT** use other ways even if you know them. The convergence criterion is the same as that in PA1.

Answer the questions and discuss your findings here

Step 2: Code the objective function, the gradient, and the **Hessian** (you can use the Python lambda tool just like in PA1, or the regular function environment if you don't like that). As a reminder, logistic regression has the following model

$$y_i \in \{0, 1\}, \quad p(y_i = 1) = \text{sigmoid}(\mathbf{a}_i^T \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{a}_i^T \mathbf{x}}}, \quad p(y_i = 0) = 1 - p(y_i = 1).$$

The likelihood can be written as

$$p(y_i \mid \mathbf{a}_i^T \mathbf{x}) = p(y_i = 1 \mid \mathbf{a}_i^T \mathbf{x})_i^y \cdot p(y_i = 0 \mid \mathbf{a}_i^T \mathbf{x})^{1-y_i}$$

So we are solving the following optimization problem that minimizes the total negative log-likelihood

minimize

$$\underset{\mathbf{x}}{\text{minimize}} \qquad -\sum_{i=1}^{M} (y_i \log \frac{1}{1 + e^{-\mathbf{a}_i^T \mathbf{x}}} + (1 - y_i) \log \frac{1}{1 + e^{\mathbf{a}_i^T \mathbf{x}}}),$$

or equivalently

$$minimize_{\mathbf{x}} \sum_{i=1}^{M} \log(1 + e^{-(2y_i - 1)a_i^T \mathbf{x}})$$

where y_i is the i^{th} entry of the observation vector $\mathbf{y}$ and $\mathbf{a}_i$ is a column vector corresponding to the i^{th} row of the matrix of covariates **A**.

```
"""
  Add your code here
"""
#sigmoid
sig = lambda A,x: 1/(1+np.exp(-np.dot(A,x)))
#obj func
f = lambda y,A,x: np.sum(y*np.log(sig(A,x))+(1-y)*np.log(1-sig(A,x)))
#fp
fp = lambda y,A,x: np.dot(A.T,sig(A,x)-y)
#fpp
fpp = lambda y,A,x: np.dot(np.dot(A.T,np.diag(sig(A,x)*(1-sig(A,x)))),A)
```

Answer the questions and discuss your findings here

Step 3: Generate data. Set numpy's random seed to 0. Then, generate the matrix of covariates $\mathbf{A} \in \mathbb{R}^{M \times N}$, which has i.i.d. entries distributed as N (0, 1). Use the same method, generate the regression coefficient vector $\mathbf{x} \in \mathbb{R}^N$ as well. Then, generate the observation vector $\mathbf{y} \in \{0, 1\}^M$ using np.random.binomial().

```
"""
  Add your code here
"""
import numpy as np
np.random.seed(0)
M = 100
N = 20
A = np.random.normal(loc=0,scale=1,size=(M,N))
np.random.seed(0)
x = np.random.normal(loc=0,scale=1,size=(N,))
y = np.random.binomial(1,sig(A,x),(M,))
print(A.shape,x.shape,y.shape)
print(x)
print(y)
print(A)
```

```
(100, 20) (20,) (100,)
[ 1.76405235  0.40015721  0.97873798  2.2408932   1.86755799 -0.97727788
  0.95008842 -0.15135721 -0.10321885  0.4105985   0.14404357  1.45427351
  0.76103773  0.12167502  0.44386323  0.33367433  1.49407907 -0.20515826
  0.3130677  -0.85409574]
[1 0 0 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 1 0 0
 0 1 1 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 0 1 1
 1 0 1 1 1 0 0 1 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1]
[[ 1.76405235e+00  4.00157208e-01  9.78737984e-01 ... -2.05158264e-01
   3.13067702e-01 -8.54095739e-01]
 [-2.55298982e+00  6.53618595e-01  8.64436199e-01 ...  1.20237985e+00
  -3.87326817e-01 -3.02302751e-01]
 [-1.04855297e+00 -1.42001794e+00 -1.70627019e+00 ...  3.02471898e-01
  -6.34322094e-01 -3.62741166e-01]
 ...
 [ 8.73311836e-01  1.19973618e+00  4.56153036e-01 ... -1.24021634e+00
   9.00054243e-01  1.80224223e+00]
 [-2.08285103e-01  1.57437124e+00  1.98989494e-01 ...  4.32837621e-01
  -8.08717532e-01 -1.10412399e+00]
 [-7.89102180e-01  1.24845579e-03 -1.59939788e-01 ...  1.58433847e-01
  -1.14190142e+00 -1.31097037e+00]]
```

Answer the questions and discuss your findings here

Step 4: Run your code. Apply both algorithms to the logistic regression objective. The algorithm parameters (maxit and tol) should be the same as that in PA1. Note that for logistic regression, it could be difficult to

choose the ss_init parameter. For this assignment, we will set it to be 400 divided by the square of the maximum singular value of the matrix **A**.

```
"""
  Add your code here
"""
def gd_ls(f,fp,y,A,xinit,ss_init,maxit,tol):
  """
  Note that ss changes to ss_init
  ss_init is the starting stepsize for backtracking
  """
    # Add your code here
  x = xinit
  ct = 0
  cur_tol = abs(f(y,A,x))
  prev_tol = 0.0
  ss = ss_init
  fx=[]
  fx.append(cur_tol)
  while ct < maxit and cur_tol >= tol:
    while f(y,A,x-ss*fp(y,A,x)) > f(y,A,x) - 1/2*ss*np.inner(fp(y,A,x),fp(y,A,x)):
      ss = ss/2
    x = x - ss*fp(y,A,x)
    ct += 1
    prev_tol = cur_tol
    cur_tol = abs(f(y,A,x))
    fx.append(cur_tol)
    cur_tol = abs((cur_tol-prev_tol)/prev_tol)
  return x, ct

u,s,vh = np.linalg.svd(A)
ss_init = 400/(np.square(np.amax(s)))
xinit = np.zeros(N)
xls = gd_ls(f,fp,y,A,xinit,ss_init,10000,1e-15)
xnt = nt(f,fp,fpp,y,A,xinit,10000,1e-15)
print("LS\n",xls)
print("NT\n",xnt)
print(x)
```

```
LS
 (array([ 48.81943608,   4.67657603,  12.58837087,  56.03962391,
        29.50667472, -16.38712597,  17.03693783,  -2.04414434,
        -0.47993344,   8.25706035, -12.59821509,  31.52346759,
        29.25432761,  22.81256126,   9.82146308,  -4.40408241,
         6.35225958,  -8.2375815 , -28.56418461, -24.26669906]), 1)
NT
 (array([ 9.29715375, -2.93760139,  1.55430749,  4.61423448,  6.53645981,
       -3.36532208,  3.95160385, -0.84207273, -0.44436575,  1.36220467,
        1.16672552,  4.42050278,  3.56137935, -0.34041223,  4.29900428,
       -0.84629118,  1.90515842,  1.67737936, -3.17731472, -1.32345056]), 6)
[ 1.76405235  0.40015721  0.97873798  2.2408932   1.86755799 -0.97727788
  0.95008842 -0.15135721 -0.10321885  0.4105985   0.14404357  1.45427351
  0.76103773  0.12167502  0.44386323  0.33367433  1.49407907 -0.20515826
  0.3130677  -0.85409574]
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
```

Answer the questions and discuss your findings here

1) First, set M = 100 and N = 20. At convergence, print out the final cost, time taken, and number of iterations used by both algorithms. You should see that the final costs are more or less the same. Which algorithm is faster overall? Which algorithm converges in fewer iterations? Which algorithm has a longer per-iteration run time? Why?

```
"""
  Add your code here
"""
import timeit
import numpy as np
M = 100
N = 20

np.random.seed(0)
A = np.random.normal(loc=0,scale=1,size=(M,N))
np.random.seed(0)
x = np.random.normal(loc=0,scale=1,size=(N,))
y = np.random.binomial(1,sig(A,x),(M,))

u,s,vh = np.linalg.svd(A)
ss_init = 400/(np.square(np.amax(s)))
xinit = np.zeros(N)
start = timeit.default_timer()
xls = gd_ls(f,fp,y,A,xinit,ss_init,10000,1e-15)
stop = timeit.default_timer()
print("LS\n",stop-start,xls)

xinit = np.zeros(N)
start = timeit.default_timer()
xnt = nt(f,fp,fpp,y,A,xinit,10000,1e-15)
stop = timeit.default_timer()
print("NT\n",stop-start,xnt)
```

```
LS
 0.002688463999220403 (array([ 48.81943608,   4.67657603,  12.58837087,  56.(
        29.50667472, -16.38712597,  17.03693783,  -2.04414434,
        -0.47993344,   8.25706035, -12.59821509,  31.52346759,
        29.25432761,  22.81256126,   9.82146308,  -4.40408241,
         6.35225958,  -8.2375815 , -28.56418461, -24.26669906]), 1)
NT
 0.008266393000667449 (array([ 9.29715375, -2.93760139,  1.55430749,  4.6142
       -3.36532208,  3.95160385, -0.84207273, -0.44436575,  1.36220467,
        1.16672552,  4.42050278,  3.56137935, -0.34041223,  4.29900428,
       -0.84629118,  1.90515842,  1.67737936, -3.17731472, -1.32345056]), 6)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
```

Answer the questions and discuss your findings here

LS algorithm converges in less iteration. Newton method has a longer per-iteration run time. It's because it needs to calculate the inverse of fpp which is costly.

2) Change the problem dimension to M = 10000 and N = 20. What happens now? How about in the case of N = 100? Which algorithm is faster overall? Why?

```
"""
  Add your code here
"""
import timeit
```

```
M=10000
print("N = 20\n")
N=20
np.random.seed(0)
A = np.random.normal(loc=0,scale=1,size=(M,N))
np.random.seed(0)
x = np.random.normal(loc=0,scale=1,size=(N,))
y = np.random.binomial(1,sig(A,x),(M,))

# LS ALGORITHM N = 20
start = timeit.default_timer()
u,s,vh = np.linalg.svd(A)
ss_init = 400/(np.square(np.amax(s)))
xinit = np.zeros(N)
xls = gd_ls(f,fp,y,A,xinit,ss_init,10000,1e-15)
stop = timeit.default_timer()
print("LS\n",stop-start,xls)

# NT ALGORITHM N = 20
start = timeit.default_timer()
xnt = nt(f,fp,fpp,y,A,xinit,10000,1e-15)
stop = timeit.default_timer()
print("NT\n",stop-start,xnt)

print("N = 100\n")
N = 100
np.random.seed(0)
A = np.random.normal(loc=0,scale=1,size=(M,N))
np.random.seed(0)
x = np.random.normal(loc=0,scale=1,size=(N,))
y = np.random.binomial(1,sig(A,x),(M,))

# LS ALGORITHM N = 100
u,s,vh = np.linalg.svd(A)
ss_init = 400/(np.square(np.amax(s)))
xinit = np.zeros(N)
start = timeit.default_timer()
xls = gd_ls(f,fp,y,A,xinit,ss_init,10000,1e-15)
stop = timeit.default_timer()
print("LS\n",stop-start,xls)

# NT ALGORITHM N = 100
start = timeit.default_timer()
xnt = nt(f,fp,fpp,y,A,xinit,10000,1e-15)
stop = timeit.default_timer()
print("NT\n",stop-start,xnt)
```

```
N = 20

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
LS
 6.8241336080027395 (array([ 56.97222088,  10.7458065 ,  28.821117  ,  65.677
        55.12423706, -33.08953358,  28.90669292,  -6.71041127,
        -4.90220452,  11.55092329,   1.37145959,  45.11474031,
        21.41937128,   3.55152209,  14.81209604,   7.98002032,
        44.82031589,  -6.85871991,   8.55445417, -24.74949719]), 1)
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-20-a21a27bc24f6> in <module>()
     23 # NT ALGORITHM N = 20
     24 start = timeit.default_timer()
---> 25 xnt = nt(f,fp,fpp,y,A,xinit,10000,1e-15)
     26 stop = timeit.default_timer()
     27 print("NT\n",stop-start,xnt)

                                  1 frames
<ipython-input-16-73107904cc67> in <lambda>(y, A, x)
      9 fp = lambda y,A,x: np.dot(A.T,sig(A,x)-y)
     10 #fpp
---> 11 fpp = lambda y,A,x: np.dot(np.dot(A.T,np.diag(sig(A,x)*(1-sig(A,x)))

KeyboardInterrupt:
```

SEARCH STACK OVERFLOW

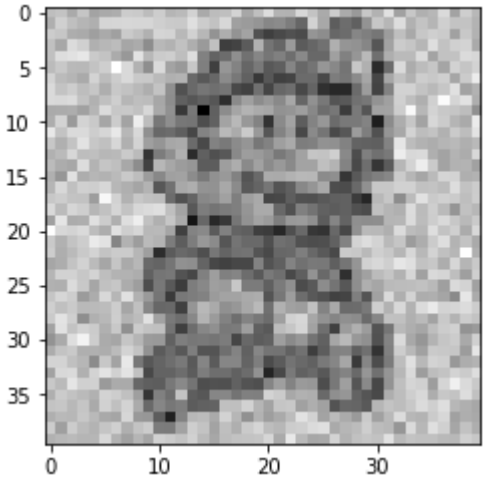
Answer the questions and discuss your findings here Newton algorithm

LS algorithm is faster overall because calculationg inverse of Hessian is costly.

3) Load the given data file noisy1.npz. Select one algorithm to recover $\mathbf{x}$ using the same approach as that in PA1. Which algorithm are you going to select? Why? Reshape it to a 40×40 matrix. Visualize it as an image in grayscale using matplotlib. What do you think is the original image?

```
"""
  Add your code here
"""
import timeit
import numpy as np
import matplotlib.pyplot as plt
# from google.colab import files
# uploaded = files.upload()
file = np.load("noisy1.npz")
A = file['A']
y = file['y']
xinit = np.zeros(1600)
start = timeit.default_timer()
# u,s,vh = np.linalg.svd(A)
# ss_init = 400/(np.square(np.amax(s)))
x, it = nt(f,fp,fpp,y,A,xinit,10000,1e-15)
# x = gd_ls(f,fp,y,A,xinit,ss_init,10000,1e-15)
im = x.reshape(40,40)
plt.gray()
plt.imshow(im)
stop = timeit.default_timer()
print(stop-start)
```

```
23.6817701459986
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:7: RuntimeWarnir
  import sys
```



Answer the questions and discuss your findings here

The image is an Italian plumber who eats mushrooms in black and white

Add Colab link here:

https://colab.research.google.com/drive/1ArjzMP3KNqeNn3IMxeAErznxn4CBCqCq