Ryan Gentry

ECE 6254

13 February 2017

HW #2

1)

ADJUSTED CODE:

```
import numpy as np
import json
from sklearn.feature_extraction import text

x = open('fedpapers_split.txt').read()
papers = json.loads(x)

papersH = papers[0] # papers by Hamilton
papersM = papers[1] # papers by Madison
papersD = papers[2] # disputed papers

nH, nM, nD = len(papersH), len(papersM), len(papersD)

# This allows you to ignore certain common words in English
# You may want to experiment by choosing the second option or your own
# list of stop words, but be sure to keep 'HAMILTON' and 'MADISON' in
# this list at a minimum, as their names appear in the text of the papers
# and leaving them in could lead to unpredictable results
stop_words = text.ENGLISH_STOP_WORDS.union({'HAMILTON','MADISON'})

## Form bag of words model using words used at least 10 times
vectorizer = text.CountVectorizer(stop_words,min_df=10)
X = vectorizer.fit_transform(papersH+papersM+papersD).toarray()

# Uncomment this line to see the full list of words remaining after filtering out
# stop words and words used less than min_df times
#vectorizer.vocabulary_

# Split word counts into separate matrices
XH, XM, XD = X[:nH,:], X[nH:nH+nM,:], X[nH+nM:,:]

# Initialize vectors for P(word_j | H/M) as total occurence of a word for an other divided by total words
fH = np.zeros(len(XH[0]))
totH = 0
fM = np.zeros(len(XM[0]))
totM = 0

# Estimate probability of each word in vocabulary being used by Hamilton
```

```
for i in range(0,len(XH[0])):
   for j in range(0,len(XH)):
      fH[i] = float(fH[i])+XH[j][i]
   totH = totH + fH[i]
fH = fH/totH

# Estimate probability of each word in vocabulary being used by Madison
for i in range(0,len(XM[0])):
   for j in range(0,len(XM)):
      fM[i] = float(fM[i])+XM[j][i]
   totM = totM + fM[i]
fM = fM/totM

# Compute ratio of these probabilities
fratio = fH/fM

# Compute prior probabilities
piH = float(nH)/(nH+nM)
piM = float(nM)/(nH+nM)

Ham_tot = nH
Mad_tot = nM
for xd in XD: # Iterate over disputed documents
   rat = 1
   # Compute likelihood ratio for Naive Bayes model
   for j in range(0,len(xd)):
      rat = rat*(fratio[j])**xd[j]

   LR = (piH/piM)*rat
   if LR>1:
      print 'Hamilton'
      Ham_tot=Ham_tot+1
   else:
      print 'Madison'
      Mad_tot=Mad_tot+1
print "Hamilton wrote %d total, and %d of the disputed." % (Ham_tot, Ham_tot-nH)
print "Madison wrote %d total, and %d of the disputed." % (Mad_tot, Mad_tot-nM)
```

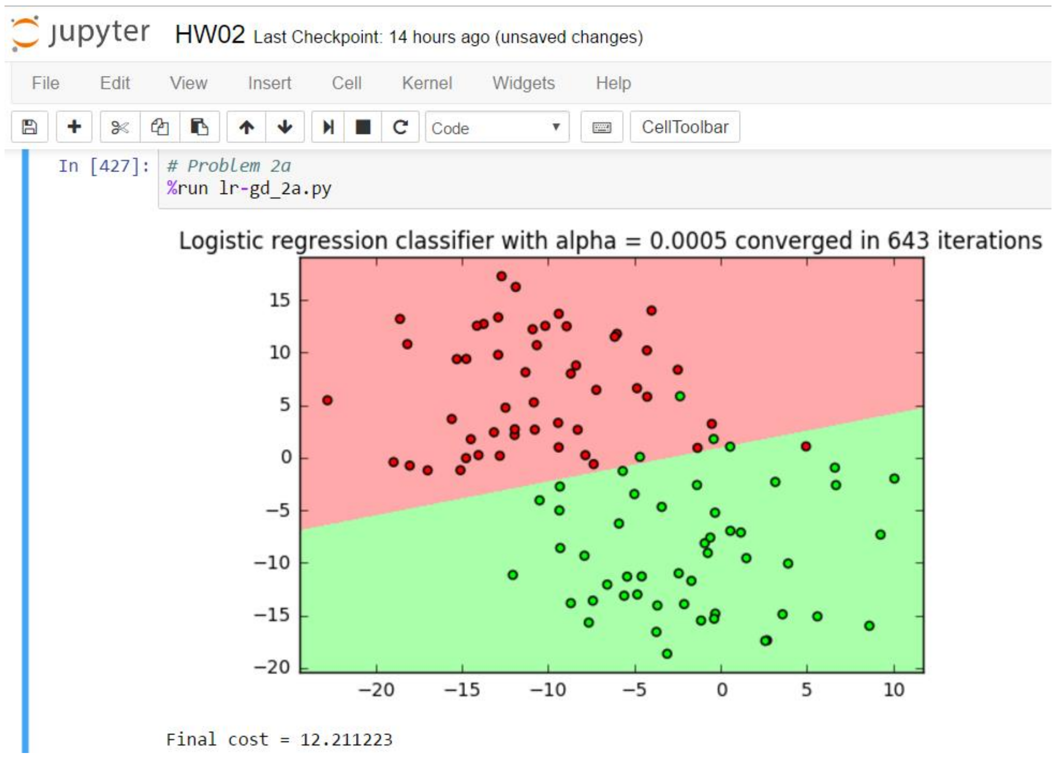
OUTPUT:

```
Madison
Madison
Madison
Madison
Hamilton
Hamilton
Hamilton
Hamilton
Hamilton
```

```
Madison
Madison
Madison
Hamilton wrote 56 total, and 5 of the disputed.
Madison wrote 24 total, and 7 of the disputed.
```

2)

a.



b.

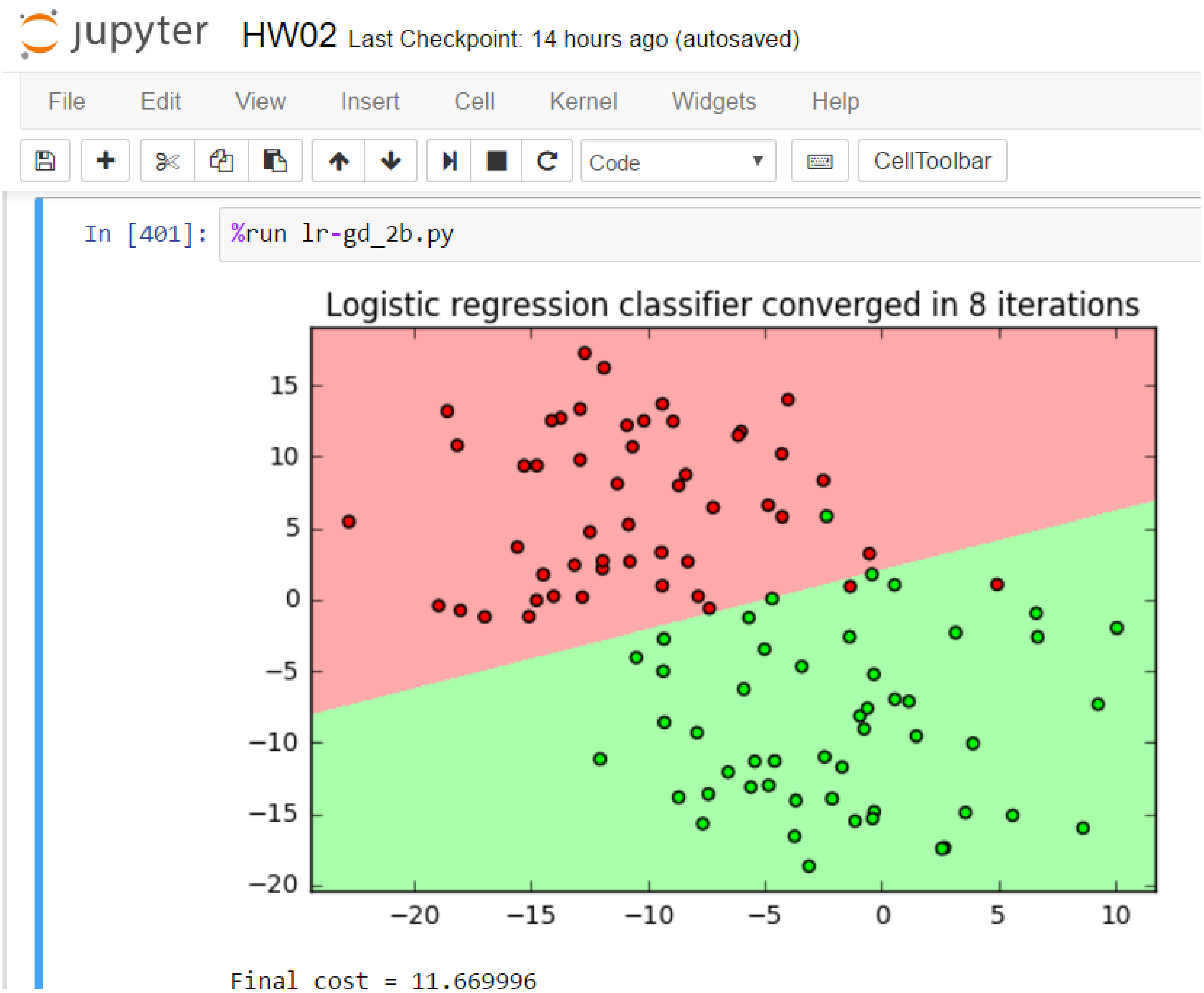
ADJUSTED CODE:

```
# function to compute inverted Hessian Matrix for Newton-Raphson method
def Newt_Raph(theta,x):
    g = logistic_func(theta,x)
    H = np.zeros((3,3))
    for i in range(0,len(x)):
        x0 = np.array([x[i]])
        xT = np.array([x[i]]).T
        H = H+np.dot(xT,x0)*g[i]*(1-g[i])
    H = np.linalg.inv(H)
    return H

# implementation of gradient descent for logistic regression
def grad_desc(theta, x, y, tol, maxiter):
```

```
nll_vec = []
nll_vec.append(neg_log_like(theta, x, y))
nll_delta = 2.0*tol
iter = 0
while (abs(nll_delta) > tol) and (iter < maxiter):
    alpha = Newt_Raph(theta,x)
    theta = theta - (alpha.dot(log_grad(theta, x, y)))
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = nll_vec[-2]-nll_vec[-1]
    iter += 1
return theta, np.array(nll_vec), iter
```

OUTPUT:



c.

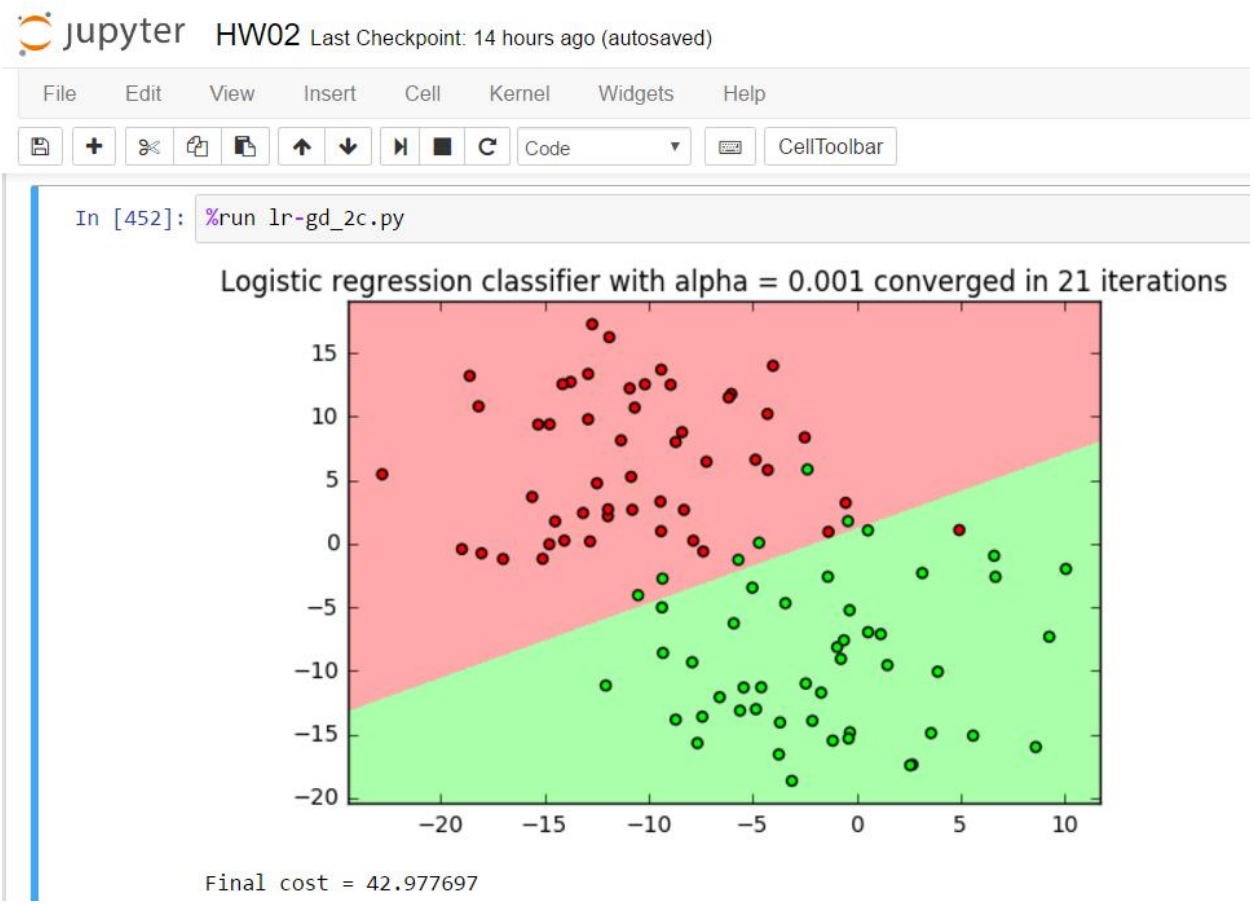
ADJUSTED CODE:

```
# function to compute the gradient of the negative log-likelihood
```

```
def log_grad(theta, x, y):
    g = logistic_func(theta,x)
    n = np.random.permutation(100)[0]
    xT = np.array([x[n]]).T
    yn = np.array([y[n]])
    gn = np.array([g[n]])
    return -xT.dot(yn-gn)

# implementation of gradient descent for logistic regression
def grad_desc(theta, x, y, alpha, tol, maxiter):
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    while (nll_delta > tol) and (iter < maxiter):
        theta = theta - (alpha * log_grad(theta, x, y))
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    return theta, np.array(nll_vec), iter
```

OUTPUT:



d.

ADJUSTED CODE:

```
# implementation of gradient descent for logistic regression
def grad_desc(theta, x, y, alpha, tol, maxiter):
    start = time.time()
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    while (abs(nll_delta) > tol) and (iter < maxiter):
        theta = theta - (alpha * log_grad(theta, x, y))
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    end = time.time()
    time_gd = end - start
    return theta, np.array(nll_vec), iter, time_gd

# function to compute inverted Hessian Matrix for Newton-Raphson method
def Newt_Raph(theta,x):
    g = logistic_func(theta,x)
    H = np.zeros((3,3))
    for i in range(0,len(x)):
        x0 = np.array([x[i]])
        xT = np.array([x[i]]).T
        H = H+np.dot(xT,x0)*g[i]*(1-g[i])
    H = np.linalg.inv(H)
    return H

# implementation of gradient descent for logistic regression
def grad_desc_N(theta, x, y, tol, maxiter):
    start = time.time()
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 2.0*tol
    iter = 0
    while (abs(nll_delta) > tol) and (iter < maxiter):
        alpha = Newt_Raph(theta,x)
        theta = theta - (alpha.dot(log_grad(theta, x, y)))
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    end = time.time()
    time_N = end - start
    return theta, np.array(nll_vec), iter, time_N

# function to compute the gradient of the negative log-likelihood
```

```
def log_grad_SGD(theta, x, y):
    g = logistic_func(theta,x)
    n = np.random.permutation(100)[0]
    xT = np.array([x[n]]).T
    yn = np.array([y[n]])
    gn = np.array([g[n]])
    return -xT.dot(yn-gn)

# implementation of gradient descent for logistic regression
def grad_desc_SGD(theta, x, y, alpha, tol, maxiter):
    start = time.time()
    nll_vec = []
    nll_vec.append(neg_log_like(theta, x, y))
    nll_delta = 3*tol
    iter = 0
    while (nll_delta > tol) and (iter < maxiter):
        theta = theta - (alpha * log_grad_SGD(theta, x, y))
        nll_vec.append(neg_log_like(theta, x, y))
        nll_delta = nll_vec[-2]-nll_vec[-1]
        iter += 1
    end = time.time()
    time_SGD = end - start
    return theta, np.array(nll_vec), iter, time_SGD

# function to compute output of LR classifier
def lr_predict(theta,x):
    # form Xtilde for prediction
    shape = x.shape
    Xtilde = np.zeros((shape[0],shape[1]+1))
    Xtilde[:,0] = np.ones(shape[0])
    Xtilde[:,1:] = x
    return logistic_func(theta,Xtilde)

## Generate dataset
np.random.seed(2017) # Set random seed so results are repeatable
x,y = datasets.make_blobs(n_samples=100000,n_features=2,centers=2,cluster_std=6.0)

## build classifier
# form Xtilde
shape = x.shape
xtilde = np.zeros((shape[0],shape[1]+1))
xtilde[:,0] = np.ones(shape[0])
xtilde[:,1:] = x

# Initialize theta to zero
theta = np.zeros(shape[1]+1)
alpha = .000005
alpha_SGD = alpha*100
tol = 1e-3
```

maxiter = 10000

Run Gradient Descent
theta_gd,cost_gd,iters_gd,time_gd = grad_desc(theta,xtilde,y,alpha,tol,maxiter)
print("Standard gradient descent with alpha = %.6f converged in %d iterations and %.6f s" % (alpha,iters_gd,time_gd))
print("Final cost = %.6f" % cost_gd[-1])

Run Newton's Method
theta_N,cost_N,iters_N,time_N = grad_desc_N(theta,xtilde,y,tol,maxiter)
print("Newton's Method converged in %d iterations and %.6f s" % (iters_N,time_N))
print("Final cost = %.6f" % cost_N[-1])

Run Stochastic Gradient Descent
theta_SGD,cost_SGD,iters_SGD,time_SGD = grad_desc_SGD(theta,xtilde,y,alpha_SGD,tol,maxiter)
print("Stochastic gradient descent with alpha = %.4f converged in %d iterations and %.6f s" % (alpha_SGD,iters_SGD,time_SGD))
print("Final cost = %.6f" % cost_SGD[-1])

OUTPUT:

```
In [4]: %run lr-gd_2d.py

Standard gradient descent with alpha = 0.000005 converged in 10000 iterations and 978.888109 s
Final cost = 27843.517227
Newton's Method converged in 7 iterations and 13.916237 s
Final cost = 23771.854285
Stochastic gradient descent with alpha = 0.0005 converged in 14 iterations and 1.340017 s
Final cost = 58624.284878
```