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

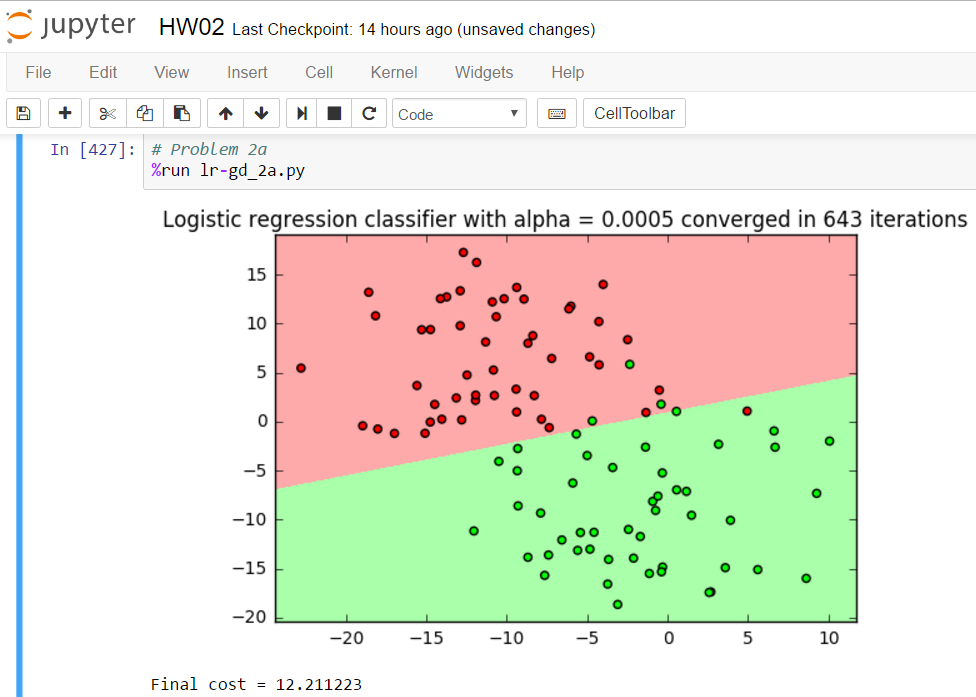
Madison

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)))

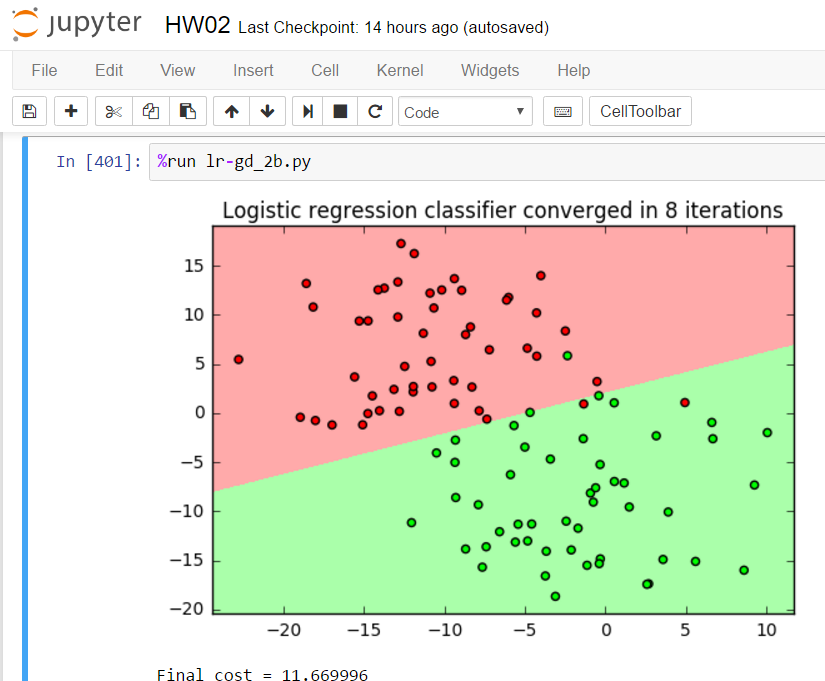
nll\_vec.append(neg\_log\_like(theta, x, y))

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))

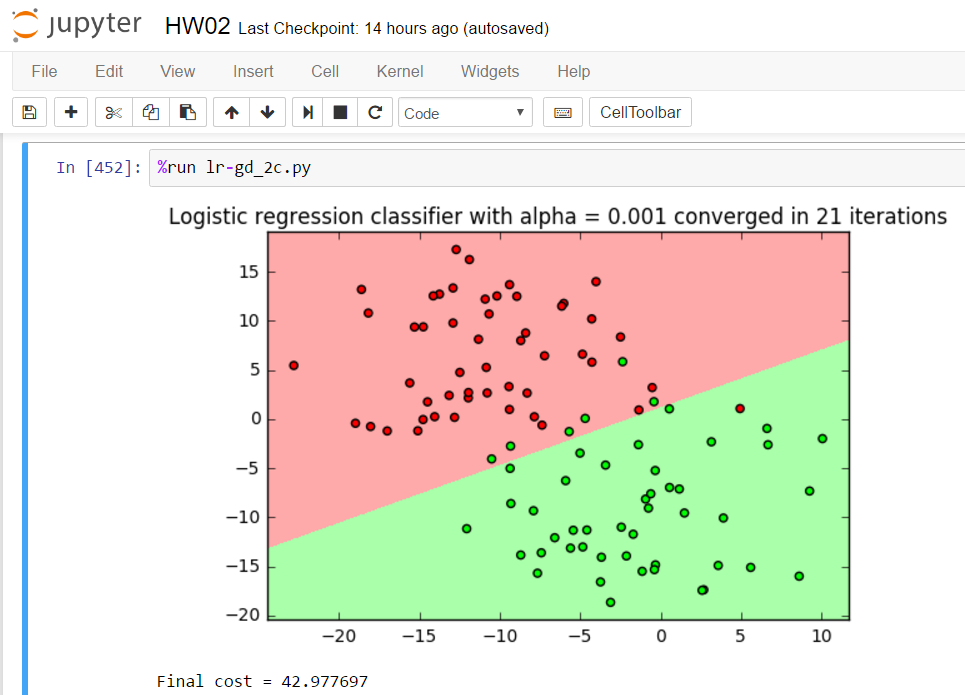
nll\_vec.append(neg\_log\_like(theta, x, y))

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:

