Q3

March 8, 2024

1 Appendix

1.1 Question 3

[ ]:
```
#Imports
import numpy as np
import scipy.io as io
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from scipy.special import expit
from sklearn.model_selection import GridSearchCV
```

[ ]:
```
SEED = 2

#Seeding RNG with seed = 0
rng = np.random.Generator(np.random.PCG64(seed=SEED))
```

1.1.1 Part 2

[ ]:
```
data = io.loadmat('data.mat')
#print(data['description'])
samples = data['X']
labels = data['y']
test = data['X_test']

print("samples: ", samples.shape)
print("labels: ", labels.shape)
```

```
samples:  (5000, 12)
labels:  (5000, 1)
```

[ ]:
```
#Split
xTrainRaw, xTestRaw, yTrain, yTest = train_test_split(samples, labels,␣
 ↪test_size=0.20, random_state=SEED, shuffle=True)

print("xTrain: ", xTrainRaw.shape)
print("xTest: ", xTestRaw.shape)
print("yTrain: ", yTrain.shape)
```

```
print("yTest: ", yTest.shape)
```

```
xTrain:  (4000, 12)
xTest:  (1000, 12)
yTrain:  (4000, 1)
yTest:  (1000, 1)
```

[ ]:
```
#Data editing
#Normalize data
xTrain = (xTrainRaw - xTrainRaw.mean()) / xTrainRaw.std()
xTest = (xTestRaw - xTrainRaw.mean()) / xTrainRaw.std()
test = (test - xTrainRaw.mean()) / xTrainRaw.std()

#Adding ficticious dim
ones = np.ones((xTrain.shape[0], 1))
xTrain = np.hstack((xTrain, ones))
print("xTrain with fict dim: ", xTrain.shape)

ones = np.ones((xTest.shape[0], 1))
xTest = np.hstack((xTest, ones))
print("xTest with fict dim: ", xTrain.shape)

ones = np.ones((test.shape[0], 1))
test = np.hstack((test, ones))
print("test data with fict dim: ", test.shape)
```

```
xTrain with fict dim:  (4000, 13)
xTest with fict dim:  (4000, 13)
test data with fict dim:  (1000, 13)
```

[ ]:
```
def validate(w, xTest, yTest):
    validationY = np.round(expit(xTest.dot(w)))
    validationY = np.reshape(validationY, (validationY.shape[0]))
    print(yTest.shape)
    yTest = np.reshape(yTest, (yTest.shape[0]))
    # print(validationY.reshape(1, validationY.shape[0]))
    # print(yTest.reshape(1, validationY.shape[0]))
    correct = 0
    total = len(validationY)
    for i in range(0, len(yTest)):
        if validationY[i] == yTest[i]:
            correct += 1
    return correct/total
```

[ ]:
```
#Batch Gradient
def batchGrad(x, y, w, reg):
    result = y - expit(x.dot(w)) #logisticFn(x, w)
    result = -x.T.dot(result) + reg * w
```

```
    return result

#Take step
def update(w, step, grad):
    w = w - step * grad
    return w

def cost(x, y, w):
    result = y * np.log(expit(np.clip(x.dot(w), a_min = 0.0001,  a_max=1)))
    result = result + (1 - y) * np.log(1 - expit(np.clip(x.dot(w), a_min = 0.
 ↪0001, a_max=1 )))
    result = np.sum(result)
    return -result
```

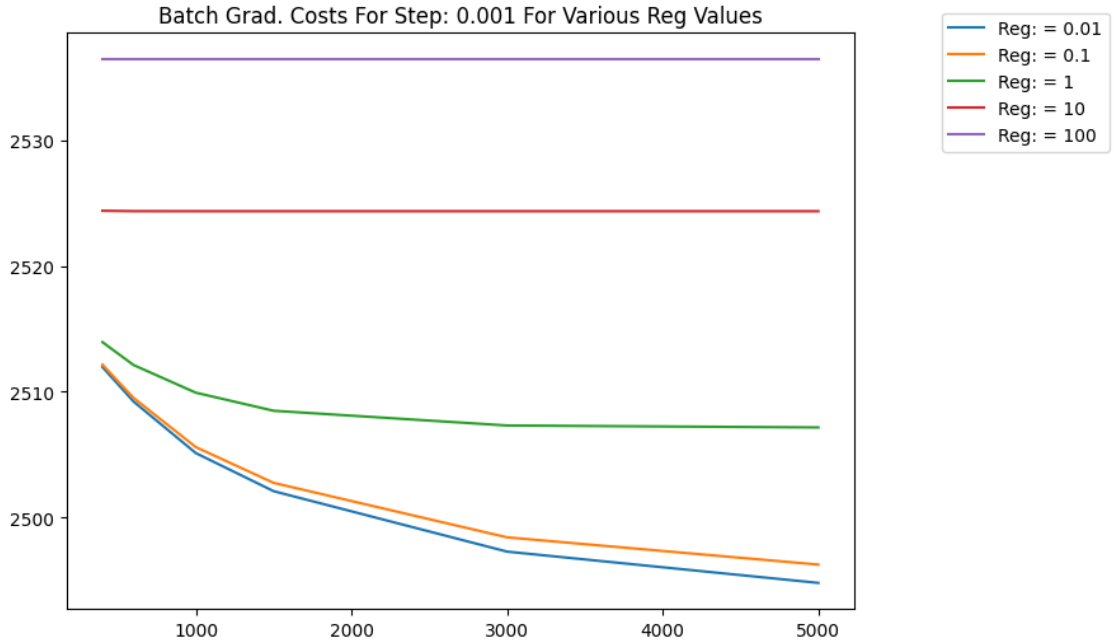
[ ]:
```
def train(tolerance, max_iter, gradFn, trainSize, step, reg):
    #Start guess at 0
    w = np.zeros((1, xTrain.shape[1])).T

    #Find first gradient
    iter = 0

    while iter < max_iter:
        grad = gradFn(xTrain[:trainSize], yTrain[:trainSize], w, reg)
        w = update(w, step, grad)
        iter += 1
        if np.all(abs(grad)) < tolerance:
            print("Met Tolarence")
            break
    return w
```

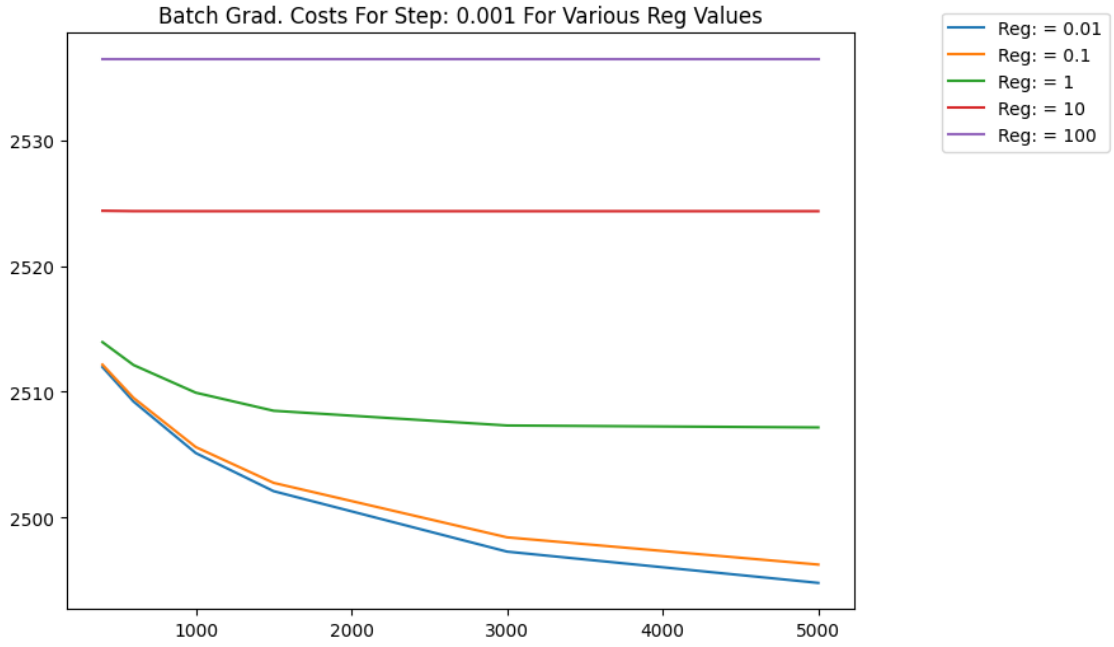
[ ]:
```
iter = [400, 600, 1000, 1500, 3000, 5000]
stepSize = [0.001, 0.001, 0.0001]
reg = [0.01, 0.1, 1, 10, 100]
batchCostsForStep = []
# param_grid = [
#   {'reg': [0.001, 0.01, 0.1, 1, 10], 'stepSize' : [0.001, 0.001, 0.0001]},
#   {'iter': [100, 200, 400, 600, 1000, 1500, 3000, 5000]},
#  ]
for step in stepSize:
    costDict = {}
    for r in reg:
        currCost = []
        for i in iter:
            w = train(0.1, i, batchGrad, 4000, step, r)
            c = cost(xTrain[:4000], yTrain[:4000], w)
            currCost.append(c)
        costDict[f'Reg: = {r}'] = currCost
```

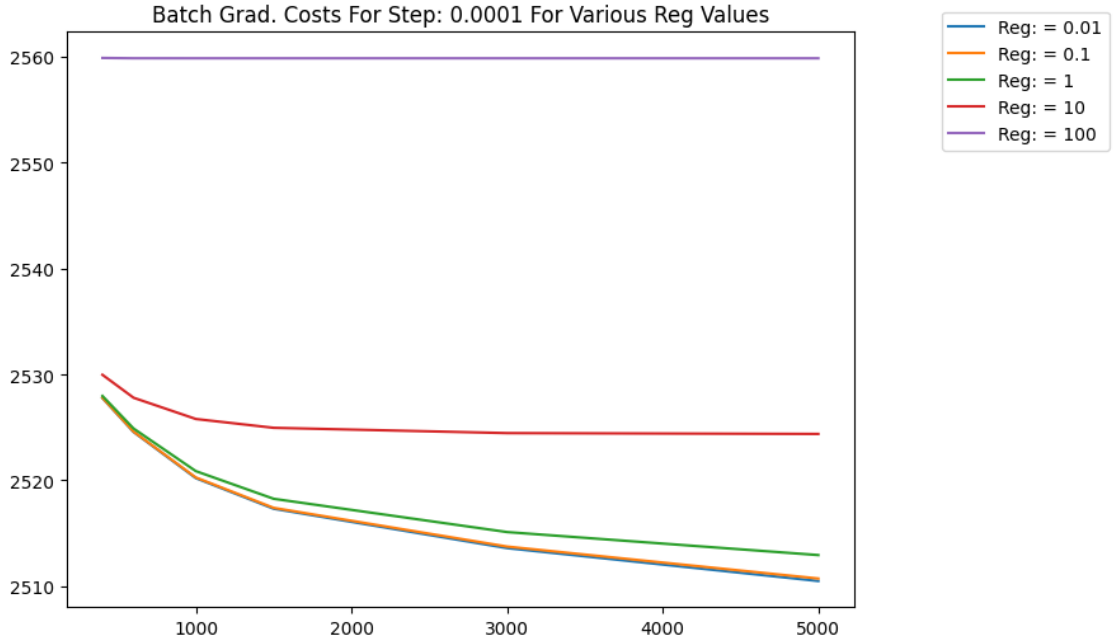
```
    batchCostsForStep.append(costDict)
```

```
Met Tolarence
Met Tolarence
Met Tolarence
Met Tolarence
```

[ ]:
```
for i in range(0, len(stepSize)):
    plt.figure(figsize=(8, 20))
    for key in batchCostsForStep[i].keys():
        plt.subplot(3, 1, i+1)
        plt.plot(iter, batchCostsForStep[i][key], label = key)
        plt.title(f"Batch Grad. Costs For Step: {stepSize[i]} For Various Reg␣
 ↪Values")
        plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```







[ ]:
```
w = train(0.1, 50000, batchGrad, 4000, 0.001, 0.01)
validate(w, xTest, yTest)
```

```
(1000, 1)
```

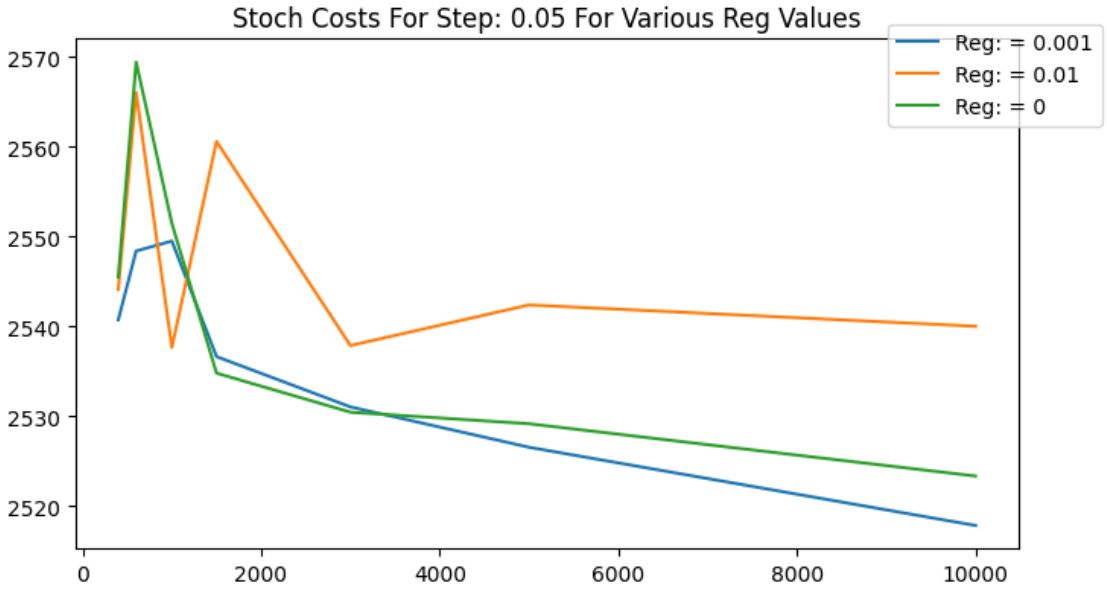
[ ]:
```
0.95
```

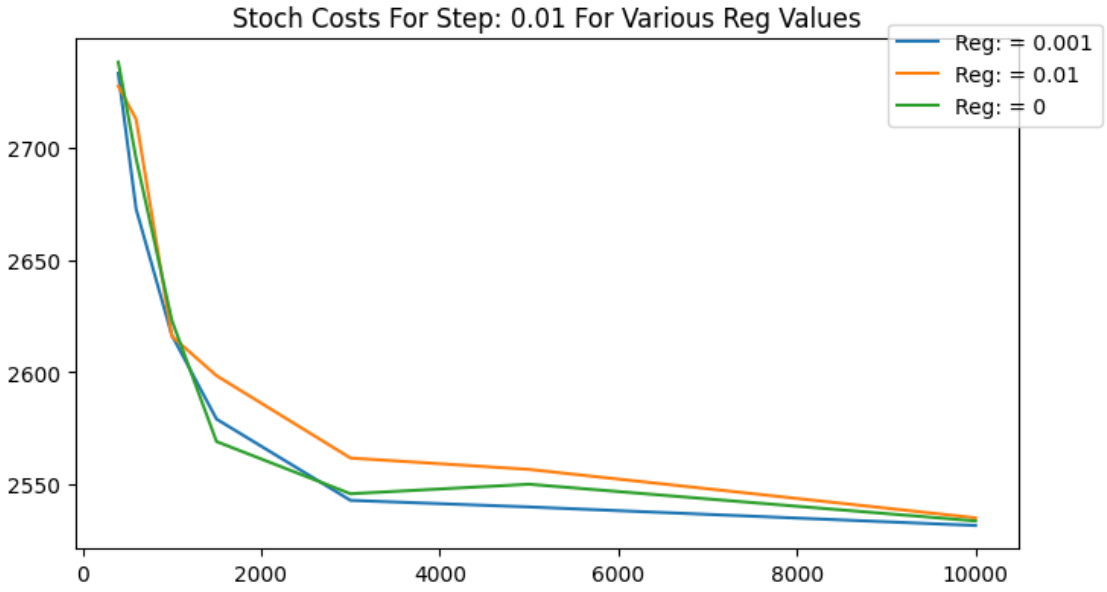
1.1.2 Section 4

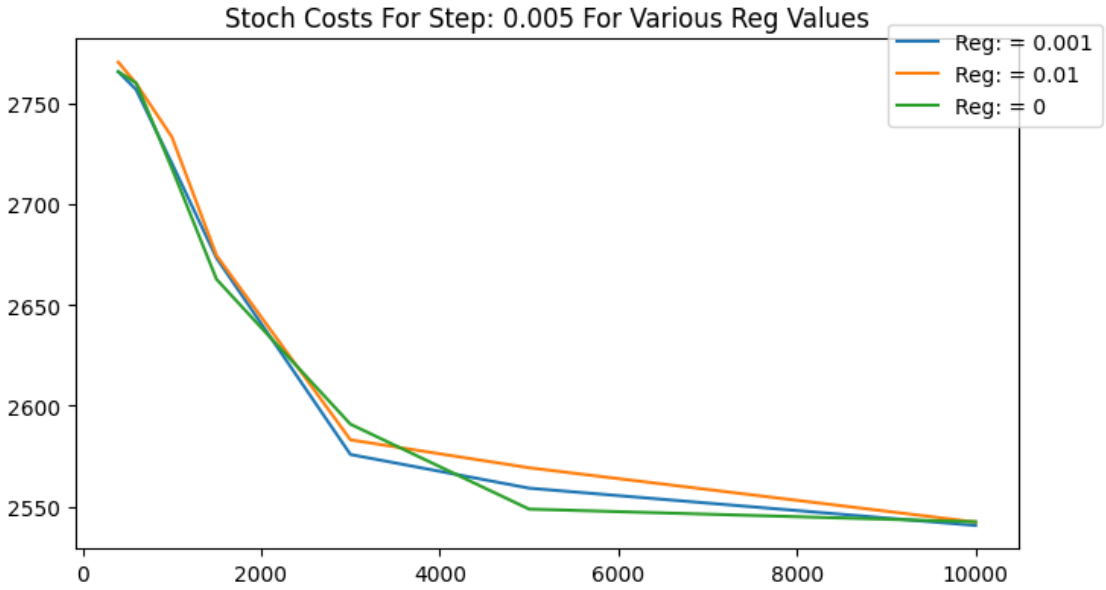
[ ]:
```
#Stochastic Gradient
def stoGrad(x, y, w, reg):
    randIdx = rng.integers(0, x.shape[0])

    xi = x[randIdx]
    yi = y[randIdx]

    result = yi - expit(xi.dot(w)) #logisticFn(x, w)
    result = np.reshape((result * xi), (13,1))
    # print((reg*w).shape)
    # print(result.shape)
    result = -result + reg * w
    return result
```

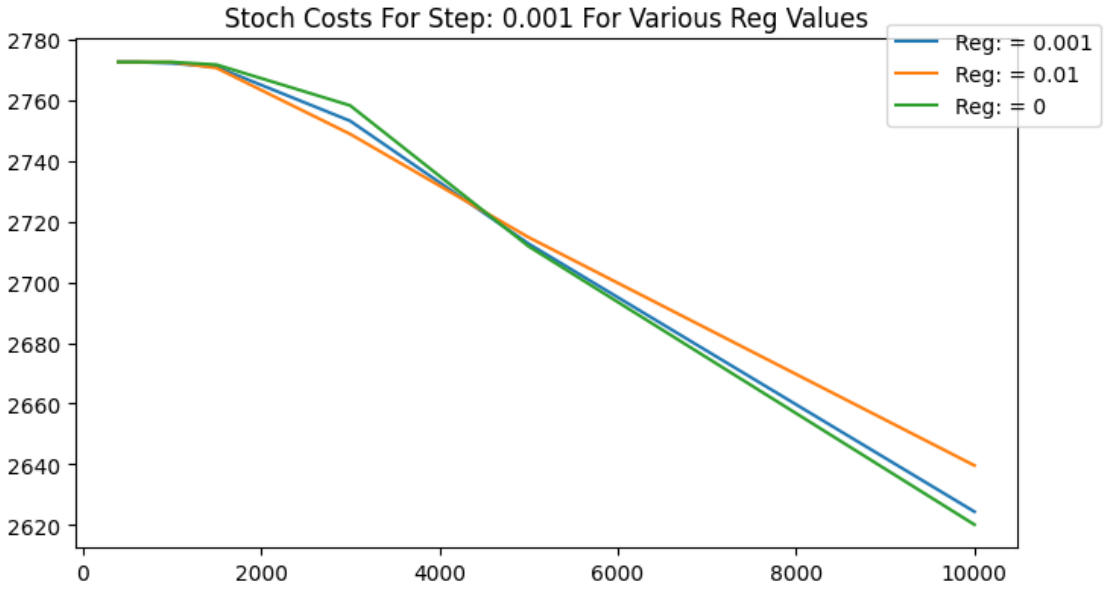
[ ]:
```
#w = train(0.1, 20, stoGrad, 4000, 0.0001, 0)

iter = [400, 600, 1000, 1500, 3000, 5000, 10000, 50000]
stepSize = [0.05, 0.01, 0.005, 0.001]
reg = [0.001, 0.01, 0]
stoCostsForStep = []
for step in stepSize:
    costDict = {}
    for r in reg:
        currCost = []
        for i in iter:
            w = train(0.01, i, stoGrad, 4000, step, r)
            c = cost(xTrain[:4000], yTrain[:4000], w)
            currCost.append(c)
        costDict[f'Reg: = {r}'] = currCost
    stoCostsForStep.append(costDict)
```

[ ]:
```
for i in range(0, len(stepSize)):
    plt.figure(figsize=(8, 20))
    for key in stoCostsForStep[i].keys():
        plt.subplot(len(stepSize), 1, i+1)
        plt.plot(iter, stoCostsForStep[i][key], label = key)
        plt.title(f"Stoch Costs For Step: {stepSize[i]} For Various Reg Values")
        plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```


Stoch Costs For Step: 0.05 For Various Reg Values
Reg: = 0.001
Reg: = 0.01
Reg: = 0
2570
2560
2550
2540
2530
2520
0
2000
4000
6000
8000
10000


Stoch Costs For Step: 0.01 For Various Reg Values
Reg: = 0.001
Reg: = 0.01
Reg: = 0
2700
2650
2600
2550
0
2000
4000
6000
8000
10000





[ ]:
```
w1 = train(0.1, 100000, stoGrad, 4000, 0.01, 0.001)
validate(w1, xTest, yTest)
```

```
(1000, 1)
```

[ ]: 0.924

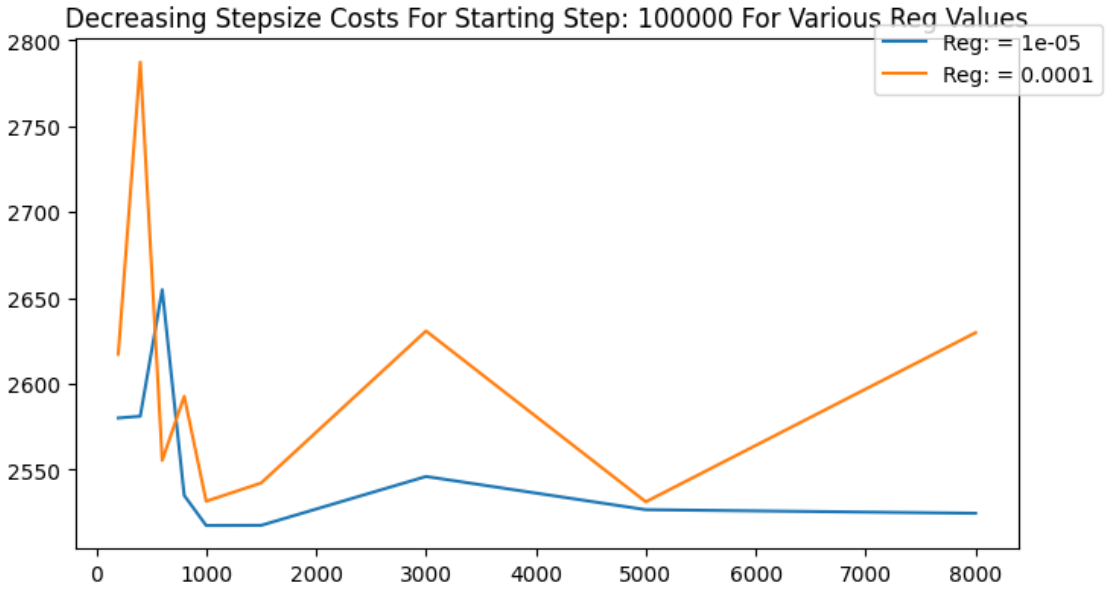
1.1.3 Section 5

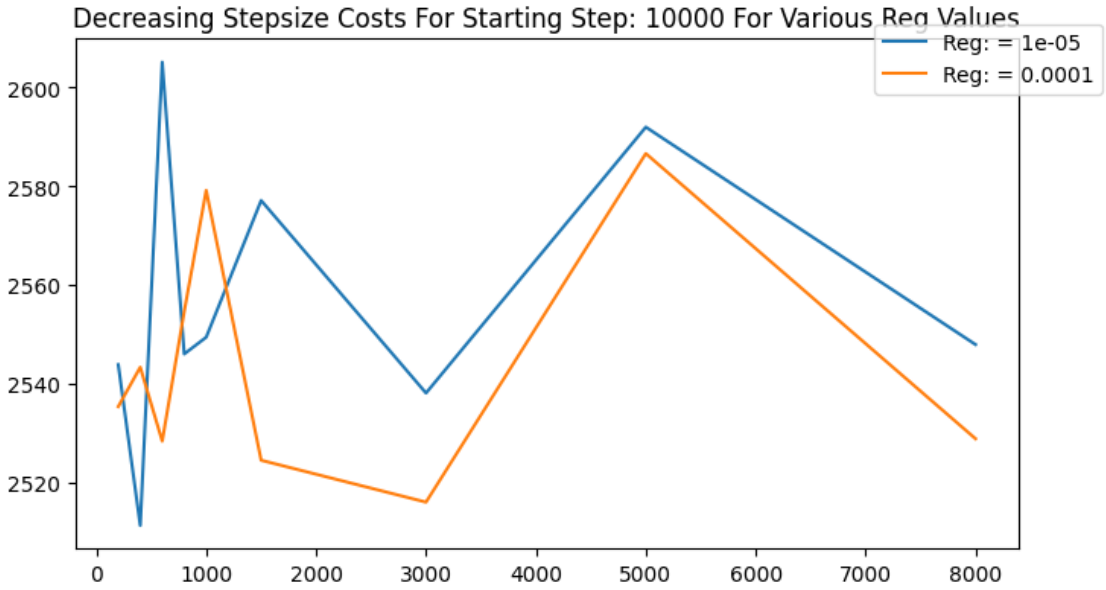
[ ]:
```
#Take step
def decreasingUpdate(w, startingStep, iter, grad):
    #w = w - max(startingStep/iter, 0.0001) * grad
    w = w - startingStep/iter * grad
    return w
```

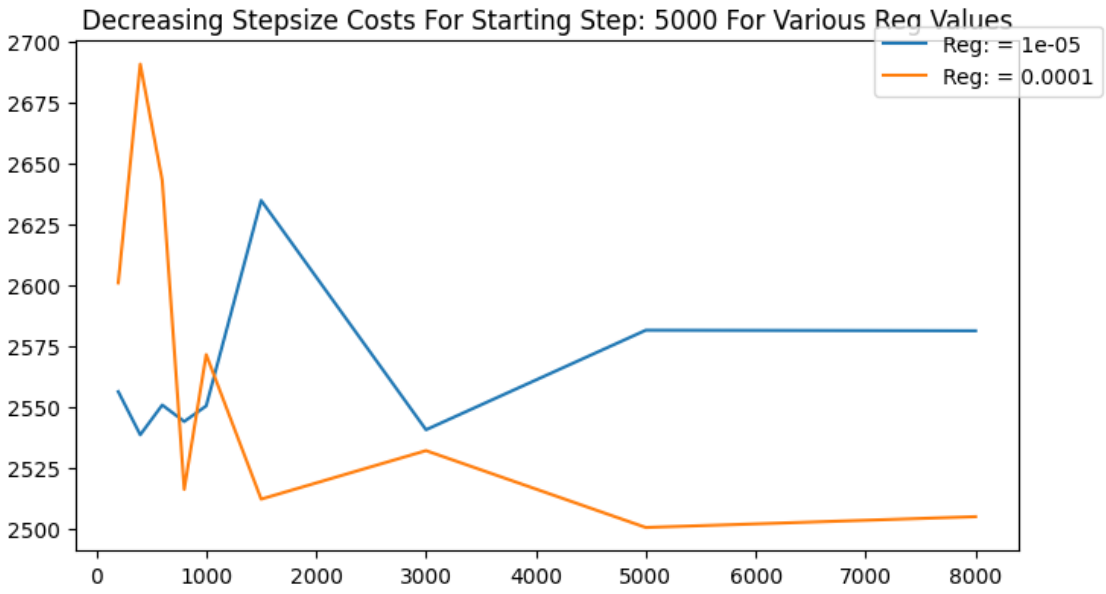
[ ]:
```
def decreasingTrain(tolerance, max_iter, gradFn, trainSize, startingStep, reg):
    #Start guess at 0
    w = np.zeros((1, xTrain.shape[1])).T

    #Find first gradient
    iter = 1

    while iter < max_iter:
        grad = gradFn(xTrain[:trainSize], yTrain[:trainSize], w, reg)
        w = decreasingUpdate(w, startingStep, iter, grad)
        iter += 1
        if np.all(abs(grad)) <= tolerance:
            print("Met Tolarence")
            break
    return w
```

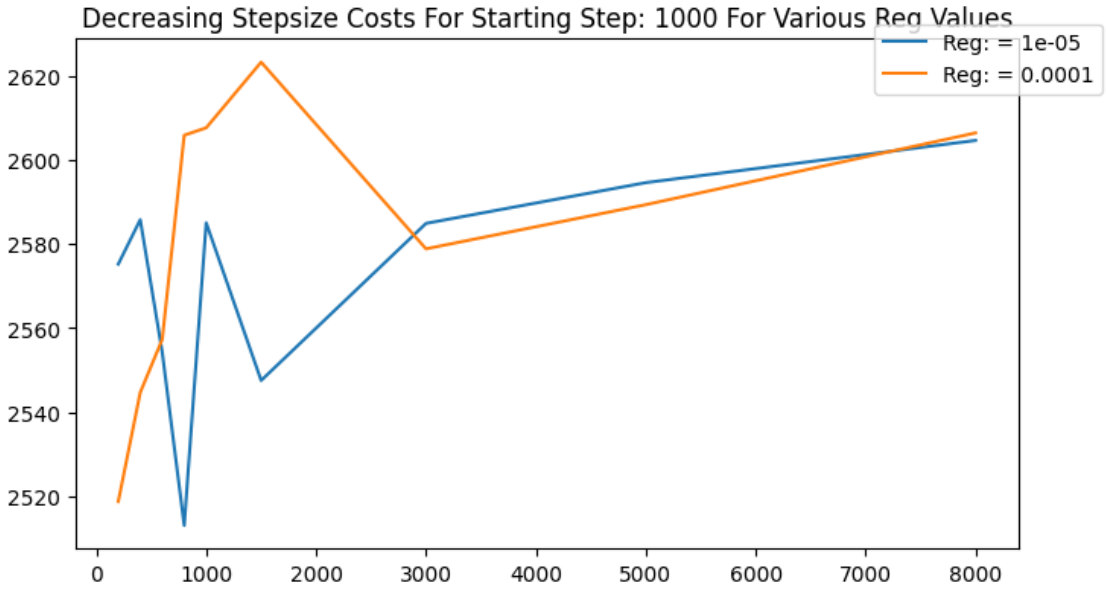
[ ]:
```
iter = [200, 400, 600, 800, 1000, 1500, 3000, 5000, 8000]
startingStep = [100000, 10000, 5000, 1000]
reg = [0.00001, 0.0001]
decreasingCostsForStep = []
for step in startingStep:
    costDict = {}
    for r in reg:
        currCost = []
        for i in iter:
            w = decreasingTrain(0, i, stoGrad, 4000, step, r)
            c = cost(xTrain[:4000], yTrain[:4000], w)
            currCost.append(c)
        costDict[f'Reg: = {r}'] = currCost
    decreasingCostsForStep.append(costDict)
```

[ ]:
```
for i in range(0, len(startingStep)):
    plt.figure(figsize=(8, 20))
    for key in decreasingCostsForStep[i].keys():
        plt.subplot(len(startingStep), 1, i+1)
        plt.plot(iter, decreasingCostsForStep[i][key], label = key)
        plt.title(f"Decreasing Stepsize Costs For Starting Step:␣
 ↪{startingStep[i]} For Various Reg Values")
        plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```


Decreasing Stepsize Costs For Starting Step: 100000 For Various Reg Values
Reg: = 1e-05
Reg: = 0.0001
2800
2750
2700
2650
2600
2550
0
1000
2000
3000
4000
5000
6000
7000
8000


Decreasing Stepsize Costs For Starting Step: 10000 For Various Reg Values
Reg: = 1e-05
Reg: = 0.0001
2600
2580
2560
2540
2520
0
1000
2000
3000
4000
5000
6000
7000
8000





[ ]:
```
w = decreasingTrain(0.00001, 100000, stoGrad, 4000, 100000, 0.00001)
```

[ ]:
```
validate(w, xTest, yTest)
```

```
(1000, 1)
```

[ ]: 0.94

[ ]:
```
final = train(0.1, 50000, batchGrad, 4000, 0.001, 0.01)
validate(final, xTest, yTest)
```

```
(1000, 1)
```

[ ]:
```
0.95
```

[ ]:
```
result = np.round(expit(test.dot(final)))
print(result.shape)
result = np.reshape(result, (result.shape[0]))
import pandas as pd
list = [*range(1, len(result)+1)]
outputDict = {"Id":list, "Category": result}
df = pd.DataFrame(outputDict)
df.to_csv('wineResult.csv', index=False)
```

```
(1000, 1)
```

[ ]:
```

```