# CSE 847 – HW 4 – Andrew Haas

GitHub link: https://github.com/AndrewEHaas/cse847hw4

First, I implemented the following python code to drive my regression:

```
# ==== Additional Methods ==== #

sigmoid = lambda data : 1/(1 + np.exp(-1*data))
predict = lambda data, weights : np.where(sigmoid(data @ weights) > .5, 1, 0)

def loss(data: Array, weights: Array, labels: Array) -> float:
    s = sigmoid(data @ weights)
    return (-1/data.shape[0])*np.sum(labels*np.log(s) + (1-labels)*np.log(1-s))

def gradient(data: Array, weights: Array, labels: Array) -> Array:
    s = sigmoid(data @ weights)
    g = np.zeros(weights.shape)
    for i in range(data.shape[0]):
        g += np.reshape((labels[i,:] - s[i,:])*data[i,:], g.shape)
    return (-1/data.shape[0])*g

def l1_loss(data: Array, weights: Array, labels: Array, par: float) -> float:
    return loss(data, weights, labels) + par*np.linalg.norm(weights, ord=1)/data.shape[0]

def l1_gradient(data: Array, weights: Array, labels: Array, par: float) -> float:
    return gradient(data, weights, labels) + par*np.sign(weights)/data.shape[0]

# ======================= #
```

These methods were then called by the following functions. First, standard logistic regression (0,1 encoding):

```
def logistic_train(data: Array, labels: Array, lr: float, epsilon: float = 1e-5, maxiter: int = 1000) ->
Array:
    """
    parameters:
        data   : n * (d + 1) with n samples and d features, where col d+1 is all ones (corresponding to
intercept)

        labels : n * 1 vector of class labels (0 or 1)

        lr : learning rate

        epsilon: optional parameter specifying convergence - if the change in absolute difference in
predictions, from one iteration to the next
                 when averaged across input features is less than epsilon, halt (default 1e-5)

        maxiter: optional parameter that specifies the maximum number of iterations (default 1000)

    return: 1 * (d + 1) vector of weights corresponding to columns of data - last entry is bias term
    """
    # instantiate weight vector, store previous weight iteration
    weights = np.zeros((data.shape[1],1))
    prev = np.copy(weights)

    #ls = [] # list to store loss values at each iteration

    for iter in range(maxiter):

        # update weights
        weights -= lr*gradient(data, weights, labels)

        # compute and store loss
        #ls.append(loss(data, weights, labels))

        # check break condition, halting of necessary
        if np.linalg.norm(prev - weights, ord=2) < epsilon:
            return weights
        prev = np.copy(weights)

    #plt.plot(range(maxiter), ls, label='learning_rate: ' + str(lr))
    #plt.xlabel('iteration')
    #plt.ylabel('loss function')
    #plt.legend(loc='best')
    #plt.savefig('test.jpg')
    return weights
```

(note: the plotting code shown here is commented, though used in one test that will be shown shortly)

Then, $l_1$ regularized logistic regression, performed via sub gradient descent:

```
def logistic_l1_train(data: Array, labels: Array, par: float, lr:float, epsilon: float = 1e-5, maxiter:
int = 5000):
    """
    parameters:
        data   : n * (d + 1) with n samples and d features, where col d+1 is all ones (corresponding to
intercept)

        labels : n * 1 vector of class labels (0 or 1)

        par : regularization parameter

        lr : learning rate

        epsilon: optional parameter specifying convergence - if the change in absolute difference in
predictions, from one iteration to the next
                when averaged across input features is less than epsilon, halt (default 1e-5)

        maxiter: optional parameter that specifies the maximum number of iterations (default 1000)

    return: 1 * (d + 1) vector of weights corresponding to columns of data - last entry is bias term
    """
    # instantiate weight vector, store previous weight iteration
    weights = np.zeros((data.shape[1],1))
    prev = np.copy(weights)

    #ls = [] # list to store loss values

    for iter in range(maxiter):

        # update weights
        weights -= lr*l1_gradient(data, weights, labels, par)

        # compute and store loss
        #ls.append(l1_loss(data, weights, labels, par))

        # check break condition, halting if necessary
        if np.linalg.norm(prev - weights, ord=2) < epsilon:
            return weights
        prev = np.copy(weights)

    #plt.plot(range(maxiter), ls, label='par: ' + str(par))
    #plt.xlabel('iteration')
    #plt.ylabel('loss function')
    #plt.legend(loc='best')
    #plt.savefig('test.jpg')
    return weights
```

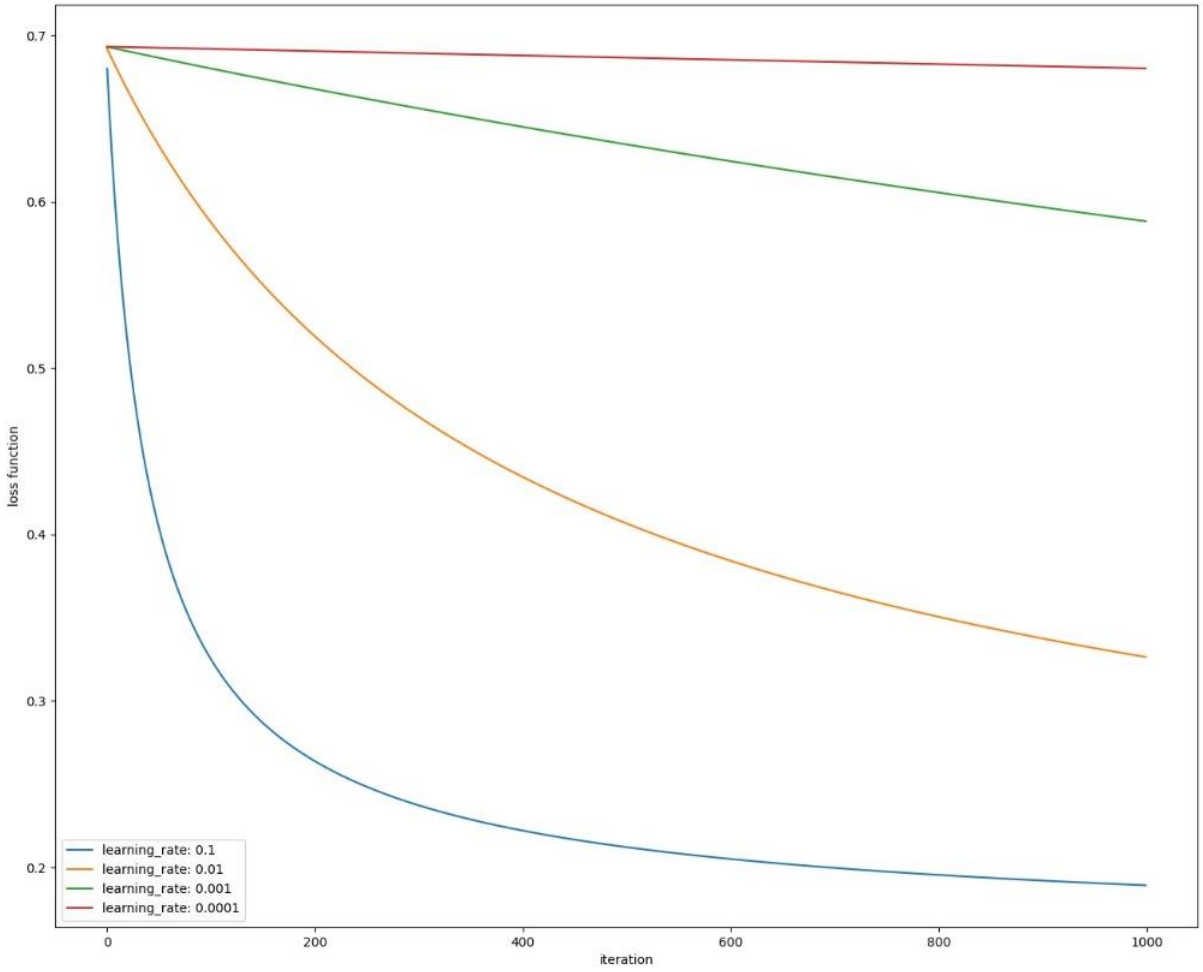
Using this code, I implemented the following tests.

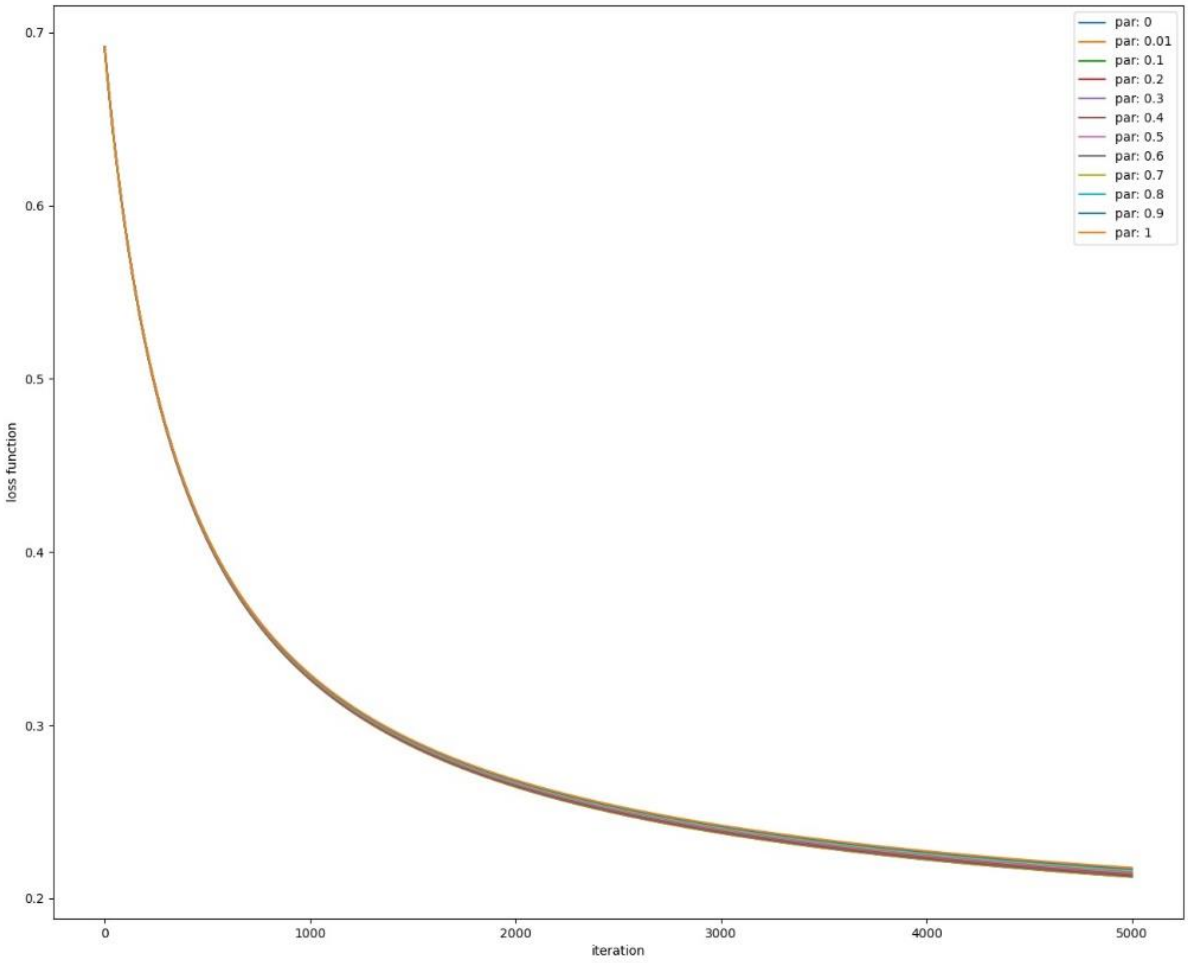
**Problem 1:**

First, I had to determine whether my code even worked. To do this, I made convergence plots on the first dataset using both types of regression, and a wide variety of parameters. The following code was used to drive the test:

```
def test1a():
    # simply looks at convergence for various parameters of dataset 1

    # read data
    data = np.loadtxt('data_spam.txt')
    labels = np.loadtxt('labels_spam.txt')

    # append column of 1 ones for bias term
    data = np.concatenate((data, np.ones((data.shape[0],1))), axis=1)

    # reshape labels
    labels = np.reshape(labels, (4601, 1))

    plt.figure(figsize=(16,13), dpi=100)
    for lr in [.1, .01, .001, .0001]:
        logistic_train(data, labels, lr)

    plt.figure(figsize=(16,13), dpi=100)
    for par in [0, .01, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1]:
        logistic_l1_train(data, labels, par, .01, maxiter=5000)
```

Producing the following output, and verifying convergence:




par: 0
par: 0.01
par: 0.1
par: 0.2
par: 0.3
par: 0.4
par: 0.5
par: 0.6
par: 0.7
par: 0.8
par: 0.9
par: 1
0.7
0.6
0.5
0.4
0.3
0.2
loss function
0
1000
2000
3000
4000
5000
iteration

Next, I performed the test specified in part one of the assignment – splitting the spam dataset into a testing and training set and uses varying portions of the training set to test our algorithm.

```
def test1b():
    # tests accuracy with various training sets

    # read data
    data = np.loadtxt('data_spam.txt')
    labels = np.loadtxt('labels_spam.txt')

    # append column of 1 ones for bias term
    data = np.concatenate((data, np.ones((data.shape[0],1))), axis=1)

    # reshape labels
    labels = np.reshape(labels, (4601, 1))

    # split data and labels for testing
    x_test, y_test = data[2000:, :], labels[2000:, :]

    table = pt.PrettyTable()
    table.field_names = ['Training Size', 'Testing Misclassifications']

    for training_size in [200, 500, 800, 1000, 1500, 2000]:
        w = logistic_train(data[:training_size,:], labels[:training_size], .05, maxiter=7500)
        prediction = predict(x_test, w)
        temp = 0
        for i in range(y_test.shape[0]):
            if prediction[i,:] != y_test[i,:]:
                temp += 1
        table.add_row([training_size, temp])

    print(table)
```

Which generated the following output (notice this test was performed without regularization, with a learning rate of .05 and 7500 iterations:

```
+---------------+----------------------------+
| Training Size | Testing Misclassifications |
+---------------+----------------------------+
|      200      |            198             |
|      500      |            179             |
|      800      |            175             |
|      1000     |            175             |
|      1500     |            164             |
|      2000     |            159             |
+---------------+----------------------------+
```

**Problem 2:**

The following code was used to test problem two. I considered a coefficient zero if it were less than $.0001$, and used a step size of $.01$.

```
def test2():
    # read data
    data = loadmat('ad_data.mat')
    x_train, y_train = data['X_train'], data['y_train']
    x_test, y_test = data['X_test'], data['y_test']

    # change encoding
    y_train = np.where(y_train == 1, 1, 0)
    y_test  = np.where(y_test == 1, 1, 0)

    table = pt.PrettyTable()
    table.field_names = ['Reg Param', 'Testing Misclassifications', 'Nonzero Weights']
    for par in [0, .01, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1]:
        w = logistic_l1_train(x_train, y_train, par, .01, epsilon=1e-20, maxiter=5000)
        prediction = predict(x_test, w)

        # count misclassifications
        temp1 = 0
        for i in range(y_test.shape[0]):
            if prediction[i,:] != y_test[i,:]:
                temp1 += 1
        # count nonzero weights
        temp2 = 0
        for i in range(w.shape[0]):
            if abs(w[i,:]) > 1e-3:
                temp2 += 1

        table.add_row([par, temp1, temp2])

    print(table)
```

Which produced the following table, showing behavior much as one would expect as the regularization parameter grew:

| Reg Param | Testing Misclassifications | Nonzero Weights |
|---|---|---|
| 0 | 19 | 116 |
| 0.01 | 19 | 116 |
| 0.1 | 20 | 105 |
| 0.2 | 19 | 86 |
| 0.3 | 19 | 75 |
| 0.4 | 19 | 68 |
| 0.5 | 19 | 60 |
| 0.6 | 19 | 48 |
| 0.7 | 19 | 45 |
| 0.8 | 19 | 41 |
| 0.9 | 19 | 38 |
| 1 | 19 | 31 |