Name: Adam Shaar

Submission:

1- Run all cells (this is important, the results will remain there for us to look)

2- Download .ipynb

3- Download .py

4- Use Save as or Print to create a PDF version of the notebook

5- Create a directory named: firstname_lastname_lr (e.g. pedram_rooshenas_lr)

6- Put all three .ipynb, .py, and .pdf into the directory. (*** Don't forget the PDF and .py ***)

7- Zip (don't use rar) and Submit on Gradescope

```
#Mounting Google Drive:
#After running this cell a popup window will appear and requesting to select your  Google account and give the access permission.
#You can either use your personal Google account or your UIC Google account.
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
path="/content/gdrive/MyDrive/mlcourse/hw2/"
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('ggplot')
%matplotlib inline
import re
import pandas as pd
```

Numpy is library for scientific computing in Python. It has efficient implementation of n-dimensional array (tensor) manupulations, which is useful for machine learning applications.

```
import numpy as np
```

We can convert a list into numpy array (tensor)

```
b = [[1, 2, 4], [2, 6, 9]]
a = np.array(b)
a
```

```
array([[1, 2, 4],
       [2, 6, 9]])
```

We can check the dimensions of the array

```
a.shape
```

```
(2, 3)
```

We can apply simple arithmetic operation on all element of a tensor

```
a * 3
```

```
array([[ 3,  6, 12],
       [ 6, 18, 27]])
```

You can transpose a tensor

```
print(a.T.shape)
a.T
```

```
(3, 2)
array([[1, 2],
       [2, 6],
       [4, 9]])
```

You can apply aggregate functions on the whole tensor

```
np.sum(a)
```

```
24
```

or on one dimension of it

```
np.sum(a, axis=0)
```

```
array([ 3,  8, 13])
```

```
np.sum(a, axis=1)
```

```
array([ 7, 17])
```

We can do element-wise arithmetic operation on two tensors (of the same size)

```
c1 = np.array([[1, 2, 4], [2, 6, 9]])
c2 = np.array([[2, 3, 5], [1, 2, 1]])
c1 * c2
```

```
array([[ 2,  6, 20],
       [ 2, 12,  9]])
```

If you want to multiply all columns of a tensor by vector (for example if you want to multiply all data features by their lables) you need a trick. This multiplication shows up in calculating the gradients.

```
a = np.array([[1, 2, 4], [2, 6, 9]])
b = np.array([1,-1])
print(a)
print(b)
```

```
[[1 2 4]
 [2 6 9]]
[ 1 -1]
```

Here we want to multiply the first row of a by 1 and the second row of a by -1. Simply multiplying a by b does not work because a and b do not have the same dimension

```
a * b
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call
last)
<ipython-input-127-9bc1a869709f> in <module>
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3)
(2,)
```

To do this multiplication we first have to assume b has one column and then repeat the column of b with the number of columns in a. We use tile function to do that

```
b_repeat = np.tile(b,  (a.shape[1],1)).T
print(b_repeat.shape)
b_repeat
```

```
(2, 3)
array([[ 1,  1,  1],
       [-1, -1, -1]])
```

Now we can multiply each column of a by b:

```
a * b_repeat
```

```
array([[ 1,  2,  4],
       [-2, -6, -9]])
```

You can create inital random vector using numpy (using N(0,1)):

```
mu = 0 #mean
sigma = 1 #standard deviation
r = np.random.normal(mu,sigma, 1000) #draws 1000 samples from a normal distribution
```

We can apply functions on tensors

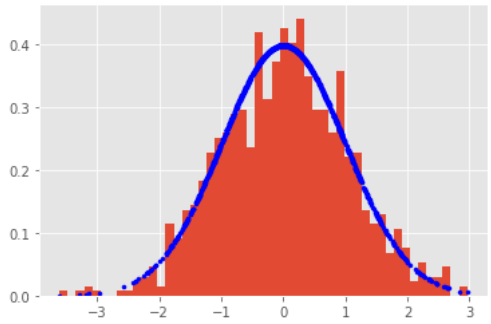
```
#implementation of Normal distribution
def normal(x, mu, sigma):
    return np.exp( -0.5 * ((x-mu)/sigma)**2)/np.sqrt(2.0*np.pi*sigma**2)

#probability of samples on the Normal distribution
probabilities = normal(r, mu, sigma)
```

Numpy has useful APIs for analysis. Here we plot the histogram of samples and also plot the probabilies to see if the samples follow the normal distribution.

```
counts, bins = np.histogram(r,50,density=True)
plt.hist(bins[:-1], bins, weights=counts)
plt.scatter(r, probabilities, c='b', marker='.')
```

```
<matplotlib.collections.PathCollection at 0x7fd9eda2d160>
```




```
def read_data(filename):
    f = open(filename, 'r')
    p = re.compile(',')
    xdata = []
    ydata = []
    header = f.readline().strip()
    varnames = p.split(header)
    namehash = {}
    for l in f:
        li = p.split(l.strip())
        xdata.append([float(x) for x in li[:-1]])
        ydata.append(float(li[-1]))

    return np.array(xdata), np.array(ydata)
```

Assuming our data is x is available in numpy we use numpy to implement logistic regression

```
(xtrain_whole, ytrain_whole) = read_data(path + 'spambase-train.csv')
(xtest, ytest) = read_data(path + 'spambase-test.csv')
```

```
print("The shape of xtrain:", xtrain_whole.shape)
print("The shape of ytrain:", ytrain_whole.shape)
print("The shape of xtest:", xtest.shape)
print("The shape of ytest:", ytest.shape)
```

```
The shape of xtrain: (3601, 54)
The shape of ytrain: (3601,)
The shape of xtest: (1000, 54)
The shape of ytest: (1000,)
```

before training make we normalize the input data (features)

```
xmean = np.mean(xtrain_whole, axis=0)
xstd = np.std(xtrain_whole, axis=0)
xtrain_normal_whole = (xtrain_whole-xmean) / xstd
xtest_normal = (xtest-xmean) / xstd
```

We need to create a validation set. We create an array of indecies and permute it.

```
premute_indicies = np.random.permutation(np.arange(xtrain_whole.shape[0]))
```

We keep the first 2600 data points as the training data and rest as the validation data

```
xtrain_normal = xtrain_normal_whole[premute_indicies[:2600]]
ytrain = ytrain_whole[premute_indicies[:2600]]
xval_normal = xtrain_normal_whole[premute_indicies[2600:]]
yval = ytrain_whole[premute_indicies[2600:]]
```

Initiallizing the weights and bias with random values from N(0,1)

```
weights = np.random.normal(0, 1, xtrain_normal.shape[1]);
bias = np.random.normal(0,1,1)
```

```
#the sigmoid function
def sigmoid(v):
    #return np.exp(-np.logaddexp(0, -v)) #numerically stable implementation of sigmoid function
    return 1.0 / (1+np.exp(-v))
```

We can use dot-product from numpy to calculate the margin and pass it to the sigmoid function

```
#w: weight vector (numpy array of size n)
#b: numpy array of size 1
#returns p(y=1|x, w, b)
def prob(x, w, b):
    return sigmoid(np.dot(x,w) + b);
```

You can also calculate l_2 penalty using linalg library of numpy

```
np.linalg.norm(weights)
```

```
7.22076369886441
```

$$\text{Cross Entropy Loss} = -\frac{1}{|D|}[\sum_{(y^i,\mathbf{x}^i)\in} y^i \log p(y=1|\mathbf{x}^i;\mathbf{w},b) + (1-y^i)\log(1-p(y=1|\mathbf{x}^i;\mathbf{w},b))] + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

```
#w: weight vector (numpy array of size n)
#x: training data points (only attributes)
#y_prob: p(y|x, w, b)
#y_true: class variable data
#lambda_: l2 penalty coefficient
#returns the cross entropy loss
def loss(w, x, y_prob, y_true, lambda_):
    return ((-1 / len(x)) * np.sum(y_true * np.log(y_prob) + (1 - y_true) * np.log(1 - y_prob))) + (lambda_ / 2) * np.sum(np.squar



#x: input variables (data of size m x n with m data point and n features)
#w: weight vector (numpy array of size n)
#y_prob: p(y|x, w, b)
#y_true: class variable data
#lambda_: l2 penalty coefficient
#returns tuple of gradient w.r.t w and w.r.t to bias

def grad_w_b(x, w, y_prob, y_true, lambda_):
    grad_w = ((1/ x.shape[1]) * np.dot(x.T, y_prob - y_true)) + lambda_ * w
    grad_b = (1 / len(y_true)) * np.sum(y_prob - y_true)
    return (grad_w,grad_b)



#lambda_ is the coeffienct of l2 norm penalty
#learning_rate is learning rate of gradient descent algorithm
#max_iter determines the maximum number of iterations if the gradients descent does not converge.
#continue the training while gradient > 0.1 or the number steps is less max_iter

#returns model as tuple of (weights,bias)

def fit(x, y_true, learning_rate, lambda_, max_iter, verbose=0):
    weights = np.random.normal(0, 1, x.shape[1]);
    bias = np.random.normal(0,1,1)
    iter = 0
    grad_w_norm = 1
    #change the condition appropriately
    while iter < max_iter and grad_w_norm > 0.1:
        y_prob = prob(x, weights, bias)
        grad_w, grad_b = grad_w_b(x, weights, y_prob, y_true, lambda_)
        weights -= learning_rate * grad_w
        bias -= learning_rate * grad_b
        weights_norm = np.linalg.norm(weights)
        grad_w_norm = np.linalg.norm(grad_w)
        if verbose: #verbose is used for debugging purposes
            loss_ = loss(weights, x, y_prob, y_true, lambda_)
            print(f"Iteration Number: {iter}, Loss: {loss_:5f}, l2 norm of gradients: {grad_w_norm:5f}, l2 norm of weights: {weigh
            #print iteration number, loss, l2 norm of gradients, l2 norm of weights
        iter += 1
    return (weights, bias)


def accuracy(x, y_true, model):
    w, b = model
    return np.sum((prob(x, w, b)>0.5).astype(np.float64) == y_true)  / y_true.shape[0]


learning_rate = 0.001
lambda_ = 1.0

model = fit(xtrain_normal, ytrain, learning_rate, lambda_, 10000, verbose=1) #keep the verbose on here for your submissions
```

```
Iteration Number: 2586, Loss: 2.140462, 12 norm of gradients: 0.105427, 12 norm of weights: 1.964446
Iteration Number: 2587, Loss: 2.140417, 12 norm of gradients: 0.105286, 12 norm of weights: 1.964424
Iteration Number: 2588, Loss: 2.140373, 12 norm of gradients: 0.105146, 12 norm of weights: 1.964402
Iteration Number: 2589, Loss: 2.140328, 12 norm of gradients: 0.105006, 12 norm of weights: 1.964380
Iteration Number: 2590, Loss: 2.140283, 12 norm of gradients: 0.104866, 12 norm of weights: 1.964358
Iteration Number: 2591, Loss: 2.140238, 12 norm of gradients: 0.104726, 12 norm of weights: 1.964336
Iteration Number: 2592, Loss: 2.140194, 12 norm of gradients: 0.104587, 12 norm of weights: 1.964314
Iteration Number: 2593, Loss: 2.140149, 12 norm of gradients: 0.104448, 12 norm of weights: 1.964292
Iteration Number: 2594, Loss: 2.140105, 12 norm of gradients: 0.104309, 12 norm of weights: 1.964270
Iteration Number: 2595, Loss: 2.140060, 12 norm of gradients: 0.104170, 12 norm of weights: 1.964248
Iteration Number: 2596, Loss: 2.140016, 12 norm of gradients: 0.104032, 12 norm of weights: 1.964226
Iteration Number: 2597, Loss: 2.139972, 12 norm of gradients: 0.103894, 12 norm of weights: 1.964204
Iteration Number: 2598, Loss: 2.139927, 12 norm of gradients: 0.103756, 12 norm of weights: 1.964182
Iteration Number: 2599, Loss: 2.139883, 12 norm of gradients: 0.103618, 12 norm of weights: 1.964160
Iteration Number: 2600, Loss: 2.139839, 12 norm of gradients: 0.103480, 12 norm of weights: 1.964138
Iteration Number: 2601, Loss: 2.139795, 12 norm of gradients: 0.103343, 12 norm of weights: 1.964116
Iteration Number: 2602, Loss: 2.139751, 12 norm of gradients: 0.103206, 12 norm of weights: 1.964094
Iteration Number: 2603, Loss: 2.139707, 12 norm of gradients: 0.103069, 12 norm of weights: 1.964072
Iteration Number: 2604, Loss: 2.139663, 12 norm of gradients: 0.102933, 12 norm of weights: 1.964051
Iteration Number: 2605, Loss: 2.139619, 12 norm of gradients: 0.102796, 12 norm of weights: 1.964029
Iteration Number: 2606, Loss: 2.139575, 12 norm of gradients: 0.102660, 12 norm of weights: 1.964007
Iteration Number: 2607, Loss: 2.139531, 12 norm of gradients: 0.102524, 12 norm of weights: 1.963986
Iteration Number: 2608, Loss: 2.139487, 12 norm of gradients: 0.102389, 12 norm of weights: 1.963964
Iteration Number: 2609, Loss: 2.139443, 12 norm of gradients: 0.102253, 12 norm of weights: 1.963942
Iteration Number: 2610, Loss: 2.139400, 12 norm of gradients: 0.102118, 12 norm of weights: 1.963921
Iteration Number: 2611, Loss: 2.139356, 12 norm of gradients: 0.101983, 12 norm of weights: 1.963899
Iteration Number: 2612, Loss: 2.139312, 12 norm of gradients: 0.101848, 12 norm of weights: 1.963877
Iteration Number: 2613, Loss: 2.139269, 12 norm of gradients: 0.101713, 12 norm of weights: 1.963856
Iteration Number: 2614, Loss: 2.139225, 12 norm of gradients: 0.101579, 12 norm of weights: 1.963834
Iteration Number: 2615, Loss: 2.139182, 12 norm of gradients: 0.101445, 12 norm of weights: 1.963813
Iteration Number: 2616, Loss: 2.139138, 12 norm of gradients: 0.101311, 12 norm of weights: 1.963791
Iteration Number: 2617, Loss: 2.139095, 12 norm of gradients: 0.101178, 12 norm of weights: 1.963770
Iteration Number: 2618, Loss: 2.139052, 12 norm of gradients: 0.101044, 12 norm of weights: 1.963748
Iteration Number: 2619, Loss: 2.139008, 12 norm of gradients: 0.100911, 12 norm of weights: 1.963727
Iteration Number: 2620, Loss: 2.138965, 12 norm of gradients: 0.100778, 12 norm of weights: 1.963706
Iteration Number: 2621, Loss: 2.138922, 12 norm of gradients: 0.100645, 12 norm of weights: 1.963684
Iteration Number: 2622, Loss: 2.138879, 12 norm of gradients: 0.100513, 12 norm of weights: 1.963663
Iteration Number: 2623, Loss: 2.138836, 12 norm of gradients: 0.100380, 12 norm of weights: 1.963642
Iteration Number: 2624, Loss: 2.138793, 12 norm of gradients: 0.100248, 12 norm of weights: 1.963620
Iteration Number: 2625, Loss: 2.138750, 12 norm of gradients: 0.100116, 12 norm of weights: 1.963599
Iteration Number: 2626, Loss: 2.138707, 12 norm of gradients: 0.099985, 12 norm of weights: 1.963578
```

```
print("Train accuracy: ", accuracy(xtrain_normal, ytrain, model))
```

```
Train accuracy:  0.9192307692307692
```

```
#grid search for finding the best hyperparams and model

best_model = None
best_val = -1
for lr in [0.01, 0.001, 0.0001, 0.00001]:
    for la in [5, 2, 1, 0.1, 0.01]:
        model = fit(xtrain_normal, ytrain, lr, la, 10000, verbose=0)
        val_acc = accuracy(xval_normal, yval, model)
        print(lr, la, val_acc)
        if val_acc > best_val:
            best_val = val_acc
            best_model = model
```

```
0.01 5 0.8621378621378621
0.01 2 0.9110889110889111
0.01 1 0.929070929070929
0.01 0.1 0.9240759240759241
0.01 0.01 0.9280719280719281
0.001 5 0.9090909090909091
0.001 2 0.919080919080919
0.001 1 0.9140859140859141
0.001 0.1 0.9330669330669331
0.001 0.01 0.9240759240759241
0.0001 5 0.9110889110889111
0.0001 2 0.913086913086913
0.0001 1 0.9240759240759241
0.0001 0.1 0.919080919080919
0.0001 0.01 0.9070929070929071
1e-05 5 0.8161838161838162
1e-05 2 0.7452547452547452
1e-05 1 0.7442557442557443
1e-05 0.1 0.7902097902097902
1e-05 0.01 0.7572427572427572
```

```
print("Test accuracy: ", accuracy(xtest_normal, ytest, best_model))
```

```
print( 'Test accuracy:  ', accuracy(xtest_normal, ytest, best_model))
```

```
Test accuracy:  0.941
```

✓ 0s completed at 7:37 PM