```
Name: Nachiket Patwardhan   |  student_ID: 801208762
```

In [189]:
```
import numpy as np
import sklearn.datasets as ds
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.

In [190]:
```
import numpy as np
```

We can convert a list into numpy array (tensor)

In [191]:
```
b = [[1, 2, 4], [2, 6, 9]]
a = np.array(b)
a
```

Out[191]:
```
array([[1, 2, 4],
       [2, 6, 9]])
```

We can check the dimensions of the array

In [192]:
```
a.shape
```

Out[192]:
```
(2, 3)
```

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

In [193]:
```
a * 3
```

Out[193]:
```
array([[ 3,  6, 12],
       [ 6, 18, 27]])
```

You can transpose a tensor

In [194]:
```
print(a.T.shape)
a.T
```

```
(3, 2)
```

Out[194]:
```
array([[1, 2],
       [2, 6],
       [4, 9]])
```

You can apply aggregate functions on the whole tensor

In [195]:
```
np.sum(a)
```

Out[195]:
```
24
```

or on one dimension of it

In [196]:
```
np.sum(a, axis=0)
```

Out[196]:
```
array([ 3,  8, 13])
```

In [197]:
```
np.sum(a, axis=1)
```

Out[197]:
```
array([ 7, 17])
```

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

In [198]:
```
c1 = np.array([[1, 2, 4], [2, 6, 9]])
c2 = np.array([[2, 3, 5], [1, 2, 1]])
c1 * c2
```

Out[198]:
```
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.

In [199]:
```
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

In [200]:
```
a * b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-200-9bc1a869709f> 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

In [201]:
```
b_repeat = np.tile(b,  (a.shape[1],1)).T
print(b_repeat.shape)
b_repeat
```

```
(2, 3)
```

Out[201]:
```
array([[ 1,  1,  1],
       [-1, -1, -1]])
```

Now we can multiply each column of a by b:

In [202]:
```
a * b_repeat
```

Out[202]:
```
array([[ 1,  2,  4],
       [-2, -6, -9]])
```

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

In [203]:
```
mu = 0 #mean
sigma = 1 #standard deviation
r = np.random.normal(mu,sigma, 1000) #draws 1000 samples from a normal distributi
```

We can apply functions on tensors

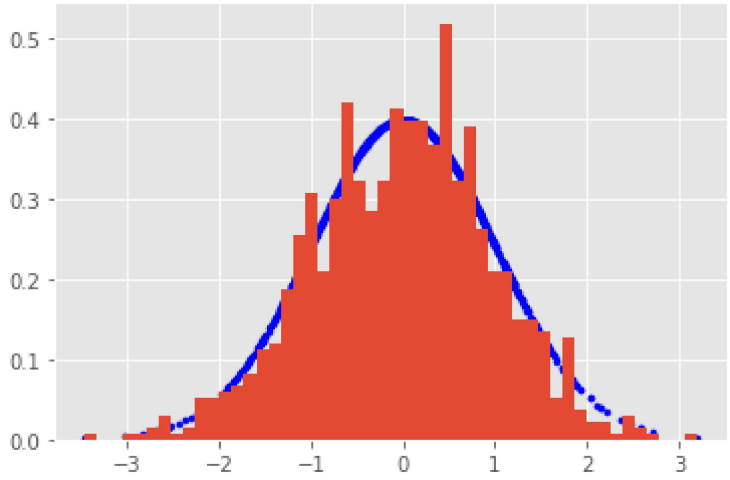
In [204]:
```
#implementation of Normal distribution
def normal(x, mu, sigma):
    return np.exp( -0.5 * ((x-mu)/sigma)**2)/np.sqrt(2.0*np.pi*sigma**2)

#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.

In [205]:
```
counts, bins = np.histogram(r,50,density=True)
plt.hist(bins[:-1], bins, weights=counts)
plt.scatter(r, probabilities, c='b', marker='.')
```

Out[205]:
```
<matplotlib.collections.PathCollection at 0x132a0a830d0>
```



In [206]:
```
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

In [207]:
```
(xtrain_whole, ytrain_whole) = read_data('spambase-train.csv')
(xtest, ytest) = read_data('spambase-test.csv')
```

In [208]:
```
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)

In [209]:
```
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.

In [210]:
```
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

In [211]:
```
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)

In [212]:
```
weights = np.random.normal(0, 1, xtrain_normal.shape[1]);
bias = np.random.normal(0,1,1)
```

In [213]:
```
#the sigmoid function
def sigmoid(z):
    s = 1 / (1 + np.exp(-z))
    return s
```

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

In [214]:
```
#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):
    z = sigmoid(np.dot(x,w) + b)
    #print(z)
    return z
```

You can also calculate l_2 penalty using linalg library of numpy

In [215]:
```
np.linalg.norm(weights)
```

Out[215]:
```
7.412178281536248
```

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

In [216]:
```
#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 the cross entropy loss
def loss(w, y_prob, y_true, lambda_):
    Cross_Entropy_loss = -(1) * np.sum((y_true * np.log10(y_prob)) + ((1 - y_true
    return Cross_Entropy_loss
```

In [217]:
```
 #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 = np.zeros(w.shape)
    #m = len(x)
    grad_b = 0.0
    grad_w = np.dot(x.T, (y_prob - y_true)) + (2 * lambda_ * w)
    grad_b = np.sum(y_prob - y_true)

    return(grad_w, grad_b)
```

In [224]:
```
#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 do
#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)
    #raise NotImplementedError
    costs = []
    i = 0
    while(i < max_iter):
        #To calculate y_prob
        y_prob = prob(x, weights, bias)

        Cross_Entropy_loss = loss(weights, y_prob, y_true, lambda_)

        #costs = np.append(cost)

        grad_w, grad_b = grad_w_b(x, weights, y_prob, y_true, lambda_)

        #print(grad_w)
        #print(grad_b)

        #Condition to break if grad_w and grad_b is less than 0.1
        if((np.linalg.norm(grad_w) < 0.1) and (np.linalg.norm(grad_b) < 0.1)):
            break
        else:
            weights = weights - learning_rate * grad_w

            bias = bias - learning_rate * grad_b

        i = i+1


    return (weights, bias)
```

In [225]:
```
def accuracy(x, y_true, model):
    w, b = model
    return np.sum((prob(x, w, b)>0.5).astype(float) == y_true)  / y_true.shape[0]
```

In [226]:
```
learning_rate = 0.001
lambda_ = 1.0

model = fit(xtrain_normal, ytrain, learning_rate, lambda_, 10000, verbose=1)
```

In [227]:
```
print("Train accuracy: ", accuracy(xtrain_normal, ytrain, model))
```

```
Train accuracy:  0.9338461538461539
```

In [228]:
```
#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)
        val_acc = accuracy(xval_normal, yval, model)
        print(lr, la, val_acc)
        if val_acc > best_val:
            best_val = val_acc
            best_model = model
```

```
<ipython-input-216-e5e81e887299>:7: RuntimeWarning: divide by zero encountered
in log10
  Cross_Entropy_loss = -(1) * np.sum((y_true * np.log10(y_prob)) + ((1 - y_tru
e) * np.log10(1 - y_prob))) + ((lambda_/2) * (w**2))
<ipython-input-216-e5e81e887299>:7: RuntimeWarning: invalid value encountered i
n multiply
  Cross_Entropy_loss = -(1) * np.sum((y_true * np.log10(y_prob)) + ((1 - y_tru
e) * np.log10(1 - y_prob))) + ((lambda_/2) * (w**2))

0.01 5 0.7922077922077922
0.01 2 0.903096903096903
0.01 1 0.9150849150849151
0.01 0.1 0.9050949050949051
0.01 0.01 0.8561438561438561
0.001 5 0.9300699300699301
0.001 2 0.9330669330669331
0.001 1 0.9340659340659341
0.001 0.1 0.9340659340659341
0.001 0.01 0.9340659340659341
0.0001 5 0.9300699300699301
0.0001 2 0.9330669330669331
0.0001 1 0.9340659340659341
0.0001 0.1 0.9340659340659341
0.0001 0.01 0.9340659340659341
1e-05 5 0.9300699300699301
1e-05 2 0.9330669330669331
1e-05 1 0.9340659340659341
1e-05 0.1 0.936063936063936
1e-05 0.01 0.9340659340659341
```

In [229]:
```
print("Test accuracy: ", accuracy(xtest_normal, ytest, best_model))
```

```
Test accuracy:  0.941
```

In [ ]:
```

```