Homework 1

Student ID: 2020321249

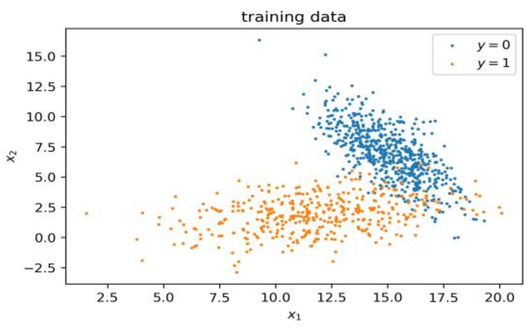
Name: 남혜린 Hyelin Nam

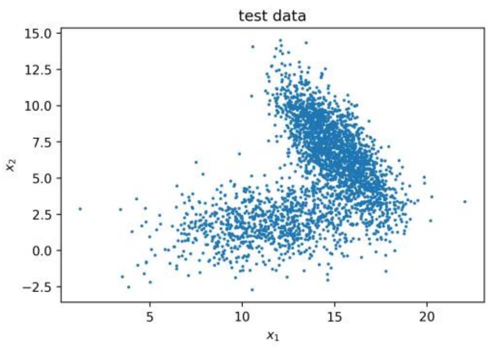
(i)

a) Training data, Test data

I can easily see data from each class are distributed discriminately.

Even though the class of test data are given, I assumed that I have no information of them, in order to purely inference the class of given test data with built model. However, I can easily predict the bottom samples and data distributed diagonally at right side belong to different classes.





(ii)

First, I predict classes of each data sample, with different methods such as MLE or model output. Each will be described in detail at problems below.

Then count the difference of predicted class and ground truth. This devided by number of total inferenced data means error rate. (1-error rate)*100 is accuracy in percentage.

(iii)

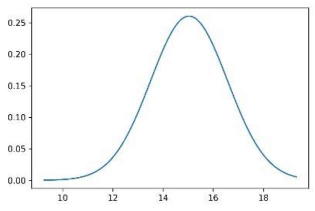
a) Maximum Likelihood Estimation

In Naïve Bayes decision output, likelihood of a sample is in class 0 and that in class 1 is compared. The class with bigger likelihood value is the prediction result.

$$prediction(x) = \begin{cases} 0 & ,if\ likelihood\ p(x|\ y=0) > p(x|\ y=1) \\ 1 & ,if\ likelihood\ p(x|\ y=1) > p(x|\ y=0) \end{cases}$$

When calculating likelihood, each features of sample are independently distributed. Therefore, likelihood is multiplication of likelihood of each feature.

$$p(x|\ y=0) = p(x_1|y=0) \cdot p(x_2|y=0)$$

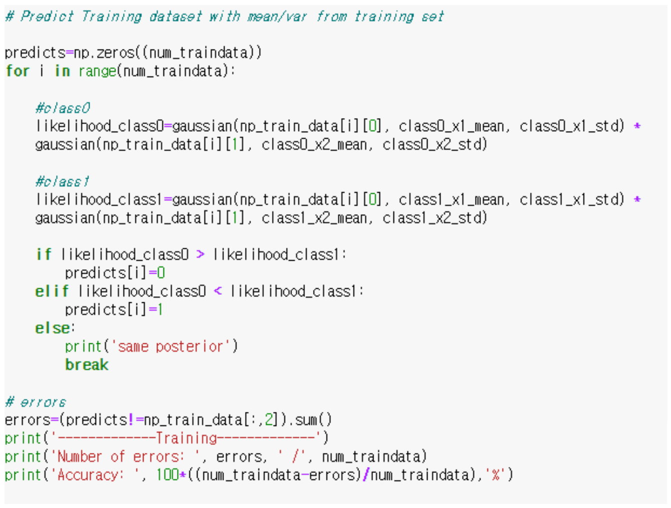


Here, each feature follows Gaussian distribution. This is plot of likelihood

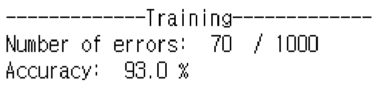
$p(x_1|y=0)$, which follows $N\sim(\mu(x_1|y=0), \sigma^2(x_1|y=0))$

b) Predict Train dataset with mean/var from training set

Like above, I measured likelihood of each data by calculating gaussian pdf with mean and std of training set belong to each class. It resulted 93% of accuracy, with number of errors 70 out of 1,000 training set.



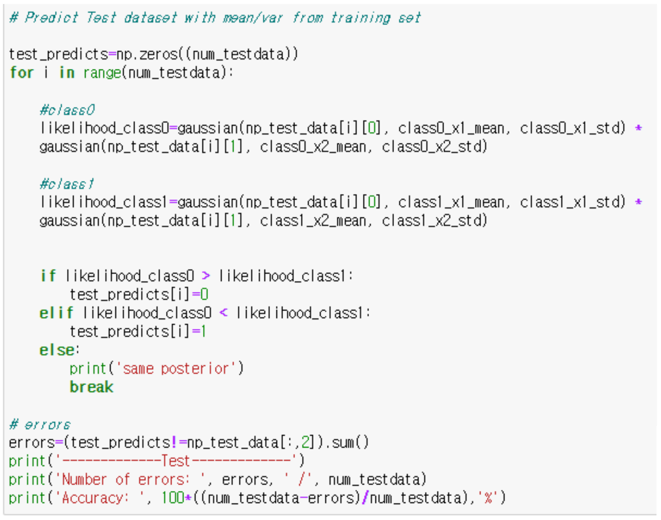
```
# Predict Training dataset with mean/var from training set

predicts=np.zeros((num_traindata))
for i in range(num_traindata):

    #class0
    likelihood_class0=gaussian(np_train_data[i][0], class0_x1_mean, class0_x1_std) *
    gaussian(np_train_data[i][1], class0_x2_mean, class0_x2_std)

    #class1
    likelihood_class1=gaussian(np_train_data[i][0], class1_x1_mean, class1_x1_std) *
    gaussian(np_train_data[i][1], class1_x2_mean, class1_x2_std)

    if likelihood_class0 > likelihood_class1:
        predicts[i]=0
    elif likelihood_class0 < likelihood_class1:
        predicts[i]=1
    else:
        print('same posterior')
        break

# errors
errors=(predicts!=np_train_data[:,2]).sum()
print('-------------Training-------------')
print('Number of errors: ', errors, ' /', num_traindata)
print('Accuracy: ', 100*((num_traindata-errors)/num_traindata),'%')
```



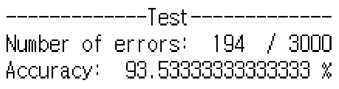
```
-------------Training-------------
Number of errors:  70  / 1000
Accuracy:  93.0 %
```

c) Predict Test dataset with mean/var from training set

Likewise, with gaussian distribution generated with training dataset, I measured likelihood of each test data. If the likelihood of the datum with condition of class 0 is bigger than that of class 1, I made a decision with the sample as class 0. It resulted 93.53% of accuracy, with number of errors 194 out of 1,000 training set.



```
# Predict Test dataset with mean/var from training set

test_predicts=np.zeros((num_testdata))
for i in range(num_testdata):

    #class0
    likelihood_class0=gaussian(np_test_data[i][0], class0_x1_mean, class0_x1_std) *
    gaussian(np_test_data[i][1], class0_x2_mean, class0_x2_std)

    #class1
    likelihood_class1=gaussian(np_test_data[i][0], class1_x1_mean, class1_x1_std) *
    gaussian(np_test_data[i][1], class1_x2_mean, class1_x2_std)


    if likelihood_class0 > likelihood_class1:
        test_predicts[i]=0
    elif likelihood_class0 < likelihood_class1:
        test_predicts[i]=1
    else:
        print('same posterior')
        break

# errors
errors=(test_predicts!=np_test_data[:,2]).sum()
print('-------------Test-------------')
print('Number of errors: ', errors, ' /', num_testdata)
print('Accuracy: ', 100*((num_testdata-errors)/num_testdata),'%')
```



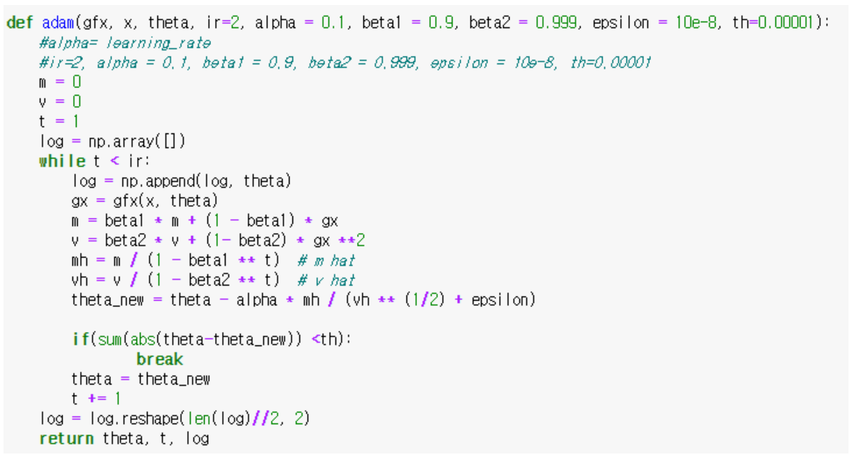
```
-------------Test-------------
Number of errors:  194  / 3000
Accuracy:  93.53333333333333 %
```

d) Training

Also, we can define $\theta = [\theta_1, \theta_2]^T$ that comprises each mean and std of Gaussian distribution that can describe the likelihood probability of data.

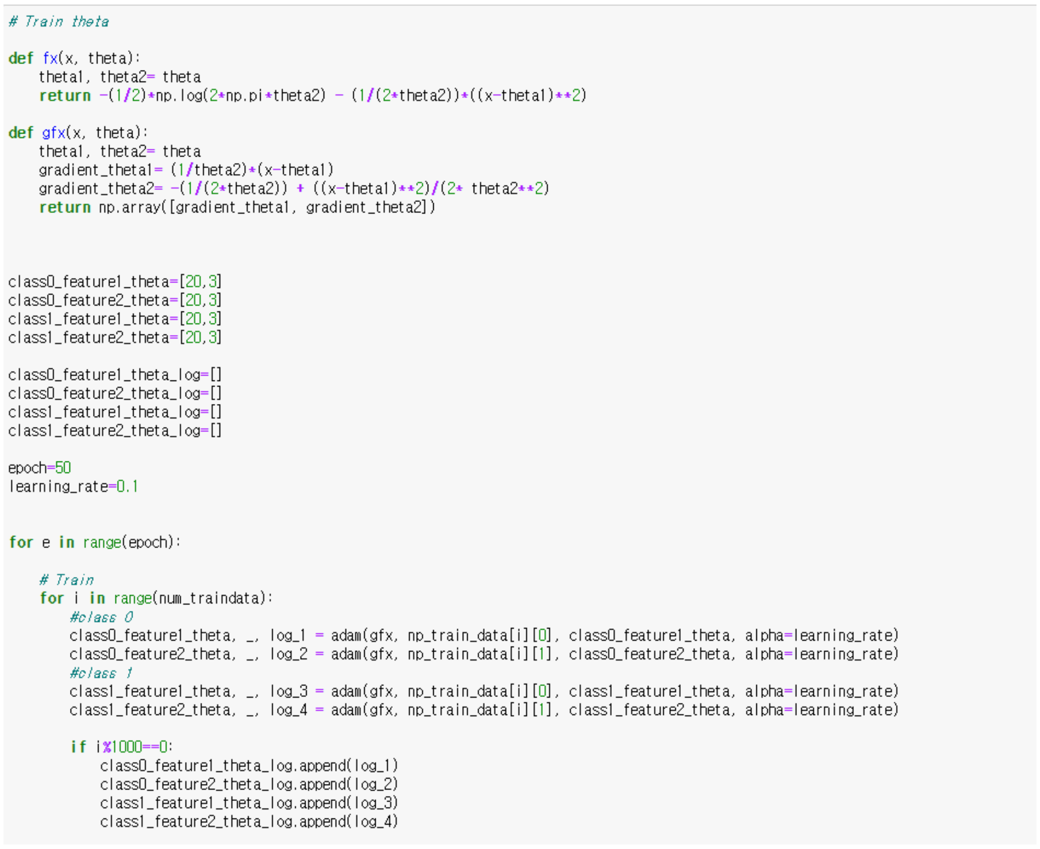
I calculated gradient of log-likelihood as below, and updated the θ with adam optimizer.

Adam optimizer with defined gradient (gfx). x means input, theta is the updating parameters, and alpha is learning rate



```
def adam(gfx, x, theta, ir=2, alpha = 0.1, beta1 = 0.9, beta2 = 0.999, epsilon = 10e-8, th=0.00001):
    #alpha= learning_rate
    #ir=2, alpha = 0.1, beta1 = 0.9, beta2 = 0.999, epsilon = 10e-8, th=0.00001
    m = 0
    v = 0
    t = 1
    log = np.array([])
    while t < ir:
        log = np.append(log, theta)
        gx = gfx(x, theta)
        m = beta1 * m + (1 - beta1) * gx
        v = beta2 * v + (1- beta2) * gx **2
        mh = m / (1 - beta1 ** t)  # m hat
        vh = v / (1 - beta2 ** t)  # v hat
        theta_new = theta - alpha * mh / (vh ** (1/2) + epsilon)

        if(sum(abs(theta-theta_new)) <th):
                break
        theta = theta_new
        t += 1
    log = log.reshape(len(log)//2, 2)
    return theta, t, log
```

Training



```
# Train theta

def fx(x, theta):
    theta1, theta2= theta
    return -(1/2)*np.log(2*np.pi*theta2) - (1/(2*theta2))*((x-theta1)**2)

def gfx(x, theta):
    theta1, theta2= theta
    gradient_theta1= (1/theta2)*(x-theta1)
    gradient_theta2= -(1/(2*theta2)) + ((x-theta1)**2)/(2* theta2**2)
    return np.array([gradient_theta1, gradient_theta2])



class0_feature1_theta=[20,3]
class0_feature2_theta=[20,3]
class1_feature1_theta=[20,3]
class1_feature2_theta=[20,3]

class0_feature1_theta_log=[]
class0_feature2_theta_log=[]
class1_feature1_theta_log=[]
class1_feature2_theta_log=[]

epoch=50
learning_rate=0.1


for e in range(epoch):

    # Train
    for i in range(num_traindata):
        #class 0
        class0_feature1_theta, _, log_1 = adam(gfx, np_train_data[i][0], class0_feature1_theta, alpha=learning_rate)
        class0_feature2_theta, _, log_2 = adam(gfx, np_train_data[i][1], class0_feature2_theta, alpha=learning_rate)
        #class 1
        class1_feature1_theta, _, log_3 = adam(gfx, np_train_data[i][0], class1_feature1_theta, alpha=learning_rate)
        class1_feature2_theta, _, log_4 = adam(gfx, np_train_data[i][1], class1_feature2_theta, alpha=learning_rate)

        if i%1000==0:
            class0_feature1_theta_log.append(log_1)
            class0_feature2_theta_log.append(log_2)
            class1_feature1_theta_log.append(log_3)
            class1_feature2_theta_log.append(log_4)
```

```python
    # Test with training data itself
    predicts=np.zeros((num_traindata))
    for i in range(num_traindata):
        #class0
        log_likelihood_class0_feature1=fx(np_train_data[i][0], class0_feature1_theta)
        log_likelihood_class0_feature2=fx(np_train_data[i][1], class0_feature2_theta)
        log_likelihood_class0= log_likelihood_class0_feature1 + log_likelihood_class0_feature2
        #class1
        log_likelihood_class1_feature1=fx(np_train_data[i][0], class1_feature1_theta)
        log_likelihood_class1_feature2=fx(np_train_data[i][1], class1_feature2_theta)
        log_likelihood_class1= log_likelihood_class1_feature1 + log_likelihood_class1_feature2

        if log_likelihood_class0 > log_likelihood_class1:
            predicts[i]=0
        elif log_likelihood_class0 < log_likelihood_class1:
            predicts[i]=1
        else:
            print('same posterior')
            break


    # errors
    errors=(predicts!=np_train_data[:,2]).sum()
    print('Epoch: ',e)
    print('Number of errors: ', errors, ' /', num_traindata)
    print('Accuracy: ', 100*((num_traindata-errors)/num_traindata),'%')


    if e==epoch-1:
        for log_idx in range(len(class0_feature1_theta_log)):
            plotLog(class0_feature1_theta_log[log_idx])
        plt.title('class0_feature1_theta')
        plt.legend('',frameon=False)
        plt.show()

        for log_idx in range(len(class0_feature2_theta_log)):
            plotLog(class0_feature2_theta_log[log_idx])
        plt.title('class0_feature2_theta')
        plt.legend('',frameon=False)
        plt.show()
```

```python
        for log_idx in range(len(class0_feature2_theta_log)):
            plotLog(class0_feature2_theta_log[log_idx])
        plt.title('class0_feature2_theta')
        plt.legend('',frameon=False)
        plt.show()

        for log_idx in range(len(class1_feature1_theta_log)):
            plotLog(class1_feature1_theta_log[log_idx])
        plt.title('class1_feature1_theta')
        plt.legend('',frameon=False)
        plt.show()

        for log_idx in range(len(class1_feature2_theta_log)):
            plotLog(class1_feature2_theta_log[log_idx])
        plt.title('class1_feature2_theta')
        plt.legend('',frameon=False)
        plt.show()
```

The Training accuracy was 89.7%, with 103 errors.

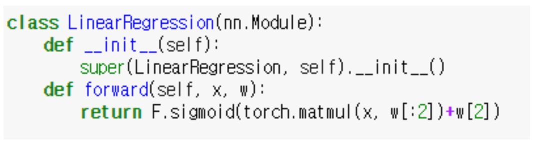
The Test accuracy was 80.93% ,with 572 errors.

(iv, vii)

In linear regression, I built an network with 2-dimension weight and bias. For loss function, I use Mean Squre Error loss. With optimizing the weights with SGD optimizer, a linear decision boundary came out. It seems weakly performing, since the line do not discriminate data from different classes. For reasons, I can predict wrong initial value of weights, since I generated with random values, not proper activation function, sigmoid. Also, other loss functions such as cross entropy may work better.

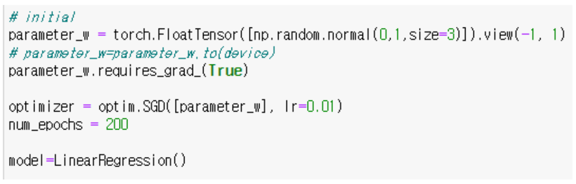
When the model output is closer to 1 than 0, in other words, the output is larger than 0.5, the datum is predicted to class 0. The training accuracy is 61.19% and the test accuracy s 51.93%

Model



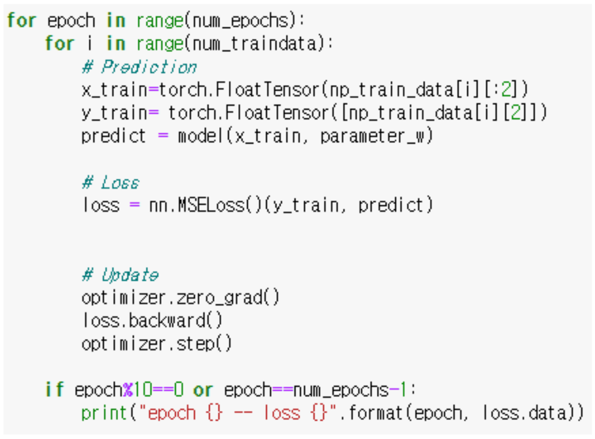
```
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
    def forward(self, x, w):
        return F.sigmoid(torch.matmul(x, w[:2])+w[2])
```

Parameters



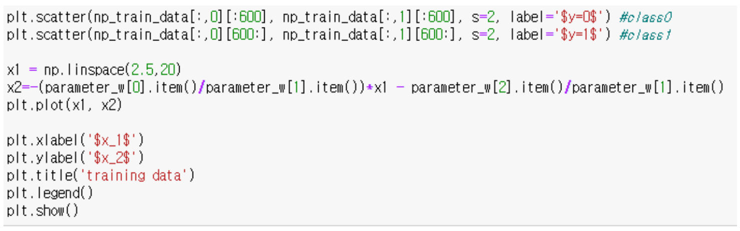
```
# initial
parameter_w = torch.FloatTensor([np.random.normal(0,1,size=3)]).view(-1, 1)
# parameter_w=parameter_w.to(device)
parameter_w.requires_grad_(True)

optimizer = optim.SGD([parameter_w], lr=0.01)
num_epochs = 200

model=LinearRegression()
```

Training

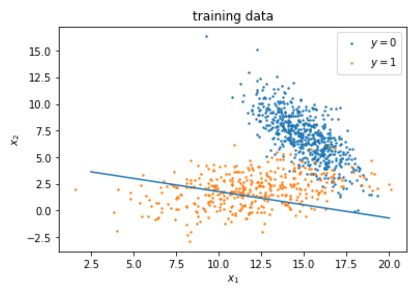


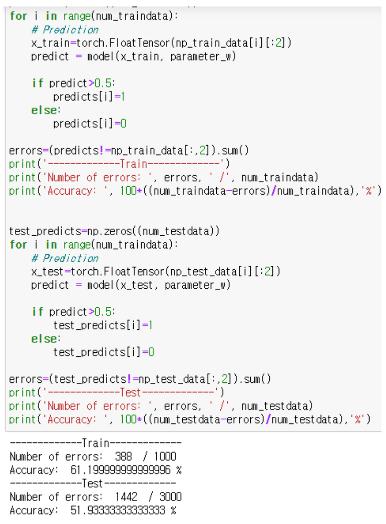
```
for epoch in range(num_epochs):
    for i in range(num_traindata):
        # Prediction
        x_train=torch.FloatTensor(np_train_data[i][:2])
        y_train= torch.FloatTensor([np_train_data[i][2]])
        predict = model(x_train, parameter_w)

        # Loss
        loss = nn.MSELoss()(y_train, predict)


        # Update
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if epoch%10==0 or epoch==num_epochs-1:
        print("epoch {} -- loss {}".format(epoch, loss.data))
```

Results



```
plt.scatter(np_train_data[:,0][:600], np_train_data[:,1][:600], s=2, label='$y=0$') #class0
plt.scatter(np_train_data[:,0][600:], np_train_data[:,1][600:], s=2, label='$y=1$') #class1

x1 = np.linspace(2.5,20)
x2=-(parameter_w[0].item()/parameter_w[1].item())*x1 - parameter_w[2].item()/parameter_w[1].item()
plt.plot(x1, x2)

plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('training data')
plt.legend()
plt.show()
```





```
for i in range(num_traindata):
    # Prediction
    x_train=torch.FloatTensor(np_train_data[i][:2])
    predict = model(x_train, parameter_w)

    if predict>0.5:
        predicts[i]=1
    else:
        predicts[i]=0

errors=(predicts!=np_train_data[:,2]).sum()
print('------------Train------------')
print('Number of errors: ', errors, ' /', num_traindata)
print('Accuracy: ', 100*((num_traindata-errors)/num_traindata),'%')


test_predicts=np.zeros((num_testdata))
for i in range(num_traindata):
    # Prediction
    x_test=torch.FloatTensor(np_test_data[i][:2])
    predict = model(x_test, parameter_w)

    if predict>0.5:
        test_predicts[i]=1
    else:
        test_predicts[i]=0

errors=(test_predicts!=np_test_data[:,2]).sum()
print('------------Test------------')
print('Number of errors: ', errors, ' /', num_testdata)
print('Accuracy: ', 100*((num_testdata-errors)/num_testdata),'%')
```

```
------------Train------------
Number of errors:  388  / 1000
Accuracy:  61.199999999999996 %
------------Test------------
Number of errors:  1442  / 3000
Accuracy:  51.93333333333333 %
```

(v, vi)

In k-NN, when predicting for one sample, I draw a bound with k nearest samples in Euclidean distance samples. If the number of data belong to class 0 is more than that of class1 in the bound, the corresponding sample is predicted to class 0. After predicting with train data, I draw a Voronoi cell with regard to the prediction, then in inference with test dataset, each test sample follows the class of nearest dot, which is a data in the cell where the test sample is located.

Euclidean distance

```
def distance(a,b):
    # Euclidean distance

    temp=a-b
    distance = np.sqrt(np.dot(temp.T, temp))

    return distance
```

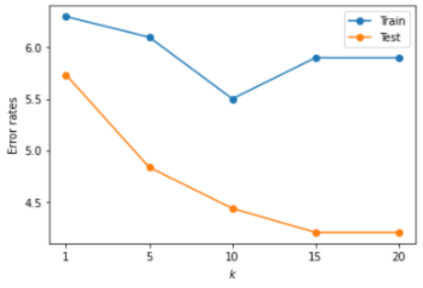
Calculating distance

```
# Train
# Calculate distances
distances=np.zeros((num_traindata, num_traindata))
for i in range(num_traindata):
    for j in range(num_traindata):
        if i==j:
            distances[i][j]=10000
        elif distances[j][i]!=0:
            distances[i][j]=distances[j][i]
        else:
            distances[i][j]= distance(np_train_data[i][:2], np_train_data[j][:2])


# Test
# Calculate distances
test_distances=np.zeros((num_testdata, num_traindata))
for i in range(num_testdata):
    for j in range(num_traindata):
        test_distances[i][j]= distance(np_test_data[i][:2], np_train_data[j][:2])
```

Train and test with different ks.

```
Ks=[1,5,10,15,20]
train_errors=[]
test_errors=[]

for k in Ks:
    print('k=',k,'-----------------------------------')
    # Train with train data
    predicts=np.copy(np_train_data)
    for i in range(num_traindata):
        distances_i=distances[i]
        indexes_i=distances_i.argsort()[:k]
        labels_i=np_train_data[:,2][indexes_i]

        if (labels_i==0).sum() > (labels_i==1).sum():
            predicts[i][2]=0
        elif (labels_i==1).sum() > (labels_i==0).sum():
            predicts[i][2]=1

    # Training error
    errors=(predicts[:,2]!=np_train_data[:,2]).sum()
    train_errors.append(errors)
    print('Training:')
    print('Number of errors: ', errors)
    print('Accuracy: ', 100*((num_traindata-errors)/num_traindata),'%')


    # Predict each test data
    test_predicts=np.copy(np_test_data)
    for i in range(num_testdata):
        distances_i=test_distances[i]
        indexes_i=distances_i.argsort()[0]
        labels_i=predicts[:,2][indexes_i]

        if labels_i==0:
            test_predicts[i][2]=0
        elif labels_i==1:
            test_predicts[i][2]=1

    # Test error
    errors=(test_predicts[:,2]!=np_test_data[:,2]).sum()
    test_errors.append(errors)
    print('Test:')
    print('Number of errors: ', errors)
    print('Accuracy: ', 100*((num_traindata-errors)/num_testdata),'%')
```

```
k= 1 -----------------------------------
Training:
Number of errors:  63
Accuracy:  93.7 %
Test:
Number of errors:  172
Accuracy:  27.6 %
k= 5 -----------------------------------
Training:
Number of errors:  61
Accuracy:  93.89999999999999 %
Test:
Number of errors:  145
Accuracy:  28.499999999999996 %
k= 10 -----------------------------------
Training:
Number of errors:  55
Accuracy:  94.5 %
Test:
Number of errors:  133
Accuracy:  28.9 %
k= 15 -----------------------------------
Training:
Number of errors:  59
Accuracy:  94.1 %
Test:
Number of errors:  126
Accuracy:  29.133333333333333 %
k= 20 -----------------------------------
Training:
Number of errors:  59
Accuracy:  94.1 %
Test:
Number of errors:  126
Accuracy:  29.133333333333333 %
```



(viii, ix) is included in the above answers.