NNFL ASSIGNMENT 1
Harshul Gupta
2018B5A31058H

Question1 :
https://colab.research.google.com/drive/1KzTvbjm0qkJMrqFXOu1CPlqKp3jvZbvh?authuser=1

Question 2 :

https://colab.research.google.com/drive/1A8MviRrRoJI7IS9J30k2MrUkSdYxsEKy?authuser=1

question 3 :

https://colab.research.google.com/drive/1InYAJlXlQHJLyC-U5l9fZyVQvSdK2MsC?authuser=1

question 4:

https://colab.research.google.com/drive/1sKxqC-ZDsOmTeVJY-PrrEUHflF6IIS8D?authuser=1

Question 5

https://colab.research.google.com/drive/1DY_bZUle9h1ZHaNydp0d3_7L6qTs1vmT?authuser=1

Question 6:

https://colab.research.google.com/drive/1wbQvXMZIWtI5R9NB4j_X-8jMJ5bioKck?authuser=1

question 8

https://colab.research.google.com/drive/13YHDkb41IBk0ffltm-nfw6IlhCjUowuy?authuser=1

Question 9

https://colab.research.google.com/drive/1mB0a1NOPquzPGdyJM7Yqff-f8VLU9srb?authuser=1

Question 10

https://colab.research.google.com/drive/1fNYXaVE2t6fCYY_YeN9HQTTAy8ZTTpGB?authuser=1

Question 11

https://colab.research.google.com/drive/1T2QsXuN2jv-PkaBC2DOYcivhDbUrXix_?authuser=1#

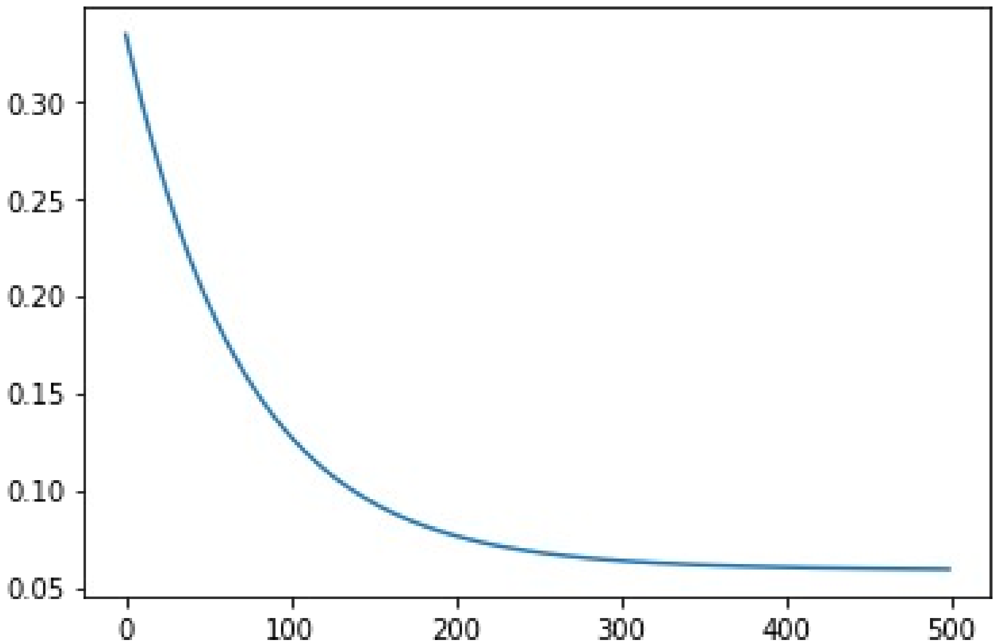
NNFL assignment 1

Harshul Gupta
2018B5A31058H

Question 1

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt

from google.colab import files
uploaded = files.upload()

def cost_function(X,y,w):        ###define·cost·function·
    hypothesis = np.dot(X,w.T)
###calculation·of·hypothesis·for·all·instances···
    J = (1/(2*len(y))) * np.sum((hypothesis - y)**2)
    return J

def batch_gradient_descent(X,y,w,alpha,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every
iteration

  for i in range(iters):

    hypothesis = np.dot(X,w.T)
    w = w - (alpha/len(y)) * np.dot((hypothesis-y), X)
    cost_history[i] = cost_function(X,y,w)

  return w,cost_history

def stochastic_gradient_descent(X,y,w,alpha, iters):

  cost_history = np.zeros(iters)

  for i in range(iters):

    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+1]
    ind_y = y[rand_index:rand_index+1]
    w = w - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w)

  return w, cost_history

def MB_gradient_descent(X,y,w,alpha, iters, batch_size):
  cost_history = np.zeros(iters)
  for i in range(iters):
    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+batch_size]
    ind_y = y[rand_index:rand_index+batch_size]
    w = w - (alpha/batch_size) * (ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w)
```

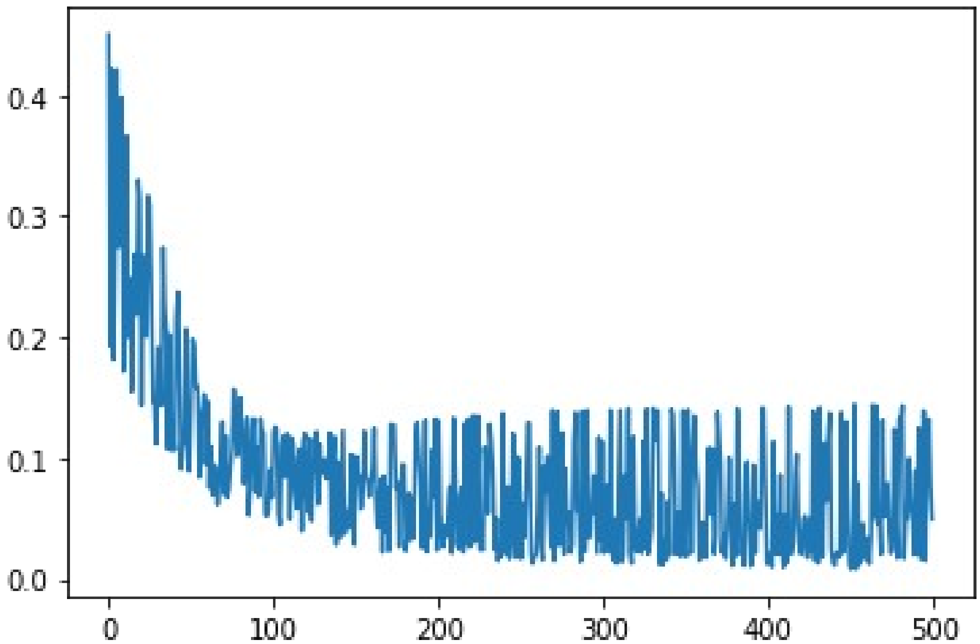
```python
    return w, cost_history

data = pd.read_excel(uploaded['data_q1.xlsx'])

print(data)

datan =data.values
X=datan[:,[0,1]]
m = X.shape[0] #no of examples
xmin = np.min(X, axis = 0)
xmax = np.max(X, axis = 0)
X = (X- xmin)/(xmax-xmin) #Normalization
pp = np.ones([m, 1]) # vector containg ones as all elements
X = np.append(pp,X, axis=1) #Column of ones
y=datan[:,2] #output
ymin = np.min(y, axis = 0)
ymax = np.max(y, axis = 0)
y = (y- ymin)/(ymax-ymin)

print(X)

w= np.zeros((X.shape[1])) ###weight initialization
print(w)

[0. 0. 0.]

alpha=0.005
iters=500
batch_w,J_his = batch_gradient_descent(X,y,w,alpha,iters)

plt.plot(range(iters),J_his)
plt.show()
```




```
print(batch_w)
```

```
[0.52799868 0.20112163 0.24080326]
```

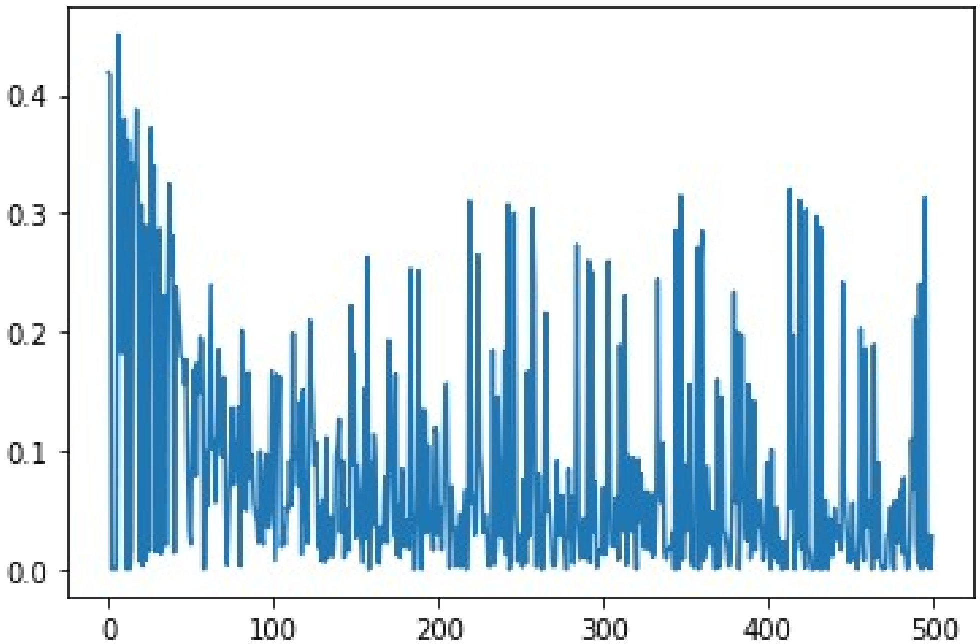
```
alpha=0.01
iters=500
batch_size=30
mini_batch_w,J_mini_batch = MB_gradient_descent(X,y,w,alpha,iters,
batch_size)

plt.plot(range(iters),J_mini_batch)
plt.show()
```




```
print(mini_batch_w)
```

```
[0.53264732 0.17424192 0.25812162]
```

```
n_epochs=500
alpha=0.01
w_n,J_sgd = stochastic_gradient_descent(X,y,w, alpha, n_epochs)
plt.plot(range(n_epochs),J_sgd)
plt.show()
```



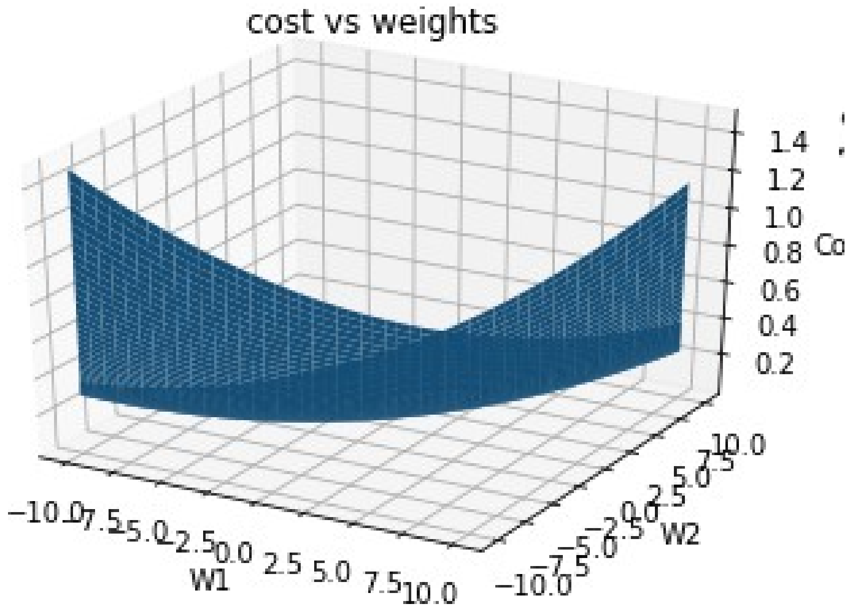

```
print(w_n) #SCD
```

```
[0.56750678 0.15126243 0.26207921]
```

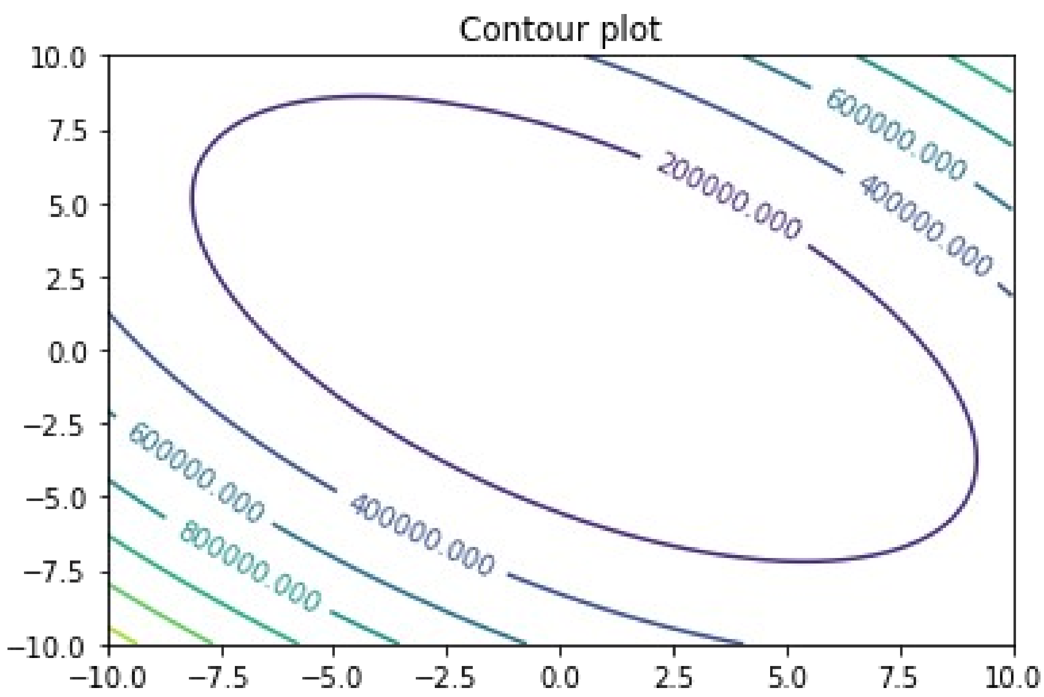
```
w1 = np.linspace(-10,10,500)
w2 = np.linspace(-10,10,500)
j = np.zeros((500,500))
w_model = np.zeros((3,1))
for xn in range(len(w1)):
  for yn in range(len(w2)):
    w_model[1] = w1[xn]
    w_model[2] = w2[yn]
    y_pred = np.dot(X,w_model)
    squared_errors = (y_pred - y)**2
    sum_squared_errors = np.sum(squared_errors)
    j[xn][yn] = sum_squared_errors
```

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(w1, w2, j)
ax.set_xlabel('W1')
```

```
ax.set_ylabel('W2')
ax.set_zlabel('Cost')
ax.set_title('cost vs weights')
fig.show()
plt.show()
```




```
plt.title('Contour plot')
contours = plt.contour(w1, w2, j)
plt.clabel(contours, inline=1, fontsize=10)
plt.show()
```



Question 2

```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from google.colab import files
uploaded = files.upload()

<IPython.core.display.HTML object>

Saving data_q2_q3.xlsx to data_q2_q3.xlsx

#linear regression with the L2-norm regularization (Ridge regression)
#approach using BGD, SGD, and MBGD algorithms.

def cost_function(X,y,w,reg):        ###define·cost·function·
    hypothesis = np.dot(X,w.T)
###calculation·of·hypothesis·for·all·instances···
    J = (1/(2*len(y))) * np.sum((hypothesis - y)**2) +
(reg/2)*np.sum(np.square(w))
    return J

def batch_gradient_descent(X,y,w,alpha,reg,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every
iteration

  for i in range(iters):

    hypothesis = np.dot(X,w.T)
    w = w*(1-alpha*reg) - (alpha/len(y)) * np.dot((hypothesis-y), X)
    cost_history[i] = cost_function(X,y,w,reg)

  return w,cost_history

def stochastic_gradient_descent(X,y,w,alpha,reg, iters):

  cost_history = np.zeros(iters)

  for i in range(iters):

    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+1]
    ind_y = y[rand_index:rand_index+1]
    w = w*(1-alpha*reg) - alpha * (ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost = cost_function(ind_x,ind_y,w,reg)

  return w, cost
```

```python
def MB_gradient_descent(X,y,w,alpha, iters, reg,batch_size):
  cost_history = np.zeros(iters)
  for i in range(iters):
    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+batch_size]
    ind_y = y[rand_index:rand_index+batch_size]
    w = w*(1-alpha*reg) - (alpha/batch_size) *
(ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost = cost_function(ind_x,ind_y,w)

  return w, cost
```

```python
# A function to implement min-max normalization

def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan
```

```python
data = pd.read_excel(uploaded['data_q2_q3.xlsx'])
```

```python
#test_train_validaion_split

data = norm(data) #all features of data are now normalized

df = data.sample(len(data)) #randomized data frame
df = df.values
train_len = int(0.7 * len(data))

test_len = int(0.2 * len(data))

X_train = df[0: train_len,[0,1,2,3]]                      #train
X_test = df[train_len: train_len+test_len,[0,1,2,3]]     #test
X_vald = df[train_len+test_len:,[0,1,2,3]]                 #validation
l = len(X_train)
m = len(X_test)
n = len(X_vald)
X_train = np.append(np.ones([l, 1]),X_train, axis=1) #Column of ones
X_test = np.append(np.ones([m, 1]),X_test, axis=1) #Column of ones
X_vald = np.append(np.ones([n, 1]),X_vald, axis=1) #Column of ones

y_train = df[0: train_len,4]                    #train
y_test = df[train_len: train_len+test_len,4]     #test
y_vald = df[train_len+test_len:,4]               #validation
```

```python
X_train.shape[1]
```

5

```
iters=500
#grid search
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    batch_w,j_his =
batch_gradient_descent(X_train,y_train,w,alpha,reg,iters)
    train_errors.append(j_his[iters -1])
    W_values.append(batch_w)
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i

print(min_index)
print( 'optimal w values',W_values[min_index])
```

```
990
[0.22491164 0.1602437  0.10383832 0.1062071  0.09526949]
```

```
W_min = W_values[min_index]

#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error

Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for SGD with L2 = ",Mean_square_error)
print("MAE for SGD with L2= ",Mean_abs_error)
print("CC for SGD with L2 = ",corr_coff[0,1])
```

```
MSE for SGD with L2 =  0.17657093127011386
MAE for SGD with L2=  0.38809203453229507
CC for SGD with L2 =  0.728852950682363
```

```
iters=500
#grid search
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
```

```
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    w_SGD,j_his =
stochastic_gradient_descent(X_train,y_train,w,alpha,reg,iters)
    train_errors.append(j_his)
    W_values.append(w_SGD)
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i


print('optimal w values',W_values[min_index])
W_min = W_values[min_index]
#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error

Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for SGD with L2 = ",Mean_square_error)
print("MAE for SGD with L2= ",Mean_abs_error)
print("CC for SGD with L2 = ",corr_coff[0,1])
```

```
optimal w values [0.04801007 0.0325622  0.02358152 0.02417115
0.01875058]
MSE for SGD with L2 =  0.4988711384427089
MAE for SGD with L2=  0.6103949903945742
CC for SGD with L2 =  0.7030664551957814
```

```
iters=500
#grid search
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    w_MBGD,j_his =
MB_gradient_descent(X_train,y_train,w,alpha,reg,iters,30)
    train_errors.append(j_his[iters-1])
    W_values.append(w_SGD)
```

```
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i


print('optimal w values',W_values[min_index])
W_min = W_values[min_index]
#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error

Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for MBGD with L2 = ",Mean_square_error)
print("MAE for MBGD with L2= ",Mean_abs_error)
print("CC for MBGD with L2 = ",corr_coff[0,1])
```

Question 3

```
import math
import numpy as np
import pandas as pd

from google.colab import files
uploaded = files.upload()

<IPython.core.display.HTML object>

Saving data_q2_q3.xlsx to data_q2_q3.xlsx

#linear regression with the L2-norm regularization (Least angle
regression)
#approach using BGD, SGD, and MBGD algorithms.

def cost_function(X,y,w,reg):        ###define·cost·function·
    hypothesis = np.dot(X,w.T)
###calculation·of·hypothesis·for·all·instances···
    J = (1/(2*len(y))) * np.sum((hypothesis - y)**2) +
(reg/2)*np.sum(np.square(w))
    return J

def batch_gradient_descent(X,y,w,alpha,reg,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every
iteration

  for i in range(iters):

    hypothesis = np.dot(X,w.T)
    w = w - (alpha*reg*0.5)*(np.sign(w)) - (alpha/len(y)) *
np.dot((hypothesis-y), X)
    cost_history[i] = cost_function(X,y,w,reg)

  return w,cost_history

def stochastic_gradient_descent(X,y,w,alpha,reg, iters):

  cost_history = np.zeros(iters)

  for i in range(iters):

    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+1]
    ind_y = y[rand_index:rand_index+1]
    w = w - alpha  * (alpha*reg*0.5)*(np.sign(w))
(ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost_history[i] = cost_function(ind_x,ind_y,w,reg)

  return w, cost_history
```

```
def MB_gradient_descent(X,y,w,alpha, iters, reg,batch_size):
  cost_history = (np.zeros(iters)).astype(float)
  for i in range(iters):
    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+batch_size]
    ind_y = y[rand_index:rand_index+batch_size]

    w = w - alpha  * ((alpha*reg*0.5)/batch_size)*(np.sign(w))
(ind_x.T.dot(ind_x.dot(w) - ind_y))
    cost_history = cost_function(ind_x,ind_y,w)

  return w, cost

# A function to implement min-max normalization
def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan

data = pd.read_excel(uploaded['data_q2_q3.xlsx'])

#test_train_validaion_split

data = norm(data) #all features of data are now normalized

df = data.sample(len(data)) #randomized sample
df = df.values
train_len = int(0.7 * len(data))

test_len = int(0.2 * len(data))

X_train = df[0: train_len,[0,1,2,3]]                      #train
X_test = df[train_len: train_len+test_len,[0,1,2,3]]     #test
X_vald = df[train_len+test_len:,[0,1,2,3]]                #validation
l = len(X_train)
m = len(X_test)
n = len(X_vald)
X_train = np.append(np.ones([l, 1]),X_train, axis=1) #Column of ones
X_test = np.append(np.ones([m, 1]),X_test, axis=1) #Column of ones
X_vald = np.append(np.ones([n, 1]),X_vald, axis=1) #Column of ones

y_train = df[0: train_len,4]                    #train
y_test = df[train_len: train_len+test_len,4]     #test
y_vald = df[train_len+test_len:,4]               #validation

X_train.shape[1]
```

```
5
```

```
iters=500
#Grid Search with hyperparametrs alpha and Reg
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    batch_w,j_his =
batch_gradient_descent(X_train,y_train,w,alpha,reg,iters)
    train_errors.append(j_his[iters -1])
    W_values.append(batch_w)
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i

print(min_index)
print('optimal w values for BGD-L2',W_values[min_index])
```

```
990
optimal w values for BGD-L2 [0.21514831 0.14591479 0.09640734
0.08152917 0.07988704]
```

```
W_min = W_values[min_index]
```

```
#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error
Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for BCD with L1= ",Mean_square_error)
print("MAE for BCD with L1 = ",Mean_abs_error)
print("CC for BCD with L1= ",corr_coff[0,1])
```

```
MSE for BCD with L1=  0.22092608971417466
MAE for BCD with L1 =  0.44464049387081317
CC for BCD with L1=  0.4499533688220539
```

```
iters=500
#Grid Search with hyperparametrs alpha and Reg
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
```

```
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    w_SGD,j_his =
stochastic_gradient_descent(X_train,y_train,w,alpha,reg,iters)
    train_errors.append(j_his[iters -1])
    W_values.append(w_SGD)
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i

print(min_index)
print('optimal w values for SGD-L2',W_values[min_index])

#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error
Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for BCD with L1= ",Mean_square_error)
print("MAE for BCD with L1 = ",Mean_abs_error)
print("CC for BCD with L1= ",corr_coff[0,1])

iters=500
#Grid Search with hyperparametrs alpha and Reg
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    w_SGD,j_his =
stochastic_gradient_descent(X_train,y_train,w,alpha,reg,iters)
    train_errors.append(j_his[iters -1])
    W_values.append(w_SGD)
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i
```

```
print(min_index)
print('optimal w values for SGD-L2',W_values[min_index])

#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error
Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for BCD with L1= ",Mean_square_error)
print("MAE for BCD with L1 = ",Mean_abs_error)
print("CC for BCD with L1= ",corr_coff[0,1])

iters=500
batch_size = 30
#Grid Search with hyperparametrs alpha and Reg
alpha_vals=np.linspace(0.0001,0.001,100)
reg_vals=np.linspace(0.1,1,10)
train_errors = []
W_values = []
i=0
for alpha in alpha_vals:
  for reg in reg_vals:
    w= np.zeros((5))
    w_SGD,j_his = 
MB_gradient_descent(X_train,y_train,w,alpha,reg,iters,batch_size)
    train_errors.append(j_his[iters -1])
    W_values.append(w_SGD)
    i = i+1

#best model
min_index = 0
for i in range(len(train_errors)):
  if(train_errors[i] < train_errors[min_index]):
    min_index = i

print(min_index)
print('optimal w values for SGD-L2',W_values[min_index])

#Mean square error
y_pred = np.dot(X_test,W_min.T)
Mean_square_error =  1/len(y_pred) * np.sum((y_pred - y_test)**2)
#Mean square error
Mean_abs_error =   1/len(y_pred) * np.sum(np.abs(y_pred - y_test))
corr_coff = np.corrcoef(y_pred,y_test)
print("MSE for BCD with L1= ",Mean_square_error)
print("MAE for BCD with L1 = ",Mean_abs_error)
print("CC for BCD with L1= ",corr_coff[0,1])
```

Question 4

```
import math
import numpy as np
import pandas as pd
```

```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_q4_q5.xlsx to data_q4_q5.xlsx
```

```
def sigmoid(x):
  x = x.astype(float)
  z = np.exp(-x)
  sig = 1 / (1 + z)
  return sig

def set(y):
    for i in range(len(y)):
        if(y[i]>0.5):
            y[i] = 1
        if(y[i]<0.5):
            y[i] = 0
    return y

def set_type(y) :
  for i in range(len(y)):
        if(y[i] == 'M'):
            y[i] = 1
        if(y[i] == 'B'):
            y[i] = 0
  return y


#def cost_function(X,y,w,reg):        ###define·cost·function·
  #  hypothesis = sigmoid(np.dot(X,w.T))
###calculation·of·hypothesis·for·all·instances···
  #  J = -(1/len(y)) * ()
  #  return J

#BCD for logistic regression


def batch_gradient_descent(X,y,w,alpha,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every
iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
```

```
    w = w -  alpha * np.dot((hypothesis-y), X) #weight updation

  return w

def stochastic_gradient_descent(X,y,w,alpha, iters):

  for i in range(iters):
    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+1]
    ind_y = y[rand_index:rand_index+1]
    z = np.dot(ind_x,w.T)
    hypothesis = sigmoid(z)
    w = w - alpha * np.dot((hypothesis-ind_y), ind_x)

  return w,

def MB_gradient_descent(X,y,w,alpha, iters, batch_size):
  for i in range(iters):
    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+batch_size]
    ind_y = y[rand_index:rand_index+batch_size]
    w = w - (alpha/batch_size) * (ind_x.T.dot(sigmoid(ind_x.dot(w)) -
ind_y))

  return w

# A function to implement min-max normalization
def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan

data = pd.read_excel(uploaded['data_q4_q5.xlsx'])
#dataframe = data.drop(columns = 'diagnosis')




df = data.sample(len(data)) #randomized sample
df = df.values
train_len = int(0.7 * len(data))

test_len = int(0.2 * len(data))

X_train = df[0: train_len,[i for i in range(30)]]
X_test = df[train_len: train_len+test_len,[i for i in range(30)]]
#test
X_vald = df[train_len+test_len:,[i for i in range(30)]]
#validation
```

```
l = len(X_train)
m = len(X_test)
n = len(X_vald)


y_train = df[0: train_len,30]                    #train
y_test = df[train_len: train_len+test_len,30]     #test
y_vald = df[train_len+test_len:,30] #val

X_train = norm(X_train)
X_test = norm(X_test)     #test
X_vald = norm(X_vald)               #validation

X_train = np.append(np.ones([l, 1]),X_train, axis=1) #Column of ones
X_test = np.append(np.ones([m, 1]),X_test, axis=1) #Column of ones
X_vald = np.append(np.ones([n, 1]),X_vald, axis=1) #Column of ones

X_train[0]

array([1.0, 0.2443312601008864, 0.5744771660264618,
0.24701110162254483,
       0.14111483294683502, 0.49251660224006344, 0.365796863228546,
       0.2642924086223055, 0.34160039761431416, 0.4267676767676768,
       0.35375055481580125, 0.11370631902951293, 0.18073727015558697,
       0.10224756160768977, 0.05716675854571529, 0.20627528299962603,
       0.1819629284705741, 0.09786043449637918, 0.3284950343773873,
       0.1580499458733062, 0.12865102304708298, 0.34402093898249625,
       0.6051341217190654, 0.3278416566557977, 0.20717491584590447,
       0.7523608267846529, 0.36752335768547895, 0.321405750798722,
       0.5896907216494846, 0.4319315751960085, 0.31523022432113346],
      dtype=object)

y_train

array(['M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'B',
'B',
       'B', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'B', 'M',
'B',
       'M', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B',
'B',
       'M', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B', 'M',
'M',
       'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'M',
'B',
       'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'M',
'B',
       'B', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'B', 'B', 'M',
'B',
       'B', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'M',
'B',
       'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M', 'M', 'B',
```

```
'B',
       'B', 'M', 'M', 'B', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B',
'M',
       'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
'B',
       'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'B',
'M',
       'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'M', 'M',
'B',
       'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'M',
       'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M',
'M',
       'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'M', 'B',
'M',
       'B', 'M', 'M', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'M', 'B',
'M',
       'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'B',
'B',
       'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'M', 'B', 'M',
'M',
       'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'M', 'B',
'M',
       'B', 'B', 'M', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'B', 'B',
'M',
       'M', 'B', 'B', 'B', 'M', 'M', 'M', 'B', 'B', 'B', 'B', 'M',
'B',
       'B', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'M', 'B',
'M',
       'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B',
'B',
       'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B',
'M',
       'B', 'B', 'M', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'B',
'B',
       'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
'M',
       'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
'B',
       'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B',
'B',
       'M', 'B', 'M', 'B', 'B', 'M', 'M', 'M', 'B', 'M', 'B', 'M',
'B',
       'B', 'B', 'M', 'B', 'M', 'B', 'M', 'B'], dtype=object)
```

```
X_train.shape
```

```
(398, 31)
```

```
y_test = set_type(y_test)
y_train = set_type(y_train)
y_vald = set_type(y_vald)

y_train

array([1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1,
0,
       1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
1,
       0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0,
0,
       0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1,
0,
       0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0,
0,
       1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0,
1,
       0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
0,
       0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1,
0,
       1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,
0,
       1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1,
0,
       0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1,
0,
       0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0,
       1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0,
0,
       1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1,
0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
0,
       1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,
0,
       1, 0], dtype=object)

alpha=0.005
iters=500
w= np.zeros((X_train.shape[1])) ###weight initialization
w_BCD = batch_gradient_descent(X_train,y_train,w,alpha,iters)
print(w_BCD)
```

```
[-9.938069827186878 0.8951614744746299 2.088418078176403
0.977901778510686
 1.4716923721479214 0.7178860316271198 0.33871809152230326
 3.078331589531312 4.1803988902237545 -0.03578783521443474
 -2.799967812137131 2.7893960634013157 0.3447398665342109
 2.0471847078355467 1.8819162302390373 -0.17916063436683838
 -1.371418484402272 -0.6443426388828907 0.2562349294318265
 -0.5255638699504118 -1.4147616123139855 2.918546848169572
 2.874463877189223 2.51366278507917 3.007390549099292
1.4185386430975444
 0.37558709610872115 1.711211189714206 3.8000024798188137
 1.3281800153591738 -0.23384586139783037]
```

```
y_pred = sigmoid(np.dot(X_test,w_BCD.T))
y_pred = set(y_pred)
```

```
y_pred
```

```
array([1., 1., 1., 1., 0., 0., 1., 0., 1., 1., 0., 0., 0., 0., 1., 0.,
1.,
       1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 0.,
0.,
       1., 1., 0., 1., 1., 1., 1., 1., 0., 1., 0., 0., 0., 1., 1., 1.,
0.,
       0., 0., 0., 0., 1., 1., 1., 1., 1., 0., 1., 1., 0., 0., 0., 0.,
1.,
       1., 0., 1., 0., 0., 1., 0., 0., 1., 1., 1., 1., 0., 0., 1., 0.,
0.,
       0., 1., 0., 1., 1., 0., 0., 1., 1., 0., 0., 1., 1., 0., 1., 0.,
0.,
       0., 1., 1., 0., 0., 0., 0., 0., 1., 1., 1.])
```

```
y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
```

```
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
confmat = np.asarray(confmat)
tp = confmat[1][1]
tn = confmat[0][0]
fp = confmat[0][1]
fn = confmat[1][0]
```

```
Predicted  0.0  1.0
Actual
0           56    5
1            1   51
```

```
Acc = (tp+tn)/(tp+tn+fp+fn)
SE = tp/(tp+fn)
SP = tn/(tn+fp)
print('Accuracy = ',Acc)
```

```
print('Sensitivity = ',SE)
print('specificity = ',SP)
```

```
Accuracy =  0.9469026548672567
Sensitivity =  0.9807692307692307
specificity =  0.9180327868852459
```

```
#Logistic regression with socastic gradient descent
alpha=0.005
iters=500
w= np.zeros((X_train.shape[1])) ###weight initialization
w_SGD = stochastic_gradient_descent(X_train,y_train,w,alpha,iters)
w_SGD = np.array(w_SGD)
print(w_SGD)
```

```
[[-0.31349653456717086 0.04592714133948313 -0.029469242832653023
  0.049529138714109806 0.05909478758599255 -0.051015081669462135
  0.03477597283347468 0.08587491523142753 0.09861083088102914
  -0.06521295400398364 -0.10022118892670437 0.03487356122985107
  -0.05053817239240319 0.032252579915281236 0.03540893958866682
  -0.05662538485403935 -0.006916069753025564 -0.0033792124178667884
  -0.013394302215421311 -0.06638712000238434 -0.030594232858995592
  0.08139454806587208 -0.02308790931263196 0.08114333062394326
  0.08880545608052351 -0.05520242143033854 0.03463316194119539
  0.05562595154882657 0.08373072515231177 -0.025710307844776323
  -0.02327390553730539]]
```

```
w_SGD = np.squeeze (w_SGD)
print(w_SGD)
```

```
[-0.31349653456717086 0.04592714133948313 -0.029469242832653023
 0.049529138714109806 0.05909478758599255 -0.051015081669462135
 0.03477597283347468 0.08587491523142753 0.09861083088102914
 -0.06521295400398364 -0.10022118892670437 0.03487356122985107
 -0.05053817239240319 0.032252579915281236 0.03540893958866682
 -0.05662538485403935 -0.006916069753025564 -0.0033792124178667884
 -0.013394302215421311 -0.06638712000238434 -0.030594232858995592
 0.08139454806587208 -0.02308790931263196 0.08114333062394326
 0.08880545608052351 -0.05520242143033854 0.03463316194119539
 0.05562595154882657 0.08373072515231177 -0.025710307844776323
 -0.02327390553730539]
```

```
y_pred = sigmoid(np.dot(X_test,w_SGD.T))
y_pred = set(y_pred)
```

```
y_pred
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
1.,
       1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
       0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
```

```
0.,
       0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
0.,
       0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')

confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
confmat = np.asarray(confmat)
tp = confmat[1][1]
tn = confmat[0][0]
fp = confmat[0][1]
fn = confmat[1][0]
Acc = (tp+tn)/(tp+tn+fp+fn)
SE = tp/(tp+fn)
SP = tn/(tn+fp)
print('Accuracy = ',Acc)
print('Sensitivity = ',SE)
print('specificity = ',SP)
```

```
Predicted  0.0  1.0
Actual
0           61    0
1           42   10
Accuracy =  0.6283185840707964
Sensitivity =  0.19230769230769232
specificity =  1.0
```

```
alpha=0.005
iters=500
batch_size = 30
w= np.zeros((X_train.shape[1])) ###weight initialization
w_MBGD = MB_gradient_descent(X_train,y_train,w,alpha,iters,batch_size)
print(w_MBGD)
```

```
[-0.2922359886027979 0.046326282625936036 -0.024952982074459663
 0.04959415906732672 0.05835255748879707 -0.04014953372485376
 0.03193938566682174 0.08548327852325091 0.09803174164268283
 -0.05970971105524067 -0.08723790567671012 0.0387879546442997
 -0.04540592417977377 0.035298406169141296 0.03823131596553538
 -0.050577430031411325 -0.005798488329350273 0.0008447994355502685
 -0.007388553886715478 -0.05984671198525166 -0.02511388207026195
 0.07801645767621608 -0.020492954467550395 0.07712780038044803
 0.08465457465320876 -0.04571811333382822 0.026336244059559297
```

```
 0.05370824098656535 0.0811598779832903 -0.028746874880265824
 -0.0192961626569748]
```

```
y_pred = sigmoid(np.dot(X_test,w_SGD.T))
y_pred = set(y_pred)
y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
confmat = np.asarray(confmat)
tp = confmat[1][1]
tn = confmat[0][0]
fp = confmat[0][1]
fn = confmat[1][0]
Acc = (tp+tn)/(tp+tn+fp+fn)
SE = tp/(tp+fn)
SP = tn/(tn+fp)
print('Accuracy = ',Acc)
print('Sensitivity = ',SE)
print('specificity = ',SP)
```

```
Predicted  0.0  1.0
Actual
0           61    0
1           42   10
Accuracy =  0.6283185840707964
Sensitivity =  0.19230769230769232
specificity =  1.0
```

Question 5

```
import math
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold, train_test_split
```

```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_q4_q5.xlsx to data_q4_q5 (1).xlsx
```

```
def sigmoid(x):
  x = x.astype(float)
  z = np.exp(-x)
  sig = 1 / (1 + z)
  return sig
```

```
def set(y):
    for i in range(len(y)):
        if(y[i]>0.5):
            y[i] = 1
        if(y[i]<0.5):
            y[i] = 0
    return y
```

```
def set_type(y) :
  for i in range(len(y)):
        if(y[i] == 'M'):
            y[i] = 1
        if(y[i] == 'B'):
            y[i] = 0
  return y
```

```
# A function to implement min-max normalization
def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan
```

```
def batch_gradient_descent(X,y,w,alpha,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every
iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w -  alpha * np.dot((hypothesis-y), X) #weight updation
```

```
    return w

data = pd.read_excel(uploaded['data_q4_q5.xlsx'])
#dataframe = data.drop(columns = 'diagnosis')

df = data.values
X = df[:,:-1]
y = df[:,-1]
y = set_type(y)
X = norm(X) #normalization
l = len(X)
X = np.append(np.ones([l, 1]),X, axis=1)

#Implementing cross validation
alpha=0.05
iters=500
k = 5
kf = KFold(n_splits=k, random_state=None)
acc_score = []

for train_index , test_index in kf.split(X):
    X_train , X_test = X[train_index,:],X[test_index,:]
    y_train , y_test = y[train_index] , y[test_index]
    w= np.zeros((X_train.shape[1])) ###weight initialization
    w_BCD = batch_gradient_descent(X_train,y_train,w,alpha,iters) #BCD
for logistic regression

    y_pred = np.dot(X_test,w_BCD.T)
    y_pred = set(y_pred)
    y_actual = pd.Series(y_test, name='Actual')
    y_pred = pd.Series(y_pred, name='Predicted')
    confmat = pd.crosstab(y_actual, y_pred)
    confmat = np.asarray(confmat)
    print(confmat)
    tp = confmat[1][1]
    tn = confmat[0][0]
    fp = confmat[0][1]
    fn = confmat[1][0]
    acc = (tp+tn)/(tp+tn+fp+fn)
    acc_score.append(acc)

avg_acc_score = sum(acc_score)/k
```

```
[[46  0]
 [ 3 65]]
[[65  0]
 [ 6 43]]
[[74  0]
 [ 3 37]]
```

```
[[84  1]
 [ 0 29]]
[[86  1]
 [ 0 26]]
```

```
print(avg_acc_score)
```

```
0.9754230709517155
```

Question 6(one vs all)

```
import math
import numpy as np
import pandas as pd
```

```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_q6_q7.xlsx to data_q6_q7.xlsx
```

```
def sigmoid(x):
  x = x.astype(float)
  z = np.exp(-x)
  sig = 1 / (1 + z)
  return sig
```

```
def set(y):
    for i in range(len(y)):
        if(y[i]>=0.5):
            y[i] = 1
        if(y[i]<0.5):
            y[i] = 0
    return y
```

```
# A function to implement min-max normalization
def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan
```

```
#BCD for logistic regression


def batch_gradient_descent(X,y,w,alpha,iters):

  cost_history = np.zeros(iters) # cost function for each iteration

  #initalize our cost history list to store the cost function on every
iteration

  for i in range(iters):
    z = np.dot(X,w.T)
    hypothesis = sigmoid(z)
    w = w -  alpha * np.dot((hypothesis-y), X) #weight updation

  return w
```

```
def stochastic_gradient_descent(X,y,w,alpha, iters):

  for i in range(iters):
    rand_index = np.random.randint(len(y))
```

```
    ind_x = X[rand_index:rand_index+1]
    ind_y = y[rand_index:rand_index+1]
    z = np.dot(ind_x,w.T)
    hypothesis = sigmoid(z)
    w = w - alpha * np.dot((hypothesis-ind_y), ind_x)

  return w,

def MB_gradient_descent(X,y,w,alpha, iters, batch_size):
  for i in range(iters):
    rand_index = np.random.randint(len(y))
    ind_x = X[rand_index:rand_index+batch_size]
    ind_y = y[rand_index:rand_index+batch_size]
    w = w - (alpha/batch_size) * (ind_x.T.dot(sigmoid(ind_x.dot(w)) -
ind_y))

  return w

# A function to implement min-max normalization
def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan

data = pd.read_excel(uploaded['data_q6_q7.xlsx'])

data.sample(5)

     Column1  Column2  Column3  Column4  Column5  Column6  Column7
Column8
116    18.96    16.20   0.9077    6.051    3.897    4.334    5.750
2
173    11.40    13.08   0.8375    5.136    2.763    5.588    5.089
3
13     13.78    14.06   0.8759    5.479    3.156    3.136    4.872
1
129    17.55    15.66   0.8991    5.791    3.690    5.366    5.661
2
53     14.33    14.28   0.8831    5.504    3.199    3.328    5.224
1

df = data.sample(len(data)) #randomized sample
df = df.values
train_len = int(0.7 * len(data))

test_len = int(0.2 * len(data))

X_train = df[0: train_len,[0,1,2,3,4,5,6]]
X_test = df[train_len: train_len+test_len,[0,1,2,3,4,5,6]]     #test
X_vald = df[train_len+test_len:,[0,1,2,3,4,5,6]]
```

```
#validation
l = len(X_train)
m = len(X_test)
n = len(X_vald)


y_train = df[0: train_len,7]                    #train
y_test = df[train_len: train_len+test_len,7]     #test
y_vald = df[train_len+test_len:,7] #val

X_train = norm(X_train)
X_test = norm(X_test)     #test
X_vald = norm(X_vald)              #validation
X_train = np.append(np.ones([l, 1]),X_train, axis=1) #Column of ones
X_test = np.append(np.ones([m, 1]),X_test, axis=1) #Column of ones
X_vald = np.append(np.ones([n, 1]),X_vald, axis=1) #Column of ones

y1_tr = [1 for i in range(len(y_train))]
y2_tr = [1 for i in range(len(y_train))]
y3_tr = [1 for i in range(len(y_train))]
for i in range(len(y_train)):
    if(y_train[i] != 1):
        y1_tr[i] = 0
    if(y_train[i] != 2):
        y2_tr[i] = 0
    if(y_train[i] != 3):
        y3_tr[i] = 0

alpha=0.005
iters=500
w= np.zeros((X_train.shape[1])) ###weight initialization
w_m1 = batch_gradient_descent(X_train,y1_tr,w,alpha,iters)
y_p1 = np.dot(X_test,w_m1.T)
y_p1 = set(y_p1)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m2 = batch_gradient_descent(X_train,y2_tr,w,alpha,iters)
y_p2 = np.dot(X_test,w_m2.T)
y_p2 = set(y_p2)

w= np.zeros((X_train.shape[1])) ###weight initialization
w_m3 = batch_gradient_descent(X_train,y3_tr,w,alpha,iters)
y_p3 = np.dot(X_test,w_m3.T)
y_p3 = set(y_p3)

cval = [0 for i in range(len(y_test))]
for i in range(len(y_test)):
    if (y_p1[i] == 1):
        cval[i] = 1.0
    if (y_p2[i] == 1):
        cval[i] = 2.0
```

```
    if (y_p3[i] == 1):
        cval[i] = 3.0


for i in range(len(cval)):
    if (cval[i] == 0):
        cval[i] = 'None'

y_actual = pd.Series(y_test, name='Actual')
y_pred = pd.Series(cval, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
```

```
Predicted  1.0  2.0  3.0  None
Actual
1.0          9    0    1     3
2.0          1   12    0     0
3.0          2    0   14     0
```

```
confmat = np.asarray(confmat)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2]
[2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
```

```
Overall Accuracy : 0.8333333333333334
Accuracy of class 1 : 0.6923076923076923
Accuracy of class 2 : 0.9230769230769231
Accuracy of class 3 : 0.875
```

Question 8

```
import math
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold, train_test_split
```

```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_q4_q5.xlsx to data_q4_q5.xlsx
```

```
def LRT_rule(x_ts,X,y):
  p1 = len([i for (i, val) in enumerate(y) if val == 1])
  p2 = len([i for (i, val) in enumerate(y) if val == 2])
  py1, py2 = p1/(len(y)), p2/(len(y))
  x1 = np.array([X[i] for (i, val) in enumerate(y) if val == 1])
  x2 = np.array([X[i] for (i, val) in enumerate(y) if val == 2])
  m1 = np.mean(x1, axis=0)
  m2 = np.mean(x2, axis=0)
  #x1 = x1.astype(float)
  #x2 = x2.astype(float)
  cov1 = np.cov((x1.T).astype(float))
  cov2 = np.cov((x2.T).astype(float))
  coeff1 = 1/(((2*3.14)**2)*np.linalg.det(cov1)**0.5)
  coeff2 = 1/(((2*3.14)**2)*np.linalg.det(cov2)**0.5)
  R1 = -0.5*np.dot(np.dot((x_ts - m1),np.linalg.inv(cov1)),(x_ts -
m1).T)
  R2 = -0.5*np.dot(np.dot((x_ts - m2),np.linalg.inv(cov2)),(x_ts -
m2).T)
  #R1 = R1.astype(float)
  #R2 = R2.astype(float)
  l1 = coeff1*np.exp(R1)
  l2 = coeff2*np.exp(R2)
  if (l1/p2) > (l2/p1):
    return 1
  else:
    return 2
```

```
def confmat(y_pred,y_ts):
    a, b, c, d = 0, 0, 0, 0
    for i in range(len(y_ts)):
        if y_ts[i] == 1:
            if y_pred[i] == 1:
                a = a + 1
            if y_pred[i] == 2:
                b = b + 1
        if y_ts[i] == 2:
            if y_pred[i] == 1:
                c = c + 1
            if y_pred[i] == 2:
```

```
                d = d + 1
    return a, b, c, d
```

```
data = pd.read_excel(uploaded['data_q4_q5.xlsx'])

print(data)
```

```
     radius_mean  texture_mean  ...  fractal_dimension_worst
diagnosis
0          17.99         10.38  ...                  0.11890
M
1          20.57         17.77  ...                  0.08902
M
2          19.69         21.25  ...                  0.08758
M
3          11.42         20.38  ...                  0.17300
M
4          20.29         14.34  ...                  0.07678
M
..           ...           ...  ...                      ...          ..
.
564        21.56         22.39  ...                  0.07115
M
565        20.13         28.25  ...                  0.06637
M
566        16.60         28.08  ...                  0.07820
M
567        20.60         29.33  ...                  0.12400
M
568         7.76         24.54  ...                  0.07039
B

[569 rows x 31 columns]
```

```
df = data.values
X = df[:,:-1]
y = df[:,-1]

for i in range(len(y)):
        if(y[i] == 'M'):
            y[i] = 1
        if(y[i] == 'B'):
            y[i] = 2

kf = KFold(n_splits=5, random_state=None)
acc_score = []
sens_score = []
spec_score = []
```

```python
for train_index , test_index in kf.split(X):
    X_train , X_test = X[train_index,:],X[test_index,:]
    y_train , y_test = y[train_index] , y[test_index]
    y_pred = []

    for i in range(len(X_test)):
      y_pred.append(LRT_rule(X_test[i],X_train,y_train))

    a, b, c, d = confmat(y_pred,y_test)
    acc = (a+d)/(a+b+c+d)
    sens = (a)/(a+b)
    spec = (d)/(d+c)
    acc_score.append(acc)
    sens_score.append(sens)
    spec_score.append(spec)

avg_acc_score = sum(acc_score)/5
avg_sensivity = sum(sens_score)/5
avg_specivity = sum(spec_score)/5
print('we are assuming class 1 to be M and class2 to be B')
print('tp: ',a,'fp: ',c,'tn: ',d,'fn: ',b)
print('average accuracy: ',avg_acc_score)
print('average sensitivity: ',avg_specivity)
print('average specificity: ',avg_specivity)
```

```
we are assuming class 1 to be M and class2 to be B
tp:  26 fp:  5 tn:  82 fn:  0
average accuracy:  0.9613258810743673
average sensitivity:  0.9705954898299902
average specificity:  0.9705954898299902
```

Question 9

```
import math
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold, train_test_split
```

```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_q6_q7.xlsx to data_q6_q7.xlsx
```

```
# a function to implement LRT
def MAP_rule(x_ts,x,y):
    p1 = len([i for (i, val) in enumerate(y) if val == 1])
    p2 = len([i for (i, val) in enumerate(y) if val == 2])
    p3 = len([i for (i, val) in enumerate(y) if val == 3])
    # priors P(y)
    p1, p2, p3 = p1/(len(y)), p2/(len(y)), p3/(len(y))
    x1 = np.array([x[i] for (i, val) in enumerate(y) if val == 1])
    x2 = np.array([x[i] for (i, val) in enumerate(y) if val == 2])
    x3 = np.array([x[i] for (i, val) in enumerate(y) if val == 3])
    # evidence P(x)
    e1, e2, e3 = len(x1)/(len(x)), len(x2)/(len(x)), len(x3)/(len(x))
    m1 = np.mean(x1,axis = 0)
    m2 = np.mean(x2, axis = 0)
    m3 = np.mean(x3,axis = 0)
    cov1 = np.cov((x1.T).astype(float))
    cov2 = np.cov((x2.T).astype(float))
    cov3 = np.cov((x3.T).astype(float))
    coeff1 = 1/(((2*3.14)**2)*np.linalg.det(cov1)**0.5)
    coeff2 = 1/(((2*3.14)**2)*np.linalg.det(cov2)**0.5)
    coeff3 = 1/(((2*3.14)**2)*np.linalg.det(cov3)**0.5)
    # likelihoods P(x|y)
    l1 = coeff1*np.exp(-0.5*np.dot(np.dot((x_ts -
m1),np.linalg.inv(cov1)),(x_ts - m1).T))
    l2 = coeff2*np.exp(-0.5*np.dot(np.dot((x_ts -
m2),np.linalg.inv(cov2)),(x_ts - m2).T))
    l3 = coeff3*np.exp(-0.5*np.dot(np.dot((x_ts -
m3),np.linalg.inv(cov3)),(x_ts - m3).T))
    # Posteriors P(y|x)
    prob1, prob2, prob3 = (l1*p1)/e1, (l2*p2)/e2, (l3*p3)/e3
    if max(prob1,prob2,prob3) == prob1:
        return 1
    elif max(prob1,prob2,prob3) == prob2:
        return 2
    else:
        return 3
```

```
data = pd.read_excel('data_q6_q7.xlsx',header=None)
data = np.asarray(data)
X = data[:,:-1]
y = data[:,-1]

kf = KFold(n_splits=5, random_state=None)
acc_score = []
sens_score = []
spec_score = []

for train_index , test_index in kf.split(X):
    X_train , X_test = X[train_index,:],X[test_index,:]
    y_train , y_test = y[train_index] , y[test_index]
    y_pred = []

    for i in range(len(X_test)):
      y_pred.append(MAP_rule(X_test[i],X_train,y_train))

    a, b, c, d = confmat(y_pred,y_test)
    acc = (a+d)/(a+b+c+d)
    sens = (a)/(a+b)
    spec = (d)/(d+c)
    acc_score.append(acc)
    sens_score.append(sens)
    spec_score.append(spec)

print(sum(acc_score)/5)
print(sum(sens_score)/5)
print(sum(spec_score)/5)
```

Question 10

```
import pandas as pd
import math
import numpy as np
from sklearn.model_selection import train_test_split
```

```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_q6_q7.xlsx to data_q6_q7.xlsx
```

```
# A function to return a column of the data at the specified index
def col(array, i):
    return [row[i] for row in array]

# A function to calculate the mean of an array
def mean(array):
    m = []
    for i in range(7):
        m.append(sum(col(array,i))/len(col(array,i)))
    return m

# a function to implement LRT
def rule(x_ts,x,y):
    x1 = np.array([x[i] for (i, val) in enumerate(y) if val == 1])
    x2 = np.array([x[i] for (i, val) in enumerate(y) if val == 2])
    x3 = np.array([x[i] for (i, val) in enumerate(y) if val == 3])
    m1 = mean(x1)
    m2 = mean(x2)
    m3 = mean(x3)
    cov1 = np.cov((x1.T).astype(float))
    cov2 = np.cov((x2.T).astype(float))
    cov3 = np.cov((x3.T).astype(float))
    coeff1 = 1/(((2*3.14)**2)*np.linalg.det(cov1)**0.5)
    coeff2 = 1/(((2*3.14)**2)*np.linalg.det(cov2)**0.5)
    coeff3 = 1/(((2*3.14)**2)*np.linalg.det(cov3)**0.5)
    # likelihoods P(x|y)
    l1 = coeff1*np.exp(-0.5*np.dot(np.dot((x_ts -
m1),np.linalg.inv(cov1)),(x_ts - m1).T))
    l2 = coeff2*np.exp(-0.5*np.dot(np.dot((x_ts -
m2),np.linalg.inv(cov2)),(x_ts - m2).T))
    l3 = coeff3*np.exp(-0.5*np.dot(np.dot((x_ts -
m3),np.linalg.inv(cov3)),(x_ts - m3).T))
    if max(l1,l2,l3) == l1:
        return 1
    elif max(l1,l2,l3) == l2:
        return 2
    else:
        return 3
```

```python
def confmat(y_pred,y_ts):
    a, b, c, d = 0, 0, 0, 0
    for i in range(len(y_ts)):
        if y_ts[i] == 1:
            if y_pred[i] == 1:
                a = a + 1
            if y_pred[i] == 2:
                b = b + 1
        if y_ts[i] == 2:
            if y_pred[i] == 1:
                c = c + 1
            if y_pred[i] == 2:
                d = d + 1
    return a, b, c, d

# input the data csv
data = pd.read_excel('data_q6_q7.xlsx',header=None)
data = np.asarray(data)

x = data[:,:-1]
y = data[:,-1]
x_tr, x_ts, y_tr, y_ts = train_test_split(x, y, test_size=0.3)

y_pred = []
for i in range(len(x_ts)):
    y_pred.append(rule(x_ts[i],x_tr,y_tr))

y_actual = pd.Series(y_ts, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
confmat = pd.crosstab(y_actual, y_pred)
print(confmat)
confmat = np.asarray(confmat)
Acc = (confmat[0][0] + confmat[1][1] + confmat[2]
[2])/sum(sum(confmat))
Acc1 = confmat[0][0]/sum(confmat[0])
Acc2 = confmat[1][1]/sum(confmat[1])
Acc3 = confmat[2][2]/sum(confmat[2])
print('Overall Accuracy : ' + str(Acc))
print('Accuracy of class 1 : ' + str(Acc1))
print('Accuracy of class 2 : ' + str(Acc2))
print('Accuracy of class 3 : ' + str(Acc3))
```

```
Predicted   1   2   3
Actual
1          23   0   2
2           0  17   0
3           0   0  22
Overall Accuracy : 0.96875
Accuracy of class 1 : 0.92
```

```
Accuracy of class 2 : 1.0
Accuracy of class 3 : 1.0
```

Question 11

```
import pandas as pd
import cmath as math
import numpy as np
import random
from random import randint
import matplotlib.pyplot as plt

from google.colab import files
uploaded = files.upload()

<IPython.core.display.HTML object>

Saving data_q11.xlsx to data_q11.xlsx

def norm(datan):
  data_min = np.min(datan, axis = 0)
  data_max = np.max(datan, axis = 0)
  datan = (datan- data_min)/(data_max-data_min)
  return datan

# A function to calculate the distance between two points
#n = no. of features
def dist(x,y,n):
    sum = 0
    for a in range(n):
        sum = sum + (x[a]-y[a])**2
    return (math.sqrt(sum)).real

# A function to calculate the distances from initialized centroids
def distcen(data,cen):
    distc = [0 for x in range(len(data))]
    Dist = []
    for j in range (20) :
        for i in range(len(data)):
           distc[i] = (dist(data[i],data[cen[j]],data.shape[1]))
           Dist.append(dist)

    return Dist

data = pd.read_excel(uploaded['data_q11.xlsx'])
data = np.asarray(data)
data = norm(data)
X = data

m = X.shape[0]
n = X.shape[1]

def recalculate_clusters(X, centroids, k):
    """ Recalculates the clusters """
    # Initiate empty clusters
    clusters = {}
    # Set the range for value of k (number of centroids)
```

```python
    for i in range(k):
        clusters[i] = []
    # Setting the plot points using dataframe (X) and the vector norm
(magnitude/length)
    for data in X:
        # Set up list of euclidian distance and iterate through
        euc_dist = []
        for j in range(k):
            euc_dist.append(np.linalg.norm(data - centroids[j]))
        # Append the cluster of data to the dictionary
        clusters[euc_dist.index(min(euc_dist))].append(data)
    return clusters

def recalculate_centroids(centroids, clusters, k):
    """ Recalculates the centroid position based on the plot """
    for i in range(k):
        # Finds the average of the cluster at given index
        centroids[i] = np.average(clusters[i], axis=0)
    return centroids


def k_means_clustering(X, centroids={}, k=20, repeats=10):
    """ Calculates full k_means_clustering algorithm """
    for i in range(k):
        # Sets up the centroids based on the data
        centroids[i] = X[i]

    # Outputs the recalculated clusters and centroids
    print(f'First and last of {repeats} iterations')
    for i in range(repeats):
        clusters = recalculate_clusters(X, centroids, k)
        centroids = recalculate_centroids(centroids, clusters, k)

    return clusters,centroids
```

```python
final_cluster,final_centroids = k_means_clustering(X)
```

```
First and last of 10 iterations
```

```python
final_centroids
```

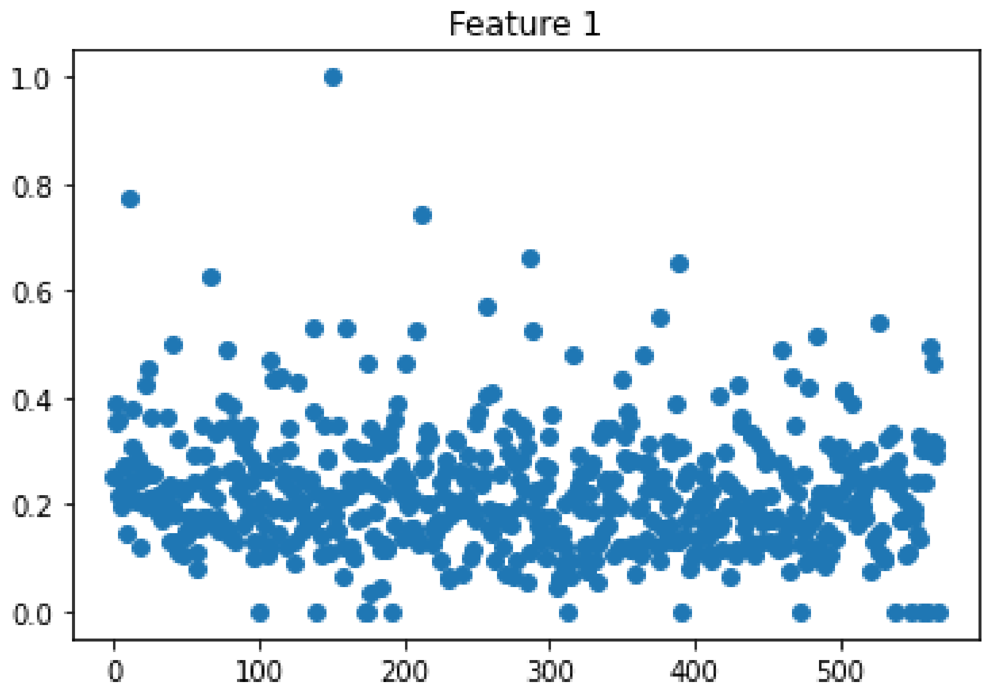
```
{0: array([0.26013352, 0.13874618, 0.07833328, 0.48347011, 0.33191493,
        0.45267916, 0.30743793, 0.37470596, 0.19465491, 0.2431062 ,
        0.51796658, 0.22790024, 0.12405515]),
 1: array([0.38482667, 0.24778029, 0.16480971, 0.59602899, 0.40984808,
        0.58809577, 0.41205024, 0.5397626 , 0.47583462, 0.51622404,
        0.85034364, 0.42361522, 0.30263676]),
 2: array([0.42242849, 0.86407384, 0.23156516, 0.39879047, 0.45549041,
        0.41921908, 0.22369003, 0.76391732, 0.69278459, 0.58107029,
        0.8733677 , 0.88192391, 0.53312344]),
 3: array([0.44701174, 0.41246412, 0.14353387, 0.50403178, 0.41111407,
```

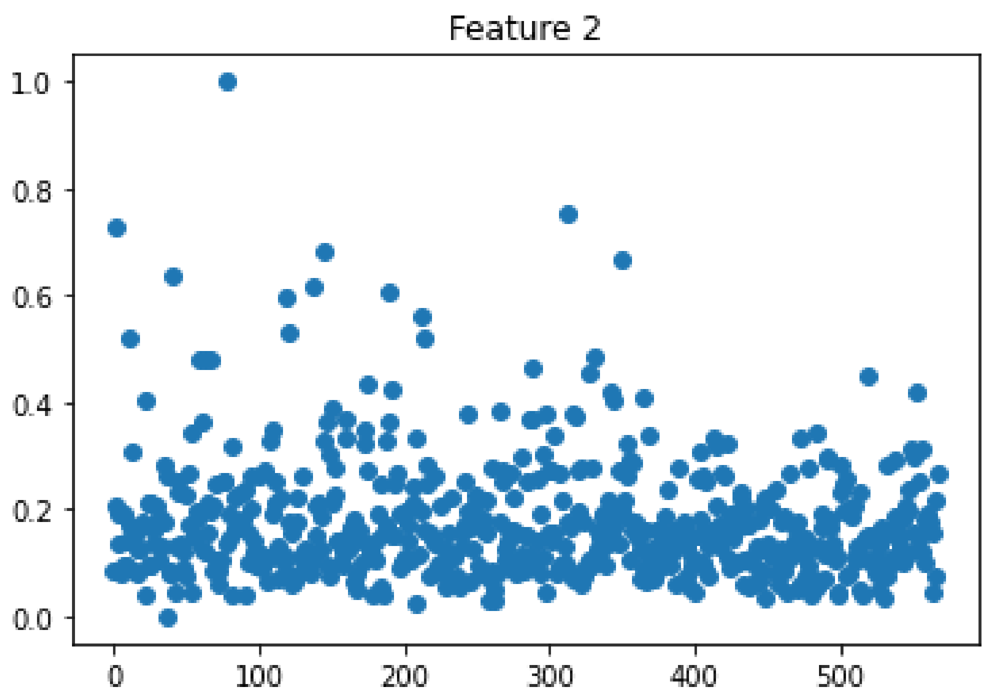
```
       0.49038797, 0.32265861, 0.31882718, 0.2259947 , 0.25730165,
       0.60495418, 0.29867271, 0.11645678]),
 4: array([0.26607786, 0.17019615, 0.27247437, 0.18108324, 0.28414845,
       0.19316201, 0.08635224, 0.60253913, 0.42351631, 0.43772963,
       0.46180842, 0.31076286, 0.4978191 ]),
 5: array([0.27416712, 0.12976449, 0.08923653, 0.55840152, 0.50863813,
       0.52703488, 0.37557977, 0.47116125, 0.29852437, 0.35187392,
       0.65106177, 0.29027481, 0.19578906]),
 6: array([0.23644472, 0.13714682, 0.11546791, 0.25315427, 0.49201537,
       0.24873417, 0.12743827, 0.59978758, 0.30958595, 0.28013845,
       0.47578465, 0.31638905, 0.29275165]),
 7: array([0.29164404, 0.37791974, 0.1198272 , 0.33475631, 0.43034826,
       0.33425746, 0.18292044, 0.5392077 , 0.43580423, 0.45468584,
       0.67468499, 0.63306831, 0.32494498]),
 8: array([0.2712635 , 0.14082287, 0.31733068, 0.25471363, 0.76385928,
       0.23527068, 0.1293256 , 0.75368157, 1.        , 0.88258786,
       0.75945017, 0.55213877, 1.        ]),
 9: array([0.14400921, 0.14756225, 0.05326102, 0.21808828, 0.51534771,
       0.19988354, 0.10627325, 0.29916336, 0.10786515, 0.09613648,
       0.21055612, 0.19769822, 0.10912925]),
 10: array([0.2887706 , 0.14434063, 0.13669451, 0.34882604,
0.3833822 ,
       0.34333383, 0.19174695, 0.52410355, 0.42571625, 0.40574681,
       0.6516323 , 0.34420461, 0.33449429]),
 11: array([0.56117531, 0.31317736, 0.53641901, 0.17578956,
0.27407605,
       0.17986177, 0.08507013, 0.29862716, 0.27583898, 0.3592208 ,
       0.40540664, 0.19990363, 0.3267961 ]),
 12: array([0.31727126, 0.23816362, 0.17000271, 0.20618774,
0.31891325,
       0.200729  , 0.09802184, 0.295935  , 0.20751836, 0.18813399,
       0.32846005, 0.20020205, 0.18433319]),
 13: array([0.38602008, 0.29737716, 0.24264472, 0.22070438,
0.49109808,
       0.24484287, 0.10812033, 0.62213564, 0.70866684, 0.72479233,
       0.73738832, 0.48038636, 0.50609996]),
 14: array([0.24824777, 0.13049708, 0.14851743, 0.43027392,
0.72756752,
       0.44427843, 0.25318276, 0.60179621, 0.66521459, 0.61130192,
       0.76357388, 0.46300677, 0.49254449]),
 15: array([0.21023484, 0.10978294, 0.06066113, 0.32862211,
0.44749467,
       0.31215752, 0.18003781, 0.38765749, 0.22578427, 0.2390316 ,
       0.44231819, 0.26110143, 0.16158425]),
 16: array([0.24967323, 0.12779544, 0.11403272, 0.44978062,
0.53520345,
       0.44295035, 0.27698584, 0.70233771, 0.43985214, 0.47107295,
       0.70063001, 0.38205533, 0.33129455]),
 17: array([0.29082213, 0.11440451, 0.08331735, 0.79884827,
0.52108875,
```

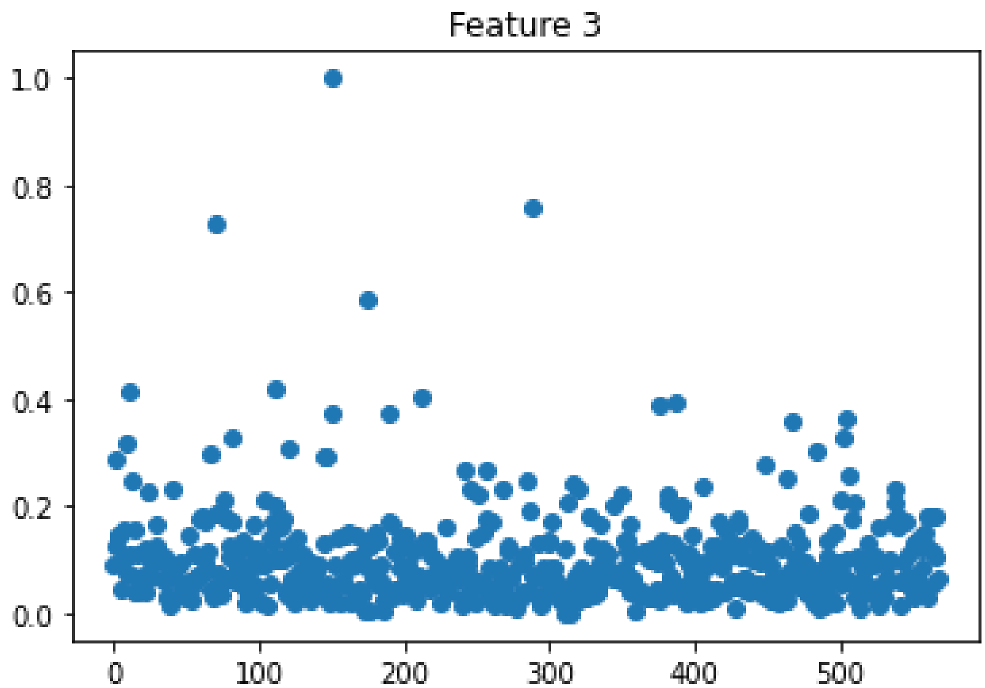
```
        0.77059366, 0.65522697, 0.45238724, 0.34482056, 0.38701577,
        0.78344072, 0.2600409 , 0.18108438]),
 18: array([0.17902546, 0.14323749, 0.07435225, 0.21571671,
0.24216915,
        0.20347889, 0.10340137, 0.42200737, 0.17239211, 0.14976424,
        0.31138185, 0.2494078 , 0.17368601]),
 19: array([0.14102734, 0.2186487 , 0.07053706, 0.15183011,
0.23925511,
        0.13766225, 0.0687003 , 0.3026542 , 0.07655394, 0.04982317,
        0.139014  , 0.20261223, 0.11605169])}
```

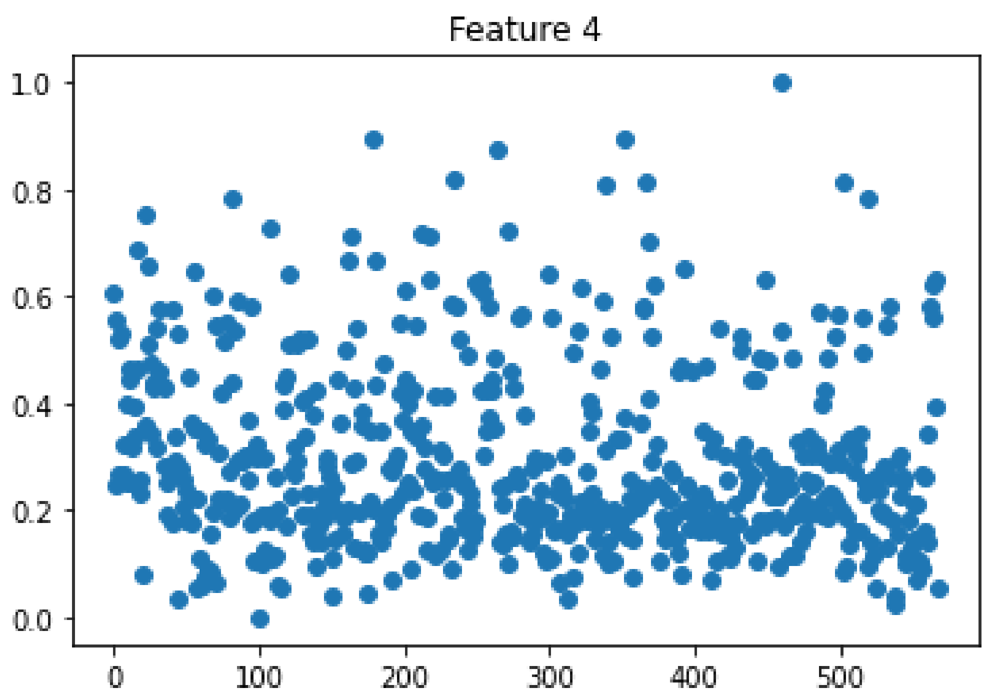
```
plt.scatter(np.arange(len(data[:,0])),data[:,0])
plt.title('Feature 1')
plt.show()
plt.scatter(np.arange(len(data[:,1])),data[:,1])
plt.title('Feature 2')
plt.show()
plt.scatter(np.arange(len(data[:,2])),data[:,2])
plt.title('Feature 3')
plt.show()
plt.scatter(np.arange(len(data[:,3])),data[:,3])
plt.title('Feature 4')
plt.show()
plt.scatter(np.arange(len(data[:,4])),data[:,4])
plt.title('Feature 5')
plt.show()
plt.scatter(np.arange(len(data[:,5])),data[:,5])
plt.title('Feature 6')
plt.show()
plt.scatter(np.arange(len(data[:,6])),data[:,6])
plt.title('Feature 7')
plt.show()
plt.scatter(np.arange(len(data[:,7])),data[:,7])
plt.title('Feature 8')
plt.show()
plt.scatter(np.arange(len(data[:,8])),data[:,8])
plt.title('Feature 9')
plt.show()
plt.scatter(np.arange(len(data[:,9])),data[:,9])
plt.title('Feature 10')
plt.show()
plt.scatter(np.arange(len(data[:,10])),data[:,10])
plt.title('Feature 11')
plt.show()
plt.scatter(np.arange(len(data[:,11])),data[:,11])
plt.title('Feature 12')
plt.show()
plt.scatter(np.arange(len(data[:,12])),data[:,12])
```

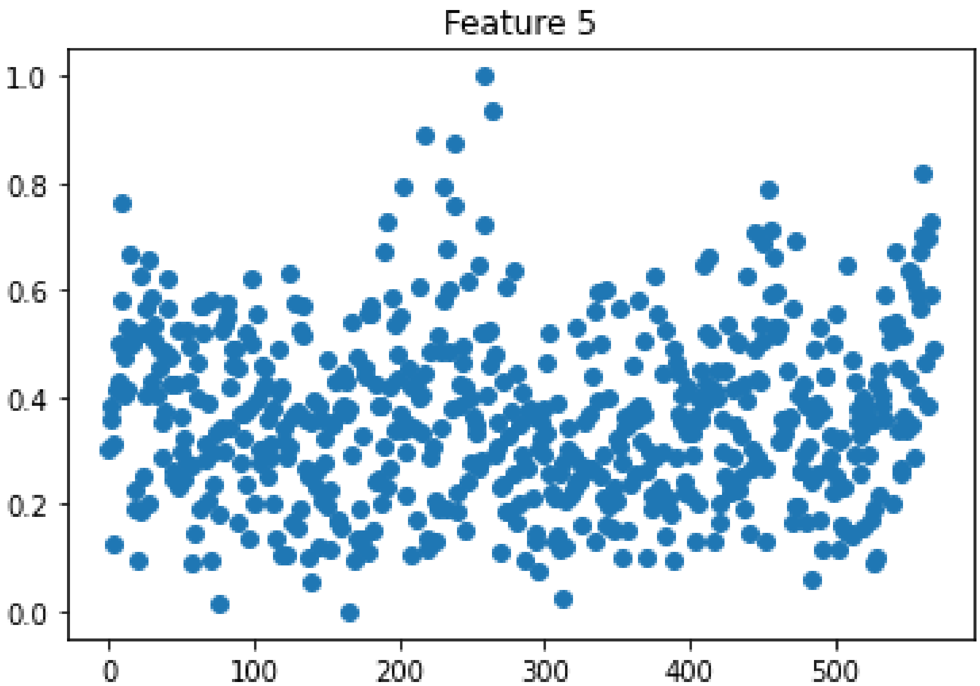
```
plt.title('Feature 13')
plt.show()
```

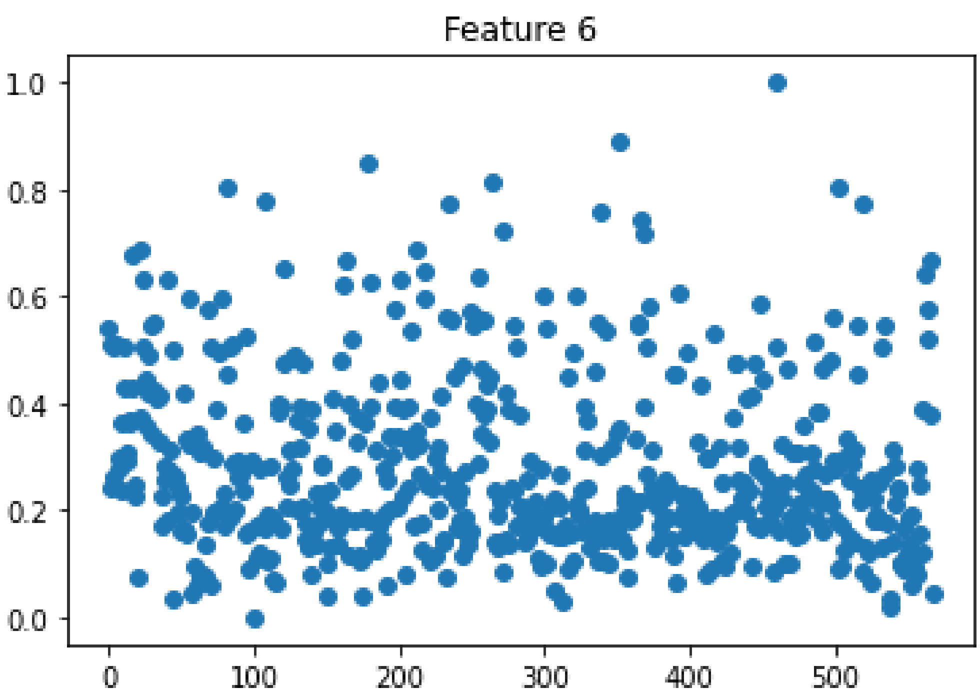


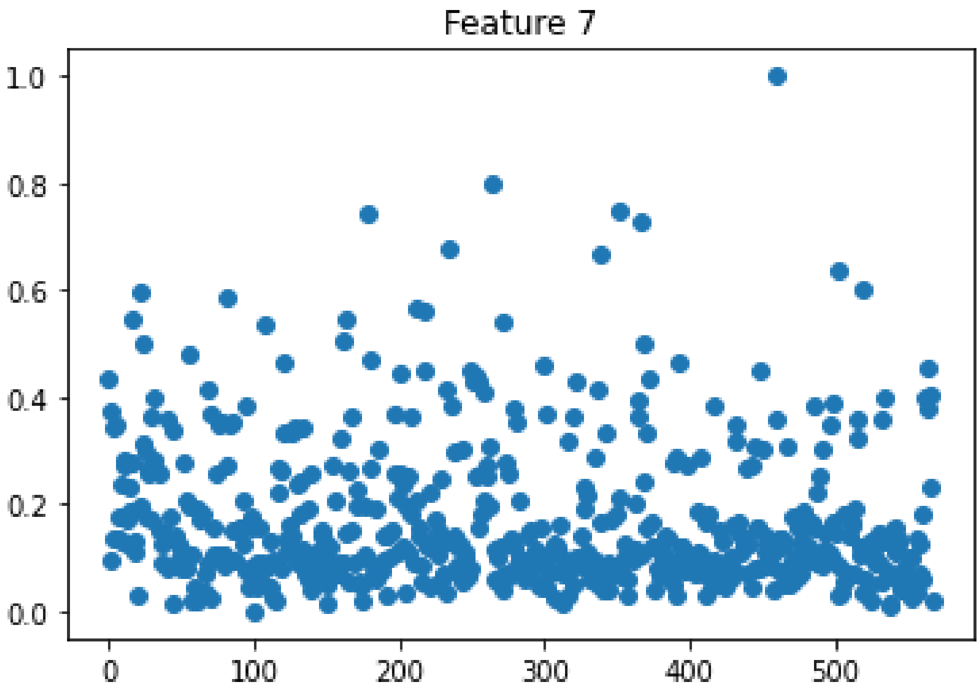


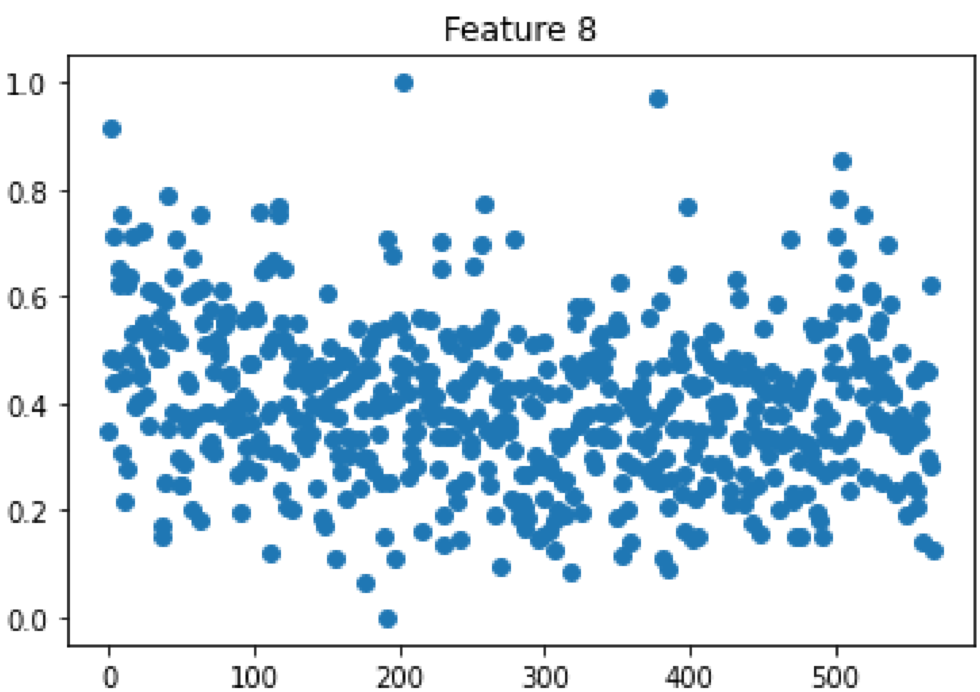

Feature 3
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500

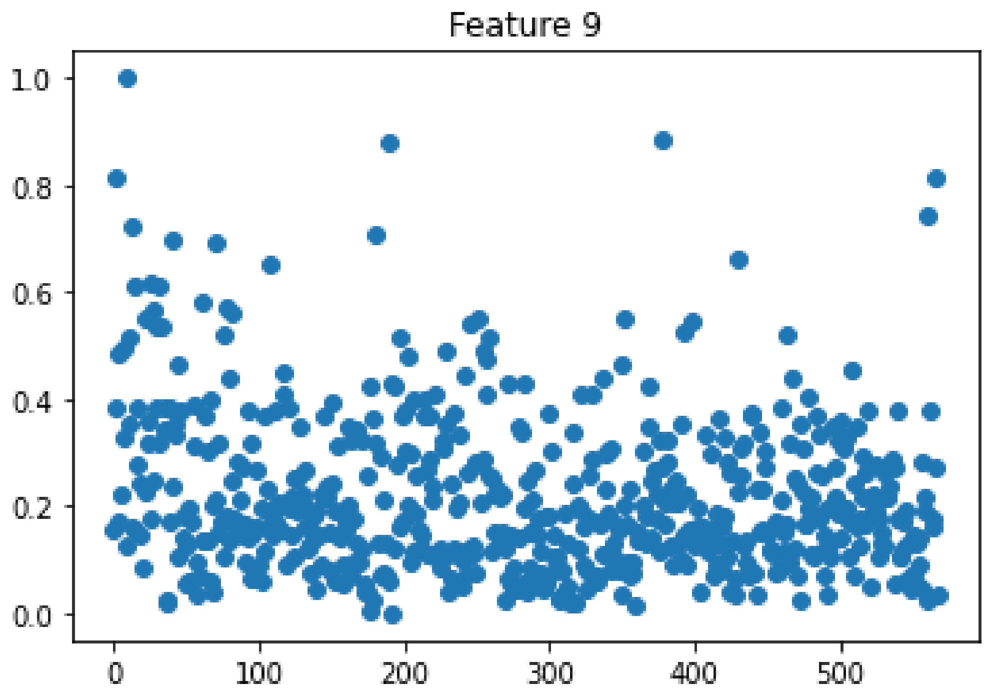

Feature 4
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500


Feature 5
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500

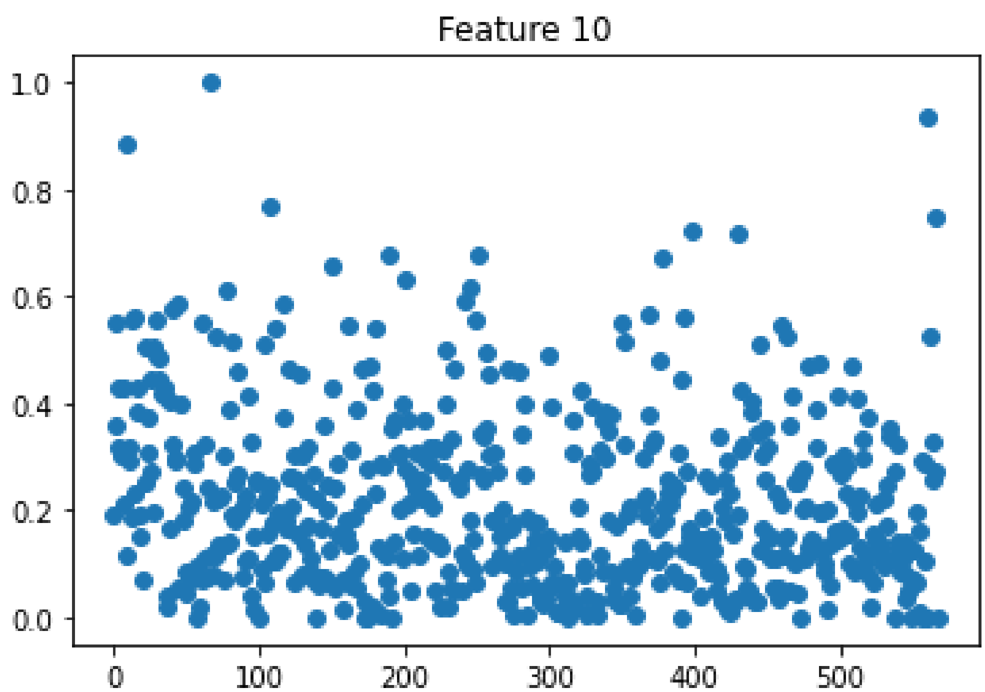

Feature 6
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500


Feature 7
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500

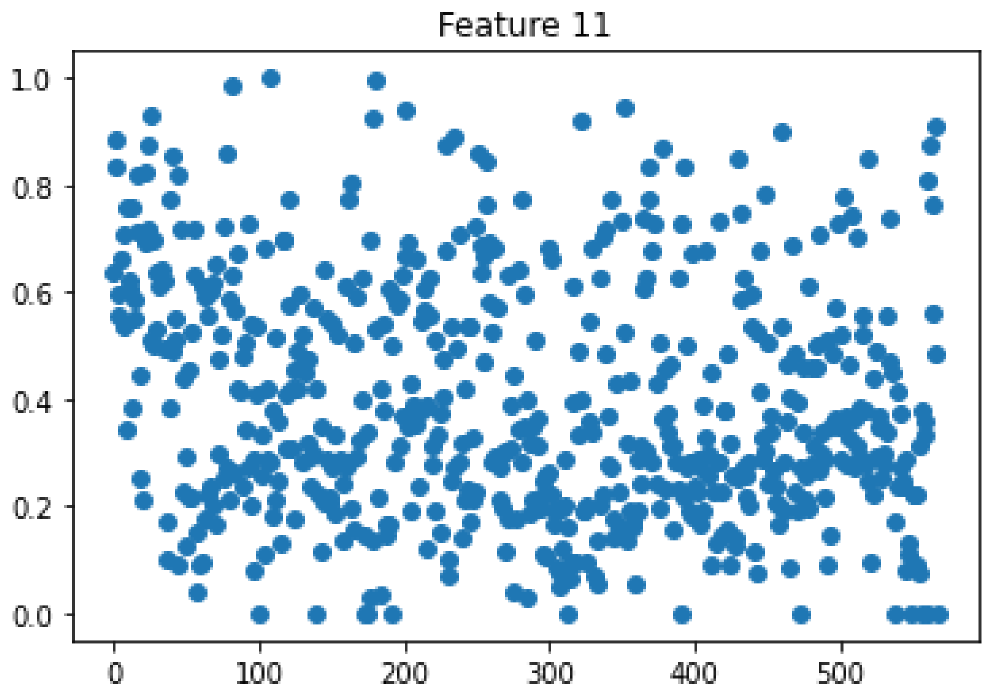

Feature 8
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500

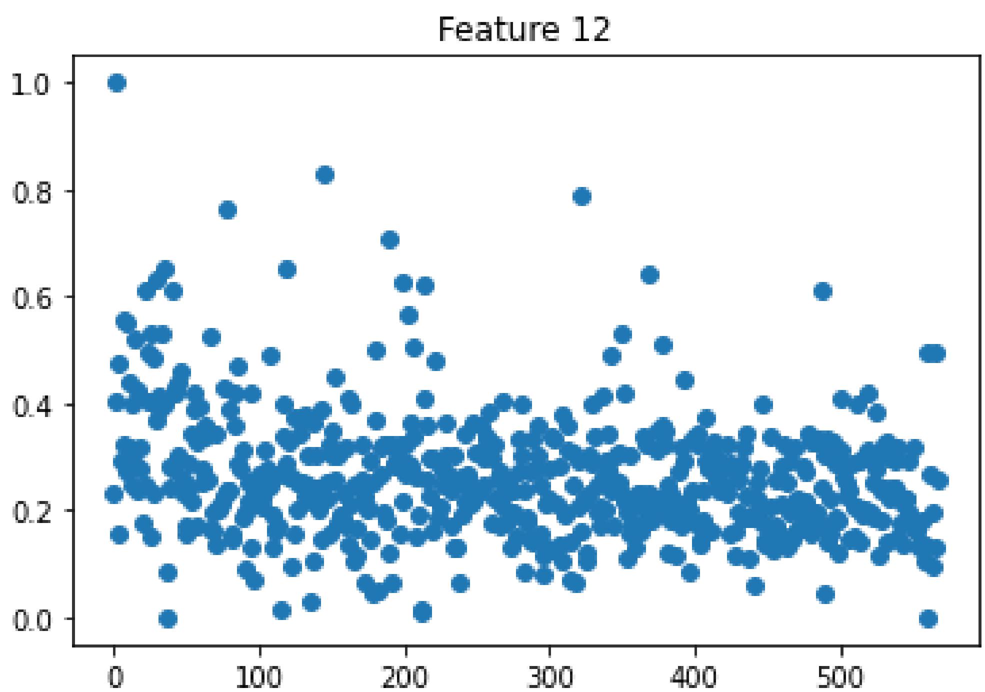


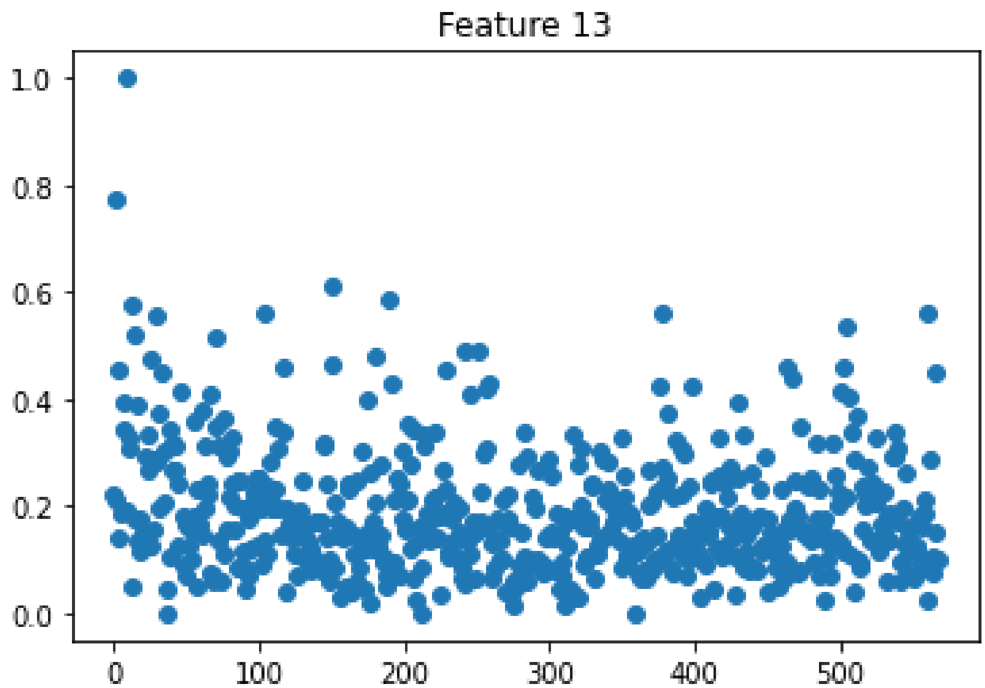
Feature 9
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500



Feature 10
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500


Feature 11
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500


Feature 12
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500


Feature 13
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500