Neural Networks and Fuzzy Logic

BITS F312

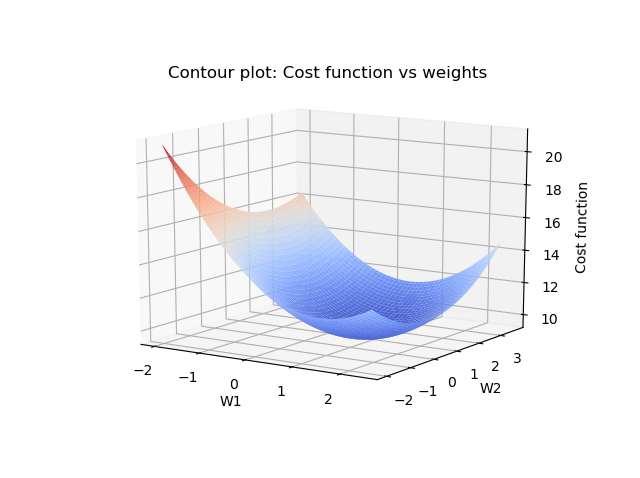
Assignment 1

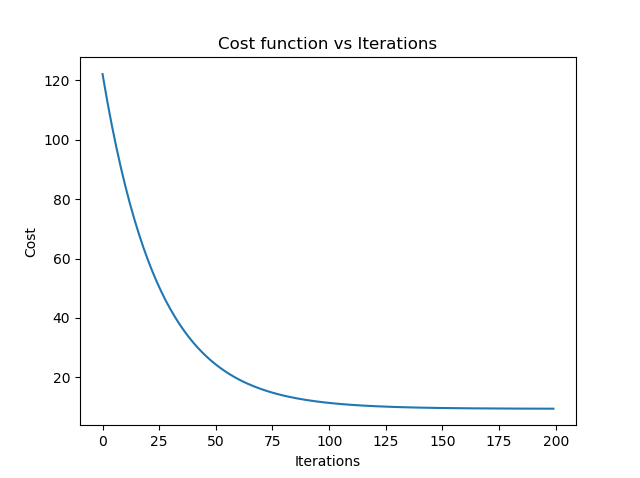
Name: Tejas Radhakrishnan

ID No.: 2017AAPS1736H

# 

## 1. Batch Gradient Descent





##Batch gradient descent

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 200

w = [0, 0]

b = 0

lr = 2e-2

J = []

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(error) :

return np.sum(0.5 \* (error \*\*2) / n)

#Training loop

for i in range(num\_epochs) :

error = np.zeros\_like(y)

y\_pred = np.sum(w \* X, axis = 1) + b

error = y\_pred - y

w[0] -= lr \* np.sum(error \* X[:, 0])/n

w[1] -= lr \* np.sum(error \* X[:, 1])/n

b -= lr \* np.sum(error)/n

J.append(eval\_cost(error))

w1\_history.append(w[0])

w2\_history.append(w[1])

#Plots

plt.plot(J)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.title('Cost function vs Iterations')

print(eval\_cost(error))

print(w, b)

ww1 = np.linspace(min(w1\_history)-2,max(w1\_history)+2,100)

ww2 = np.linspace(min(w2\_history)-2,max(w2\_history)+2,100)

J\_cont = np.zeros((100, 100))

for i1, w1 in enumerate(ww1):

for i2, w2 in enumerate(ww2):

cost = 0

for i in range(n):

error = np.zeros\_like(y)

y\_pred = w1 \* X[:, 0] + w2 \* X[:, 1] + b

error = y\_pred - y

cost = eval\_cost(error)

J\_cont[i1, i2] = cost

www1,www2 = np.meshgrid(ww1,ww2)

fig = plt.figure()

ax = fig.add\_subplot(111,projection = '3d')

ax.plot\_surface(www1,www2,J\_cont, cmap=cm.coolwarm)

ax.set\_xlabel('W1')

ax.set\_ylabel('W2')

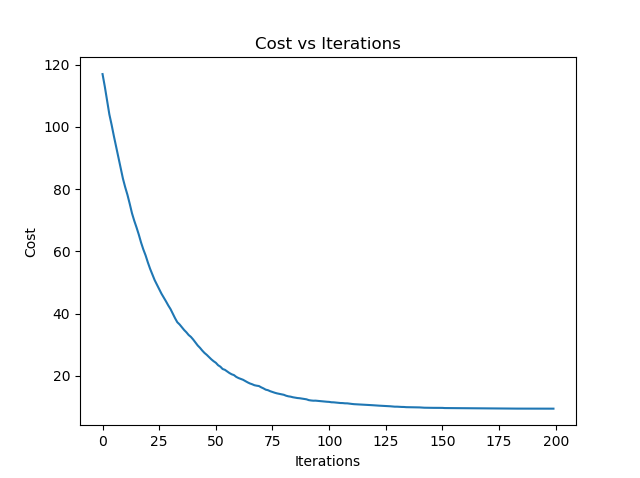
ax.set\_zlabel('Cost function')

plt.title("Contour plot: Cost function vs weights")

plt.show()

## 

## 2. (a) Mini Batch Gradient Descent



## Mini batch gradient descent

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 200

w = [0, 0]

b = 0

lr = 2e-2

J = []

batch\_size = 16

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(X, y, w, b) :

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

return 0.5 \* np.sum(error \*\*2) /n

#Training function to update weights once

def train(X, y, n) :

global w, b

error = np.zeros\_like(y)

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

w[0] -= lr \* np.sum(error \* X[:, 0])/n

w[1] -= lr \* np.sum(error \* X[:, 1])/n

b -= lr \* np.sum(error)/n

w1\_history.append(w[0])

w2\_history.append(w[1])

#Training loop

for j in range(num\_epochs):

rand\_sample = np.random.randint(n, size = batch\_size)

batch\_X = X[rand\_sample, :]

batch\_y = y[rand\_sample]

train(batch\_X, batch\_y, batch\_size)

J.append(eval\_cost(X, y, w, b))

#Plots

plt.plot(J)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.title('Cost vs Iterations')

print(J[-1])

print(w, b)

ww1 = np.linspace(min(w1\_history)-2,max(w1\_history)+2,100)

ww2 = np.linspace(min(w2\_history)-2,max(w2\_history)+2,100)

J\_cont = np.zeros((100, 100))

for i1, w1 in enumerate(ww1):

for i2, w2 in enumerate(ww2):

cost = 0

for i in range(n):

error = np.zeros\_like(y)

y\_pred = w1 \* X[:, 0] + w2 \* X[:, 1] + b

error = y\_pred - y

cost = eval\_cost(X, y, [w1, w2], b)

J\_cont[i1, i2] = cost

www1,www2 = np.meshgrid(ww1,ww2)

fig = plt.figure()

ax = fig.add\_subplot(111,projection = '3d')

ax.plot\_surface(www1,www2,J\_cont, cmap=cm.coolwarm)

ax.set\_xlabel('W1')

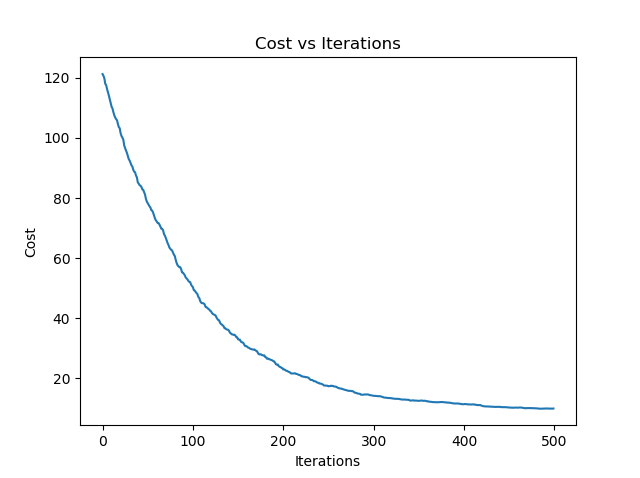
ax.set\_ylabel('W2')

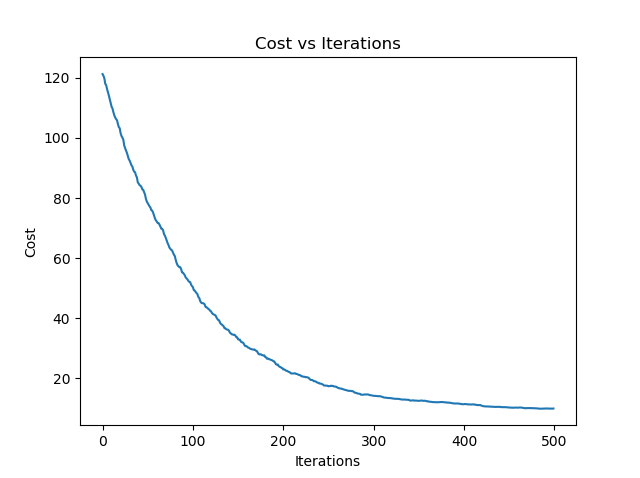
ax.set\_zlabel('Cost function')

plt.title("Contour plot: Cost function vs weights")

plt.show()

## 2.(b) Stochastic Gradient Descent



## Stochastic gradient descent

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 500

w = [0, 0]

b = 0

lr = 5e-3

J = []

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(X, y, w, b) :

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

return 0.5 \* np.sum(error \*\*2)/n

#Training loop

for i in range(num\_epochs) :

j = np.random.randint(n)

y\_pred = X[j, 0] \* w[0] + X[j, 1] \* w[1] + b

error = y\_pred - y[j]

w[0] += -(lr \* error \* X[j, 0])

w[1] += -(lr \* error \* X[j, 1])

b += -(lr \* error)

J.append(eval\_cost(X, y, w, b))

w1\_history.append(w[0])

w2\_history.append(w[0])

#Plots

plt.plot(J)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.title('Cost vs Iterations')

print(J[-1])

print(w, b)

ww1 = np.linspace(min(w1\_history)-2,max(w1\_history)+2,100)

ww2 = np.linspace(min(w2\_history)-2,max(w2\_history)+2,100)

J\_cont = np.zeros((100, 100))

# plt.plot(w1\_history, w2\_history, J)

# plt.show()

for i1, w1 in enumerate(ww1):

for i2, w2 in enumerate(ww2):

cost = 0

for i in range(n):

error = np.zeros\_like(y)

y\_pred = w1 \* X[:, 0] + w2 \* X[:, 1] + b

error = y\_pred - y

cost = eval\_cost(X, y, [w1, w2], b)

cost = np.sum(cost)

J\_cont[i1, i2] = cost

www1,www2 = np.meshgrid(ww1,ww2)

fig = plt.figure()

ax = fig.add\_subplot(111,projection = '3d')

ax.plot\_surface(www1,www2,J\_cont, cmap=cm.coolwarm)

ax.set\_xlabel('W1')

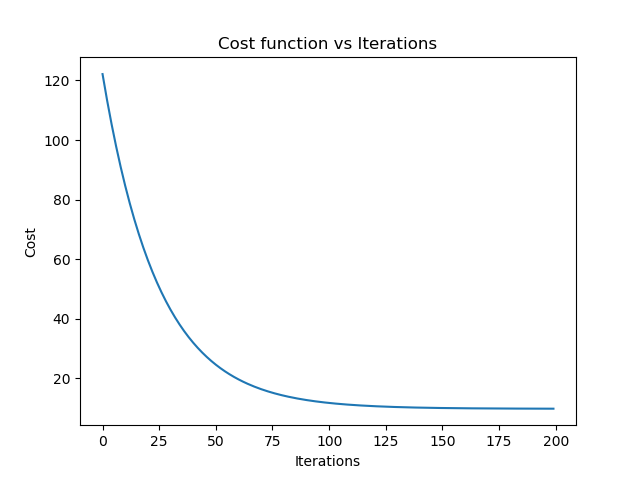
ax.set\_ylabel('W2')

ax.set\_zlabel('Cost function')

plt.title("Contour plot: Cost function vs weights")

plt.show()

## 3. (a) Ridge Regression (Batch Gradient Descent)



## Ridge regression (Batch gradient descent)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 200

w = [0, 0]

b = 0

lr = 2e-2

reg = 0.3

J = []

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(error) :

global w

return np.sum(0.5 \* (error \*\*2) / n) + 0.5 \* reg \* (w[0]\*\*2 + w[1]\*\*2)

#Training loop

for i in range(num\_epochs) :

error = np.zeros\_like(y)

y\_pred = np.sum(w \* X, axis = 1) + b

error = y\_pred - y

w[0] -= lr \* (np.sum(error \* X[:, 0])/n + reg \* w[0])

w[1] -= lr \* (np.sum(error \* X[:, 1])/n + reg \* w[1])

b -= lr \* np.sum(error)/n

J.append(eval\_cost(error))

w1\_history.append(w[0])

w2\_history.append(w[1])

#Plots

plt.plot(J)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.title('Cost function vs Iterations')

print(eval\_cost(error))

print(w, b)

ww1 = np.linspace(min(w1\_history)-2,max(w1\_history)+2,100)

ww2 = np.linspace(min(w2\_history)-2,max(w2\_history)+2,100)

J\_cont = np.zeros((100, 100))

for i1, w1 in enumerate(ww1):

for i2, w2 in enumerate(ww2):

cost = 0

for i in range(n):

error = np.zeros\_like(y)

y\_pred = w1 \* X[:, 0] + w2 \* X[:, 1] + b

error = y\_pred - y

cost = eval\_cost(error)

J\_cont[i1, i2] = cost

www1,www2 = np.meshgrid(ww1,ww2)

fig = plt.figure()

ax = fig.add\_subplot(111,projection = '3d')

ax.plot\_surface(www1,www2,J\_cont, cmap=cm.coolwarm)

ax.set\_xlabel('W1')

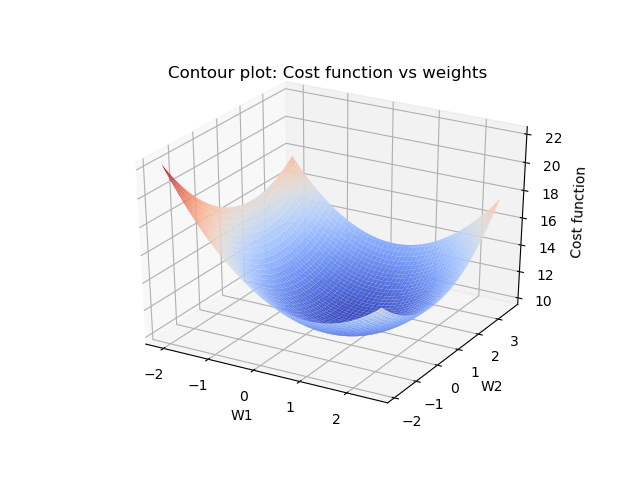
ax.set\_ylabel('W2')

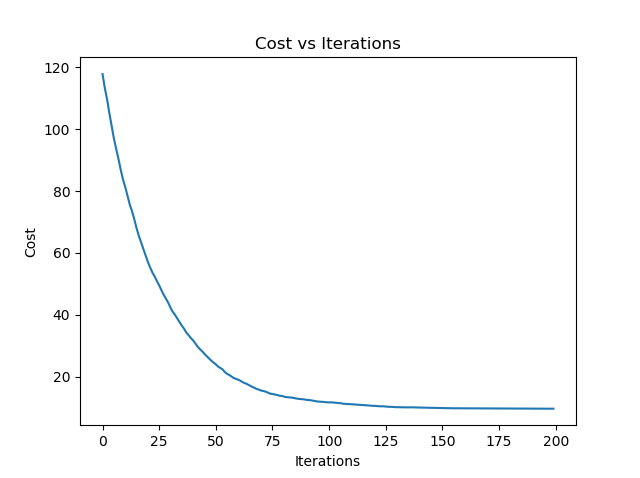
ax.set\_zlabel('Cost function')

plt.title("Contour plot: Cost function vs weights")

plt.show()

## 3. (b) Ridge Regression (Minibatch Gradient Descent)





## Ridge regression (minibatch)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 200

w = [0, 0]

b = 0

lr = 2e-2

reg = 0.3

J = []

batch\_size = 16

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(X, y, w, b) :

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

return 0.5 \* np.sum(error \*\*2) /n + 0.5 \* reg \* (w[0]\*\*2 + w[1]\*\*2)

#Training function to update weights once

def train(X, y, n) :

global w, b

error = np.zeros\_like(y)

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

w[0] -= lr \* (np.sum(error \* X[:, 0])/n + reg \* w[0])

w[1] -= lr \* (np.sum(error \* X[:, 1])/n + reg \* w[1])

b -= lr \* np.sum(error)/n

w1\_history.append(w[0])

w2\_history.append(w[1])

#Training loop

for j in range(num\_epochs):

rand\_sample = np.random.randint(n, size = batch\_size)

batch\_X = X[rand\_sample, :]

batch\_y = y[rand\_sample]

train(batch\_X, batch\_y, batch\_size)

J.append(eval\_cost(X, y, w, b))

#PLots

plt.plot(J)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.title('Cost vs Iterations')

print(J[-1])

print(w, b)

ww1 = np.linspace(min(w1\_history)-2,max(w1\_history)+2,100)

ww2 = np.linspace(min(w2\_history)-2,max(w2\_history)+2,100)

J\_cont = np.zeros((100, 100))

for i1, w1 in enumerate(ww1):

for i2, w2 in enumerate(ww2):

cost = 0

for i in range(n):

error = np.zeros\_like(y)

y\_pred = w1 \* X[:, 0] + w2 \* X[:, 1] + b

error = y\_pred - y

cost = eval\_cost(X, y, [w1, w2], b)

J\_cont[i1, i2] = cost

www1,www2 = np.meshgrid(ww1,ww2)

fig = plt.figure()

ax = fig.add\_subplot(111,projection = '3d')

ax.plot\_surface(www1,www2,J\_cont, cmap=cm.coolwarm)

ax.set\_xlabel('W1')

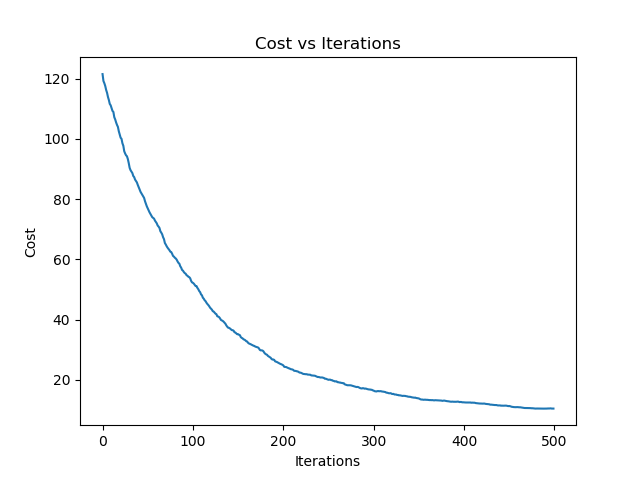
ax.set\_ylabel('W2')

ax.set\_zlabel('Cost function')

plt.title("Contour plot: Cost function vs weights")

plt.show()

## 3. (c) Ridge Regression (Stochastic Gradient Descent)



## Ridge regression (Stochastic)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

from mpl\_toolkits.mplot3d import Axes3D

from matplotlib import cm

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 500

w = [0, 0]

b = 0

lr = 5e-3

reg = 0.3

J = []

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(X, y, w, b) :

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

return 0.5 \* np.sum(error \*\*2)/n + 0.5 \* reg \* (w[0]\*\*2 + w[1]\*\*2)

#Training loop

for i in range(num\_epochs) :

j = np.random.randint(n)

y\_pred = X[j, 0] \* w[0] + X[j, 1] \* w[1] + b

error = y\_pred - y[j]

w[0] += -(lr \* (error \* X[j, 0] + reg \* w[0]))

w[1] += -(lr \* (error \* X[j, 1] + reg \* w[1]))

b += -(lr \* error)

J.append(eval\_cost(X, y, w, b))

w1\_history.append(w[0])

w2\_history.append(w[0])

#Plots

plt.plot(J)

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.title('Cost vs Iterations')

print(J[-1])

print(w, b)

ww1 = np.linspace(min(w1\_history)-2,max(w1\_history)+2,100)

ww2 = np.linspace(min(w2\_history)-2,max(w2\_history)+2,100)

J\_cont = np.zeros((100, 100))

# plt.plot(w1\_history, w2\_history, J)

# plt.show()

for i1, w1 in enumerate(ww1):

for i2, w2 in enumerate(ww2):

cost = 0

for i in range(n):

error = np.zeros\_like(y)

y\_pred = w1 \* X[:, 0] + w2 \* X[:, 1] + b

error = y\_pred - y

cost = eval\_cost(X, y, [w1, w2], b)

cost = np.sum(cost)/n

J\_cont[i1, i2] = cost

www1,www2 = np.meshgrid(ww1,ww2)

fig = plt.figure()

ax = fig.add\_subplot(111,projection = '3d')

ax.plot\_surface(www1,www2,J\_cont, cmap=cm.coolwarm)

ax.set\_xlabel('W1')

ax.set\_ylabel('W2')

ax.set\_zlabel('Cost function')

plt.title("Contour plot: Cost function vs weights")

plt.show()

## 4. Vectorized Linear Regression

|  |  |  |  |
| --- | --- | --- | --- |
| Weight Values | Vectorized | Batch | Stochastic |
| W0 | 14.90479943 | 14.64265461619355 | 13.73934498331477 |
| W1 | 0.36790549 | 0.395609981881525 | 0.265104189346941 |
| W2 | 1.6991193 | 1.658696618161573 | 1.6421193553151414 |

As we can see, the weight values from all methods are in close agreement with each other.

## Vectorized gradient descent

import numpy as np

import pandas as pd

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X2 = df[:, :-1]

temp = np.ones((X2.shape[0], X2.shape[1] + 1))

temp[:, :-1] = X2

X = temp

n = X.shape[0]

y = df[:, -1]

w = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(y))

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + w[2]

error = y\_pred - y

print(np.sum(0.5 \* (error \*\*2) / n))

print(w)

## 5. (a) Least Angle Regression (Batch Gradient descent)

|  |  |
| --- | --- |
| Weights | Values |
| W0 | 14.642654616193555 |
| W1 | 0.15838616041897743 |
| W2 | 1.4214727966990253 |

##Least Angle regression (Batch)

import numpy as np

import pandas as pd

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 200

w = [0, 0]

b = 0

lr = 2e-2

reg = 0.3

J = []

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(error) :

global w

return np.sum(0.5 \* (error \*\*2) / n) + 0.5 \* reg \* (abs(w[0]) + abs(w[1]))

#Training loop

for i in range(num\_epochs) :

error = np.zeros\_like(y)

y\_pred = np.sum(w \* X, axis = 1) + b

error = y\_pred - y

w[0] -= lr \* (np.sum(error \* X[:, 0])/n + reg \* np.sign(w[0]))

w[1] -= lr \* (np.sum(error \* X[:, 1])/n + reg \* np.sign(w[1]))

b -= lr \* np.sum(error)/n

J.append(eval\_cost(error))

w1\_history.append(w[0])

w2\_history.append(w[1])

print(eval\_cost(error))

print(w, b)

## 5. (b) Least Angle Regression (Stochastic Gradient Descent)

|  |  |
| --- | --- |
| Weights | Values |
| W0 | 13.512724547898621 |
| W1 | 0.1434771895838139 |
| W2 | 1.5617088077525068 |

## Least Angle regression (Stochastic)

import numpy as np

import pandas as pd

#Accept and normalize data, convert to numpy array

df = pd.read\_excel('data.xlsx', header = None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

X = df[:, :-1]

n = X.shape[0]

y = df[:, -1]

y\_pred = np.zeros\_like(y)

num\_epochs = 500

w = [0, 0]

b = 0

lr = 5e-3

reg = 0.3

J = []

w1\_history = []

w2\_history = []

#Cost function

def eval\_cost(X, y, w, b) :

y\_pred = X[:, 0] \* w[0] + X[:, 1] \* w[1] + b

error = y\_pred - y

return 0.5 \* np.sum(error \*\*2)/n + 0.5 \* reg \* (abs(w[0]) + abs(w[1]))

#Training loop

for i in range(num\_epochs) :

j = np.random.randint(n)

y\_pred = X[j, 0] \* w[0] + X[j, 1] \* w[1] + b

error = y\_pred - y[j]

w[0] += -(lr \* (error \* X[j, 0] + reg \* np.sign(w[0])))

w[1] += -(lr \* (error \* X[j, 1] + reg \* np.sign(w[1])))

b += -(lr \* error)

J.append(eval\_cost(X, y, w, b))

w1\_history.append(w[0])

w2\_history.append(w[0])

print(J[-1])

print(w, b)

## 

## 6. K - means Clustering

# K-means clustering

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits import mplot3d

df = pd.read\_excel('data2.xlsx', header = None)

df = df.to\_numpy()

X = df

n = X.shape[0]

dist1 = np.zeros(n)

dist2 = np.zeros(n)

class\_label = np.zeros(n)

init\_mean1 = np.random.random(4)

init\_mean2 = np.random.random(4)

def calculate\_dist(X\_in, mean) :

distance = np.sqrt(np.sum((X\_in - mean)\*\*2, axis = 1))

return distance

def calculate\_mean(X\_in, class\_label) :

X\_temp = np.reshape(class\_label, (n, -1)) \* X\_in

X1 = X\_temp \* (X\_temp>0)

X2 = -1 \*(X\_temp \* (X\_temp<0))

n1 = X1[X1>0].size/4

n2 = X2[X2>0].size/4

mean1 = np.sum(X1, axis = 0)/n1

mean2 = np.sum(X2, axis = 0)/n2

return mean1, mean2

init\_mean1 = X[np.random.randint(151)]

init\_mean2 = X[np.random.randint(151)]

mean1 = init\_mean1

mean2 = init\_mean2

prev\_sum\_mean = 0

new\_sum\_mean = 1

while(np.sum(new\_sum\_mean - prev\_sum\_mean) > 0.0001) :

prev\_sum\_mean = mean1 + mean2

dist1 = calculate\_dist(X, mean1)

dist2 = calculate\_dist(X, mean2)

class\_label = np.sign(dist1 - dist2)

mean1, mean2 = calculate\_mean(X, class\_label)

new\_sum\_mean = mean1 + mean2

LABEL\_COLOUR\_MAP = {-1 : 'r', 1 : 'b'}

label\_colour = [LABEL\_COLOUR\_MAP[l] for l in class\_label]

f = plt.figure(figsize = (100, 100))

plt.title('Class labels (Colour) vs pairs of features')

f.add\_subplot(3, 2, 1)

plt.scatter(X[:, 0], X[:, 1], c = label\_colour)

f.add\_subplot(3, 2, 2)

plt.scatter(X[:, 0], X[:, 2], c = label\_colour)

f.add\_subplot(3, 2, 3)

plt.scatter(X[:, 0], X[:, 3], c = label\_colour)

f.add\_subplot(3, 2, 4)

plt.scatter(X[:, 1], X[:, 2], c = label\_colour)

f.add\_subplot(3, 2, 5)

plt.scatter(X[:, 1], X[:, 3], c = label\_colour)

f.add\_subplot(3, 2, 6)

plt.scatter(X[:, 2], X[:, 3], c = label\_colour)

plt.show()

## 7. Logistic Regression for Binary Classification

|  |  |
| --- | --- |
| Accuracy | 1.0 |
| Sensitivity | 1.0 |
| Specificity | 1.0 |

import numpy as np

import pandas as pd

import matplotlib as plt

from math import exp, log

df = pd.read\_excel('data3.xlsx', header = None)

df = df.to\_numpy()

np.random.shuffle(df)

df = np.insert(df, 0, 1, axis = 1)

X = df[0:60, :-1]

y = df[0:60, -1]

X\_test = df[60:100, :-1]

y\_test = df[60:100, -1]

y -= 1

y\_test -= 1

w = np.zeros(5, dtype= 'float')

n = X.shape[0]

n\_test = X\_test.shape[0]

num\_epoch = 200

lr = 1e-2

J = []

def sigmoid(x) :

return 1/(1 + np.exp(-x))

def eval\_cost(h, y) :

return -(y \* np.log(h) + (1-y) \* np.log(1-h))

for i in range(num\_epoch) :

cost = 0

h = np.dot(X, w)

h = sigmoid(h)

error = h-y

cost = np.mean(eval\_cost(h, y))

grads = X.T.dot(error)

w -= lr \* grads

J.append(cost)

#print(cost)

TN, FN, TP, FP = 0, 0, 0, 0

y\_pred = np.dot(X\_test, w)

y\_pred = sigmoid(y\_pred)

y\_pred[y\_pred > 0.5] = 1

y\_pred[y\_pred <= 0.5] = 0

print(y\_pred)

TP = np.sum(y\_pred \* y\_test)

FP = np.sum(y\_pred \* (1-y\_test))

FN = np.sum((1-y\_pred) \* y\_test)

TN = np.sum((1-y\_pred) \* (1-y\_test))

print(y\_test)

sensitivity = TP/(TP + FN)

specificity = TN/(TN + FP)

accuracy = (TP + TN)/(TP + TN + FP + FN)

print(sensitivity, specificity, accuracy)

## 8. (a) Multiclass Logistic Regression (One vs All)

|  |  |
| --- | --- |
| Total Accuracy | 0.9666666666666667 |
| Class 1 Accuracy | 1.0 |
| Class 2 Accuracy | 0.9473684210526315 |
| Class 3 Accuracy | 0.9411764705882353 |

## Multiclass logistic regression (One vs All)

import numpy as np

import pandas as pd

from math import exp,log

import random

# Define activation function

def sigmoid(x):

return 1/(1+np.exp(-x))

# Function to calculate gradients for logistic regression

def getGradients(x, weights):

delta = np.zeros(len(x)-1, dtype=float)

h = np.dot(weights, x[:-1])

h = sigmoid(h)

delta = [(h-x[-1])\*x[i] for i in range(len(x)-1)]

return delta

# Training function

def train(data, num\_iter):

weights = np.zeros(np.size(data,1)-1, dtype=float)

for i in range(num\_iter):

x = data[random.randint(0,len(data)-1)]

delta = getGradients(x, weights)

weights = [weights[i] - lr\*delta[i] for i in range(len(x)-1)]

return weights

lr = 0.01

# Read, normalize and shuffle dataset

df = pd.read\_excel('data4.xlsx', header=None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

np.random.shuffle(df)

df = np.insert(df, 0, 1, axis=1)

test = df[int(0.6\*len(df)) : ]

X = df[ : int(0.6\*len(df))]

weights = np.zeros([3, np.size(df,1)-1])

accuracy = np.zeros(3)

count = [0,0,0]

# Training loop

for i in range(3):

data = df.copy()

X = data[ : int(0.6\*len(df))]

for row in X:

if row[-1] == i+1:

row[-1] = 1

else:

row[-1] = 0

weights[i] = train(X,5000)

confusion\_matrix = np.zeros((3,3))

# Testing loop

for row in test:

# Count number of instances per class

count[int(row[-1])-1] += 1

h = np.zeros(3)

for i in range(3):

h\_ = sigmoid(np.dot(weights[i], row[:-1]))

h[i] = h\_

l = np.argmax(h) + 1

confusion\_matrix[int(row[-1])-1, l-1] += 1

c1 = confusion\_matrix[0,0]/np.sum(confusion\_matrix[0])

c2 = confusion\_matrix[1,1]/np.sum(confusion\_matrix[1])

c3 = confusion\_matrix[2,2]/np.sum(confusion\_matrix[2])

total\_accuracy = (confusion\_matrix[0,0] + confusion\_matrix[1,1] + confusion\_matrix[2,2])/np.sum(confusion\_matrix)

print("Total accuracy: ", total\_accuracy)

print("Class 1 accuracy: ", c1)

print("Class 2 accuracy: ", c2)

print("Class 3 accuracy: ", c3)

print(confusion\_matrix)

## 8. (b) Multiclass Logistic Regression (One vs One)

|  |  |
| --- | --- |
| Total Accuracy | 0.9666666666666667 |
| Class 1 Accuracy | 1.0 |
| Class 2 Accuracy | 0.9523809523809523 |
| Class 3 Accuracy | 0.9545454545454546 |

## Multiclass logistic regression (One vs One)

import numpy as np

import pandas as pd

from math import exp,log

import random

# Define activation function

def sigmoid(x):

return 1/(1+np.exp(-x))

# Calculate gradients

def getGradients(x, weights):

delta = np.zeros(len(x)-1, dtype=float)

h = np.dot(weights, x[:-1])

h = sigmoid(h)

delta = [(h-x[-1])\*x[i] for i in range(len(x)-1)]

return delta

# Training function

def train(data, num\_iter):

'''Returns weights'''

weights = np.zeros(np.size(data,1)-1, dtype=float)

for i in range(num\_iter):

x = data[random.randint(0,len(data)-1)]

delta = getGradients(x, weights)

weights = [weights[i] - lr\*delta[i] for i in range(len(x)-1)]

return weights

lr = 0.05

df = pd.read\_excel('data4.xlsx', header=None)

df.iloc[:,0:7]=(df.iloc[:,0:7]-(df.iloc[:,0:7]).mean())/(df.iloc[:,0:7]).std()

df = df.to\_numpy()

np.random.shuffle(df)

df = np.insert(df, 0, 1, axis=1)

# Train test split

X, test = df[:int(0.6\*len(df))], df[int(0.6\*len(df)):]

# Preparing training data

train1, train2, train3 = [], [], []

for row in X:

if row[-1] == 1:

train1.append(row)

train2.append(row)

elif row[-1] == 2:

train1.append(row)

train3.append(row)

elif row[-1] == 3:

train2.append(row)

train3.append(row)

train1, train2, train3 = np.array(train1), np.array(train2), np.array(train3)

for row in train1:

row[-1] = 0 if row[-1] == 1 else 1

for row in train2:

row[-1] = 0 if row[-1] == 1 else 1

for row in train3:

row[-1] = 0 if row[-1] == 2 else 1

# Initializing weights

weights = np.zeros([3, np.size(df,1)-1])

# Training weights

weights[0], weights[1], weights[2] = train(train1, 4000), train(train2, 4000), train(train3, 4000)

confusion\_matrix = np.zeros((3,3))

# Training Loop

for row in test:

predicted = 0

h = []

for i in range(3):

h\_ = np.dot(row[:-1], weights[i])

h\_ = sigmoid(h\_)

h.append(round(h\_))

if h[0] == 0 and h[1] == 0: predicted = 1

if h[0] == 1 and h[2] == 0: predicted = 2

if h[1] == 1 and h[2] == 1: predicted = 3

confusion\_matrix[int(row[-1])-1, predicted-1] += 1

c1 = confusion\_matrix[0,0]/np.sum(confusion\_matrix[0])

c2 = confusion\_matrix[1,1]/np.sum(confusion\_matrix[1])

c3 = confusion\_matrix[2,2]/np.sum(confusion\_matrix[2])

total\_accuracy = (confusion\_matrix[0,0] + confusion\_matrix[1,1] + confusion\_matrix[2,2])/np.sum(confusion\_matrix)

print("Total accuracy: ", total\_accuracy)

print("Class 1 accuracy: ", c1)

print("Class 2 accuracy: ", c2)

print("Class 3 accuracy: ", c3)

print(confusion\_matrix)

## 

## 

## 

## 9. (a) 5-Fold Cross Validation Multi Class Regression (One vs All)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Accuracies | Fold 1 | Fold 2 | Fold 3 | Fold 4 | Fold 5 |
| Total Accuracy | 1.0 | 0.9777777777777779 | 0.8993265993265993 | 0.8717948717948718 | 0.875 |
| Class 1 Accuracy | 1.0 | 1.0 | 0.8888888888888888 | 1.0 | 1.0 |
| Class 2 Accuracy | 1.0 | 1.0 | 0.9 | 0.6153846153846154 | 1.0 |
| Class 3 Accuracy | 1.0 | 0.9333333333333333 | 0.9090909090909091 | 1.0 | 0.625 |

## K-fold cross validation (One vs All)

import numpy as np

import pandas as pd

from math import exp,log

import random

lr = 0.01

# Define activation function

def sigmoid(x):

return 1/(1+np.exp(-x))

# Function to calculate gradients for logistic regression

def getGradients(x, weights):

delta = np.zeros(len(x)-1, dtype=float)

h = np.dot(weights, x[:-1])

h = sigmoid(h)

delta = [(h-x[-1])\*x[i] for i in range(len(x)-1)]

return delta

# Training function

def train(data, num\_iter):

weights = np.zeros(np.size(data,1)-1, dtype=float)

for i in range(num\_iter):

x = data[random.randint(0,len(data)-1)]

delta = getGradients(x, weights)

weights = [weights[i] - lr\*delta[i] for i in range(len(x)-1)]

return weights

# Read, normalize and shuffle dataset

df = pd.read\_excel('data4.xlsx', header=None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

np.random.shuffle(df)

df = np.insert(df, 0, 1, axis=1)

# k-fold loop

for k in range(5):

data = df.copy()

test = data[k\*int(len(df)/5) : (k+1)\*int(len(df)/5)]

X = np.delete(data, np.s\_[k\*int(len(df)/5):(k+1)\*int(len(df)/5)], 0)

weights = np.zeros([3, np.size(df,1)-1])

accuracy = np.zeros(3)

count = [0,0,0]

# Training loop

for i in range(3):

data = df.copy()

X = np.delete(data, np.s\_[k\*int(len(df)/5):(k+1)\*int(len(df)/5)], 0)

for row in X:

if row[-1] == i+1:

row[-1] = 1

else:

row[-1] = 0

weights[i] = train(X,5000)

# Testing loop

for row in test:

# Count number of instances per class

count[int(row[-1])-1] += 1

h = np.zeros(3)

for i in range(3):

h\_ = np.dot(weights[i], row[:-1])

h\_ = sigmoid(h\_)

h[i] = h\_

predicted = np.argmax(h) + 1

if(predicted == row[-1]):

accuracy[int(row[-1])-1] += 1

accuracy = np.divide(accuracy, count)

print("Fold {}".format(k))

for i in range(3):

print("Class {} Accuracy: {}".format(i+1,accuracy[i]))

print("Total accuracy {}".format(np.mean(accuracy)))

## 9. (b) 5-Fold Cross Validation Multi Class Regression (One vs One)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Accuracies | Fold 1 | Fold 2 | Fold 3 | Fold 4 | Fold 5 |
| Total Accuracy | 0.9333333333333332 | 0.8727272727272727 | 1.0 | 0.9188034188034188 | 0.9166666666666666 |
| Class 1 Accuracy | 1.0 | 0.8 | 1.0 | 0.8333333333333334 | 1.0 |
| Class 2 Accuracy | 0.9 | 0.8181818181818182 | 1.0 | 0.9230769230769231 | 1.0 |
| Class 3 Accuracy | 0.9 | 1.0 | 1.0 | 0.9188034188034188 | 0.9166666666666666 |

## K-fold cross validation (One vs One)

import numpy as np

import pandas as pd

from math import exp,log

import random

lr = 0.01

def most\_frequent(List):

return max(set(List), key = List.count)

# Define activation function

def sigmoid(x):

return 1/(1+np.exp(-x))

# Calculate gradients

def getGradients(x, weights):

delta = np.zeros(len(x)-1, dtype=float)

h = np.dot(weights, x[:-1])

h = sigmoid(h)

delta = [(h-x[-1])\*x[i] for i in range(len(x)-1)]

return delta

# Training function

def train(data, num\_iter):

'''Returns weights'''

weights = np.zeros(np.size(data,1)-1, dtype=float)

for i in range(num\_iter):

x = data[random.randint(0,len(data)-1)]

delta = getGradients(x, weights)

weights = [weights[i] - lr\*delta[i] for i in range(len(x)-1)]

return weights

# Read, normalize and shuffle dataset

df = pd.read\_excel('data4.xlsx', header=None)

df.iloc[:,:-1]=(df.iloc[:,:-1]-(df.iloc[:,:-1]).mean())/(df.iloc[:,:-1]).std()

df = df.to\_numpy()

np.random.shuffle(df)

df = np.insert(df, 0, 1, axis=1)

overall\_accuracy = 0

# k-fold loop

for k in range(5):

fold\_accuracy = [0.0,0.0,0.0]

count = [0,0,0]

data = df.copy()

test = data[k\*int(len(df)/5) : (k+1)\*int(len(df)/5)]

X = np.delete(data, np.s\_[k\*int(len(df)/5):(k+1)\*int(len(df)/5)], 0)

# Initializing weights

weights = np.zeros([3, np.size(df,1)-1])

# Preparing training data

train1, train2, train3 = [], [], []

for row in X:

if row[-1] == 1:

train1.append(row)

train2.append(row)

elif row[-1] == 2:

train1.append(row)

train3.append(row)

elif row[-1] == 3:

train2.append(row)

train3.append(row)

train1, train2, train3 = np.array(train1), np.array(train2), np.array(train3)

for row in train1:

row[-1] = 0 if row[-1] == 1 else 1

for row in train2:

row[-1] = 0 if row[-1] == 1 else 1

for row in train3:

row[-1] = 0 if row[-1] == 2 else 1

# Training weights

weights[0], weights[1], weights[2] = train(train1, 3000), train(train2, 3000), train(train3, 3000)

for row in test:

count[int(row[-1])-1] += 1

h = []

for i in range(3):

h\_ = np.dot(row[:-1], weights[i])

h\_ = sigmoid(h\_)

h.append(round(h\_))

predicted = 0

if h[0] == 0 and h[1] == 0:

predicted = 1

if h[0] == 1 and h[2] == 0:

predicted = 2

if h[1] == 1 and h[2] == 1:

predicted = 3

if(predicted == row[-1]):

fold\_accuracy[int(row[-1]) - 1] += 1

fold\_accuracy = np.divide(fold\_accuracy, count)

print("Fold {}".format(k+1))

for i in range(3):

print("Class {} accuracy: {}".format(i+1, fold\_accuracy[i]))

print("Total accuracy {}".format(np.mean(fold\_accuracy)))

## 10. Likelihood Ratio Test Binary Classification

|  |  |
| --- | --- |
| Accuracy | 1.0 |
| Sensitivity | 1.0 |
| Specificity | 1.0 |

##Likelihood Ratio Test binary classification

import numpy as np

import pandas as pd

# Preparing training and testing datasets

df = pd.read\_excel( 'data3.xlsx', header=None)

df = df.to\_numpy()

class1, class2 = df[0:50, :], df[50:100, :]

X = np.zeros([60, 5], dtype=float)

test = np.zeros([40, 5], dtype=float)

X[0:30,:], X[30:60, :] = class1[0:30, :], class2[0:30, :]

test[0:20, :], test[20:40, :] = class1[0:20, :], class2[0:20, :]

# calculating means of feature vectors

means = [np.mean(X[0:30, :-1], axis=0), np.mean(X[30:60, :-1], axis=0)]

# calculation of covariance matrix for each class

cov = [np.cov((X[0:30, 0:4]).T), np.cov((X[30:60, 0:4]).T)]

# Calculate priors

priors = np.array([0.0, 0.0])

l = np.array(X[:, -1], dtype='int') - 1

priors[l] += 1

priors = priors/len(X)

# likelihood function

def likelihood(x, k):

m = np.dot((x-means[k-1]).T, np.dot(np.linalg.inv(cov[k-1]), x-means[k-1]))

p = np.exp(-0.5 \* m) / ((2 \* np.pi) \* np.linalg.det(cov[k-1]))

return p

ratio = priors[1]/priors[0]

confusion\_matrix = np.zeros([2,2])

for row in test:

l = [likelihood(row[:-1], 1), likelihood(row[:-1], 2)]

r = l[0]/l[1]

# Calculate predicted class

p = 1 if r>ratio else 2

# Update confusion matrix

confusion\_matrix[int(row[-1])-1, p-1] += 1

sensitivity = confusion\_matrix[0,0]/np.sum(confusion\_matrix[0])

specificity = confusion\_matrix[1,1]/np.sum(confusion\_matrix[1])

accuracy = (confusion\_matrix[0,0] + confusion\_matrix[1,1])/np.sum(confusion\_matrix)

print("Sensitivity: ", sensitivity)

print("Specificity: ", specificity)

print("Accuracy: ", accuracy)

print(confusion\_matrix)

## 

## 11. Maximum A Posteriori Decision Rule Multi Class Classification

|  |  |
| --- | --- |
| Total Accuracy | 0.9777777777777777 |
| Class 1 Accuracy | 1.0 |
| Class 2 Accuracy | 0.9333333333333333 |
| Class 3 Accuracy | 1.0 |

## Maximum A Posteriori decision rule multiclass classification

import pandas as pd

import numpy as np

#Preparing testing and training datasets, normalising data

df = pd.read\_excel('data4.xlsx', header=None)

df.iloc[:,0:7]=(df.iloc[:,0:7]-(df.iloc[:,0:7]).mean())/(df.iloc[:,0:7]).std()

df = df.to\_numpy()

class1, class2, class3 = df[:50], df[50:100], df[100:]

X = np.concatenate((class1[:int(0.7\*len(class1))], class2[:int(0.7\*len(class2))], class3[:int(0.7\*len(class3))]))

test = np.concatenate((class1[int(0.7\*len(class1)):], class2[int(0.7\*len(class2)):], class3[int(0.7\*len(class3)):]))

xlen = len(X)

class\_train = [(X[:int(xlen/3), :7]).T, (X[int(xlen/3):2\*int(xlen/3), :7]).T,

(X[2\*int(xlen/3):, :7]).T]

# Calculate priors

priors = np.array([0.0, 0.0, 0.0])

l = np.array(X[:, -1], dtype='int') - 1

priors[l] += 1

priors = priors/len(X)

# calculate covariance matrices

cov = [np.cov(class\_train[i]) for i in range(3)]

# calculate means for each class

means = [np.mean(class\_train[i], axis=1) for i in range(3)]

# Likelihood function

def likelihood(x, c):

den = np.sqrt(2 \* np.pi \* np.linalg.det(cov[c]))

pow = -0.5 \* np.dot(np.dot((x-means[c]).T, np.linalg.inv(cov[c])), (x-means[c]))

return float(np.exp(pow) / den)

# function to calculate posterior for a given feature vector and class

def posterior(x,c):

lh = likelihood(x,c)

prior = priors[c]

return lh \* prior

confusion\_matrix = np.zeros((3,3))

for row in test:

# Calculate predicted class

l = np.argmax([posterior(row[:7], i) for i in range(3)]) + 1

# Update confusion matrix

confusion\_matrix[int(row[-1])-1, l-1] += 1

c1 = confusion\_matrix[0,0] / np.sum(confusion\_matrix[0])

c2 = confusion\_matrix[1,1] / np.sum(confusion\_matrix[1])

c3 = confusion\_matrix[2,2] / np.sum(confusion\_matrix[2])

total\_accuracy = (confusion\_matrix[0,0] + confusion\_matrix[1,1] + confusion\_matrix[2,2]) / np.sum(confusion\_matrix)

print("Total accuracy: ", total\_accuracy)

print("Class 1 accuracy: ", c1)

print("Class 2 accuracy: ", c2)

print("Class 3 accuracy: ", c3)

## 12. Maximum Likelihood Decision Rule Multi Class Classification

|  |  |
| --- | --- |
| Total Accuracy | 0.9777777777777777 |
| Class 1 Accuracy | 1.0 |
| Class 2 Accuracy | 1.0 |
| Class 3 Accuracy | 0.9333333333333333 |

##Maximum Likelihood decision rule multiclass classification

import pandas as pd

import numpy as np

# Preparing training and testing datasets, normalising data

df = pd.read\_excel('data4.xlsx', header=None)

df.iloc[:,0:7]=(df.iloc[:,0:7]-(df.iloc[:,0:7]).mean())/(df.iloc[:,0:7]).std()

df = df.to\_numpy()

class1, class2, class3 = df[:50], df[50:100], df[100:]

np.random.shuffle(class1)

np.random.shuffle(class2)

np.random.shuffle(class3)

X = np.concatenate((class1[:int(0.7\*len(class1))], class2[:int(0.7\*len(class2))], class3[:int(0.7\*len(class3))]))

test = np.concatenate((class1[int(0.7\*len(class1)):], class2[int(0.7\*len(class2)):], class3[int(0.7\*len(class3)):]))

xlen = len(X)

class\_train = [(X[:int(xlen/3), :7]).T, (X[int(xlen/3):2\*int(xlen/3), :7]).T,

(X[2\*int(xlen/3):, :7]).T]

# calculate covariance matrices

cov = [np.cov(class\_train[0]), np.cov(class\_train[1]), np.cov(class\_train[2])]

# calculate means for each class

means = [np.mean(class\_train[0], axis=1), np.mean(class\_train[1], axis=1), np.mean(class\_train[2], axis=1)]

# Likelihood function

def likelihood(x, c):

den = np.sqrt(2 \* np.pi \* np.linalg.det(cov[c]))

pow = -0.5 \* np.dot(np.dot((x-means[c]).T,np.linalg.inv(cov[c])), (x-means[c]))

return float(np.exp(pow) / den)

# initialize confusion matrix

confusion\_matrix = np.zeros((3,3))

# testing loop

for row in test:

# calculate predicted class

l = np.argmax([likelihood(row[:7], i) for i in range(3)]) + 1

# update confusion matrix

confusion\_matrix[int(row[-1])-1, l-1] += 1

c1 = confusion\_matrix[0,0] / np.sum(confusion\_matrix[0])

c2 = confusion\_matrix[1,1] / np.sum(confusion\_matrix[1])

c3 = confusion\_matrix[2,2] / np.sum(confusion\_matrix[2])

total\_accuracy = (confusion\_matrix[0,0] + confusion\_matrix[1,1] + confusion\_matrix[2,2]) / np.sum(confusion\_matrix)

print("Total accuracy: ", total\_accuracy)

print("Class 1 accuracy: ", c1)

print("Class 2 accuracy: ", c2)

print("Class 3 accuracy: ", c3)

## 13. What I Learned From Solving This Assignment :

* Implementing various gradient descent and regularization methods for linear regression in Python using Numpy.
* Implementing K - means clustering based unsupervised classification.
* Classification of binary class and multi class data using supervised learning i.e logistic regression.
* Implementing hold-out cross validation technique to test models.
* Implementing K-fold cross validation technique.
* Likelihood ratio test, maximum a posteriori decision rule and maximum likelihood decision rule based Bayesian multi class classifiers and their implementation.