Classification Models

In [617]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import math
#from sklearn.utils import shuffle
```

This is a regression problem in which the goal is to use meteorological and other data to predict the burnt area of forest fires in the northeast region of Portugal.

Separating training data

In [618]:

```
data_c = pd.read_csv('Abalone_Age_2B.csv')
df_c = pd.DataFrame(data_c)
#np.random.seed(10)
#np.random.shuffle(df_c)
l = len(df_c)
print(df_c['Age'].max())
print(df_c['Age'].min())
age = df_c['Age']
age = np.array(age)
```

```
30
2
```

Making three Age Groups

Dividing groups into three classes.
AGE_Group :
2 to 12 is class '0'
13 to 19 is class '1'
20 to 30 is class '2'

In [619]:

```
c = []
i = 0
for i in range(l):
    if(age[i] >=2 and age[i]<=12):
        c.append(0)
    if(age[i] >=13 and age[i]<=19):
        c.append(1)
    if(age[i] >=20 and age[i]<=30):
        c.append(2)
c = np.array(c)
df_c["AGE_GROUP"] = c
#print(df_c)
```

In [620]:

```
df_c.columns = (df_c.columns.str.strip().str.upper()
              .str.replace(' ', ''))
#print(df_c)
```

Divding test data based on Age groups

In [621]:

```
n = 75
dfc_train = df_c.head(int(len(df_c)*(n/100)))
l_train = len(dfc_train)
grouped = dfc_train.groupby(df_c["AGE_GROUP"])
dfc_train0 = grouped.get_group(0)
dfc_train1 = grouped.get_group(1)
dfc_train2 = grouped.get_group(2)
```

Separating test data

In [622]:

```
n = 25
dfc_test = df_c.tail(int(len(df_c)*(n/100)))
age = dfc_test['AGE_GROUP']
age = np.array(age)
```

Useful functions

In [623]:

```
def classifier(p0, p1, p2):
    a = max(p0, p1, p2)
    if(a == p0):
        return 0
    elif(a == p1):
        return 1
    elif(a == p2):
        return 2
    else:
        return -1
```

Naive bayes Model (Univariate)

In univariate models there is only one variate.
In Naive Bayes Model we try to fit a Guassian.
We fit a guassian for every class and calculate respective (sigma, u) with maximim likelihood estimation.
Bayers Classifier :
p(C = Ck |x) = (p(x|C = Ck) * p(C = CK))/P(x)

In [624]:

```
def naive_byers(data):# function to calculate u and sigma
    u = 0
    sigma = 0
    for i in data:
        u = u + i
    u = u/len(data)
    for i in data:
            sigma += (i - u)*(i - u)
    sigma = sigma/len(data)
    sigma = math.sqrt(sigma)
    return u,sigma
```

In [625]:

```
#p(C = Ck) for all three classes
p_0 = len(dfc_train0)/l_train
p_1 = len(dfc_train1)/l_train
```

```python
p_2 = len(dfc_train2)/l_train

print(p_0)
print(p_1)
print(p_2)
```

```
0.7723499361430396
0.20306513409961685
0.02458492975734355
```

In [626]:

```python
#u, sigma for all three classes
p_x_0 = np.array(naive_byers(dfc_train0['WHOLEWEIGHT']))
p_x_1 = np.array(naive_byers(dfc_train1['WHOLEWEIGHT']))
p_x_2 = np.array(naive_byers(dfc_train2['WHOLEWEIGHT']))

print(p_x_0)
print(p_x_1)
print(p_x_2)
```

```
[0.73453721 0.4650852 ]
[1.11706682 0.46660689]
[1.23894805 0.42019658]
```

In [627]:

```python
#probability p(x) with u,sigma
def probability(u, sigma, x):
    a = ((-1)*(x-u)*(x-u))/(2*sigma*sigma)
    p = (1/(sigma*math.sqrt(2*math.pi)))*(math.exp(a))
    return p
```

We have u, sigma for all the classes.
We now use test data to predict age group.
For this we calculate p(C = Ck|x) for all the three classes and take maximum of all the three.

In [628]:

```python
age_predict_group =[]
for ind in dfc_test.index:
    p_0_x = p_0 * probability(p_x_0[0], p_x_0[1], dfc_test['WHOLEWEIGHT'][ind])
    p_1_x = p_1 * probability(p_x_1[0], p_x_1[1], dfc_test['WHOLEWEIGHT'][ind])
    p_2_x = p_2 * probability(p_x_2[0], p_x_2[1], dfc_test['WHOLEWEIGHT'][ind])
    age_predict_group.append(classifier(p_0_x, p_1_x, p_2_x))

print(age_predict_group)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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, 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, 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, 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, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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, 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, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 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, 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, 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, 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, 0, 0, 1, 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, 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, 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, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 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, 0, 1, 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, 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, 0, 0, 0, 1, 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, 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,
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, 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, 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, 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, 0, 0, 1, 1, 1, 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, 1, 0, 1, 0, 0, 1, 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, 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, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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, 0, 0, 1]
```

Accuracy for test data to predict the accuracy of the model

In [629]:

```python
i = 1
count = 0
for i in range(len(age_predict_group)):
    if(age_predict_group[i] == age[i]):
        count += 1
accuracy = (count/len(age_predict_group))*100
print(accuracy)
```

```
74.90421455938697
```

Our accuracy is 74.90421455938697.
Observation : Gives good results for large and diverse data

Multivariate Guassian

I tried few multivariate guassians and among them all Viscera weight, Whole weight and Height gave better results.
and also in logistic regression when taken these three variates better results were obtained.
So we can assume these are good variates to estimate and produces high Accuracy in Naive Bayes also.

Dividing training data based on groups and storing in matrices

In [630]:

```python
x_0 = dfc_train0.iloc[0:,0:10]
x_0['11'] = 1
x_0 = np.array(x_0)

y_0 = dfc_train0['AGE_GROUP']
y_0 = np.array(y_0)
```

In [631]:

```python
x_1 = dfc_train1.iloc[0:,0:10]
x_1['11'] = 1
x_1 = np.array(x_1)

y_1 = dfc_train1['AGE_GROUP']
y_1 = np.array(y_1)
```

In [632]:

```python
x_2 = dfc_train2.iloc[0:,0:10]
x_2['11'] = 1
x_2 = np.array(x_2)

y_2 = dfc_train2['AGE_GROUP']
```

```
y_2 = np.array(y_2)
```

Logistic Regression (Gradient descent)

In logistic Regression we try to select a curve which gives better results.
p(C = Ck |x, w) = softmax(W.T @ X)
To find weights we try to maximize probability with maximimum likelihood estimation using gradient descent.
Gradient descent : Θ(new) = Θ(old) - learingrate * d/dΘ(loss function)

In [633]:

```
def softmax(x, w):
    a = np.exp(x@w_random_0)/ (np.exp(x_0@w_random_0) + np.exp(x_0@w_random_1) + np.exp(
x_0@w_random_2))
```

In [634]:

```
learning_rate_logistic = 0.0000001
num_iters_logistic = 30
```

In [635]:

```
##To calculate weights
w_random_0 = np.zeros(11)
w_random_1 = np.zeros(11)
w_random_2 = np.zeros(11)
y_0 = np.ones(len(x_0))
y_1 = np.ones(len(x_1))
y_2 = np.ones(len(x_2))
for _ in range(num_iters_logistic):
    y_p_0 = x_0@w_random_0
    a0 = np.exp(x_0@w_random_0)/ (np.exp(x_0@w_random_0) + np.exp(x_0@w_random_1) + np.e
xp(x_0@w_random_2))
    dJ0 = x_0.T@(a0 - y_0)
    w_random_0 = w_random_0 - learning_rate_logistic*dJ0
    y_p_1 = x_1@w_random_1
    a1 = np.exp(x_1@w_random_1)/ (np.exp(x_1@w_random_0) + np.exp(x_1@w_random_1) + np.e
xp(x_1@w_random_2))
    dJ1 = x_1.T@(a1 - y_1)
    w_random_1 = w_random_1 - learning_rate_logistic*dJ1
    y_p_2 = x_2@w_random_2
    a2 = np.exp(x_2@w_random_2)/ (np.exp(x_2@w_random_0) + np.exp(x_2@w_random_1) + np.e
xp(x_2@w_random_2))
    dJ2 = x_2.T@(a2 - y_2)
    w_random_2 = w_random_2 - learning_rate_logistic*dJ2

print(w_random_0)
print(w_random_1)
print(w_random_2)
w_random = [w_random_0, w_random_1, w_random_2]
```

```
[0.00242963 0.00188144 0.00063493 0.00354732 0.00159932 0.00077897
 0.00098986 0.00131374 0.00187534 0.00164119 0.00483027]
[0.00074694 0.00058966 0.00020991 0.00142167 0.00057333 0.00030544
 0.00043357 0.00055227 0.00012607 0.00059431 0.00127265]
[9.33386065e-05 7.43385916e-05 2.69729331e-05 1.91113256e-04
 6.93358376e-05 3.86521029e-05 6.32899149e-05 8.01244453e-05
 6.00752376e-06 6.81096371e-05 1.54241606e-04]
```

Testing for test data

In [636]:

```
age_l = dfc_test['AGE_GROUP']
age_l = np.array(age_l)
x_test = dfc_test.iloc[0:,0:10]
x_test['11'] = 1
```

```
x_test = np.array(x_test)
```

In [637]:

```
def sum(weight, x):
    sum = 0
    for i in range(3):
        sum += np.exp(weight[i].T@x)
    return sum
```

In [638]:

```
agegroup_logistic = []
i = 0
for i in range(len(x_test)):
    p_0_xl = np.exp(w_random_0.T@x_test[i])/sum(w_random, x_test[i])
    p_1_xl = np.exp(w_random_1.T@x_test[i])/sum(w_random, x_test[i])
    p_2_xl = np.exp(w_random_1.T@x_test[i])/sum(w_random, x_test[i])
    agegroup_logistic .append(classifier(p_0_xl, p_1_xl, p_2_xl))
```

In [639]:

```
i = 0
count_l = 0
for i in range(len(agegroup_logistic)):
    if(agegroup_logistic[i] == age_l[i]):
        count_l += 1
accuracy_l = (count_l/len(agegroup_logistic))*100
print(accuracy_l)
```

```
76.34099616858238
```

Accuracy is 76.34099616858238 Observation : Accuracy for logistic Regression(gradient descent) is more than Naive Bayes Model(univariate).
We can say that model gives better results for multivariate

Logistic Regression (Newton Optimization)

In logistic Regression we try to select a curve which gives better results.
p(C = Ck |x, w) = softmax(W.T @ X)
To find weights we try to maximize probability with maximimum likelihood estimation using newton method.
Θ(new) = Θ(old) - H^(-1)@G
G is gradient.
G = X.T@(Y_pred - Y)
H is Hessian matrix
H = X.T @ S @ X
S = Y_pred(1 - Y_pred)

In [640]:

```
learning_rate_logistic = 0.01
num_iters_logistic = 10
correction = 0.00001
```

In [641]:

```
#function to calculate weights
def gradient_descent_logistic_newton(data_0, data_1, data_2, learning_rate, weight, num_
iters, correction):
    i, j, k = 0, 0, 0
    for i in range(num_iters):
        gra = [[0 for i in range(11)] for j in range(3)]
        gra = np.array(gra , dtype = float)
        s_0 = np.zeros(len(data_0))
        s_1 = np.zeros(len(data_1))
        s_2 = np.zeros(len(data_2))
```

```
        for j in range(len(data_0)):
            for k in range(11):
                a = np.exp((data_0[j]*weight[0]).sum())
                #print(a)
                b = np.exp((data_0[j]*weight[0]).sum()) + np.exp((data_0[j]*weight[1]).s
um()) + np.exp((data_0[j]*weight[2]).sum())
                gra[0][k] += (a/b - 1)*data_0[j][k]
                m = ((a/b) - (a/b)*(a/b))
            s_0[j] = m
        s0 = np.diag(s_0)
        h_0 = data_0.T @ s0 @ data_0
        for j in range(len(data_1)):
            for k in range(11):
                a = np.exp((data_1[j]*weight[1]).sum())
                b = np.exp((data_1[j]*weight[0]).sum()) + np.exp((data_1[j]*weight[1]).s
um()) + np.exp((data_1[j]*weight[2]).sum())
                gra[1][k] += (a/b - 1)*data_1[j][k]
                m = ((a/b) - (a/b)*(a/b))
            s_1[j] = m
        s1 = np.diag(s_1)
        h_1 = data_1.T @ s1 @ data_1
        for j in range(len(data_2)):
            for k in range(11):
                a = np.exp((data_2[j]*weight[2]).sum())
                b = np.exp((data_2[j]*weight[0]).sum()) + np.exp((data_2[j]*weight[1]).s
um()) + np.exp((data_2[j]*weight[2]).sum())
                gra[2][k] += (a/b - 1)*data_2[j][k]
                m = ((a/b) - (a/b)*(a/b))
            s_2[j] = m
        s2 = np.diag(s_2)
        h_2 = data_2.T @ s2 @ data_2
        weight[0] = weight[0] - np.linalg.inv(h_0 - correction * np.identity(11))@gra[0]
        weight[1] = weight[1] - np.linalg.inv(h_1 - correction * np.identity(11))@gra[1]
        weight[2] = weight[2] - np.linalg.inv(h_2 - correction * np.identity(11))@gra[2]
    return weight
```

In [642]:

```
weight_newton = [[0 for i in range(11)] for j in range(3)]
weight_newton = np.array(weight_newton, dtype = np.float128)
weight_newton = gradient_descent_logistic_newton(x_0, x_1, x_2, learning_rate_logistic, w
eight_newton, num_iters_logistic, correction)
print(weight_newton)
```

```
[[-9.14582321e-02 -1.31057767e-02 -5.07227149e-02  7.78157965e-03
   1.67328498e-02  5.26116704e-03  1.78978778e-03  7.51324614e+00
   7.51121086e+00  7.51364238e+00  2.25380948e+01]
 [-7.15341241e-02 -7.20992917e-03 -3.76225059e-02  6.63163460e-03
   1.41017907e-02  5.86397831e-04 -1.97832385e-03  7.51021975e+00
   7.50830874e+00  7.51076776e+00  2.25292956e+01]
 [ 1.52447363e-01  1.79595503e-02  8.47607885e-02 -1.35718713e-02
  -2.90377752e-02 -5.04493293e-03  9.59243222e-04  7.47804239e+00
   7.48177621e+00  7.47713670e+00  2.24369553e+01]]
```

In [643]:

```
agegroup_logistic_newton = []
i = 0
for i in range(len(x_test)):
    p_0_xln = np.exp(weight_newton[0].T@x_test[i])/sum(weight_newton, x_test[i])
    #print(p_0_xl)
    p_1_xln = np.exp(weight_newton[1].T@x_test[i])/sum(weight_newton, x_test[i])
    #print(p_1_xl)
    p_2_xln = np.exp(weight_newton[2].T@x_test[i])/sum(weight_newton, x_test[i])
    #print(p_2_xl)
    agegroup_logistic_newton .append(classifier(p_0_xln, p_1_xln, p_2_xln))
print(agegroup_logistic_newton)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 2, 2, 2, 2, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 2,
0, 2, 0, 0, 1, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0,
1, 2, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 2, 2, 0, 0, 0,
1, 2, 0, 0, 0, 0, 2, 1, 0, 0, 0, 2, 2, 0, 2, 2, 2, 0, 1, 1, 2, 0, 2, 0, 2, 2, 0, 0, 0, 0,
```

```
0, 0, 0, 2, 0, 0, 1, 0, 2, 0, 0, 2, 0, 0, 0, 0, 1, 2, 1, 0, 0, 0, 0, 0, 0, 1, 2, 2, 1, 1,
2, 1, 2, 2, 1, 1, 1, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 0, 0, 0, 0, 2, 0, 0,
0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 0, 0, 0,
0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 2, 2, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 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, 2, 0, 1, 2, 1,
0, 1, 2, 0, 2, 0, 1, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 1, 0, 1,
1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 2, 2, 1, 1, 1, 0, 2, 0, 2, 2, 1, 2, 1, 2, 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, 0, 0, 0, 0, 0, 0, 0,
1, 2, 0, 2, 2, 0, 0, 0, 2, 1, 2, 1, 2, 1, 2, 2, 2, 1, 2, 0, 2, 1, 2, 2, 2, 1, 2, 2, 1, 0,
1, 2, 1, 2, 1, 1, 1, 0, 1, 1, 2, 1, 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 2,
0, 2, 0, 1, 2, 1, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 2, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 0, 0, 1, 2, 1, 2, 1,
1, 1, 2, 1, 1, 1, 2, 2, 1, 0, 2, 2, 2, 0, 2, 2, 2, 2, 1, 1, 2, 2, 2, 1, 2, 2, 2, 2, 2, 0,
2, 2, 2, 1, 1, 2, 0, 2, 2, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 2, 1, 1, 2, 2, 1, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 1, 2,
2, 2, 2, 0, 1, 1, 0, 2, 2, 0, 2, 2, 2, 0, 1, 2, 1, 1, 1, 2, 2, 0, 1, 1, 2, 2, 2, 2, 2, 1,
1, 2, 2, 2, 2, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 2, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 1, 2, 2, 2, 1, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 1, 0, 0, 0, 2, 0, 0, 1, 1,
0, 0, 0, 0, 2, 0, 1, 1, 2, 0, 2, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 0, 2,
0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 2, 2, 2, 1, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0,
2, 0, 0, 0, 2, 2, 2, 0, 0, 2, 0, 0, 0, 2, 0, 2, 2, 2, 2, 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, 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, 2, 1, 0, 2, 1, 0, 2, 0, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 1, 0, 1, 1, 1, 2, 2, 2, 1, 1, 1, 2, 1, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 2, 1, 2, 2, 2, 0, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1, 2, 0, 2, 2, 2, 1, 0, 1, 0, 2, 2, 1,
1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 1, 1, 2, 1, 0, 0, 0, 0, 2, 2, 1, 0, 1, 0, 0,
2, 1, 2, 0, 1, 2, 1, 1, 2, 1, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 0, 0, 0, 0,
0, 0, 2, 1, 0, 0, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 0]
```

In [644]:

```
i = 0
count_ln = 0
for i in range(len(agegroup_logistic_newton)):
    if(agegroup_logistic_newton[i] == age_l[i]):
        count_ln += 1
accuracy_ln = (count_ln/len(agegroup_logistic_newton))*100
print(accuracy_ln)
```

```
48.46743295019157
```

Accuracy is 48.46743295019157
Which is less than both naive Bayes and logistic Regression(gradient Descent)

Conclusion

We got better results i.e Accuracy for Logistic Regression (Gradient descent) among the three methods we used.
THerefore, Logistic Regression (Gradient descent) is better model for the data