```
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
plt.rcParams['figure.figsize'] = [12, 5]
```

```
df = pd.read_csv("/content/train.csv")
```

```
df.head()
```

	Battery_Power	Clock_Speed	FC	Int_Memory	Mobile_D	Mobile_W	Cores	PC	Pixel_H	Pix
0	842	2.2	1	7	0.6	188	2	2	20	
1	1021	0.5	0	53	0.7	136	3	6	905	
2	563	0.5	2	41	0.9	145	5	6	1263	
3	615	2.5	0	10	0.8	131	6	9	1216	
4	1821	1.2	13	44	0.6	141	2	14	1208	

5 rows × 21 columns

```
df.shape
```

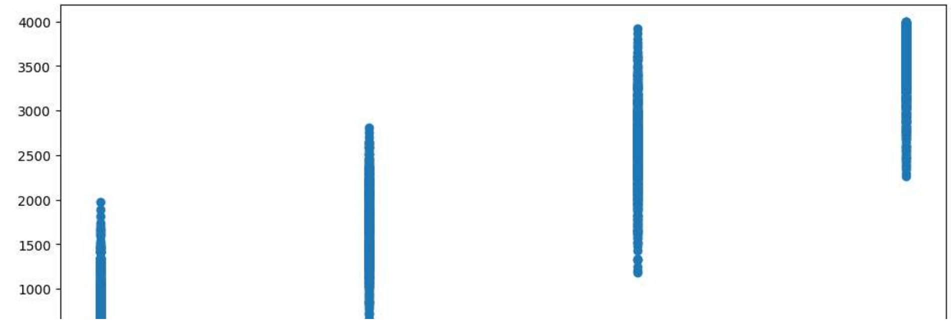
```
(2000, 21)
```

```
df.isna().sum()
```

```
Battery_Power    0
Clock_Speed      0
FC               0
Int_Memory       0
Mobile_D         0
Mobile_W         0
Cores            0
PC               0
Pixel_H          0
Pixel_W          0
Ram              0
Screen_H         0
Screen_W         0
Talk_Time        0
Four_G           0
Three_G          0
Touch_Screen     0
Dual_SIM         0
Bluetooth        0
WiFi             0
Price_Range      0
dtype: int64
```

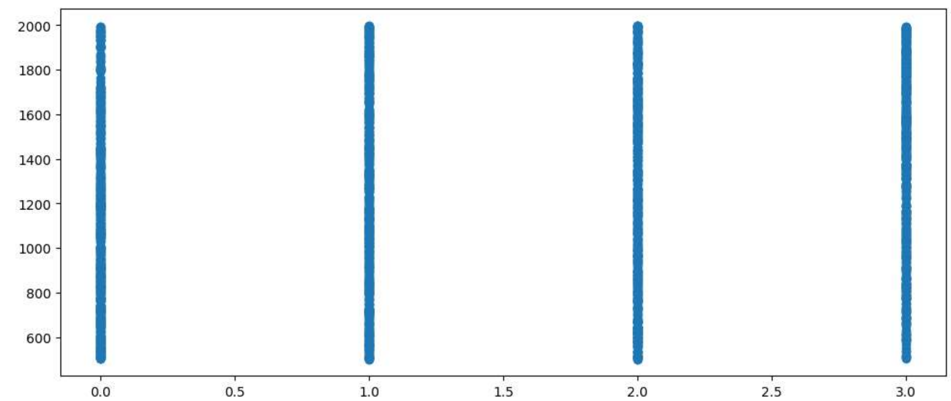
```
plt.scatter(x=df['Price_Range'],y=df['Ram'])
```

```
<matplotlib.collections.PathCollection at 0x79f3fd16e590>
```



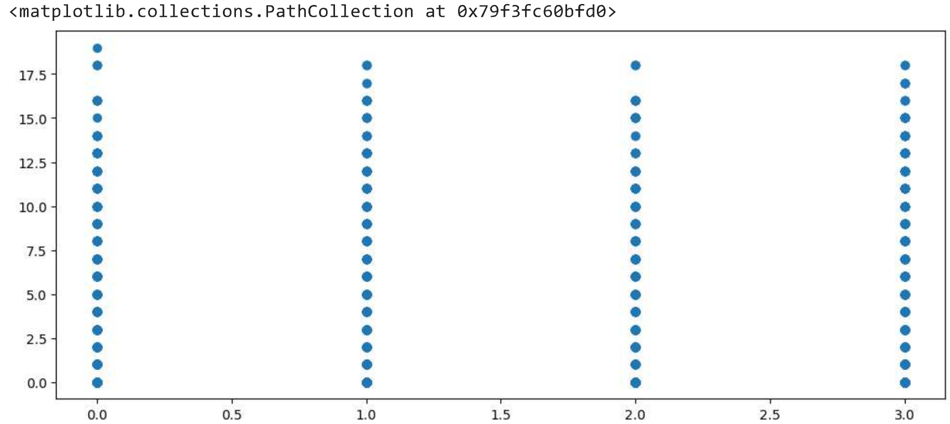
```
plt.scatter(x=df['Price_Range'],y=df['Battery_Power'])
```

```
<matplotlib.collections.PathCollection at 0x79f3fd0f9930>
```



```
plt.scatter(x=df['Price_Range'],y=df['FC'])
```

```
<matplotlib.collections.PathCollection at 0x79f3fc60bfd0>
```

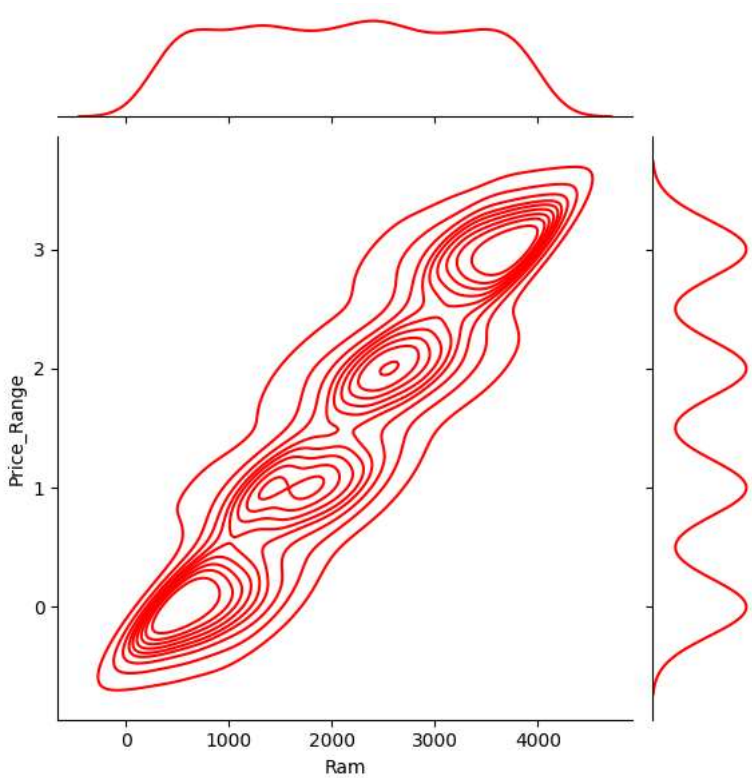


```
pip install seaborn
```

```
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages (0.12.2)
Requirement already satisfied: numpy!=1.24.0,>=1.17 in /usr/local/lib/python3.10/dist-packages (from seaborn) (1.22.4)
Requirement already satisfied: pandas>=0.25 in /usr/local/lib/python3.10/dist-packages (from seaborn) (1.5.3)
Requirement already satisfied: matplotlib!=3.6.1,>=3.1 in /usr/local/lib/python3.10/dist-packages (from seaborn) (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (0.1
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn
```

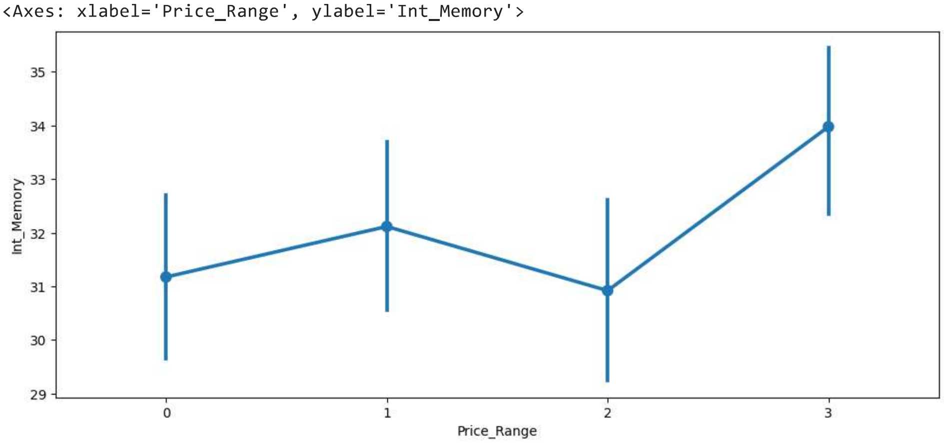
```
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (8
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seab
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25->seaborn) (2022.7.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib!=3.6.1,>
```

```
sns.jointplot(x='Ram',y='Price_Range',data=df,color='red',kind='kde');
```

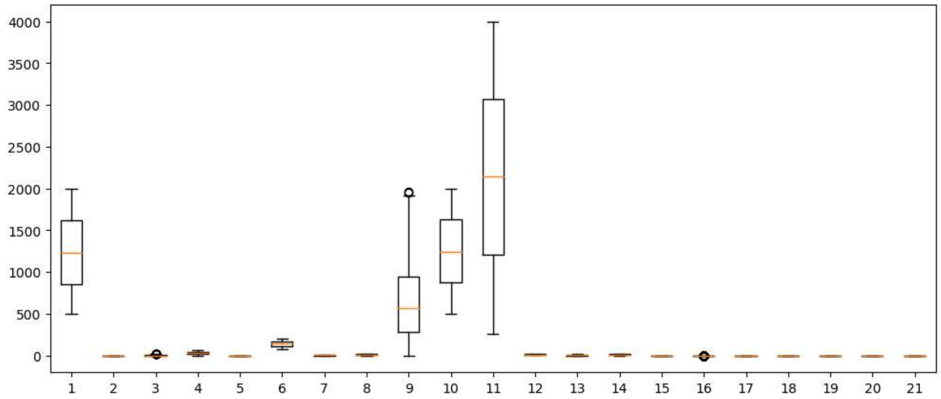



```
sns.pointplot(y="Int_Memory", x="Price_Range", data=df)
```

```
<Axes: xlabel='Price_Range', ylabel='Int_Memory'>
```




```
plt.boxplot(df)
plt.show()
```

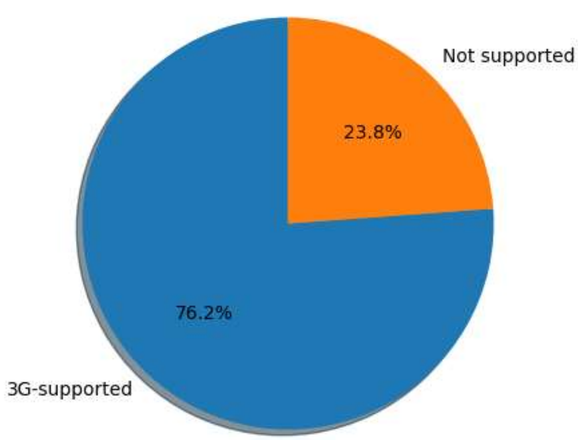


% of 3g users which supprots the phone

```
labels = ["3G-supported",'Not supported']
values=df['Three_G'].value_counts().values
```

```
def standerdize(x):
    return (x - x.mean())/x.std()
```

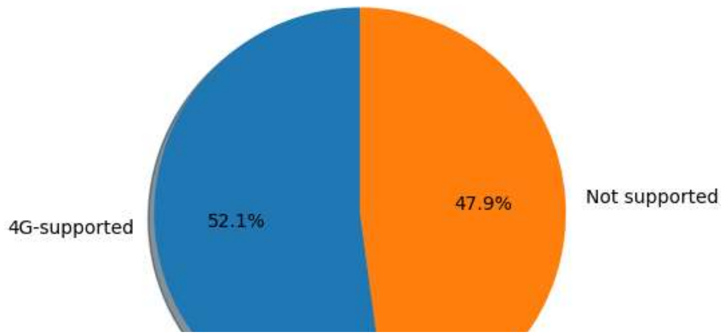
```
fig1, ax1 = plt.subplots()
ax1.pie(values, labels=labels, autopct='%1.1f%%',shadow=True,startangle=90)
plt.show()
```



% of Phones which support 4G

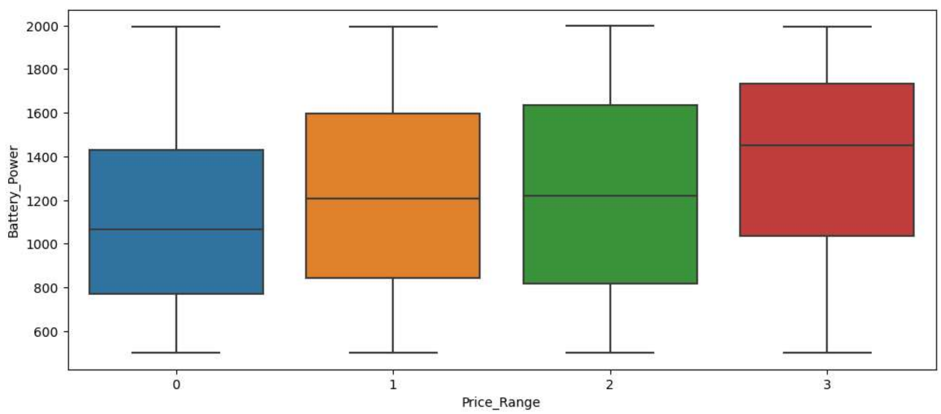
```
labels4g = ["4G-supported",'Not supported']
values4g = df['Four_G'].value_counts().values
```

```
fig1, ax1 = plt.subplots()
ax1.pie(values4g, labels=labels4g, autopct='%1.1f%%',shadow=True,startangle=90)
plt.show()
```




```
# battery power vs battery range
sns.boxplot(x="Price_Range", y="Battery_Power", data=df)
```

<Axes: xlabel='Price_Range', ylabel='Battery_Power'>




```
# mobile price vs mobile wegiht
sns.jointplot(x='Mobile_W',y='Price_Range',kind='kde',data=df,);
```

```
y=df['Price_Range']
```

```
y
```

```
0       1
1       2
2       2
3       2
4       1
       ..
1995    0
1996    2
1997    3
1998    0
1999    3
Name: Price_Range, Length: 2000, dtype: int64
```

```
x=df.drop('Price_Range',axis=1)
```

```
x
```

	Battery_Power	Clock_Speed	FC	Int_Memory	Mobile_D	Mobile_W	Cores	PC	Pixel_H
0	842	2.2	1	7	0.6	188	2	2	20
1	1021	0.5	0	53	0.7	136	3	6	905
2	563	0.5	2	41	0.9	145	5	6	1263
3	615	2.5	0	10	0.8	131	6	9	1216
4	1821	1.2	13	44	0.6	141	2	14	1208
...	...	...	...	...	...	...	...	...	...
1995	794	0.5	0	2	0.8	106	6	14	1222
1996	1965	2.6	0	39	0.2	187	4	3	915
1997	1911	0.9	1	36	0.7	108	8	3	868
1998	1512	0.9	4	46	0.1	145	5	5	336
1999	510	2.0	5	45	0.9	168	6	16	483

2000 rows × 20 columns

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

```
X_train,X_test,Y_train,Y_test=train_test_split(x,y,train_size=0.75,random_state=0)
```

```
X_train.shape
```

```
(1500, 20)
```

```
X_test.shape
```

```
(500, 20)
```

```
Y_train.shape
```

```
(1500,)
```

```
Y_test.shape
```

```
(500,)
```

```
standerdize_Dataframe = pd.DataFrame()
```

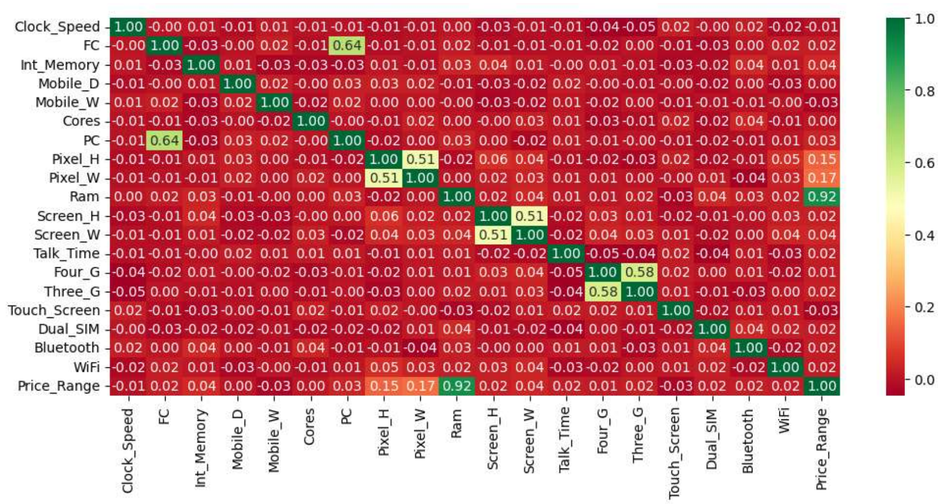
```
for c in df.columns[1:]:
    standerdize_Dataframe[c] = standerdize(df[c])
```

```
standerdize_Dataframe["Price_Range"] = df["Price_Range"]
```

```
standerdize_Dataframe.head()
```

	Clock_Speed	FC	Int_Memory	Mobile_D	Mobile_W	Cores	PC	Pixel_H	
0	0.830572	-0.762304	-1.380298	0.340654	1.348911	-1.101696	-1.305424	-1.408596	-1
1	-1.252751	-0.992642	1.154735	0.687376	-0.120029	-0.664602	-0.645827	0.585631	1
2	-1.252751	-0.531966	0.493422	1.380820	0.134210	0.209587	-0.645827	1.392336	1
3	1.198217	-0.992642	-1.214970	1.034098	-0.261274	0.646681	-0.151130	1.286428	1
4	-0.394912	2.001753	0.658751	0.340654	0.021215	-1.101696	0.673365	1.268401	-(

```
_ = sns.heatmap(standerdize_Dataframe.corr(),cmap='RdYlGn',fmt = ".2f", annot=True)
```




```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components = 5)
```

```
pca_model = pca.fit(standerdize_Dataframe[standerdize_Dataframe.columns[:-1]])
```

```
X = pca_model.transform(standerdize_Dataframe[standerdize_Dataframe.columns[:-1]])
```

```
from sklearn.model_selection import train_test_split

train_x, test_x, train_y, test_y = train_test_split(X,df.Price_Range, test_size=0.2, random_state = 13223, shuffle=True)
```

```
model_RMSE = {}
```

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

lr = LinearRegression()
lr_model = lr.fit(train_x,train_y)
```

```
pred = lr_model.predict(test_x)
model_RMSE["Linear Regression"] = mean_squared_error(pred,test_y, squared=False)
print("Root Mean Square Error: {0:.2f}".format(model_RMSE["Linear Regression"]))
```

    Root Mean Square Error: 1.02

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(n_estimators=6, random_state = 34)
rf_model = rf.fit(train_x, train_y)
pred = rf.predict(test_x)
model_RMSE["Random Forest"] = mean_squared_error(pred,test_y, squared=False)
print("Root Mean Square Error: {0:.2f}".format(model_RMSE["Random Forest"]))
```

    Root Mean Square Error: 1.15

```
from sklearn.neighbors import KNeighborsRegressor

knn = KNeighborsRegressor(n_neighbors=7)
knn_model = knn.fit(train_x, train_y)
pred = knn_model.predict(test_x)
model_RMSE["K Nearest"] = mean_squared_error(pred,test_y, squared=False)
print("Root Mean Square Error: {0:.2f}".format(model_RMSE["K Nearest"]))
```

    Root Mean Square Error: 1.10

```
from sklearn.ensemble import GradientBoostingRegressor
gb = GradientBoostingRegressor(learning_rate=0.3,random_state = 124124)
gb_model = gb.fit(train_x, train_y)
pred = gb_model.predict(test_x)
model_RMSE["Gradient Boosting"] = mean_squared_error(pred,test_y, squared=False)
print("Root Mean Square Error: {0:.2f}".format(model_RMSE["Gradient Boosting"]))
```

    Root Mean Square Error: 1.05

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import metrics
from tensorflow.keras.optimizers import RMSprop
```

```
l2 =  keras.regularizers.l2(0.001)

model = keras.Sequential([
    keras.layers.Dense(16, activation='relu', input_shape=(5,), kernel_regularizer = l2),
    keras.layers.Dense(8, activation='relu', kernel_regularizer = l2),
    keras.layers.Dense(8, activation='relu', kernel_regularizer = l2),
    keras.layers.Dense(1)
])
```

```
opt = RMSprop(learning_rate = 0.01, momentum=0.2)

model.compile(optimizer=opt, loss='mean_squared_error', metrics=[tf.keras.metrics.RootMeanSquaredError()])
```

```
model.fit(train_x, train_y, epochs=500, batch_size=16)
```

```
pred = model.predict(test_x)
model_RMSE["Neural Network"] = mean_squared_error(pred,test_y, squared=False)
print("Root Mean Square Error: {0:.2f}".format(model_RMSE["Neural Network"]))
```

```
100/100 [==============================] - 0s 2ms/step - loss: 0.9985 - root_mean_squared_error: 0.9913
Epoch 480/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9961 - root_mean_squared_error: 0.9900
Epoch 481/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0047 - root_mean_squared_error: 0.9942
Epoch 482/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0019 - root_mean_squared_error: 0.9927
Epoch 483/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0022 - root_mean_squared_error: 0.9930
Epoch 484/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0041 - root_mean_squared_error: 0.9938
Epoch 485/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9999 - root_mean_squared_error: 0.9918
Epoch 486/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0041 - root_mean_squared_error: 0.9939
Epoch 487/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9967 - root_mean_squared_error: 0.9903
Epoch 488/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9978 - root_mean_squared_error: 0.9910
Epoch 489/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9995 - root_mean_squared_error: 0.9916
Epoch 490/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9975 - root_mean_squared_error: 0.9907
Epoch 491/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9994 - root_mean_squared_error: 0.9916
Epoch 492/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9966 - root_mean_squared_error: 0.9902
Epoch 493/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9987 - root_mean_squared_error: 0.9913
Epoch 494/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0021 - root_mean_squared_error: 0.9931
Epoch 495/500
100/100 [==============================] - 0s 3ms/step - loss: 0.9993 - root_mean_squared_error: 0.9917
Epoch 496/500
100/100 [==============================] - 0s 3ms/step - loss: 1.0008 - root_mean_squared_error: 0.9925
Epoch 497/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0011 - root_mean_squared_error: 0.9925
Epoch 498/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0020 - root_mean_squared_error: 0.9929
Epoch 499/500
100/100 [==============================] - 0s 2ms/step - loss: 0.9983 - root_mean_squared_error: 0.9911
Epoch 500/500
100/100 [==============================] - 0s 2ms/step - loss: 1.0051 - root_mean_squared_error: 0.9944
13/13 [==============================] - 0s 2ms/step
Root Mean Square Error: 1.03
```

```
def MeanRegressor(models, weights, X):
    m_p = [0]*len(X)
    inverse_weights = np.ones(len(weights))/weights
    newWeights = inverse_weights / inverse_weights.sum()


    for m,w in zip(models, newWeights):
        m_p += m.predict(X).reshape(len(X))*w

    return m_p/1
```

```
pred = MeanRegressor([lr_model,rf_model, knn_model, gb_model, model],list(model_RMSE.values()),test_x)
model_RMSE["Mean Model"] = mean_squared_error(pred,test_y, squared=False)
print("Root Mean Square Error: {0:.2f}".format(model_RMSE["Mean Model"]))
```

```
13/13 [==============================] - 0s 2ms/step
Root Mean Square Error: 1.03
```

```
sns.barplot(x = list(model_RMSE.keys()), y = list(model_RMSE.values()))
```

```
<Axes: >
```


```
def pred_pipelin(data):
    for key, value in custom_data.items():
        data[key] = (value - df[key].mean()) / df[key].std()

    decom = pca_model.transform(pd.DataFrame(custom_data, index=[0]))

    return MeanRegressor([lr_model,rf_model, knn_model, gb_model, model],list(model_RMSE.values()),decom)[0]
```

```
custom_data = {
 #sales for smart phone
"Price_Range": 2,
"Battery_Power": 150.0,
"Clock_Speed": 2.5,
"FC": 15,
"Int_Memory": 64,
"Mobile_D": 122,
"Mobile_W": 153,
"Cores": 4,
"PC": 10,
"Pixel_H": 895,
"Pixel_W": 1255,
"Screen_W": 15,
"Screen_H": 5,

"Talk_Time": 15,
"Three_G": 1,
"Four_G": 0,
"Touch_Screen": 1,
"Dual_SIM": 1,
"Bluetooth": 1,
"Ram": 1452,
"WiFi": 0,





}
```

MLR

```
X1 = df['Price_Range'] #independent variable
X2 = df['Ram'] #independent variable
Y = df['Battery_Power'] #dependent variable
```

```
x1_mean = np.mean(X1)
x2_mean = np.mean(X2)
y_mean = np.mean(Y)
n = X1.count()
Ex1_2 = sum(X1**2) - (sum(X1)**2/n)
Ex2_2 = sum(X2**2) - (sum(X2)**2/n)
Ex1y = sum(X1*Y) - (sum(X1)*sum(Y)/n)
Ex2y = sum(X2*Y) - (sum(X2)*sum(Y)/n)
Ex1x2 = sum(X1*X2) - (sum(X1)*sum(X2)/n)


b1 = ((Ex2_2 * Ex1y) - (Ex1x2 * Ex2y))/((Ex1_2 * Ex2_2) - (Ex1x2**2))
b2 = ((Ex1_2 * Ex2y) - (Ex1x2 * Ex1y))/((Ex1_2 * Ex2_2) - (Ex1x2**2))
b0 = y_mean - (b1*x1_mean) - (b2*x2_mean)

print(b0)
print(b1)
print(b2)
```

```
1491.9283934589366
497.4313368528828
-0.4705539881067768
```

```
# 5. Find the Ypred
y_pred = b0 + (b1 * X1) + (b2 * X2)
y_pred
```

```
0        789.917615
1       1248.763524
2       1261.939036
3       1183.827074
4       1325.408053
            ...
1995    1177.598329
1996    1530.625363
1997    1545.738862
1998    1083.016978
1999    1140.121325
Length: 2000, dtype: float64
```

```
# 6. Calculate the SSE (sum of squared error)and RMSE (Root Mean Square Error) value
SSE = sum((Y - y_pred)**2)
print('Sum of squared error:', SSE)
```

```
Sum of squared error: 287610112.84334654
```

```
RMSE = np.sqrt(sum((Y - y_pred)**2)/len(X1))
RMSE
```

```
379.2163715106104
```

```
# 7. Calculate the coefficient of determination (r2) r-square
SSR = sum((Y - y_pred)**2)
SST = sum((Y - y_mean)**2)
r_square = 1 - (SSR/SST)
r_square
```

```
0.2548644825607733
```

```
# 9. Predict the output for a given input values
input1 = [float(i) for i in input("Enter the input values 1 to predict output : ").split()]
input2 = [float(i) for i in input("Enter the input values 2 to predict output : ").split()]
print("Input1\tInput2\tOutput")
for i in range(len(input1)):
        output = b0 + (b1 * input1[i]) + (b2 * input2[i])
        print(input1[i],"\t",input2[i],"\t",output)
```

```
Enter the input values 1 to predict output : 8
Enter the input values 2 to predict output : 64
Input1  Input2  Output
8.0      64.0    5441.263633043166
```

new linear regression

```
model=LinearRegression()
```

```
model.fit(X_train,Y_train)
```

```
▾ LinearRegression
LinearRegression()
```

```
model.score(X_train,Y_train)
```

```
0.9170426046312816
```

```
Y_pred=model.predict(X_test)
Y_pred
```

```
        1.59017421e+00,  1.33209133e+00,  4.25526275e-01,  2.98178915e+00,
        2.57545475e+00,  1.74222209e+00,  1.42211356e+00,  3.07024045e+00,
        2.35768872e+00,  1.90793417e+00,  1.93680505e+00,  1.11261199e+00,
       -1.95618924e-01,  1.75759517e+00,  1.48478731e+00,  1.33855996e+00,
        2.36427285e-01, -2.70820322e-01,  2.15226373e+00,  1.79383043e+00,
        1.84465529e+00,  2.23825573e+00, -4.91789376e-01,  1.19354441e+00,
        2.60576530e+00, -2.80343691e-01,  1.40516731e+00,  2.20625883e+00,
        2.81345608e+00,  2.47017928e-01,  1.70479450e+00, -2.35459318e-01,
        7.19644389e-01,  1.32013125e+00,  3.39668575e+00,  1.30811185e-01,
       -1.94976021e-01,  1.56403316e+00,  2.99626911e+00,  8.80838941e-01,
        1.76067569e+00,  3.32950864e-01,  1.50888555e+00,  3.78179010e-01,
        2.91819708e+00,  3.22408048e-01,  2.83166096e+00,  2.93071500e+00,
        2.21021065e+00,  2.46424185e+00,  7.55203449e-01,  1.69610236e+00,
        2.04427863e+00,  7.50043848e-01,  9.98140202e-01,  1.43722827e+00,
        1.33661227e-01,  1.20160589e+00,  1.61696861e-01,  2.62606368e+00,
        1.14298506e+00,  5.34615523e-01,  2.60075540e+00,  6.69620577e-01,
        1.67792345e-01,  7.07300465e-01,  2.37694648e+00, -8.24107299e-01,
        3.22056034e+00,  1.31004499e+00,  1.53808055e+00, -2.68314291e-01,
        1.21781328e+00,  2.61618996e+00, -3.07148227e-01,  1.51379130e+00,
        1.45468775e+00,  7.99461197e-01,  1.72010010e+00,  1.41382181e+00,
        1.35373017e+00,  3.23365815e-02,  2.06454308e+00, -3.33258292e-01,
        5.10764504e-01,  3.47423744e+00,  1.18906719e+00,  2.39189911e+00,
        3.41601545e+00,  1.81415122e+00,  2.16231862e+00,  1.28323416e-01,
        2.94247302e+00,  2.11718517e+00,  1.57277521e+00,  1.27457921e+00,
        2.58843491e+00,  2.16481809e+00,  3.03714085e+00,  3.49272717e+00,
        3.27931667e+00, -1.72076883e-01,  2.14919784e+00,  2.64229603e-01,
        2.47978708e+00, -9.75424214e-02,  9.50336160e-01,  9.00488793e-01,
        1.58097656e+00,  2.35279443e+00,  1.24115607e+00,  2.50338945e+00,
        9.93345700e-01,  1.68757386e+00,  5.10902163e-01,  1.22083168e+00,
        1.88354481e+00,  2.48416008e+00,  2.84738230e-01,  4.99772123e-01,
        7.76749724e-01,  3.56426113e+00, -9.54038396e-02,  2.48230931e+00,
       -1.64401299e-01,  1.49956797e+00,  2.08363774e+00,  9.28528936e-01,
        8.69960660e-01, -5.07287299e-01,  2.35726098e+00,  5.71487172e-01,
        4.19548709e-01,  7.74518478e-01,  3.80367332e+00, -5.74943685e-01,
        3.21080596e+00,  2.78074000e+00, -1.15102853e-01,  2.04539777e+00,
        1.29960482e+00,  3.13641417e+00,  1.53170512e+00,  1.36740217e+00,
        2.97294841e+00,  2.02387375e+00,  2.44737934e-01,  2.34595220e+00,
        2.21416500e+00,  1.88550836e+00,  3.14930515e-01,  3.02060370e-01,
        2.76805140e+00,  3.85637608e-01,  6.40365269e-01,  9.21736420e-01,
        1.28654785e+00,  2.79550632e+00,  1.76387262e+00,  2.53563626e+00,
        1.85881606e+00,  3.78486061e-01,  2.50954017e+00,  2.63136001e-01,
        1.89885466e-01,  1.03081707e+00,  2.92795085e+00,  6.24387587e-02,
        4.18902481e-01,  2.76910073e+00,  2.38066889e+00,  1.62973378e+00,
        1.79243200e+00,  2.74498138e+00, -4.03957528e-01,  4.48343709e-01,
        1.31120652e+00,  2.16231802e+00,  6.47033353e-01,  2.17006601e+00,
       -6.37089455e-01,  3.01353227e+00,  3.89076198e+00, -3.14676440e-01,
        2.38905069e+00,  2.84628943e+00, -3.26958652e-01,  2.35841494e+00,
        1.91094257e+00,  7.49011102e-01,  1.30803468e-01,  1.77785080e+00,
        2.16247104e+00,  7.51071137e-01,  3.17773253e+00,  1.89566790e+00,
        1.87652427e+00, -8.48525662e-02,  1.89263878e+00,  2.17591545e-02,
        3.40995684e+00,  2.96083764e+00,  1.99581758e+00,  1.17256854e+00,
        3.62906840e-01,  2.95883491e+00,  6.16326555e-01,  1.71903345e+00,
        1.17266502e-01,  2.99438681e-01,  1.28537717e+00,  2.95761196e+00,
        3.72206827e-01,  2.84512300e+00, -9.41552826e-02, -3.50168421e-01,
        1.41789929e+00,  1.75487527e+00,  2.50088538e-01,  1.23544617e+00,
        2.54017439e+00,  3.97392019e-01,  2.27615691e+00,  1.71471493e+00,
        9.43631006e-01,  2.08972352e+00,  2.79407043e-01,  3.03982825e+00,
        6.12831863e-01,  1.69220538e+00,  2.52426053e+00,  1.67050511e+00])
```

```
from sklearn.metrics import mean_absolute_error, mean_absolute_percentage_error, mean_squared_error
```

```
print(mean_absolute_error(Y_test,Y_pred))
```

```
print(mean_absolute_percentage_error(Y_test,Y_pred))
print(mean_squared_error(Y_test,Y_pred))
```

```
0.2747606739866767
310550246027776.6
0.10248318249564935
```

Logistic Regression

Target variables of the data set are discrete, hence, we are going to apply multiclass logistic regression model.

```
model_lr = LogisticRegression(multi_class = 'multinomial', solver = 'sag',  max_iter = 10000)
```

```
model_lr.fit(X_train,Y_train)
```

```
LogisticRegression
LogisticRegression(max_iter=10000, multi_class='multinomial', solver='sag')
```

```
model_lr.intercept_
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-32f85b5da93a> in <cell line: 1>()
----> 1 model_lr.intercept_

NameError: name 'model_lr' is not defined
```

SEARCH STACK OVERFLOW

```
model_lr.coef_
```

```
array([[-1.45168387e-03,  3.68449832e-02,  1.01318608e-03,
         2.63362043e-02,  7.63524275e-03,  3.96128100e-02,
         8.99541647e-02,  6.58937249e-02, -1.87092786e-03,
         3.09482289e-04, -4.98282961e-03,  1.48382232e-01,
         3.86353369e-02,  9.16882165e-02,  9.04036869e-03,
         1.11354193e-02,  1.34563275e-02,  9.44333226e-03,
         8.27210294e-03,  1.01342755e-02],
       [ 4.52347632e-06,  4.63079330e-03,  1.48451592e-02,
         9.14138732e-03,  5.37900410e-03,  1.19193793e-02,
        -2.32450902e-02,  1.40305153e-02, -3.26360455e-04,
         2.79284316e-04, -7.89031781e-04,  3.99101657e-02,
        -3.21712017e-03,  3.95652751e-02,  6.84643695e-03,
         6.68034024e-03,  4.49824028e-03,  8.35326085e-03,
         2.26407085e-03,  3.06021564e-03],
       [ 4.59720997e-04, -8.47403836e-03,  5.56057057e-03,
        -1.27447579e-02, -4.87745892e-03, -9.44705890e-03,
         6.53979254e-03, -1.89402055e-02,  5.82927106e-04,
        -1.48043773e-04,  1.78559385e-03, -5.63670566e-02,
        -8.56484251e-03, -3.60612739e-02, -1.11441770e-02,
        -4.69959011e-03, -1.03266659e-02, -6.81049288e-03,
        -2.51530683e-03, -1.69154941e-03],
       [ 9.87439400e-04, -3.30017381e-02, -2.14189159e-02,
        -2.27328338e-02, -8.13678792e-03, -4.20851303e-02,
        -7.32488670e-02, -6.09840347e-02,  1.61436121e-03,
        -4.40722832e-04,  3.98626754e-03, -1.31925341e-01,
        -2.68533742e-02, -9.51922177e-02, -4.74262863e-03,
        -1.31161694e-02, -7.62790187e-03, -1.09861002e-02,
        -8.02086696e-03, -1.15029417e-02]])
```

```
Y_pred=model_lr.predict(X_test)
Y_pred
```

```
array([3, 0, 2, 1, 3, 0, 0, 2, 3, 1, 1, 3, 1, 2, 3, 0, 3, 2, 2, 1, 0, 0,
       3, 1, 1, 3, 3, 1, 3, 1, 2, 0, 2, 1, 2, 3, 0, 0, 3, 2, 2, 2, 3, 3,
       2, 3, 0, 1, 3, 2, 1, 2, 0, 3, 0, 3, 3, 1, 0, 3, 3, 1, 2, 1, 1, 3,
       3, 3, 2, 2, 3, 3, 1, 0, 1, 3, 3, 2, 1, 1, 3, 1, 3, 0, 0, 0, 2, 1,
       1, 3, 1, 2, 2, 0, 0, 3, 2, 3, 0, 2, 1, 2, 2, 1, 3, 3, 3, 2, 2, 3,
       2, 0, 0, 2, 2, 3, 0, 1, 0, 0, 0, 3, 2, 2, 1, 2, 1, 0, 0, 3, 1, 3,
       3, 2, 3, 3, 3, 3, 0, 1, 1, 2, 1, 3, 0, 3, 0, 0, 2, 0, 1, 1, 1, 1,
       3, 0, 0, 3, 1, 3, 2, 2, 2, 1, 3, 3, 3, 3, 1, 0, 3, 2, 1, 3, 3, 0,
       1, 2, 3, 1, 3, 1, 0, 1, 2, 1, 2, 0, 3, 3, 1, 2, 0, 2, 2, 1, 1, 3,
       2, 0, 3, 3, 3, 0, 1, 3, 2, 2, 0, 0, 0, 1, 2, 2, 0, 0, 0, 3, 2, 2,
       3, 3, 0, 0, 3, 3, 2, 2, 0, 2, 0, 0, 0, 3, 2, 0, 2, 3, 0, 1, 0, 2,
```

```
           3, 2, 0, 0, 1, 3, 3, 2, 2, 0, 3, 1, 1, 0, 2, 2, 3, 2, 0, 0, 1, 3,
           2, 2, 2, 3, 3, 2, 1, 2, 2, 2, 1, 3, 2, 2, 2, 1, 0, 1, 1, 0, 0, 0,
           2, 3, 2, 2, 0, 1, 2, 0, 1, 2, 3, 0, 1, 0, 1, 1, 3, 0, 0, 1, 3, 1,
           2, 0, 1, 1, 3, 0, 2, 3, 2, 3, 0, 1, 3, 1, 1, 1, 0, 1, 0, 3, 1, 1,
           2, 1, 0, 1, 3, 0, 3, 1, 1, 0, 2, 3, 0, 2, 1, 1, 2, 2, 2, 0, 3, 0,
           0, 2, 1, 3, 3, 2, 2, 0, 3, 2, 3, 1, 2, 3, 3, 3, 3, 0, 2, 0, 2, 0,
           1, 1, 2, 2, 1, 3, 1, 1, 0, 1, 1, 3, 1, 0, 0, 3, 0, 3, 0, 2, 3, 1,
           1, 0, 2, 1, 0, 1, 3, 0, 3, 3, 0, 3, 3, 3, 1, 2, 3, 2, 0, 2, 1, 2,
           1, 0, 3, 1, 1, 1, 1, 3, 2, 2, 2, 0, 3, 1, 1, 2, 3, 0, 2, 3, 2, 2,
           2, 3, 0, 0, 1, 1, 1, 2, 0, 3, 3, 0, 3, 3, 0, 3, 1, 2, 0, 2, 2, 0,
           3, 3, 2, 0, 2, 0, 3, 3, 2, 2, 1, 3, 1, 2, 0, 0, 1, 2, 0, 2, 0, 0,
           1, 2, 0, 1, 2, 0, 3, 3, 2, 3, 0, 3, 1, 2, 3, 2])
```

```
from sklearn.metrics import classification_report,accuracy_score,f1_score,confusion_matrix
```

```
print(accuracy_score(Y_pred,Y_test))
print(classification_report(Y_pred,Y_test))
print(confusion_matrix(Y_pred,Y_test))
```

```
0.714
              precision    recall  f1-score   support

           0       0.87      0.91      0.89       119
           1       0.69      0.66      0.67       116
           2       0.55      0.54      0.55       128
           3       0.74      0.76      0.75       137

    accuracy                           0.71       500
   macro avg       0.71      0.72      0.71       500
weighted avg       0.71      0.71      0.71       500

[[108  11   0   0]
 [ 15  76  24   1]
 [  1  22  69  36]
 [  0   1  32 104]]
```

KNN neigbohr model is used here

```
model_knn=model_knn.fit(X_train,Y_train)
```

```
Y_pred=model_knn.predict(X_test)
Y_pred
```

```
array([3, 0, 2, 2, 3, 0, 0, 2, 3, 1, 1, 3, 0, 2, 3, 0, 3, 2, 2, 1, 0, 0,
       3, 1, 1, 2, 3, 1, 3, 1, 1, 0, 2, 0, 2, 3, 0, 0, 3, 3, 2, 1, 3, 3,
       1, 3, 0, 1, 3, 1, 1, 3, 0, 3, 0, 3, 2, 2, 0, 3, 3, 1, 3, 2, 1, 2,
       3, 2, 1, 2, 3, 2, 1, 0, 1, 3, 2, 1, 1, 2, 3, 3, 3, 0, 0, 0, 2, 0,
       2, 3, 1, 2, 3, 1, 0, 3, 3, 3, 0, 3, 1, 1, 3, 1, 3, 2, 2, 3, 2, 3,
       3, 0, 0, 1, 3, 3, 0, 1, 1, 0, 0, 3, 2, 2, 1, 1, 1, 1, 0, 2, 1, 3,
       2, 3, 3, 3, 3, 2, 0, 1, 1, 2, 1, 3, 0, 3, 0, 0, 2, 0, 1, 1, 1, 1,
       3, 0, 0, 3, 1, 3, 2, 1, 3, 1, 2, 3, 3, 2, 1, 0, 3, 2, 2, 3, 3, 0,
       2, 2, 3, 0, 2, 1, 0, 1, 3, 1, 2, 0, 2, 3, 1, 1, 0, 2, 3, 0, 1, 3,
       2, 0, 3, 3, 2, 1, 2, 3, 3, 3, 0, 0, 0, 2, 3, 3, 0, 0, 1, 3, 1, 3,
       3, 3, 0, 0, 2, 2, 3, 1, 0, 2, 0, 0, 0, 3, 3, 0, 2, 2, 1, 1, 0, 2,
       3, 3, 0, 0, 1, 3, 3, 2, 3, 0, 3, 1, 1, 0, 2, 3, 3, 2, 0, 0, 1, 2,
       3, 2, 2, 3, 1, 1, 0, 3, 3, 2, 1, 3, 3, 2, 2, 1, 0, 2, 2, 1, 0, 0,
       3, 2, 2, 2, 0, 1, 3, 0, 1, 3, 3, 0, 2, 0, 1, 1, 3, 0, 0, 2, 3, 1,
       2, 0, 2, 0, 3, 0, 3, 3, 2, 3, 1, 2, 2, 1, 1, 1, 0, 1, 0, 3, 1, 0,
       3, 0, 0, 1, 2, 0, 3, 1, 2, 0, 1, 3, 0, 2, 2, 1, 2, 1, 1, 0, 2, 0,
       0, 3, 1, 2, 3, 2, 2, 0, 3, 3, 1, 1, 3, 2, 3, 3, 3, 0, 2, 0, 3, 0,
       1, 1, 2, 2, 1, 3, 1, 2, 0, 1, 2, 2, 0, 0, 1, 3, 0, 3, 0, 1, 2, 1,
       0, 0, 2, 1, 0, 1, 3, 0, 3, 3, 0, 2, 1, 3, 2, 1, 3, 2, 0, 3, 2, 2,
       0, 0, 3, 0, 0, 1, 1, 3, 2, 3, 2, 0, 3, 0, 0, 1, 3, 0, 0, 3, 3, 2,
       2, 3, 0, 0, 1, 2, 1, 2, 0, 3, 3, 0, 2, 3, 0, 2, 2, 1, 0, 2, 2, 1,
       3, 2, 2, 0, 2, 0, 3, 3, 2, 1, 0, 3, 0, 2, 0, 0, 1, 3, 1, 3, 0, 0,
       1, 2, 0, 1, 3, 0, 2, 2, 1, 2, 0, 3, 0, 2, 3, 2])
```

```
from sklearn.metrics import classification_report,accuracy_score,f1_score,confusion_matrix
```

```
knn=accuracy_score(Y_pred,Y_test)
print(classification_report(Y_pred,Y_test))
print(confusion_matrix(Y_pred,Y_test))
knn
```

```
              precision    recall  f1-score   support

           0       0.98      0.95      0.97       128
           1       0.92      0.92      0.92       110
           2       0.86      0.90      0.88       119
           3       0.94      0.92      0.93       143

    accuracy                           0.92       500
   macro avg       0.92      0.92      0.92       500
weighted avg       0.93      0.92      0.92       500

[[122   6   0   0]
 [  2 101   7   0]
 [  0   3 107   9]
 [  0   0  11 132]]
0.924
```

0s completed at 9:28 PM