critical-analysis-2024-s1

June 1, 2024

1 Mobile Price Dataset - COMP2200 DATA SCIENCE

Name: Dang Ha Nguyen (Janice) Nguyen

ID: 47856491

Repository: GitHub Repository

1.1 AIM:

In this Project, on the basis of the mobile Specification like Battery power, 3G enabled , wifi ,Bluetooth, Ram etc, we want to predict the price range of the mobile.

1.2 DESCRIPTION:

Input variables:

- id: ID
- battery_power: Total energy a battery can store in one time measured in mAh
- blue: Has bluetooth or not
- clock_speed: speed at which microprocessor executes instructions
- dual_sim: Has dual sim support or not
- fc: Front Camera mega pixels
- four_g: Has 4G or not
- int_memory: Internal Memory in Gigabytes
- m_dep: Mobile Depth in cm
- mobile_wt: Weight of mobile phone
- n_cores: Number of cores of processor
- pc: Primary Camera mega pixels
- px_height: Pixel Resolution Height
- px_width: Pixel Resolution Width
- ram: Random Access Memory in Megabytes
- sc_h: Screen Height of mobile in cm
- sc_w: Screen Width of mobile in cm
- talk_time: longest time that a single battery charge will last when you are
- three_g: Has 3G or not
- touch_screen: Has touch screen or not
- wifi: Has wifi or not

Output variables:

- price_range: the target value we want to estimate. There are four possible values: 0,1,2,3.

1.2.1 Note that the price range has only four possible values. Thus, this is a classification problem

1.3 Library

[185]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪ConfusionMatrixDisplay, mean_squared_error
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier

import warnings
warnings.filterwarnings("ignore")
```

1.4 1. Data loading

- We print the table head of the source data to check what kind of feature data has been included.
- Note that column 'index' is not regarded as a meaningful feature here.

[186]:
```
data = pd.read_csv("data.csv").reset_index()
print("data shape is : ", data.shape)
data.head()
```

```
data shape is :  (2000, 22)
```

[186]:
```
   index  battery_power  blue  clock_speed  dual_sim  fc  four_g  int_memory  \
0      0            842     0          2.2         0   1       0           7
1      1           1021     1          0.5         1   0       1          53
2      2            563     1          0.5         1   2       1          41
3      3            615     1          2.5         0   0       0          10
4      4           1821     1          1.2         0  13       1          44

   m_dep  mobile_wt  …  px_height  px_width   ram  sc_h  sc_w  talk_time  \
0    0.6        188  …         20       756  2549     9     7         19
1    0.7        136  …        905      1988  2631    17     3          7
2    0.9        145  …       1263      1716  2603    11     2          9
3    0.8        131  …       1216      1786  2769    16     8         11
4    0.6        141  …       1208      1212  1411     8     2         15
```

```
   three_g  touch_screen  wifi  price_range
0        0             0     1            1
1        1             1     0            2
2        1             1     0            2
3        1             0     0            2
4        1             1     0            1

[5 rows x 22 columns]
```

- It shows that there are 2,000 samples and each sample has 22 features (including the target feature 'price_range')
- We further observe the statistical features of the source data by showing the mean, std, min, max, etc., statistical information as below

[187]:
```python
data.describe()
```

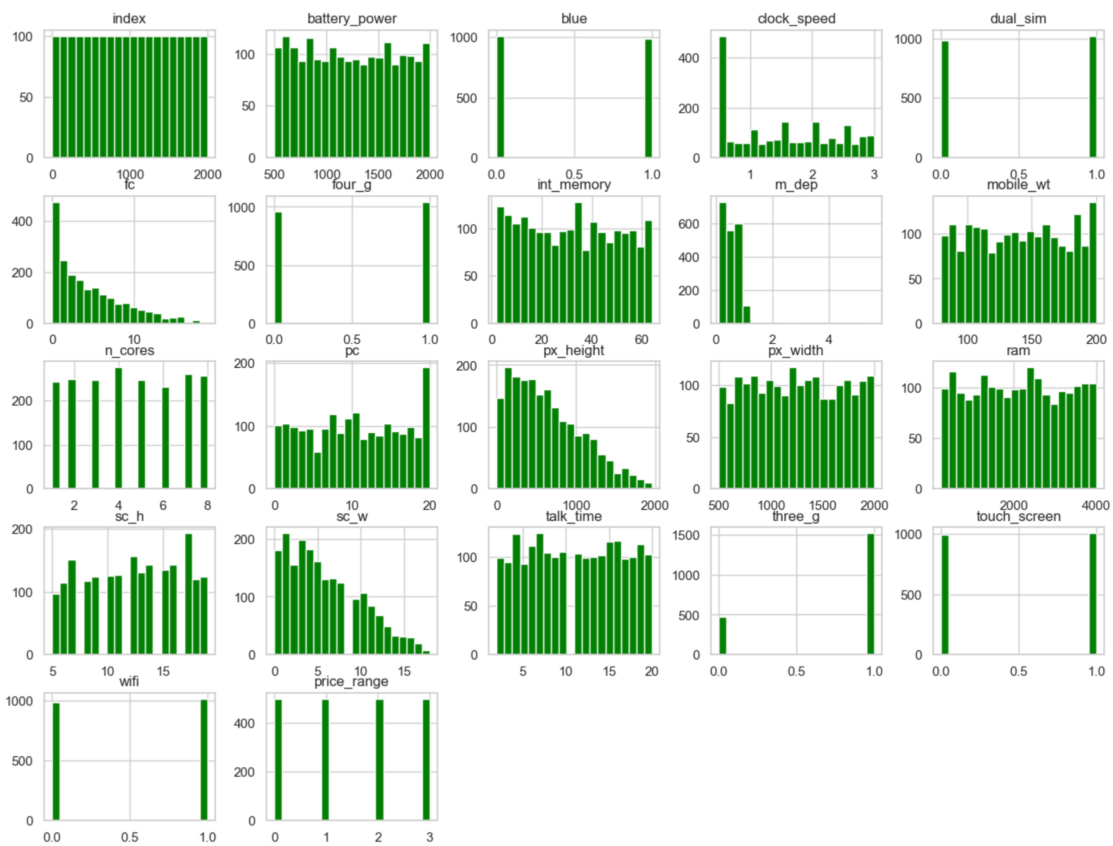
[187]:
```
             index  battery_power       blue  clock_speed     dual_sim  \
count  2000.000000    2000.000000  2000.0000  2000.000000  2000.000000
mean    999.500000    1238.518500     0.4950     1.522250     0.509500
std     577.494589     439.418206     0.5001     0.816004     0.500035
min       0.000000     501.000000     0.0000     0.500000     0.000000
25%     499.750000     851.750000     0.0000     0.700000     0.000000
50%     999.500000    1226.000000     0.0000     1.500000     1.000000
75%    1499.250000    1615.250000     1.0000     2.200000     1.000000
max    1999.000000    1998.000000     1.0000     3.000000     1.000000

                fc       four_g   int_memory        m_dep    mobile_wt  …  \
count  2000.000000  2000.000000  2000.000000  2000.000000  2000.000000  …
mean      4.309500     0.521500    32.046500     0.505250   140.249000  …
std       4.341444     0.499662    18.145715     0.314272    35.399655  …
min       0.000000     0.000000     2.000000     0.100000    80.000000  …
25%       1.000000     0.000000    16.000000     0.200000   109.000000  …
50%       3.000000     1.000000    32.000000     0.500000   141.000000  …
75%       7.000000     1.000000    48.000000     0.800000   170.000000  …
max      19.000000     1.000000    64.000000     5.600000   200.000000  …

         px_height     px_width          ram         sc_h         sc_w  \
count  2000.000000  2000.000000  2000.000000  2000.000000  2000.000000
mean    645.108000  1251.515500  2124.213000    12.306500     5.767000
std     443.780811   432.199447  1084.732044     4.213245     4.356398
min       0.000000   500.000000   256.000000     5.000000     0.000000
25%     282.750000   874.750000  1207.500000     9.000000     2.000000
50%     564.000000  1247.000000  2146.500000    12.000000     5.000000
75%     947.250000  1633.000000  3064.500000    16.000000     9.000000
max    1960.000000  1998.000000  3998.000000    19.000000    18.000000

         talk_time      three_g  touch_screen         wifi  price_range
```

count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	11.011000	0.761500	0.503000	0.507000	1.500000
std	5.463955	0.426273	0.500116	0.500076	1.118314
min	2.000000	0.000000	0.000000	0.000000	0.000000
25%	6.000000	1.000000	0.000000	0.000000	0.750000
50%	11.000000	1.000000	1.000000	1.000000	1.500000
75%	16.000000	1.000000	1.000000	1.000000	2.250000
max	20.000000	1.000000	1.000000	1.000000	3.000000

[8 rows x 22 columns]

- We also plot the feature distribution to observe the value distribution of each feature.

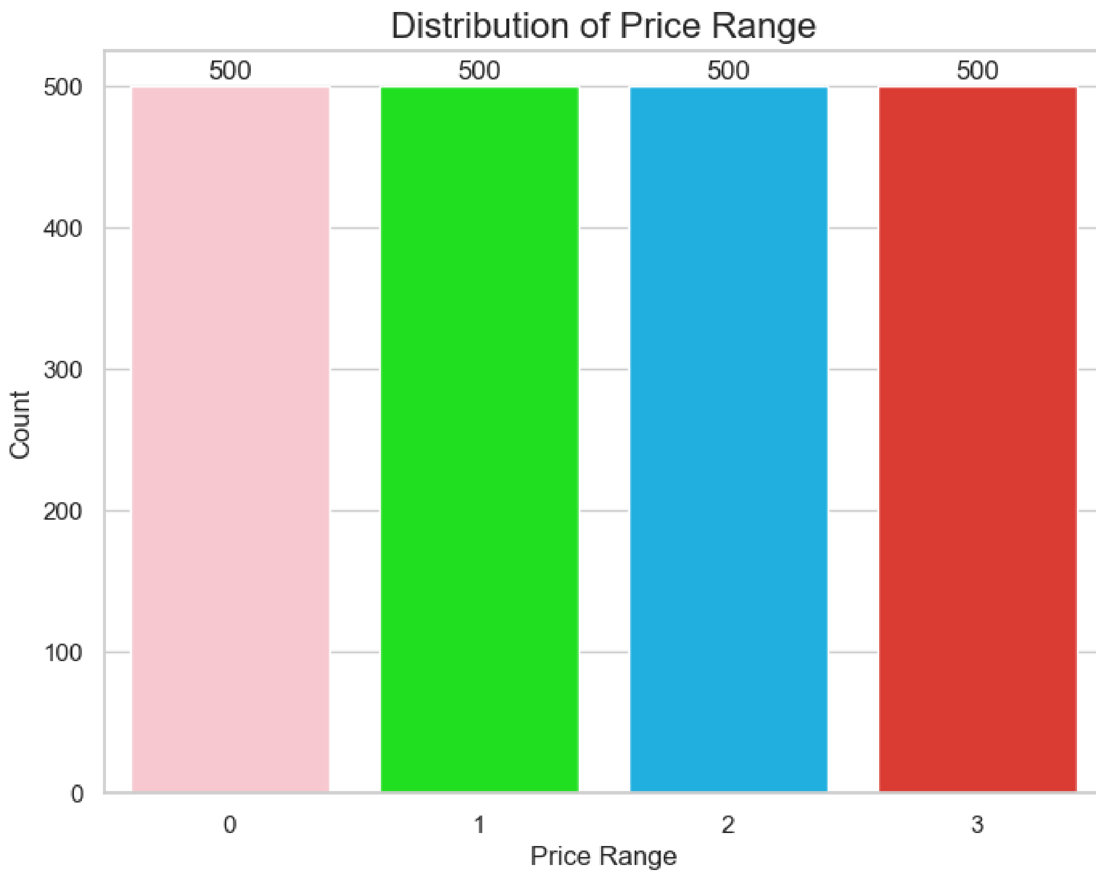
[188]:
```
data.hist(bins=20 ,figsize=(16,12), color = 'Green')
plt.show()
```



- In particular, we plot the distribution of the target variable: Price range

[189]:
```
sns.set_theme(style='whitegrid')
colors = ['#FFC0CB', '#00FF00', '#00BFFF', '#F62217']
```

```python
plt.figure(figsize=(8, 6))
sns.countplot(x='price_range', data=data, palette=colors)

plt.title('Distribution of Price Range', fontsize=16)
plt.xlabel('Price Range', fontsize=12)
plt.ylabel('Count', fontsize=12)

# Add value labels on top of each bar
for p in plt.gca().patches:
    plt.gca().text(p.get_x() + p.get_width()/2., p.get_height() + 5,
                   int(p.get_height()), fontsize=12, ha='center')

plt.show()
```



- There are four possible values of the price range with 0, 1, 2 and 3. From the distribution, we can find that the label idstribution is very balanced and even among four possible labels.

1.5 Feature Selection

We first study the correlation between the mobile price range and other features.

[190]:
```
data = data.drop(columns=['index'])
# We drop the 'index' column here since it's meaningless for prediction
correlation_matrix = data.corr()
print(correlation_matrix['price_range'])
```

```
battery_power    0.200723
blue             0.020573
clock_speed     -0.006606
dual_sim         0.017444
fc               0.021998
four_g           0.014772
int_memory       0.044435
m_dep            0.001495
mobile_wt       -0.030302
n_cores          0.004399
pc               0.033599
px_height        0.148858
px_width         0.165818
ram              0.917046
sc_h             0.022986
sc_w             0.038711
talk_time        0.021859
three_g          0.023611
touch_screen    -0.030411
wifi             0.018785
price_range      1.000000
Name: price_range, dtype: float64
```

Then we remove irrelevant features.

[191]:
```
data = data.drop(columns=['clock_speed','mobile_wt','touch_screen'])
```

[192]:
```
data.head()
```

[192]:
```
   battery_power  blue  dual_sim  fc  four_g  int_memory  m_dep  n_cores  pc  \
0            842     0         0   1       0           7    0.6        2   2
1           1021     1         1   0       1          53    0.7        3   6
2            563     1         1   2       1          41    0.9        5   6
3            615     1         0   0       0          10    0.8        6   9
4           1821     1         0  13       1          44    0.6        2  14

   px_height  px_width   ram  sc_h  sc_w  talk_time  three_g  wifi  \
0         20       756  2549     9     7         19        0     1
1        905      1988  2631    17     3          7        1     0
2       1263      1716  2603    11     2          9        1     0
3       1216      1786  2769    16     8         11        1     0
4       1208      1212  1411     8     2         15        1     0
```

```
   price_range
0            1
1            2
2            2
3            2
4            1
```

1.6 2. Data Preprocessing

1.6.1 2.1 Data Normalisetion & Train Test Split

- The cleaned normalised dataset is split into train dataset and the test dataset and we need to randomly shuffle the data set

[193]:
```
X = data.copy().drop(columns=['price_range'])
scaler = StandardScaler()
df_standardized = scaler.fit_transform(X)
df_standardized = pd.DataFrame(df_standardized, columns=X.columns)
```

- The current allocation is 75% for training and 25% for testing is not ideal and can lead to insufficient training data and unreliable model evaluation.

```
data = data.sample(frac=1, random_state=42).reset_index(drop=True)
x_ex1 = df_standardized
y_ex1 = data.copy()['price_range']
x_ex1_array = x_ex1.values
y_ex1_array = y_ex1.values
x_train = x_ex1_array[0:int((len(y_ex1_array)+1)*0.75),:]
x_test = x_ex1_array[int((len(y_ex1_array)+1)*0.75):,:]
y_train = y_ex1_array[0:int((len(y_ex1_array)+1)*0.75)]
y_test = y_ex1_array[int((len(y_ex1_array)+1)*0.75):]
```

- In this revised version, I allocate 80% of the data for training and 20% for testing

[194]:
```
X = df_standardized
y = data['price_range']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪stratify=y, random_state=42)
```

1.7 3. Neural Network

- We will train a neural network model to predict the price range target variable based on the cleaned and normalised dataset.

1.7.1 3.1 Model generation

- We firstly use the test loss and accuracy to evaluate the performance of the trained neural network model.
- From evaluation results, we can observe that:

```python
def my_logloss(true_label, predicted):
    b = np.zeros((true_label.size,true_label.max()+1))
    b[:,true_label] = 1
    N = predicted.shape[0]
    ce = -np.sum(b * np.log(predicted)) / N
    return ce

mlp = MLPClassifier(
    solver='sgd',
    activation='identity',
    random_state=42,
    hidden_layer_sizes=(20,10,5),
    learning_rate_init=0.001,
    learning_rate='constant',
    max_iter=1,
)

""" Home-made mini-batch learning
    -> not to be used in out-of-core setting!
"""
N_TRAIN_SAMPLES = x_train.shape[0]
N_EPOCHS = 25
N_BATCH = 20
N_CLASSES = np.unique(y_train)

scores_train = []
scores_test = []
train_loss = []
test_loss = []

# epoch
epoch = 0
while epoch < N_EPOCHS:
    # shuffing
    random_perm = np.random.permutation(x_train.shape[0])
    mini_batch_index = 0
    while True:
        # mini-batch
        indices = random_perm[mini_batch_index:mini_batch_index + N_BATCH]
        mlp.partial_fit(x_train[indices], y_train[indices], classes=N_CLASSES)
        mini_batch_index += N_BATCH

        if mini_batch_index >= N_TRAIN_SAMPLES:
            break

     # test record
    scores_test.append(mlp.score(x_test, y_test))
    y_pred = mlp.predict_proba(x_test)
```

```
    test_error = my_logloss(y_test, y_pred)
    test_loss.append(test_error)

    epoch += 1
# plot
plt.plot(test_loss,label='test loss')
plt.legend([ 'test loss'])
plt.xlabel('epoch')
plt.ylabel('Loss')
plt.show()

# plot
plt.plot(scores_test,label='test accuracy')
plt.legend([ 'test accuracy'])
plt.xlabel('epoch')
plt.ylabel('Accuracy')
plt.show()
```

1) The model performance is not stable.
2) As the number of iterations increases, the loss continues to decrease, but the accuracy is fluctuated.

So that in this revised version, there are some modifications to the training process:

1. Use a non-linear activation function:
   - Change `activation='identity'` to `activation='relu'` or `activation='tanh'` to introduce non-linearity in the model and capture complex patterns in the data.
2. Use a more efficient solver and learning rate schedule:
   - Change `solver='sgd'` to `solver='adam'` to use the Adam optimizer, which adapts the learning rate for each parameter and often converges faster.
   - Remove `learning_rate='constant'` and let the solver handle the learning rate schedule automatically.
3. Increase the number of iterations:
   - Increase `max_iter=1` to a higher value, such as `max_iter=200`, to allow the model to train for more iterations and potentially improve its performance.
4. Monitor the training loss:
   - Add code to calculate and store the training loss in each epoch.
   - Plot both the training and test loss to observe how the model is learning and identify any potential overfitting or underfitting.
5. Use early stopping:
   - Implement early stopping to prevent overfitting and find the optimal number of epochs.
   - Monitor the validation loss and stop training if the loss starts to increase or plateaus.

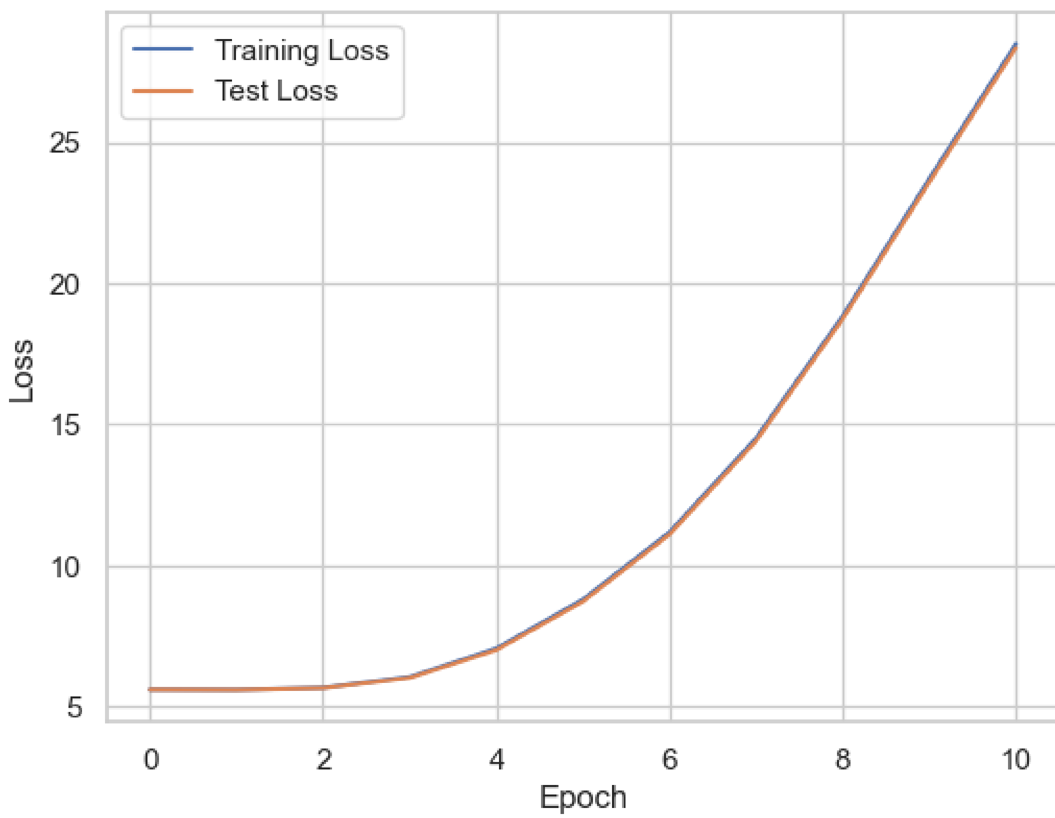
[195]:
```
def my_logloss(true_label, predicted):
    b = np.zeros((true_label.size, true_label.max() + 1))
    b[:, true_label] = 1
    N = predicted.shape[0]
    ce = -np.sum(b * np.log(predicted)) / N
    return ce
```

```
mlp = MLPClassifier(
    solver='adam',
    activation='relu',
    random_state=42,
    hidden_layer_sizes=(20, 10, 5),
    learning_rate_init=0.001,
    max_iter=200,
)

N_TRAIN_SAMPLES = X_train.shape[0]
N_EPOCHS = 25
N_BATCH = 20
N_CLASSES = np.unique(y_train)

scores_train = []
scores_test = []
train_loss = []
test_loss = []

best_test_loss = float('inf')
best_epoch = 0

epoch = 0
while epoch < N_EPOCHS:
    random_perm = np.random.permutation(X_train.shape[0])
    mini_batch_index = 0
    while True:
        indices = random_perm[mini_batch_index:mini_batch_index + N_BATCH]
        mlp.partial_fit(X_train.iloc[indices], y_train.iloc[indices],␣
 ↪classes=N_CLASSES)
        mini_batch_index += N_BATCH

        if mini_batch_index >= N_TRAIN_SAMPLES:
            break

    scores_train.append(mlp.score(X_train, y_train))
    train_pred = mlp.predict_proba(X_train)
    train_error = my_logloss(y_train, train_pred)
    train_loss.append(train_error)

    scores_test.append(mlp.score(X_test, y_test))
    y_pred = mlp.predict_proba(X_test)
    test_error = my_logloss(y_test, y_pred)
    test_loss.append(test_error)

    if test_error < best_test_loss:
```

```
        best_test_loss = test_error
        best_epoch = epoch

    epoch += 1

    if epoch - best_epoch >= 10:
        print(f"Early stopping at epoch {epoch}")
        break
```

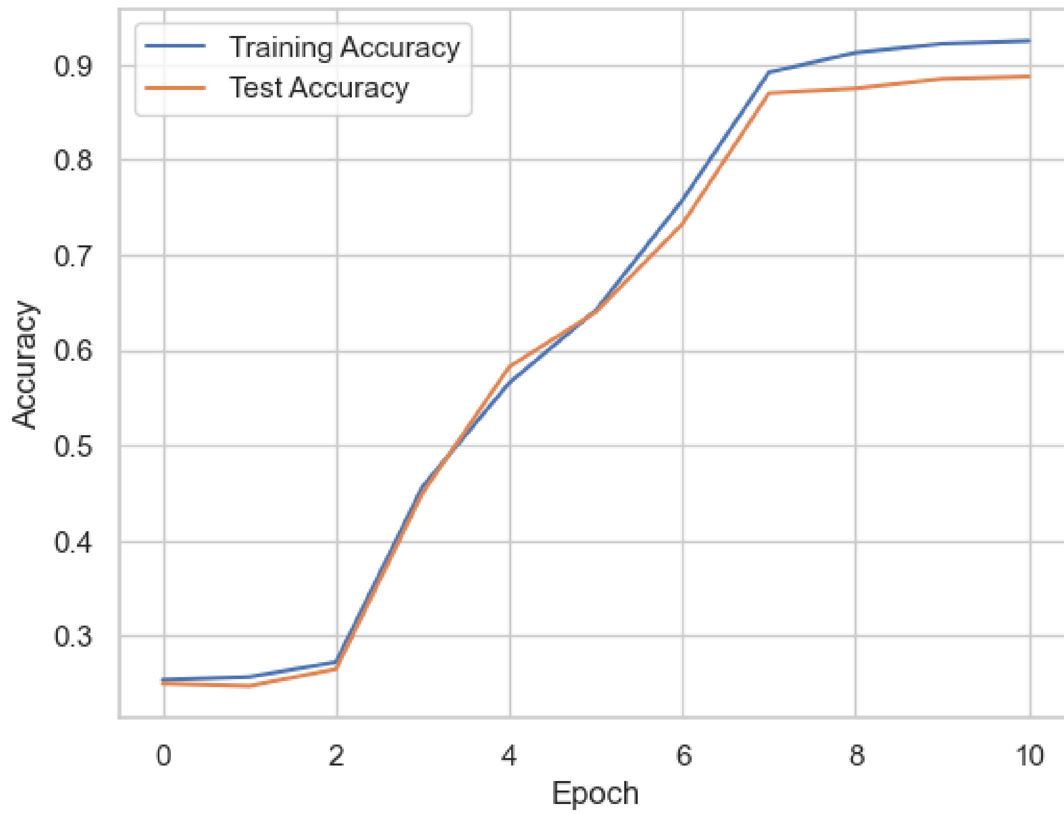
```
Early stopping at epoch 11
```

[196]:
```
# Plot training and test loss
plt.plot(train_loss, label='Training Loss')
plt.plot(test_loss, label='Test Loss')
plt.legend(['Training Loss', 'Test Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



[197]:
```
# Plot training and test accuracy
plt.plot(scores_train, label='Training Accuracy')
```

```
plt.plot(scores_test, label='Test Accuracy')
plt.legend(['Training Accuracy', 'Test Accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```



1.7.2 3.2 Model evaluation

- We further evaluate the model performance by using more metrics such as precision, recall and f1-score

[198]:
```
y_pred = mlp.predict(X_test)
report = classification_report(y_test, y_pred)
print("Classification Report:")
print(report)
```

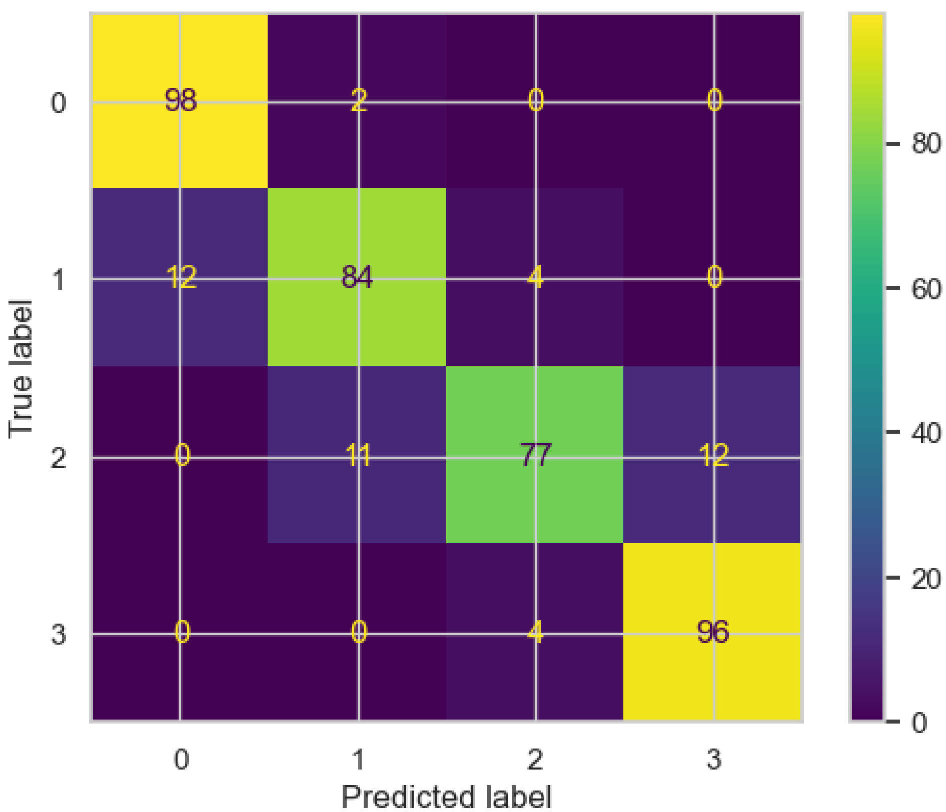
```
Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.98      0.93       100
           1       0.87      0.84      0.85       100
```

```
           2       0.91      0.77      0.83       100
           3       0.89      0.96      0.92       100

    accuracy                           0.89       400
   macro avg       0.89      0.89      0.89       400
weighted avg       0.89      0.89      0.89       400
```

[199]:

```python
y_pred = mlp.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=mlp.classes_)
disp.plot()
plt.show()
```

```
Confusion Matrix:
[[98  2  0  0]
 [12 84  4  0]
 [ 0 11 77 12]
 [ 0  0  4 96]]
```



1.8 4. Decision Tree

- Now, we train the second model, decision tree model, to predict the mobile price range, to see which model can provide better performance.

1.8.1 4.1 Model generation

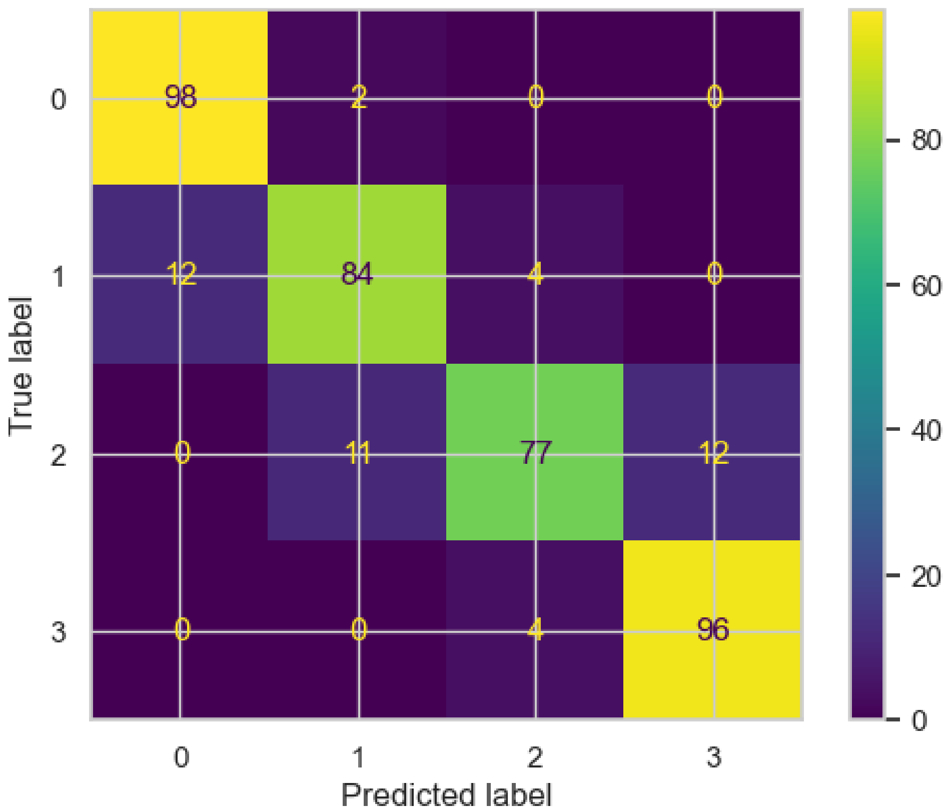
[200]:
```
tree = DecisionTreeClassifier(
    criterion='gini',
    max_depth=None,
    min_samples_split=5,
    min_samples_leaf=3,
)
tree.fit(X_train,y_train)
```

[200]:
```
DecisionTreeClassifier(min_samples_leaf=3, min_samples_split=5)
```

1.8.2 4.2 Test

[201]:
```
cm = confusion_matrix( y_test, y_pred, labels=tree.classes_)
print(cm)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=tree.classes_)
disp.plot()
plt.show()
```

```
[[98  2  0  0]
 [12 84  4  0]
 [ 0 11 77 12]
 [ 0  0  4 96]]
```



[202]:
```
tree.fit(X_train,y_train)
y_true, y_pred = y_test , tree.predict(X_test)
print('Results on the test set:')
print(classification_report(y_true, y_pred))
```

```
Results on the test set:
              precision    recall  f1-score   support

           0       0.93      0.93      0.93       100
           1       0.85      0.82      0.84       100
           2       0.77      0.85      0.81       100
           3       0.91      0.86      0.89       100

    accuracy                           0.86       400
   macro avg       0.87      0.86      0.87       400
weighted avg       0.87      0.86      0.87       400
```

It's important to note that the performance of both models is relatively high, with accuracies above 85%. The decision tree model also achieves good results, especially for class 0.

1.9 5. Performance Comparison

1. Accuracy:
    - **Neural network**: 0.89 (89%)
    - **Decision tree**: 0.86 (86%)

   The neural network has a higher overall accuracy compared to the decision tree, indicating that it correctly predicts the price range for a larger proportion of the test samples.
2. Precision:
    - **Neural network**: 0.89, 0.87, 0.91, 0.89 (for classes 0, 1, 2, 3 respectively)
    - **Decision tree**: 0.93, 0.85, 0.77, 0.91 (for classes 0, 1, 2, 3 respectively)

   The decision tree has higher precision values for classes 0 and 3, while the neural network has higher precision for classes 1 and 2. Precision measures the proportion of true positive predictions among all positive predictions for each class.
3. Recall:
    - **Neural network**: 0.98, 0.84, 0.77, 0.96 (for classes 0, 1, 2, 3 respectively)
    - **Decision tree**: 0.93, 0.82, 0.85, 0.86 (for classes 0, 1, 2, 3 respectively)

   The neural network has higher recall values for classes 0, 1, and 3, while the decision tree has higher recall for class 2. Recall measures the proportion of true positive predictions among all actual positive instances for each class.
4. F1-score:
    - **Neural network**: 0.93, 0.85, 0.83, 0.92 (for classes 0, 1, 2, 3 respectively)
    - **Decision tree**: 0.93, 0.84, 0.81, 0.89 (for classes 0, 1, 2, 3 respectively)

   The neural network has higher F1-scores for classes 1 and 3, while the decision tree has a higher F1-score for class 2. Both models have the same F1-score for class 0. The F1-score is the harmonic mean of precision and recall, providing a balanced measure of the model's performance for each class.
5. Macro and Weighted Averages:
    - **Neural network**: Macro avg - 0.89, 0.89, 0.89 | Weighted avg - 0.89, 0.89, 0.89
    - **Decision tree**: Macro avg - 0.87, 0.86, 0.87 | Weighted avg - 0.87, 0.86, 0.87

   The neural network has slightly higher macro and weighted average scores for precision, recall, and F1-score, indicating better overall performance across all classes.

In summary, the neural network model continues to outperform the decision tree model based on the updated classification reports. The neural network has higher accuracy, recall for classes 0, 1, and 3, and F1-scores for classes 1 and 3. The decision tree has higher precision for classes 0 and 3, and recall for class 2.

Both models demonstrate good performance, with accuracies above 86%. The neural network's strengths lie in its higher recall for classes 0, 1, and 3, while the decision tree's strengths are in its higher precision for classes 0 and 3, and recall for class 2.

1.9.1 4.3 Further exploration

Since these two models have a similar accuracy score on the test set, so we want to use another metric to compare them.

[203]:
```
# NN's prediction
nn_test = mlp.predict(X_test)
nn_mse = mean_squared_error(y_test, nn_test)
```

```python
print(f"Neural Network's Mean Squared Error: {nn_mse}")

# DT's prediction
dt_test = tree.predict(X_test)
dt_mse = mean_squared_error(y_test, dt_test)
print(f"Decision Tree's Mean Squared Error: {dt_mse}")
```

```
Neural Network's Mean Squared Error: 0.1125
Decision Tree's Mean Squared Error: 0.135
```

- These two model's performance under MSE are quite similar and the gap is really small.
- From that comparison, we can assert that Neural Network model is better and more powerful than that of the Decision Tree model because Neural Network model achieves a higher model accuracy.