Principle of Data Science

Final Project

Mobile Price Prediction

Submitted by:

Nawaraj Adhikari nxa6298@mavs.uta.edu Student ID: 1002166298

Patel Manan Champakbhai mxp8783@mavs.uta.edu Student Id:1002228783

Submitted to:

Mei Yang, PhD

Instructor

Division of Data Science, College of Science

For the fulfillment of the requirement for project work of 2248-ASDS-5302-003 – Principle of Data Science

The University of Texas at Arlington

Arlington, TX, USA

Abstract

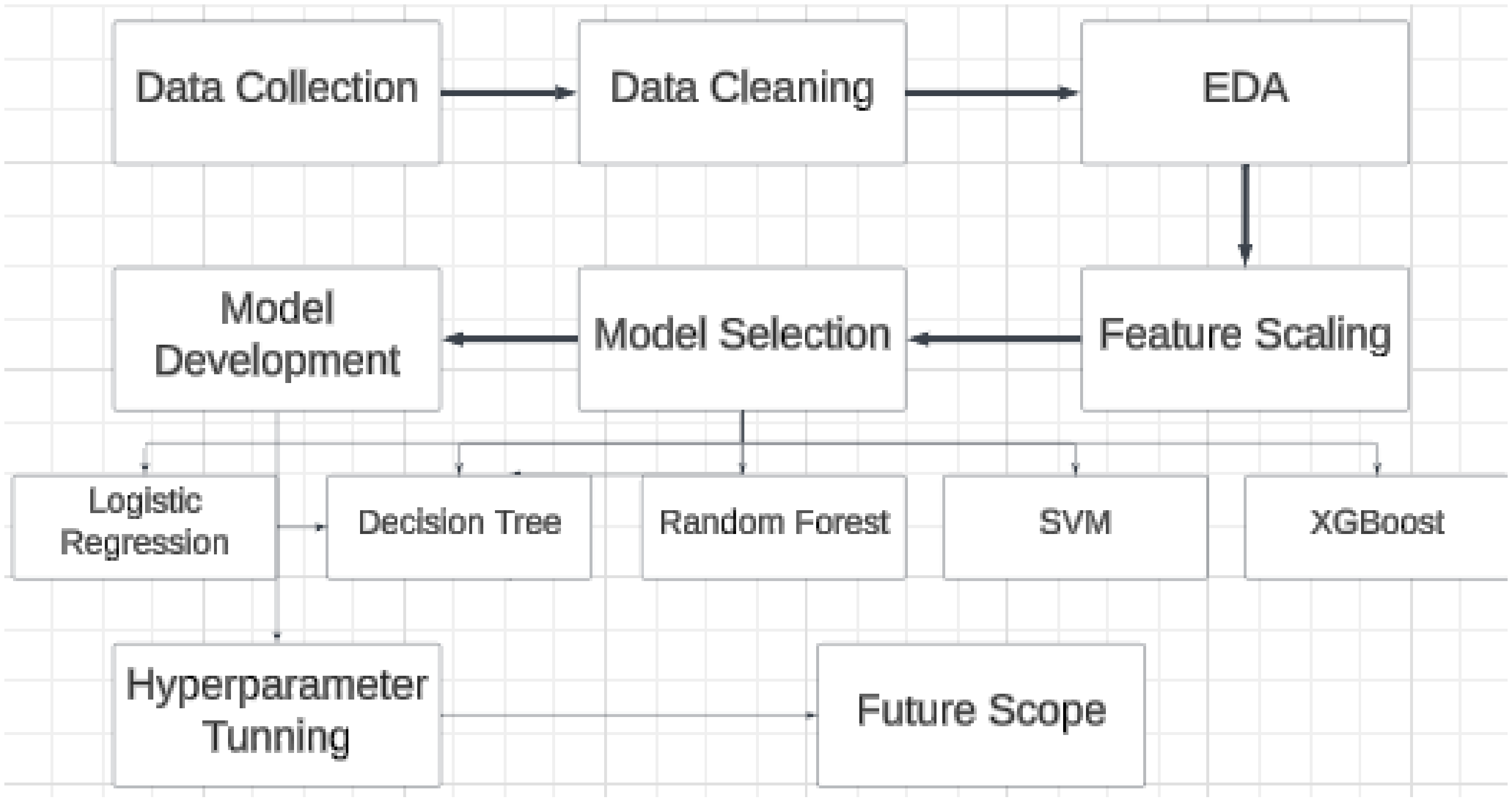
This project focuses on predicting the price range of mobile phones based on various hardware and feature specifications using machine learning techniques. The dataset consists of 21 features describing different attributes of mobile phones, such as battery capacity, RAM, screen resolution, internal memory, camera specifications, and network capabilities. The target variable, price_range, is a categorical label with four classes representing low, medium, high, and very high-cost mobile phones.

The project begins with **Exploratory Data Analysis (EDA)** to understand the relationships between features and the target variable. Key insights reveal that RAM, battery_power, px_width, and px_height are the most influential factors affecting the price range. Data preprocessing steps include handling missing values, removing duplicates, correcting inconsistencies in screen resolution data, and scaling numerical features using **StandardScaler**.

A variety of classification models are developed and evaluated, including **Logistic Regression, Decision Tree, Random Forest, XGBoost, Support Vector Classifier (SVC), and Naive Bayes**. Hyperparameter tuning is performed using **GridSearchCV** to optimize each model. The models are evaluated based on **accuracy, cross-validation scores, confusion matrices, and ROC-AUC curves**. The results indicate that **Logistic Regression** and **Support Vector Classifier (SVC)** achieves the highest accuracy of **97%**, with consistent cross-validation scores. Ensemble models like **Random Forest** and **XGBoost** also perform well, providing insights into feature importance. Key findings highlight that mobile phone with higher RAM, better battery capacity, and higher screen resolution tend to fall into higher price ranges.

Additionally, the study demonstrates that while features like RAM and battery power significantly influence the price range, factors such as camera resolution, clock speed, and dual SIM capability have a minimal impact. This insight can guide manufacturers in prioritizing features that offer the highest value to consumers. The combination of various machine learning models ensures robustness and accuracy in predictions, making this approach practical for real-world applications. Future work could incorporate additional features such as brand reputation, build quality, and customer reviews to further enhance model performance and applicability.

Workflow of the Project



The diagram outlines the key steps in a **Mobile Price Range Prediction** workflow. The process starts with **Data Collection** and **Data Cleaning** to ensure the dataset is accurate and reliable. Next, **Exploratory Data Analysis (EDA)** is performed to uncover patterns and insights, followed by **Feature Scaling** to standardize numerical values. The **Model Selection** stage involves choosing suitable machine learning algorithms, including **Logistic Regression, Decision Tree, Random Forest, SVM (Support Vector Machine), and XGBoost**. These models are then developed and optimized through **Hyperparameter Tuning** to improve performance. Once the models are evaluated, the findings lead to potential **Future Scope** improvements, such as incorporating additional features or exploring advanced techniques. This structured approach ensures comprehensive analysis, robust model development, and opportunities for enhancing predictive accuracy and applicability in real-world scenarios.

Required Libraries

The following libraries are essential for implementing the **Mobile Price Range Prediction** project. These libraries facilitate data handling, visualization, model development, and evaluation:

1. **pandas**: Used for data manipulation, cleaning, and analysis.
2. **numpy**: Supports numerical operations and array handling, which are crucial for data processing and mathematical computations.
3. **matplotlib**: A widely used library for creating static, animated, and interactive visualizations.
4. s**eaborn**: Built on top of matplotlib, this library provides enhanced statistical graphics and simplifies the creation of complex plots like heatmaps and bar plots.
5. **scikit-learn**: A comprehensive machine learning library offering tools for model development, preprocessing, and evaluation.
    - Key modules include:
        - **LogisticRegression**: For linear classification.
        - **RandomForestClassifier**: For ensemble-based classification.
        - **DecisionTreeClassifier**: For tree-based classification.
        - **SVC**: For support vector machine classification.
        - **GaussianNB**: For Naive Bayes classification.
        - **StandardScaler**: For feature scaling.
        - **GridSearchCV**: For hyperparameter tuning.
        - **train_test_split**: For splitting data into training and testing sets.
        - **accuracy_score, confusion_matrix, classification_report**: For model evaluation metrics.
6. **xgboost**: A high-performance library for gradient-boosted decision trees, used to improve prediction accuracy.
7. **warnings**: Used to suppress or manage runtime warnings for cleaner output during model execution.

Dataset Collection

Dataset Link : https://www.kaggle.com/code/sercanyesiloz/mobile-price-prediction

The dataset for the **Mobile Price Range Prediction** project was sourced from **Kaggle**, a widely recognized platform for data science and machine learning enthusiasts. Kaggle hosts a plethora of datasets contributed by the community and professionals, providing a valuable resource for data-driven projects. The specific dataset used in this project was obtained from the following link: Mobile Price Prediction by Sercan Yesiloz.

This dataset was chosen for its relevance in predicting mobile phone price categories based on hardware specifications, making it an ideal candidate for building classification models. The dataset's availability on Kaggle ensures that it has undergone initial vetting and validation by the community, ensuring a certain level of quality and reliability.

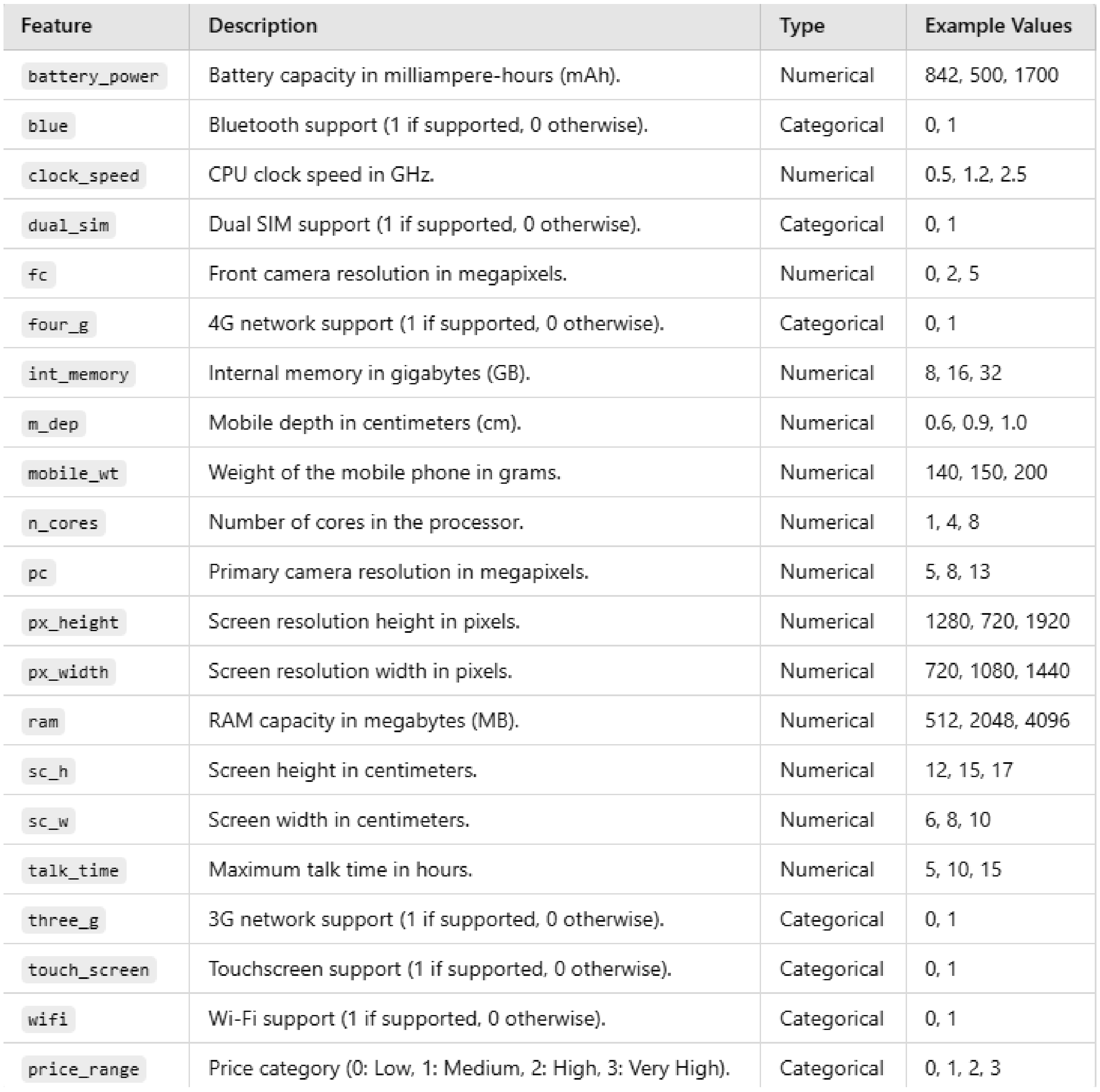
Why Kaggle Was Chosen:

1. **Accessibility**:
    - Kaggle provides free and easy access to datasets in CSV format, making it straightforward to download and integrate into projects.
2. **Community Validation**:
    - Datasets on Kaggle often come with user ratings, comments, and kernels (notebooks) showcasing different analysis techniques. This helps in understanding the data's quality and potential issues.
3. **Structured Format**:
    - The datasets on Kaggle are typically well-structured and labeled, reducing the effort needed for cleaning and preprocessing.
4. **Licensing and Usage**:
    - Kaggle datasets usually have clear licensing terms, making them suitable for educational, research, and personal projects without legal concerns.

The dataset was downloaded in CSV format, which facilitates seamless integration with Python libraries such as **pandas** for data manipulation and analysis. This ensured a smooth start to the project workflow, focusing on predictive modeling rather than extensive data collection efforts.

Dataset Overview

The dataset used in this project consists of mobile phone specifications and their corresponding price ranges, making it ideal for predictive modeling. The dataset contains 21 features describing various hardware and software attributes of mobile phones. The target variable, price_range, is a categorical label indicating the price category of the phone. Below is a detailed contextual table summarizing the dataset features:



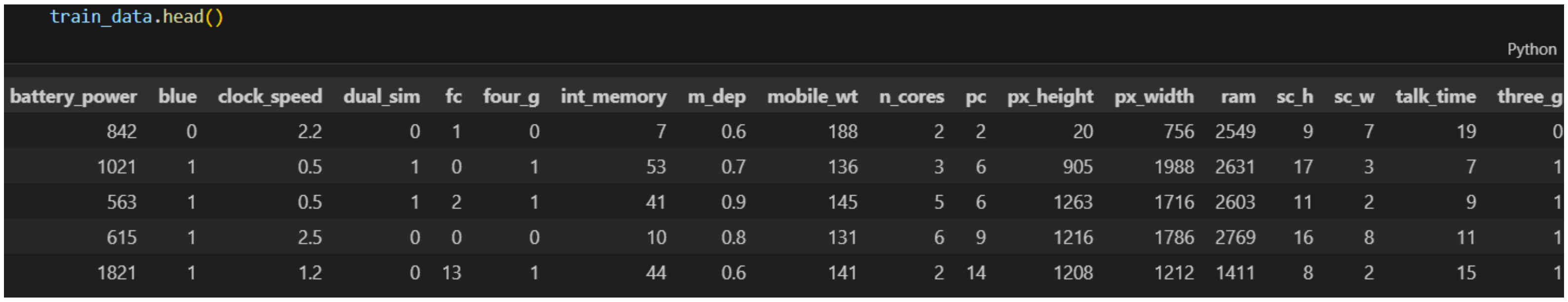
Feature	Description	Type	Example Values
`battery_power`	Battery capacity in milliampere-hours (mAh).	Numerical	842, 500, 1700
`blue`	Bluetooth support (1 if supported, 0 otherwise).	Categorical	0, 1
`clock_speed`	CPU clock speed in GHz.	Numerical	0.5, 1.2, 2.5
`dual_sim`	Dual SIM support (1 if supported, 0 otherwise).	Categorical	0, 1
`fc`	Front camera resolution in megapixels.	Numerical	0, 2, 5
`four_g`	4G network support (1 if supported, 0 otherwise).	Categorical	0, 1
`int_memory`	Internal memory in gigabytes (GB).	Numerical	8, 16, 32
`m_dep`	Mobile depth in centimeters (cm).	Numerical	0.6, 0.9, 1.0
`mobile_wt`	Weight of the mobile phone in grams.	Numerical	140, 150, 200
`n_cores`	Number of cores in the processor.	Numerical	1, 4, 8
`pc`	Primary camera resolution in megapixels.	Numerical	5, 8, 13
`px_height`	Screen resolution height in pixels.	Numerical	1280, 720, 1920
`px_width`	Screen resolution width in pixels.	Numerical	720, 1080, 1440
`ram`	RAM capacity in megabytes (MB).	Numerical	512, 2048, 4096
`sc_h`	Screen height in centimeters.	Numerical	12, 15, 17
`sc_w`	Screen width in centimeters.	Numerical	6, 8, 10
`talk_time`	Maximum talk time in hours.	Numerical	5, 10, 15
`three_g`	3G network support (1 if supported, 0 otherwise).	Categorical	0, 1
`touch_screen`	Touchscreen support (1 if supported, 0 otherwise).	Categorical	0, 1
`wifi`	Wi-Fi support (1 if supported, 0 otherwise).	Categorical	0, 1
`price_range`	Price category (0: Low, 1: Medium, 2: High, 3: Very High).	Categorical	0, 1, 2, 3

Dataset Size : 2,000 rows × 21 columns

- Now let’s import the dataset

```
train_data = pd.read_csv("train_data.csv")
test_data = pd.read_csv("test_data.csv")
```

Output:



```
train_data.head()
```

Python

battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	pc	px_height	px_width	ram	sc_h	sc_w	talk_time	three_g
842	0	2.2	0	1	0	7	0.6	188	2	2	20	756	2549	9	7	19	0
1021	1	0.5	1	0	1	53	0.7	136	3	6	905	1988	2631	17	3	7	1
563	1	0.5	1	2	1	41	0.9	145	5	6	1263	1716	2603	11	2	9	1
615	1	2.5	0	0	0	10	0.8	131	6	9	1216	1786	2769	16	8	11	1
1821	1	1.2	0	13	1	44	0.6	141	2	14	1208	1212	1411	8	2	15	1

- Now let’s Check the size of the dataset

```
print(f'Train Shape: - {train_data.shape}')
print(f'Test Shape: - {test_data.shape}')
```

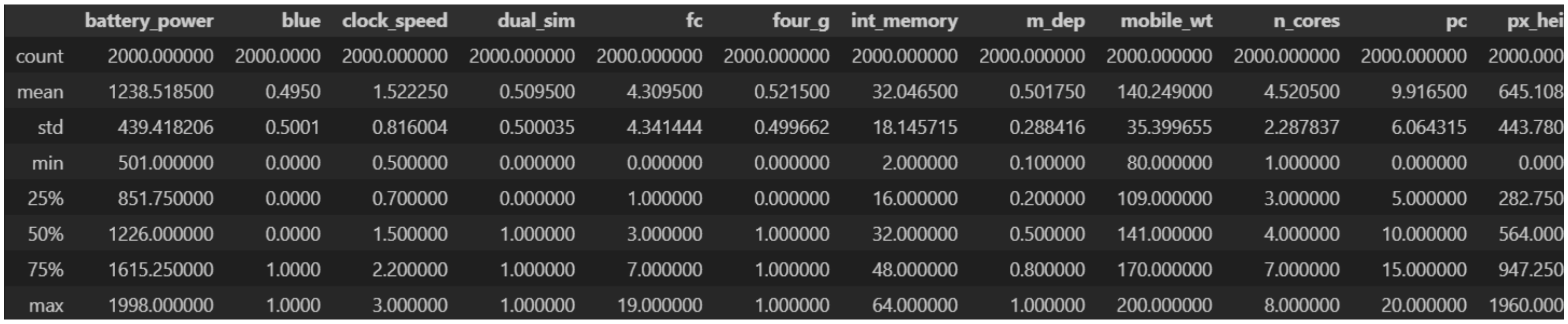
Output:

```
Train Shape: - (2000, 21)
Test Shape: - (1000, 21)
```

- Now Let’s check the summary of the dataset

```
train_data.describe()
```

Output:

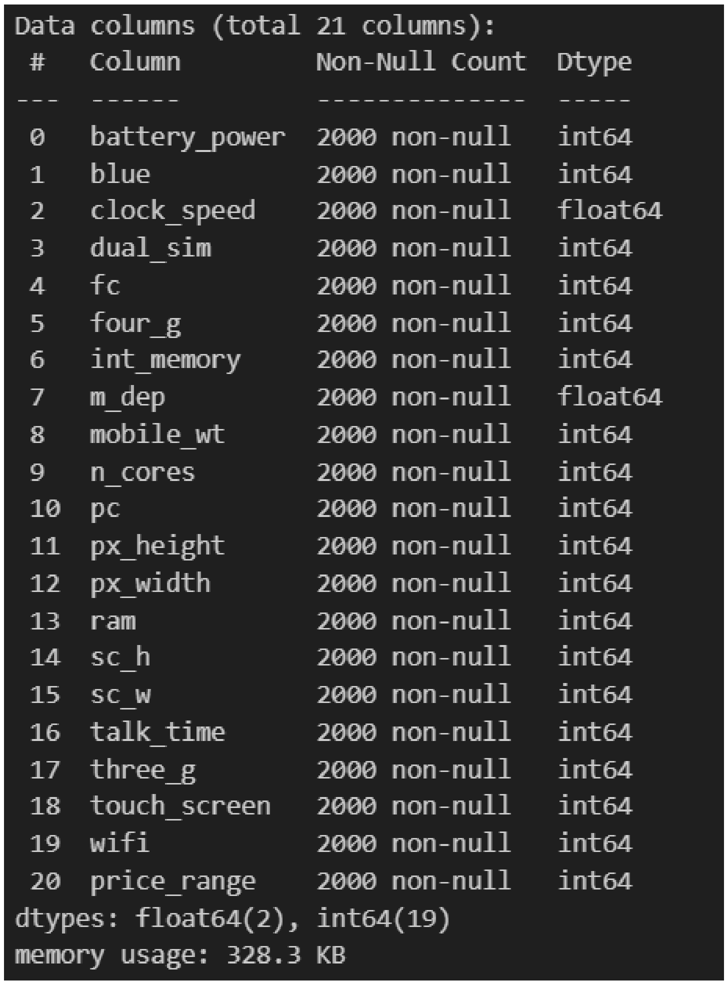


	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	pc	px_hei
count	2000.000000	2000.0000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000
mean	1238.518500	0.4950	1.522250	0.509500	4.309500	0.521500	32.046500	0.501750	140.249000	4.520500	9.916500	645.108
std	439.418206	0.5001	0.816004	0.500035	4.341444	0.499662	18.145715	0.288416	35.399655	2.287837	6.064315	443.780
min	501.000000	0.0000	0.500000	0.000000	0.000000	0.000000	2.000000	0.100000	80.000000	1.000000	0.000000	0.000
25%	851.750000	0.0000	0.700000	0.000000	1.000000	0.000000	16.000000	0.200000	109.000000	3.000000	5.000000	282.750
50%	1226.000000	0.0000	1.500000	1.000000	3.000000	1.000000	32.000000	0.500000	141.000000	4.000000	10.000000	564.000
75%	1615.250000	1.0000	2.200000	1.000000	7.000000	1.000000	48.000000	0.800000	170.000000	7.000000	15.000000	947.250
max	1998.000000	1.0000	3.000000	1.000000	19.000000	1.000000	64.000000	1.000000	200.000000	8.000000	20.000000	1960.000

- To Check the datatypes of the features

```
train_data.info()
```

Output:



```
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   battery_power  2000 non-null   int64  
 1   blue           2000 non-null   int64  
 2   clock_speed    2000 non-null   float64
 3   dual_sim       2000 non-null   int64  
 4   fc             2000 non-null   int64  
 5   four_g         2000 non-null   int64  
 6   int_memory     2000 non-null   int64  
 7   m_dep          2000 non-null   float64
 8   mobile_wt      2000 non-null   int64  
 9   n_cores        2000 non-null   int64  
 10  pc             2000 non-null   int64  
 11  px_height      2000 non-null   int64  
 12  px_width       2000 non-null   int64  
 13  ram            2000 non-null   int64  
 14  sc_h           2000 non-null   int64  
 15  sc_w           2000 non-null   int64  
 16  talk_time      2000 non-null   int64  
 17  three_g        2000 non-null   int64  
 18  touch_screen   2000 non-null   int64  
 19  wifi           2000 non-null   int64  
 20  price_range    2000 non-null   int64  
dtypes: float64(2), int64(19)
memory usage: 328.3 KB
```

The dataset contains **21 columns** with **2,000 non-null rows**, indicating there are no missing values. The features include **19 integer type (**int64**) columns** and **2 float type (**float64**) columns** (clock_speed and m_dep). The dataset encompasses mobile phone specifications such as battery_power, ram, px_height, px_width, and categorical indicators like blue, four_g, and wifi. The target variable, price_range, classifies phones into four price categories. The total memory usage is **328.3 KB**, making it efficient for processing and analysis.

Data Cleaning

In the **Mobile Price Range Prediction** project, the following data cleaning steps were performed to ensure the dataset's quality and reliability:

1. **Handling the Missing Values:**

```
train_data.isnull().sum()
```

Output:

```
battery_power    0
blue             0
clock_speed      0
dual_sim         0
fc               0
four_g           0
int_memory       0
m_dep            0
mobile_wt        0
n_cores          0
pc               0
px_height        0
px_width         0
ram              0
sc_h             0
sc_w             0
talk_time        0
three_g          0
touch_screen     0
wifi             0
price_range      0
dtype: int64
```

The output confirms that there are **no missing values** in any of the dataset columns.

Each feature, including battery_power, ram, px_height, and price_range, has **0 missing entries**. This ensures the dataset is complete and ready for analysis without the need for imputation or handling of null values.

2. **Duplicate Rows:**

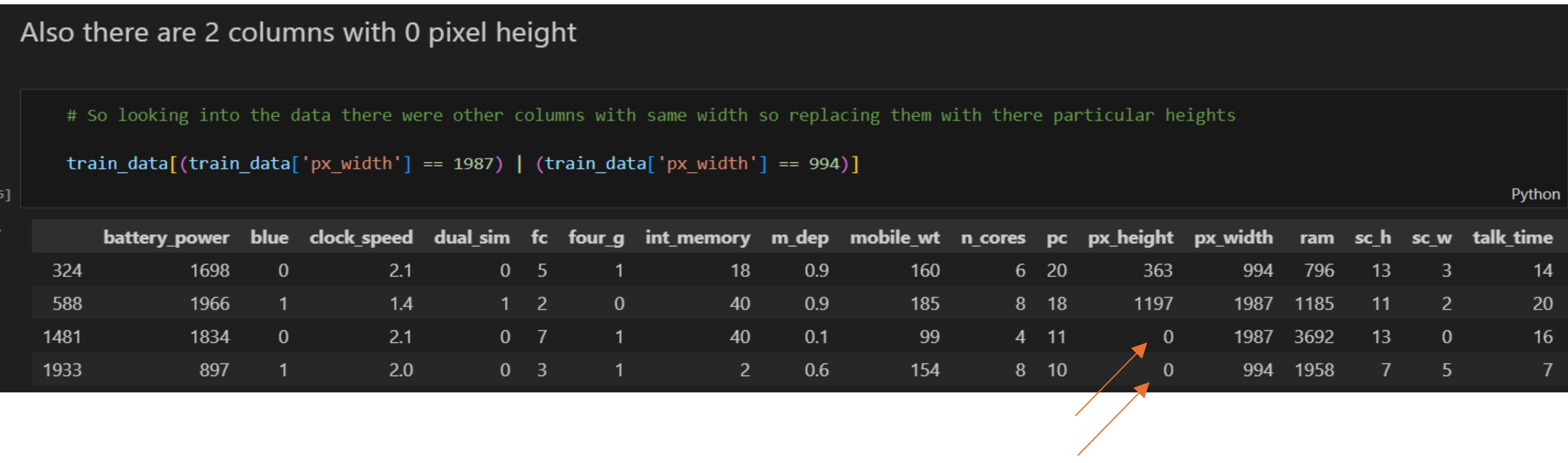
```
# Check for duplicate rows
duplicate_rows = train_data.duplicated()

# Count the number of duplicate rows
num_duplicates = duplicate_rows.sum()

# Print the result
print(f"Number of duplicate rows: {num_duplicates}")
```

Output:

```
Number of duplicate rows: 0
```

Our Dataset has no duplicate rows.

3. **Inconsistent Data Correction**:
    - Identified inconsistent values in screen resolution (px_height and px_width).
    - Corrected specific rows where px_height was **0** while px_width had valid values by replacing them with appropriate heights.

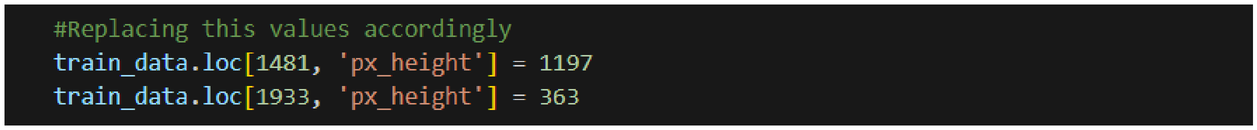


Also there are 2 columns with 0 pixel height

```
# So looking into the data there were other columns with same width so replacing them with there particular heights

train_data[(train_data['px_width'] == 1987) | (train_data['px_width'] == 994)]
```

Python

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	pc	px_height	px_width	ram	sc_h	sc_w	talk_time
324	1698	0	2.1	0	5	1	18	0.9	160	6	20	363	994	796	13	3	14
588	1966	1	1.4	1	2	0	40	0.9	185	8	18	1197	1987	1185	11	2	20
1481	1834	0	2.1	0	7	1	40	0.1	99	4	11	0	1987	3692	13	0	16
1933	897	1	2.0	0	3	1	2	0.6	154	8	10	0	994	1958	7	5	7



```
#Replacing this values accordingly
train_data.loc[1481, 'px_height'] = 1197
train_data.loc[1933, 'px_height'] = 363
```

The dataset contained **two rows** with a px_height value of **0**, which is invalid. Upon examining these rows, corresponding px_width values of **1987** and **994** were identified. The heights were corrected by replacing the px_height values with appropriate values of **1197** and **363** to maintain data consistency and accuracy.

With all data cleaning steps completed, including handling missing values, removing duplicates, correcting invalid px_height values, and removing rows with **0 screen width**, the dataset is now clean and consistent. All features are properly formatted, and there are no anomalies or inconsistencies left. This ensures the data is ready for further analysis. We can now confidently proceed with **Exploratory Data Analysis (EDA)** to uncover patterns, visualize relationships between features, and gain valuable insights for model development.

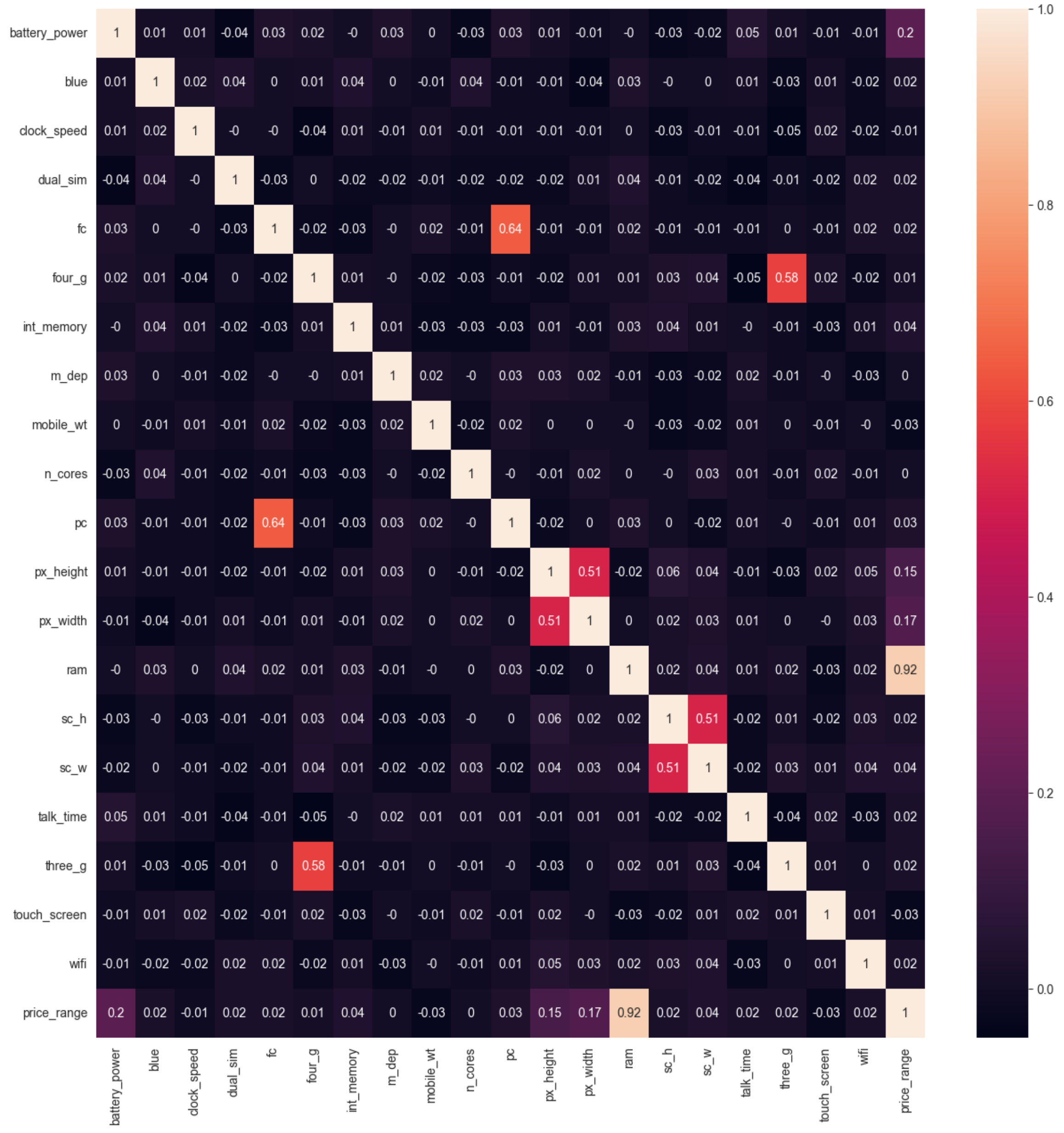
Exploratory Data Analysis

Exploratory Data Analysis (EDA) helps uncover patterns, trends, and insights within the dataset. Here are the steps and key aspects you can cover in your EDA section for the Mobile Price Range Prediction project:

1. Correlation Analysis

```
fig, ax = plt.subplots(1,1, figsize = (15,15))
sns.heatmap(np.round(train_data.corr(), 2), annot=True, ax = ax)
```

Output:



This **correlation heatmap** displays the relationships between features in the dataset, with correlation values ranging from **-1 to 1**. Lighter colors represent stronger correlations, while darker colors indicate weaker or no correlation. Key observations include:

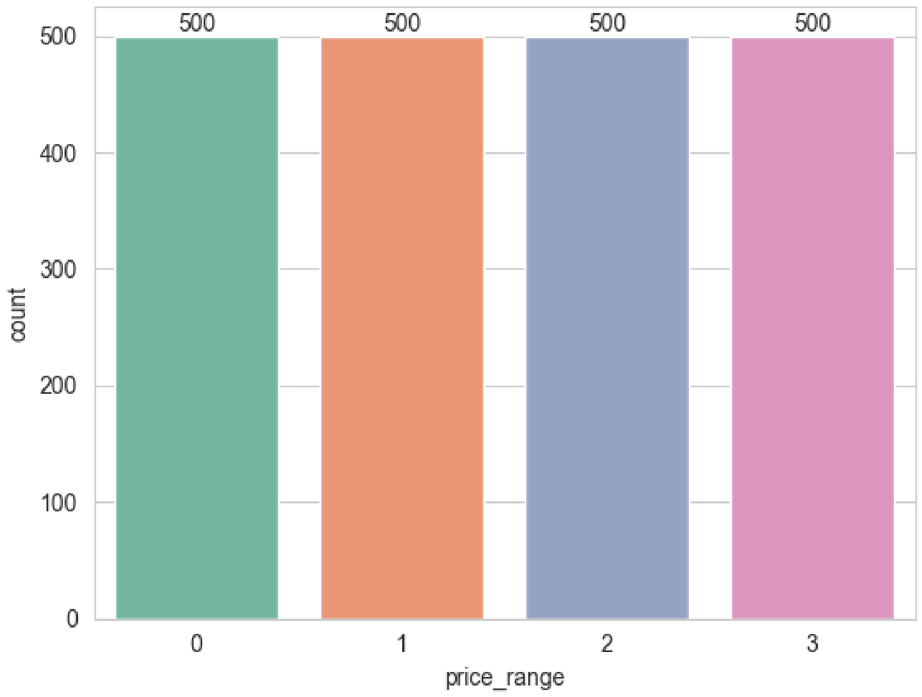
1. ram shows the strongest correlation with the target variable price_range (**0.92**), indicating RAM significantly influences the price range.
2. px_width and px_height are moderately correlated with each other (**0.51**) and price_range (**0.17** and **0.15**).
3. fc **(front camera)** and pc **(primary camera)** have a notable correlation (**0.64**), suggesting a relationship between front and primary camera resolutions.
4. Most other features have low or negligible correlation with price_range.

These insights help identify the most important features for model development.

2. **Univariate Analysis**
    - Price Range

```
ax = sns.countplot(x = train_data['price_range'])
ax.bar_label(ax.containers[0])
```

Output:



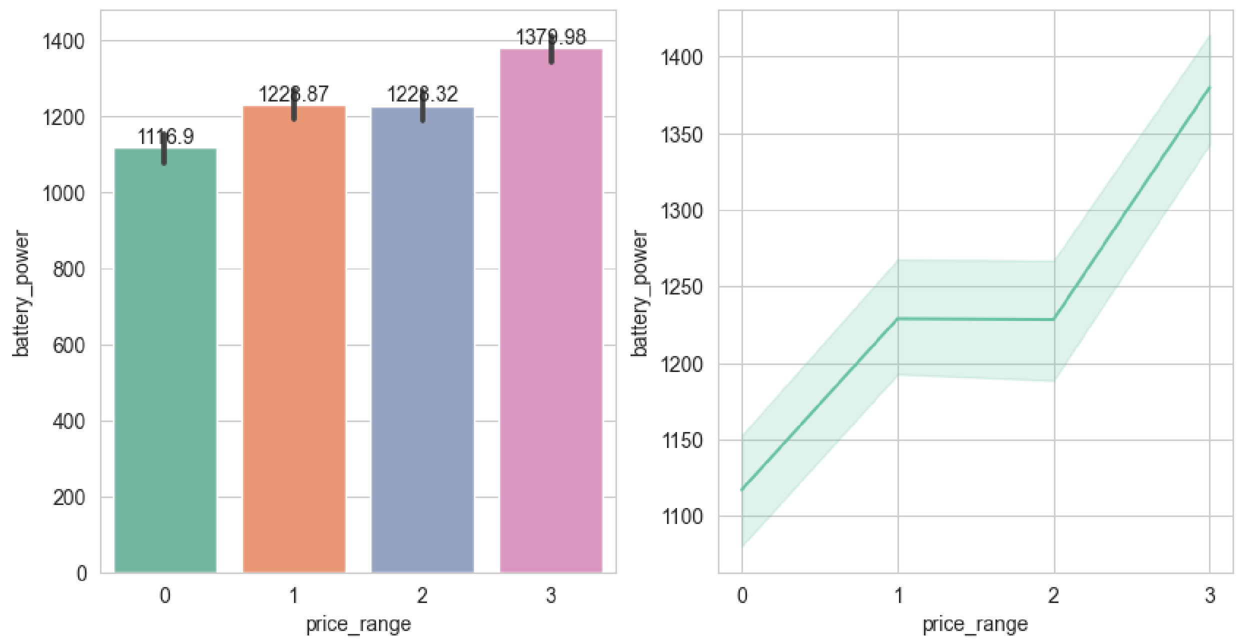
This bar plot shows the distribution of the target variable, price_range, with each category (0, 1, 2, 3) containing **500 entries**. The balanced distribution ensures that the dataset does not suffer from class imbalance, making it suitable for training machine learning models.

3. Now Visualization of every feature with the target Variable

a) Price Range Vs Battery Power

```python
fig, ax  = plt.subplots(1, 2, figsize = (10,5))
sns.barplot(x = train_data['price_range'], y = train_data['battery_power'], ax = ax[0])
ax[0].bar_label(ax[0].containers[0])

sns.lineplot(x = train_data['price_range'], y = train_data['battery_power'], ax = ax[1])
ax[1].set_xticks([0, 1, 2, 3])
```

Output:

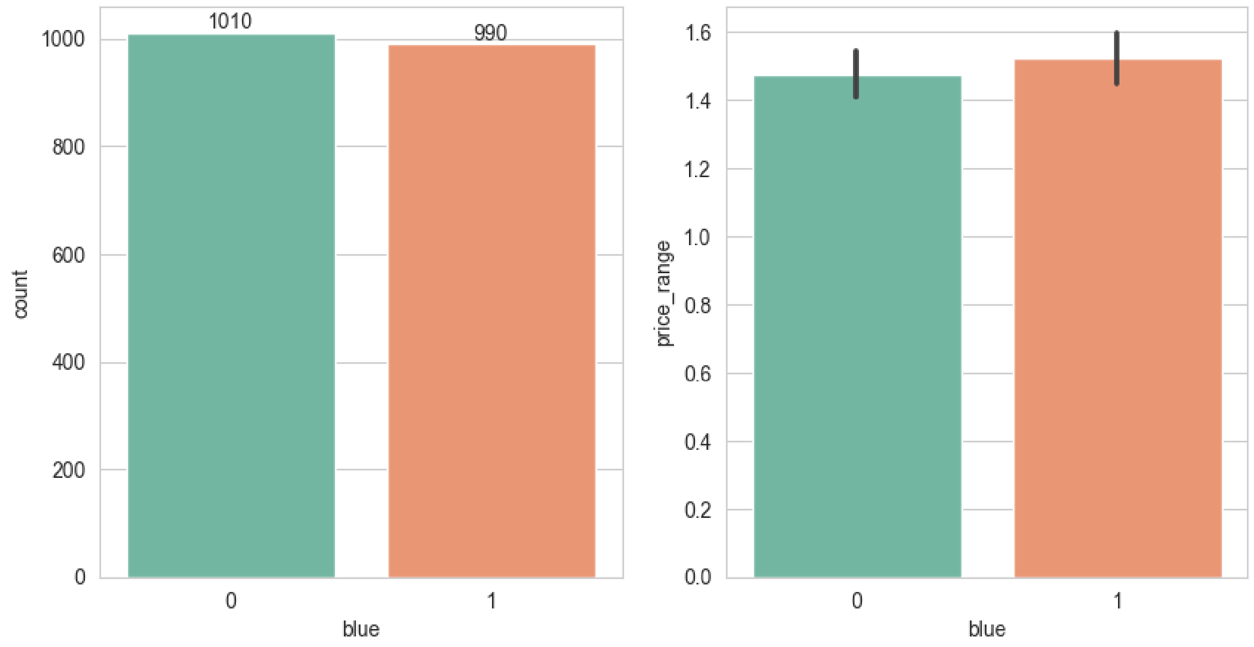


The visualizations illustrate the relationship between battery_power and price_range. The bar plot on the left shows that higher price ranges tend to have higher average battery power, with values increasing from 1116.9 mAh for the lowest price range (0) to 1379.98 mAh for the highest price range (3). The line plot on the right confirms this trend, showing a steady increase in battery power with higher price ranges, along with a shaded area indicating variability. This suggests that battery capacity is a key factor influencing mobile phone pricing, with more expensive phones generally offering better battery performance.

b) Bluetooth Vs Price range

```
fig, ax = plt.subplots(1, 2, figsize = (10, 5))

sns.countplot(x = train_data['blue'], ax = ax[0])
ax[0].bar_label(ax[0].containers[0])

sns.barplot(x = train_data['blue'], y = train_data['price_range'])
```

Output:

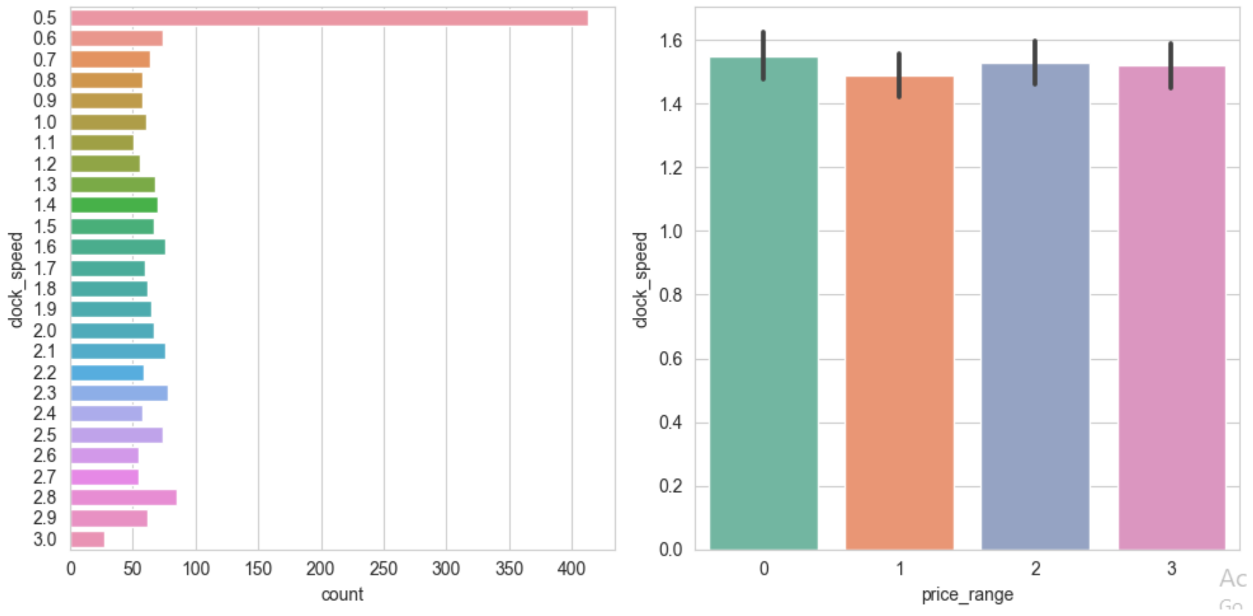


The visualizations compare the presence of Bluetooth (blue) in mobile phones and its relationship with the price_range. The bar plot on the left shows that the dataset contains 1,010 phones without Bluetooth (blue = 0) and 990 phones with Bluetooth (blue = 1), indicating a near-equal distribution. The bar plot on the right illustrates that the average price range for phones with Bluetooth is slightly higher than those without. This suggests that phones with Bluetooth functionality tend to belong to higher price categories. However, the difference is marginal, indicating that Bluetooth support does not significantly impact the price range.

c) Clock Speed Vs Price Range

```
fig, ax = plt.subplots(1, 2, figsize = (10, 5))
sns.countplot(y = train_data['clock_speed'], ax = ax[0])

sns.barplot(x = train_data['price_range'], y = train_data['clock_speed'], ax = ax[1])
ax[1].set_xticks([0,1,2,3])
plt.tight_layout()
```

Output:

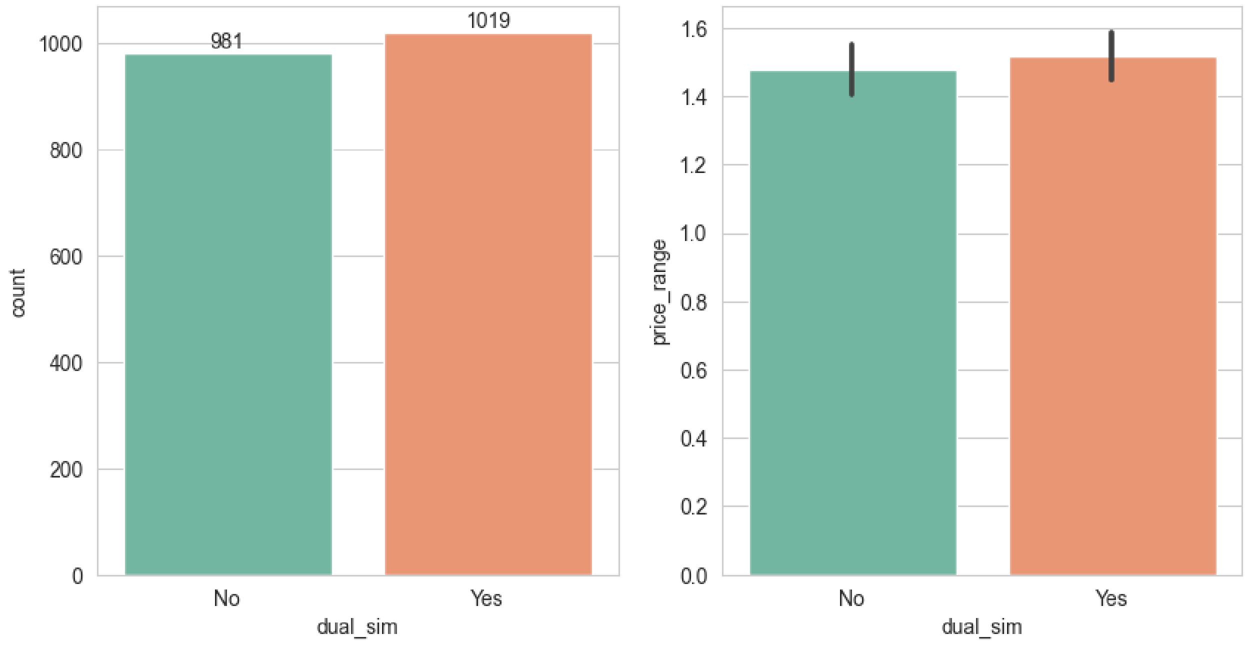


The visualizations analyze the distribution of clock_speed and its relationship with price_range. The **bar plot on the left** shows that most mobile phones have a clock speed of **0.5 GHz**, with fewer phones having higher clock speeds. The **bar plot on the right** indicates that the average clock speed remains relatively consistent across all price ranges, around **1.5 GHz**. This suggests that clock speed does not significantly influence the price range of mobile phones, as higher price categories do not show a notable increase in clock speed.

d) Dual Sim Vs Price Range

```
fig, ax = plt.subplots(1, 2, figsize = (10, 5))
sns.countplot(x = train_data['dual_sim'], ax = ax[0])
ax[0].bar_label(ax[0].containers[0])
ax[0].set_xticks([0,1], ['No', 'Yes'])

sns.barplot(x = train_data['dual_sim'], y = train_data['price_range'], ax = ax[1])
ax[1].set_xticks([0,1], ['No', 'Yes'])
```

Output:

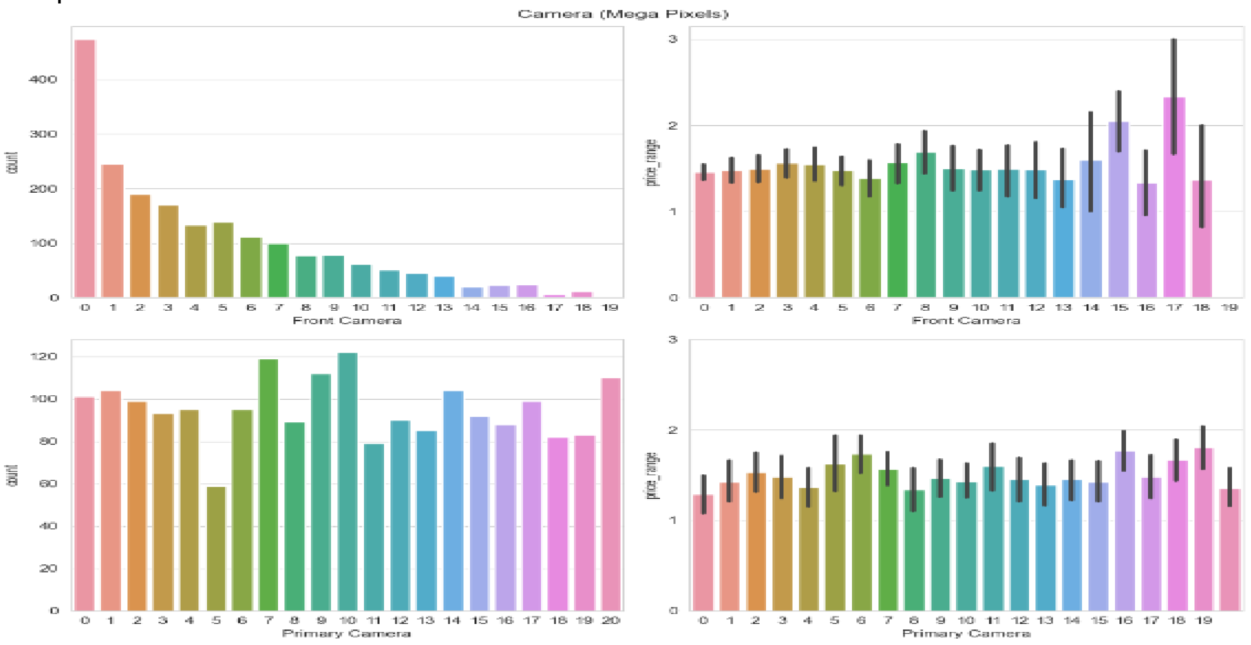


The visualizations analyze the dual_sim feature and its relationship with price_range. The **bar plot on the left** shows that **981 phones do not support dual SIM**, while **1,019 phones support dual SIM** functionality, indicating a balanced distribution. The **bar plot on the right** shows that the average price range for dual SIM phones is slightly higher than for single SIM phones. However, the difference is minimal, suggesting that the presence of dual SIM functionality does not significantly influence the mobile phone's price range.

e) Camera Vs Price Range

```
fig, ax = plt.subplots(2, 2, figsize = (10, 10))

sns.countplot(x = train_data['fc'], ax = ax[0][0])
ax[0][0].set_xlabel("Front Camera")

sns.barplot(x = train_data['fc'], y = train_data['price_range'],ax = ax[0][1])
ax[0][1].set_xticks(np.arange(0, 20))
ax[0][1].set_yticks([0, 1, 2, 3])
ax[0][1].set_xlabel("Front Camera")

sns.countplot(x = train_data['pc'], ax = ax[1][0])
ax[1][0].set_xlabel("Primary Camera")

sns.barplot(x = train_data['pc'], y = train_data['price_range'],ax = ax[1][1])
ax[1][1].set_xticks(np.arange(0, 20))
ax[1][1].set_yticks([0, 1, 2, 3])
ax[1][1].set_xlabel("Primary Camera")

fig.suptitle("Camera (Mega Pixels)")
plt.tight_layout()
```

Output:

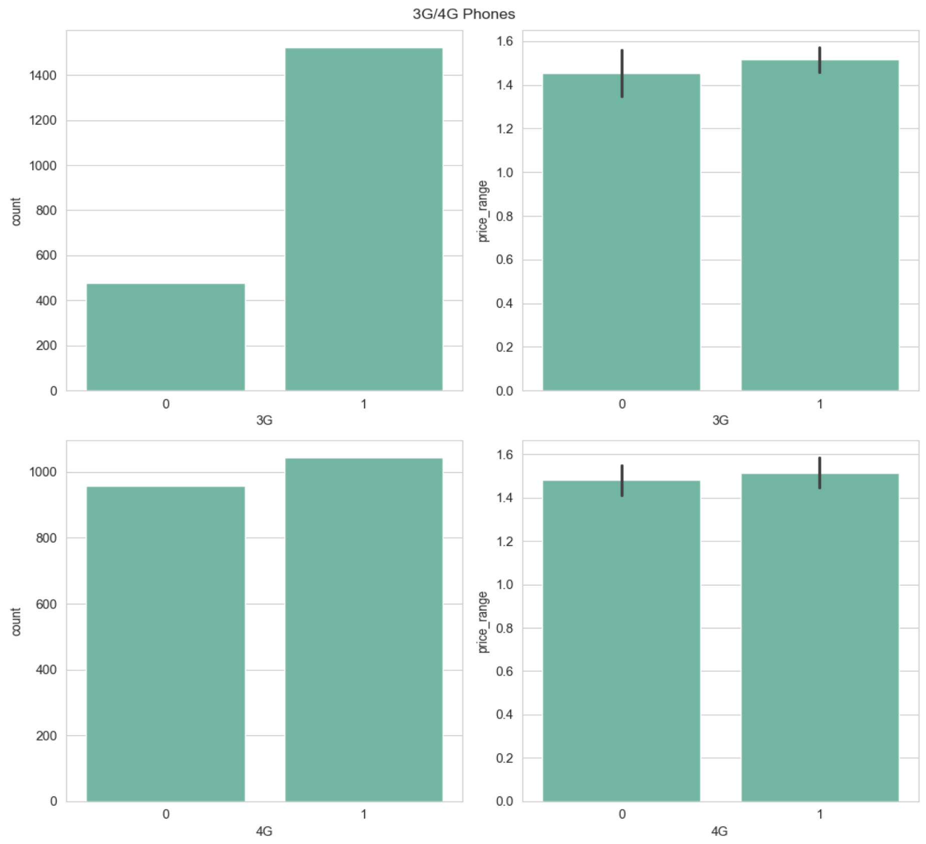


The visualizations analyze the **front camera** and **primary camera** features in relation to price_range. The **top-left plot** shows that many phones have **0 MP** front cameras, with fewer phones having higher megapixel counts. The **top-right plot** indicates that higher front camera resolutions are slightly associated with higher price ranges. The **bottom-left plot** shows a more even distribution of primary camera resolutions, with peaks around **7 MP and 10 MP**. The **bottom-right plot** suggests that the primary camera resolution does not significantly influence the price range. Overall, while higher camera resolutions are present in higher price ranges, their impact on pricing is minimal.

g) Phone Generation Vs Price range

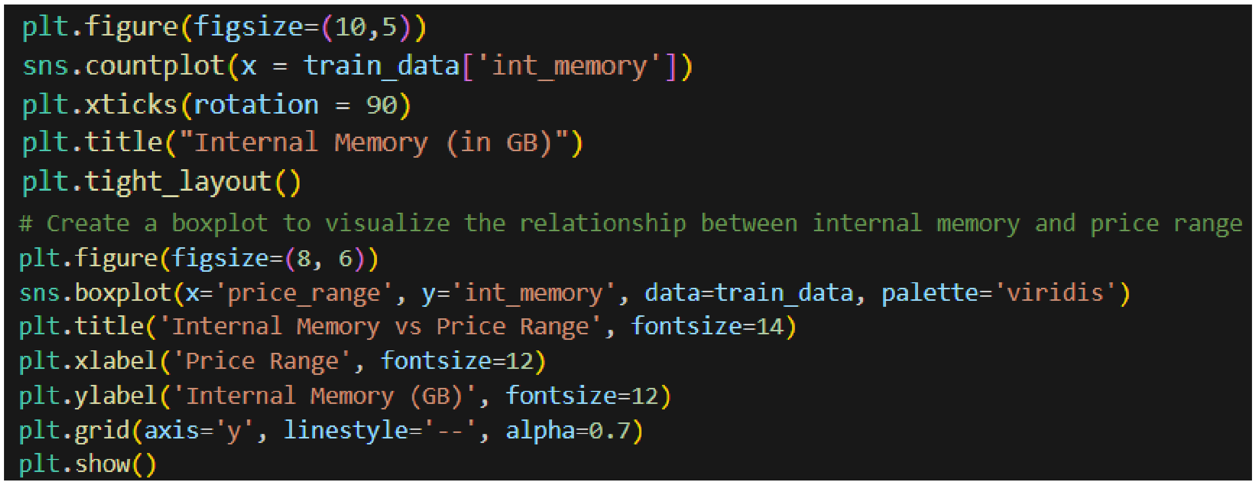
```
fig, ax = plt.subplots(2, 2, figsize = (10, 10))

sns.countplot(x = train_data['three_g'], ax = ax[0][0])
ax[0][0].set_xlabel("3G")

sns.barplot(x = train_data['three_g'], y = train_data['price_range'],ax = ax[0][1])
ax[0][1].set_xlabel("3G")

sns.countplot(x = train_data['four_g'], ax = ax[1][0])
ax[1][0].set_xlabel("4G")

sns.barplot(x = train_data['four_g'], y = train_data['price_range'],ax = ax[1][1])
ax[1][1].set_xlabel("4G")

fig.suptitle("3G/4G Phones")
plt.tight_layout()
```

Output:



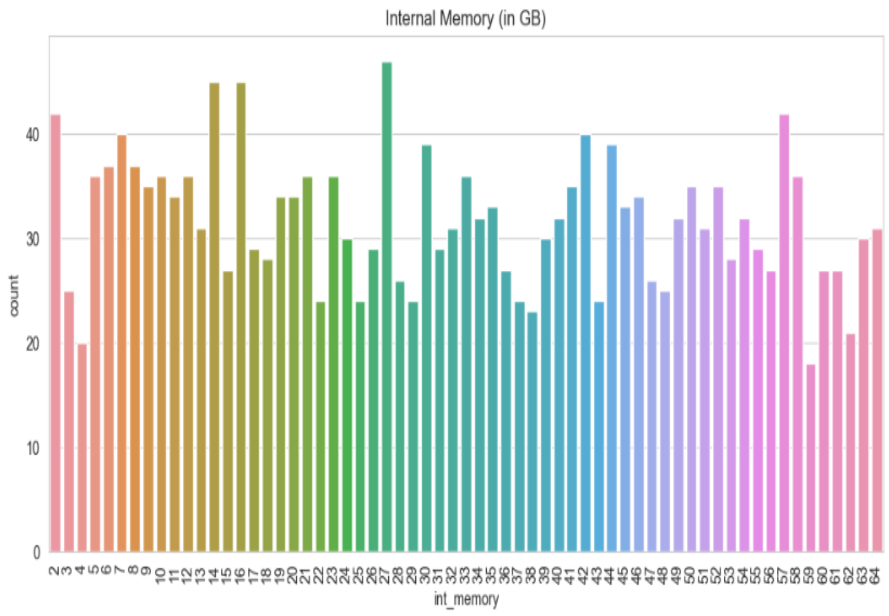
The visualizations compare the presence of **3G** and **4G** capabilities in mobile phones with their corresponding **price ranges**. The **top-left plot** shows that most phones support **3G**, while the **bottom-left plot** shows nearly even distribution for **4G** support. The bar plots on the right indicate that **3G** and **4G** capabilities have minimal impact on the average price range, suggesting that these features do not significantly influence the pricing of mobile phones.

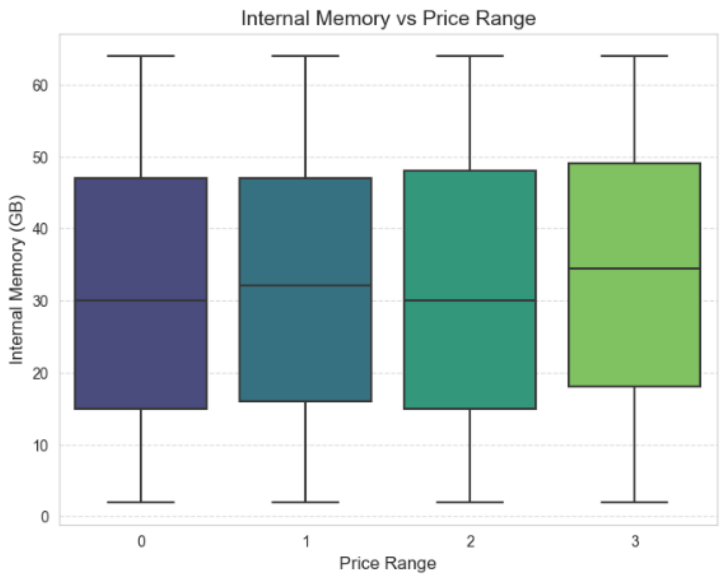
h) Internal Memory Vs Price Range



```
plt.figure(figsize=(10,5))
sns.countplot(x = train_data['int_memory'])
plt.xticks(rotation = 90)
plt.title("Internal Memory (in GB)")
plt.tight_layout()
# Create a boxplot to visualize the relationship between internal memory and price range
plt.figure(figsize=(8, 6))
sns.boxplot(x='price_range', y='int_memory', data=train_data, palette='viridis')
plt.title('Internal Memory vs Price Range', fontsize=14)
plt.xlabel('Price Range', fontsize=12)
plt.ylabel('Internal Memory (GB)', fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

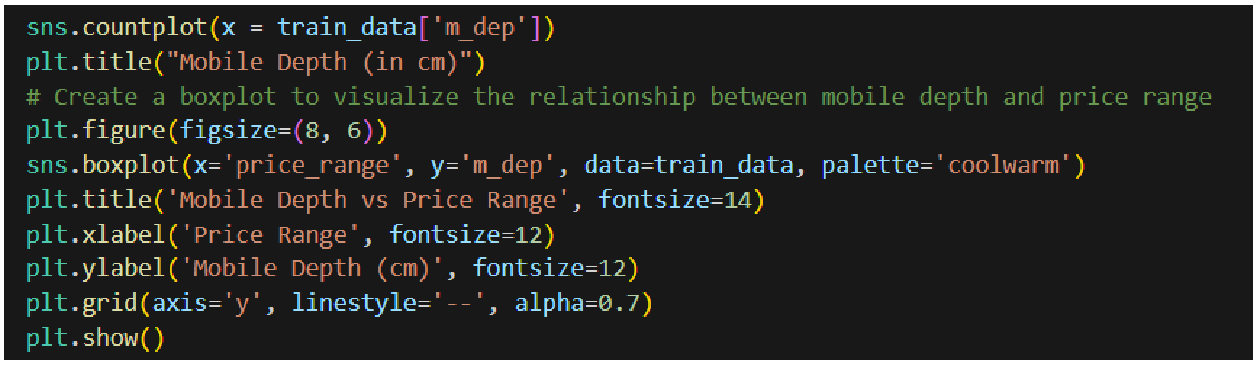
Output:





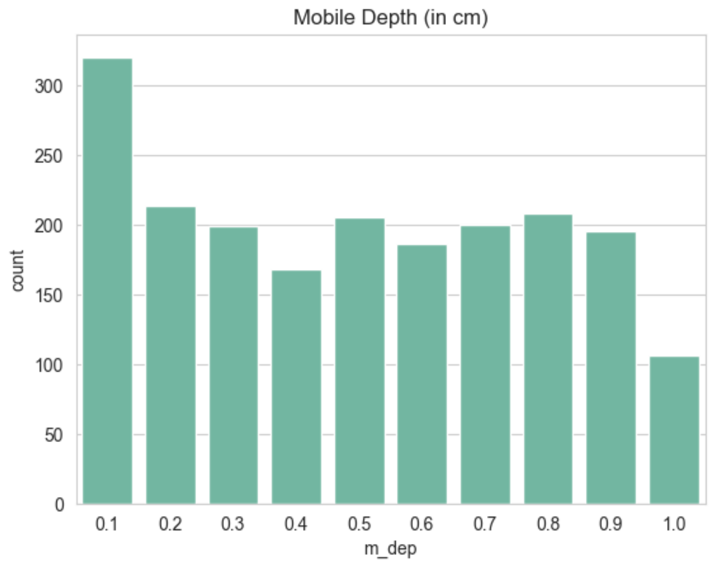
The visualizations explore the distribution of **internal memory (in GB)** and its relationship with **price range**. The **bar plot on the left** shows a wide range of internal memory values, with peaks around **14 GB, 27 GB, and 54 GB**. The **box plot on the right** indicates that higher price ranges tend to have higher median internal memory, but the variability within each price range is significant. While more expensive phones generally offer greater storage capacity, there are phones in lower price ranges that also provide relatively high internal memory, reflecting a diverse range of options across all price categories.

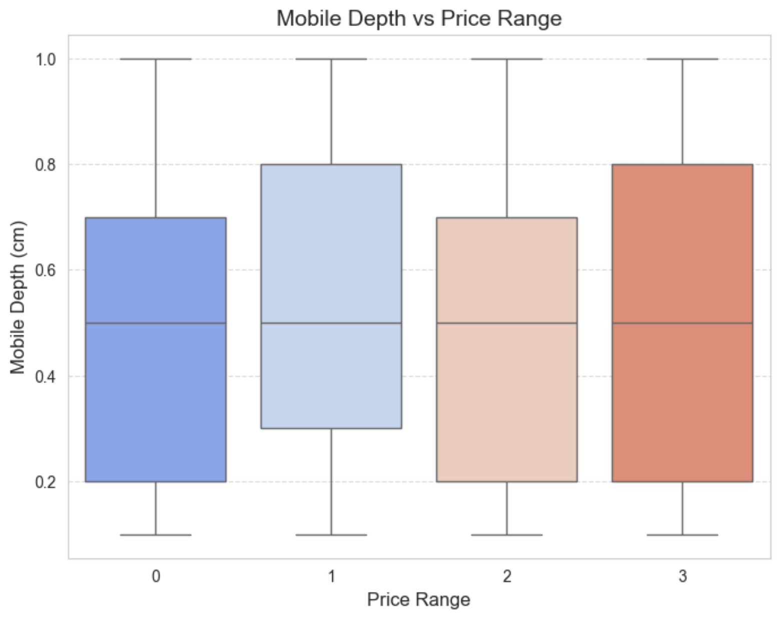
i) Depth Vs Price Range



```
sns.countplot(x = train_data['m_dep'])
plt.title("Mobile Depth (in cm)")
# Create a boxplot to visualize the relationship between mobile depth and price range
plt.figure(figsize=(8, 6))
sns.boxplot(x='price_range', y='m_dep', data=train_data, palette='coolwarm')
plt.title('Mobile Depth vs Price Range', fontsize=14)
plt.xlabel('Price Range', fontsize=12)
plt.ylabel('Mobile Depth (cm)', fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Output:



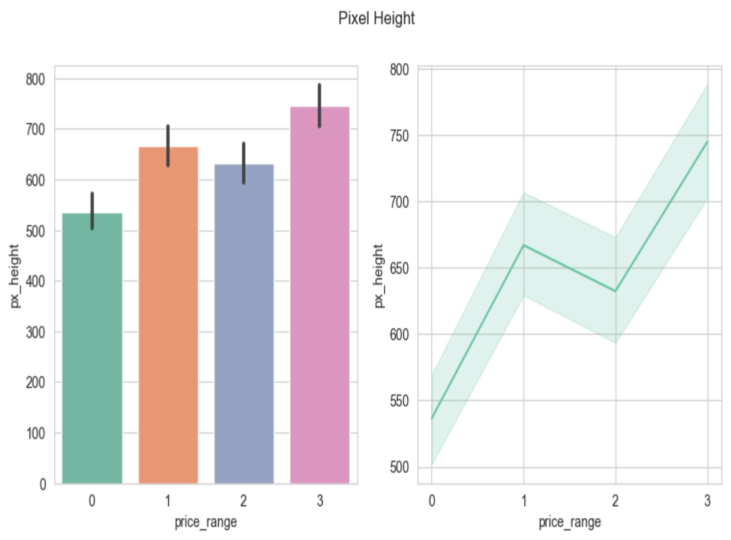


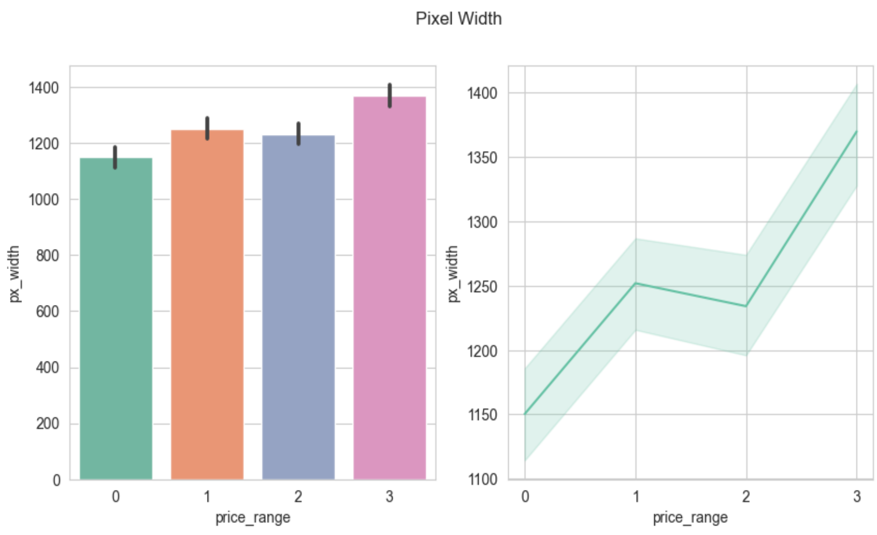
The visualizations explore mobile depth (in cm) and its relationship with price range. The bar plot on the left shows that mobile depth values are distributed between 0.1 cm to 1.0 cm, with a peak at 0.1 cm. The box plot on the right indicates that mobile depth remains consistent across all price ranges, with the median depth hovering around 0.5 cm. There is no significant variation in mobile depth with respect to the price range, suggesting that mobile depth does not strongly influence the pricing of mobile phones.

j) Pixel Height and Width Vs Price Range

```
fig, ax = plt.subplots(1, 2, figsize = (10, 5))

fig.suptitle("Pixel Height")

sns.barplot(x = train_data['price_range'], y = train_data['px_height'], ax = ax[0])

sns.lineplot(x = train_data['price_range'], y = train_data['px_height'], ax = ax[1])
ax[1].set_xticks([0,1,2,3])

fig, ax = plt.subplots(1, 2, figsize = (10, 5))

fig.suptitle("Pixel Width")

sns.barplot(x = train_data['price_range'], y = train_data['px_width'], ax = ax[0])

sns.lineplot(x = train_data['price_range'], y = train_data['px_width'], ax = ax[1])
ax[1].set_xticks([0,1,2,3])
ax[1].set_xticks([0,1,2,3])
```

Output:



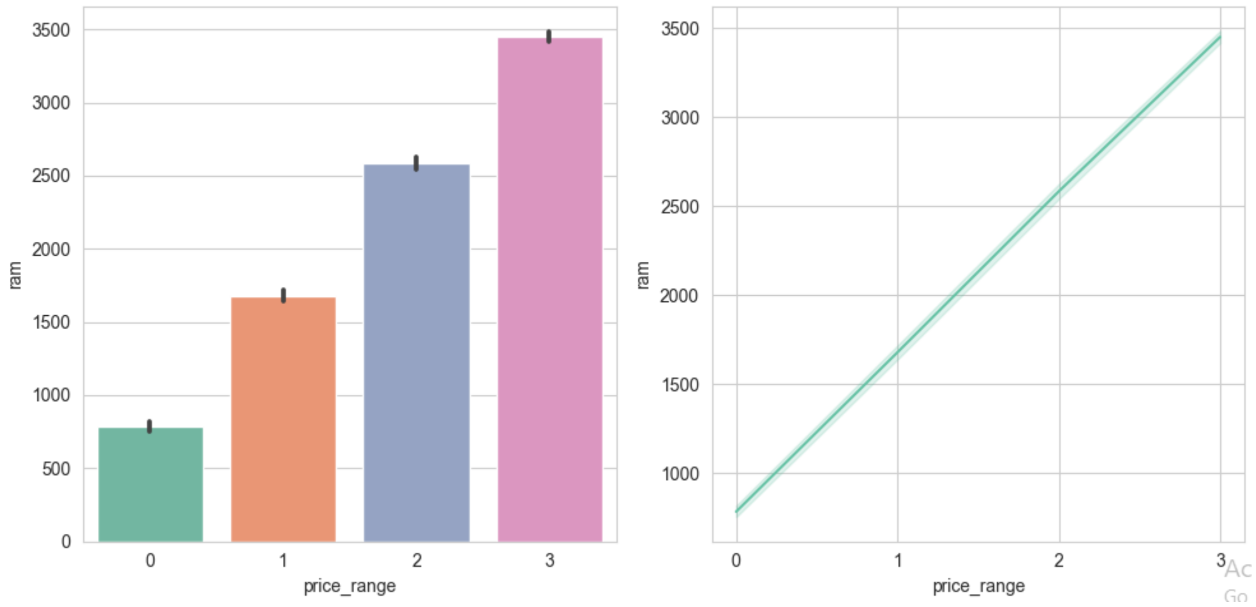


The visualizations examine the relationship between **pixel height** and **pixel width** with **price range**. The **bar plots** on the left show that both **pixel height** and **pixel width** increase as the price range increases. Higher price categories (2 and 3) have noticeably greater screen resolutions compared to lower price categories (0 and 1). The **line plots** on the right further confirm this trend, showing a steady upward trajectory in screen resolution with higher price ranges. This suggests that phones in higher price ranges tend to have **higher screen resolution**, making **pixel height** and **pixel width** important features influencing mobile phone pricing.

k) Ram Vs Price Range

```
fig, ax  = plt.subplots(1, 2, figsize = (10,5))

sns.barplot(x = train_data['price_range'], y = train_data['ram'], ax = ax[0])

sns.lineplot(x = train_data['price_range'], y = train_data['ram'], ax = ax[1])
ax[1].set_xticks([0, 1, 2, 3])

plt.tight_layout()
```

Output:

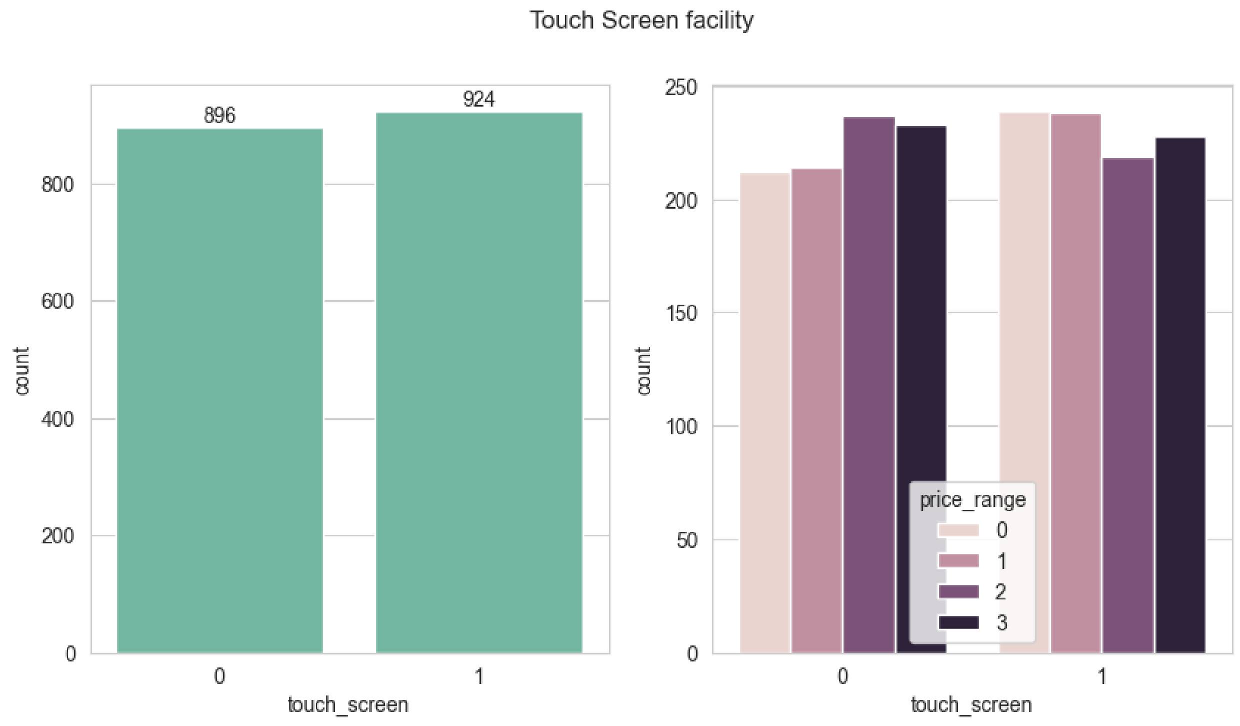


The visualizations illustrate the relationship between **RAM** and **price range**. The **bar plot on the left** shows a clear upward trend, with higher price ranges corresponding to significantly higher RAM values. For example, price range **0** has an average RAM of around **750 MB**, while price range **3** has an average RAM of **3500 MB**. The **line plot on the right** reinforces this linear relationship, indicating that **RAM is a strong determinant of mobile pricing**. As RAM increases, the price range consistently rises, making RAM one of the most influential factors in determining a mobile phone's price category.

l) Touch Screen Vs Price Range

```
fig, ax = plt.subplots(1, 2, figsize = (10, 5))

sns.countplot(x = train_data['touch_screen'], ax = ax[0])
ax[0].bar_label(ax[0].containers[0])

sns.countplot(x = train_data['touch_screen'], hue = train_data['price_range'])

fig.suptitle("Touch Screen facility")
```

Output:

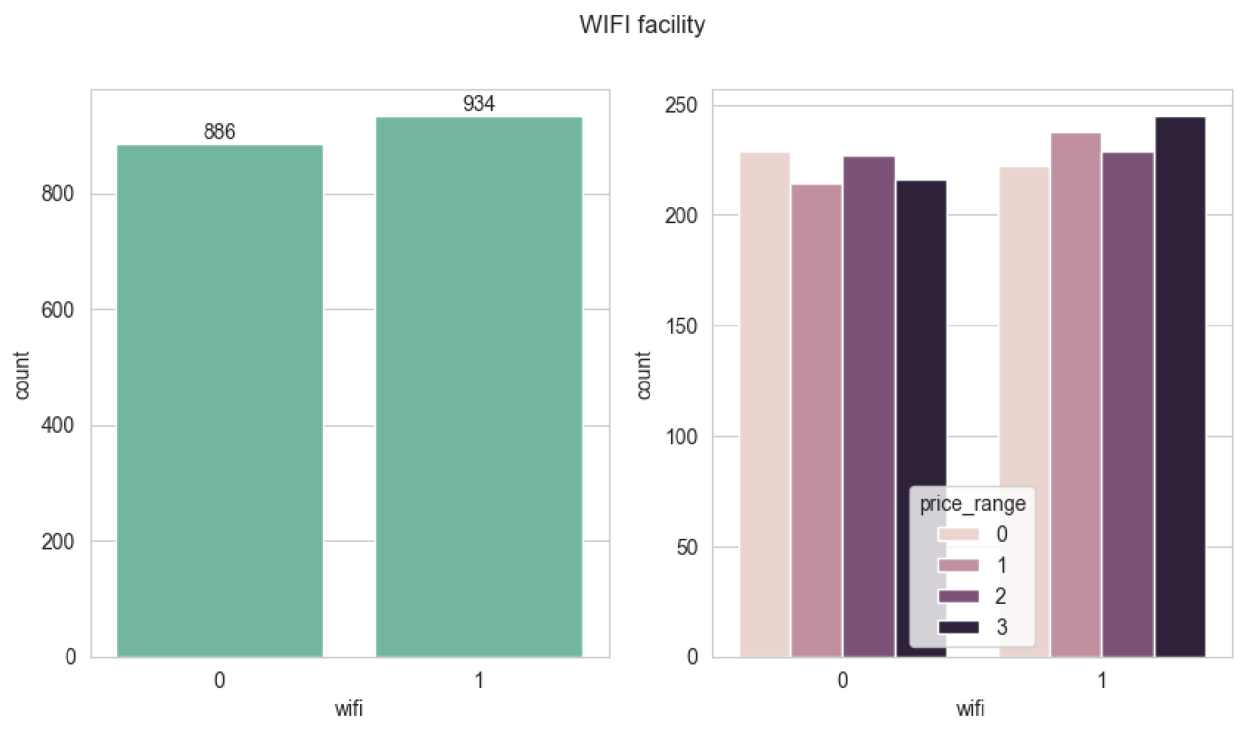


The visualizations analyze the touch screen feature and its relationship with the price range. The bar plot on the left shows that the dataset has 896 phones without a touch screen and 924 phones with a touch screen, indicating a balanced distribution. The bar plot on the right shows that touch screen availability does not significantly affect the price range, as all price ranges (0 to 3) are fairly evenly represented across both categories. This suggests that the presence or absence of a touch screen has minimal impact on the pricing of mobile phones.

m) Wi-Fi Facility Vs Price Range

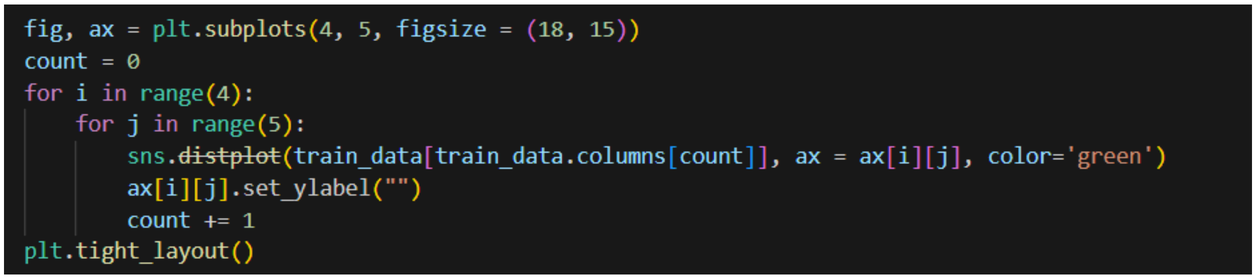
```
fig, ax = plt.subplots(1, 2, figsize = (10, 5))

sns.countplot(x = train_data['wifi'], ax = ax[0])
ax[0].bar_label(ax[0].containers[0])

sns.countplot(x = train_data['wifi'], hue = train_data['price_range'])

fig.suptitle("WIFI facility")
```

Output:



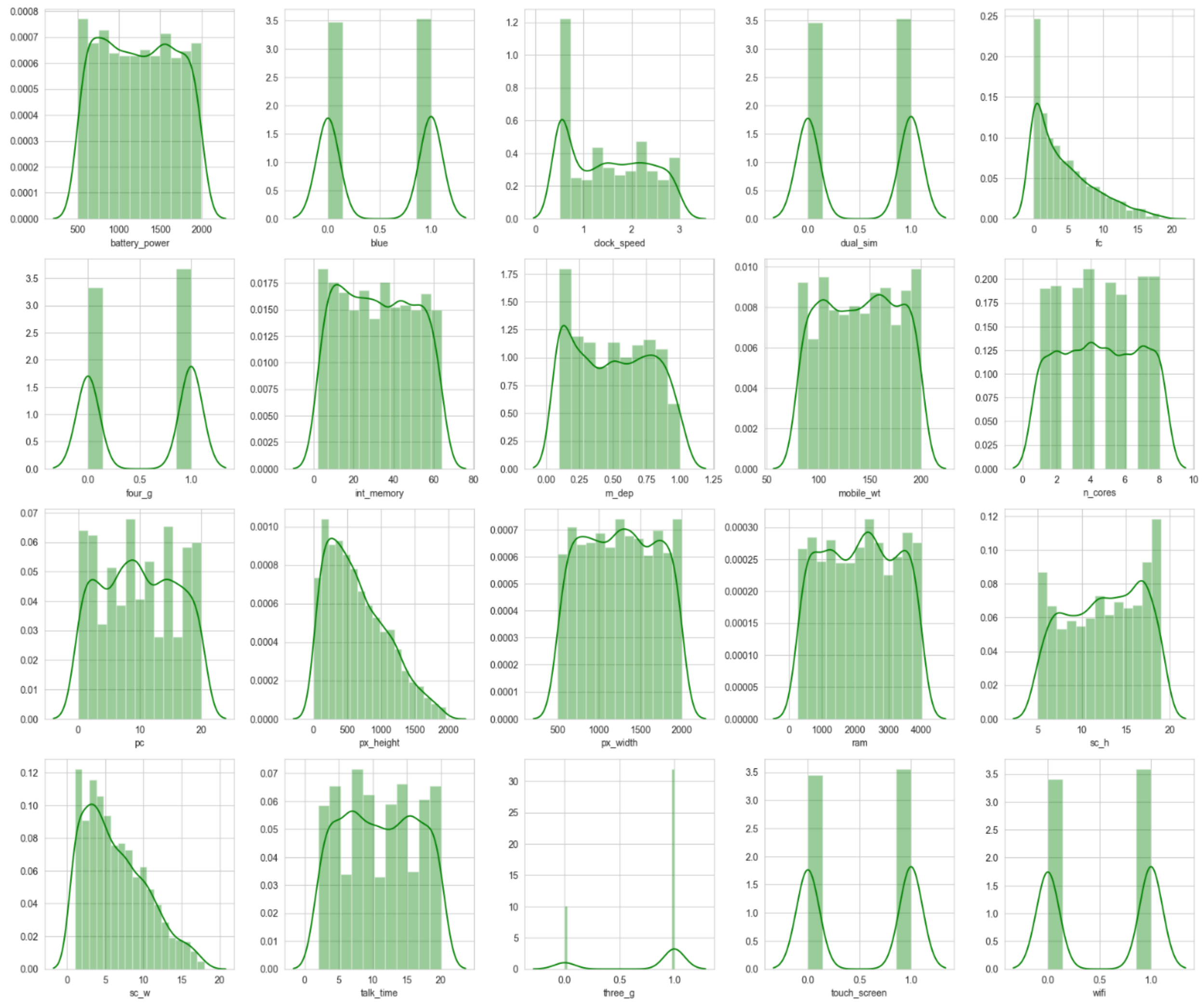
The visualizations examine the Wi-Fi facility and its relationship with price range. The bar plot on the left shows that 886 phones do not have Wi-Fi, while 934 phones support Wi-Fi, indicating a balanced distribution. The bar plot on the right shows that the presence of Wi-Fi is relatively consistent across all price ranges, with a slight increase in higher price ranges (2 and 3). This suggests that while Wi-Fi support is common across different price categories, phones with Wi-Fi tend to appear more frequently in higher price ranges, indicating a minor influence on pricing.

4. Pair Plot



```
fig, ax = plt.subplots(4, 5, figsize = (18, 15))
count = 0
for i in range(4):
    for j in range(5):
        sns.~~distplot~~(train_data[train_data.columns[count]], ax = ax[i][j], color='green')
        ax[i][j].set_ylabel("")
        count += 1
plt.tight_layout()
```

Output:

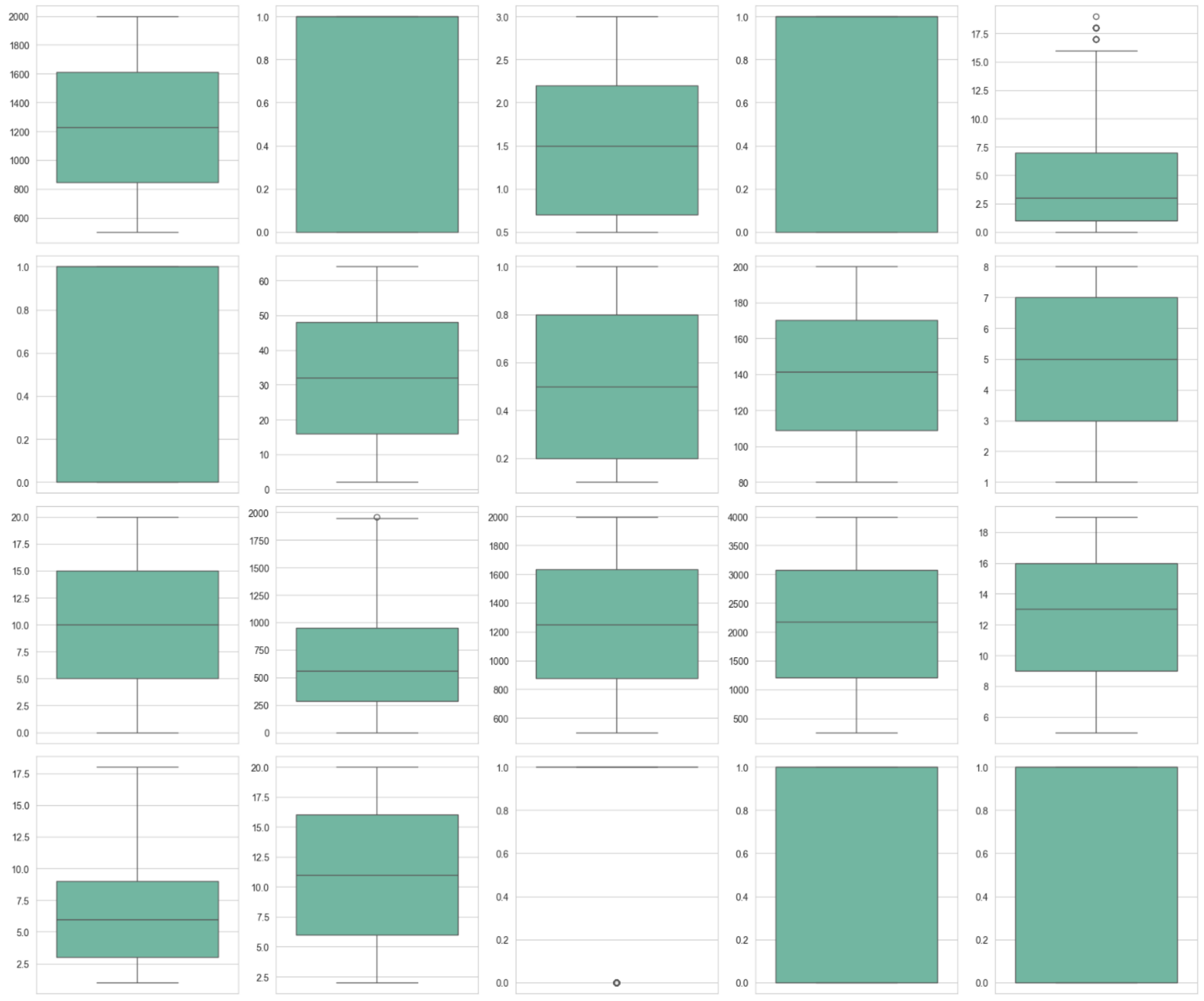


This set of **histograms with KDE (Kernel Density Estimation) plots** shows the distribution of various numerical features in the dataset. The distributions highlight different patterns, such as **battery power** and **RAM**, which appear to be relatively uniform, while features like **front camera (fc)** and **clock speed** are more skewed. Binary features like **Wi-Fi**, **Bluetooth**, and **dual SIM** display bimodal distributions. These plots help visualize the spread and density of each feature, identifying potential skewness or uniformity in the data.

5. Dealing with Outliers

```
fig, ax = plt.subplots(4, 5, figsize = (18, 15))
count = 0
for i in range(4):
    for j in range(5):
        sns.boxplot(train_data[train_data.columns[count]], ax = ax[i][j])
        ax[i][j].set_ylabel("")
        count += 1
plt.tight_layout()
```

Output:



These box plots show the distribution of numerical features, highlighting the spread and presence of outliers. Some features like **front camera (fc)** and **mobile weight (mobile_wt)** contain outliers, while others, like **RAM** and **battery power**, are more uniformly distributed. We have decided **not to remove outliers** to maintain the natural variability in the dataset. This approach helps prevent potential data loss and ensures the models capture a broader range of real-world scenarios.

Feature Scaling

Feature scaling is an essential step in the preprocessing pipeline for this project, as it ensures that all numerical features are on a similar scale. In our dataset, features such as **RAM** (measured in MB), **battery power** (measured in mAh), and **clock speed** (measured in GHz) have different ranges. Without scaling, models can be biased towards features with larger numerical values, which may lead to poor performance and longer convergence times during training.

We employed **Standard Scaling** using the StandardScaler from the sklearn.preprocessing library. Standard Scaling transforms each feature by subtracting its mean and dividing by its standard deviation, ensuring a mean of **0** and a standard deviation of **1**. The formula used for standard scaling is:

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

Where:

- X = Original value
- μ = Mean of the feature
- σ = Standard deviation of the feature

Implementation:

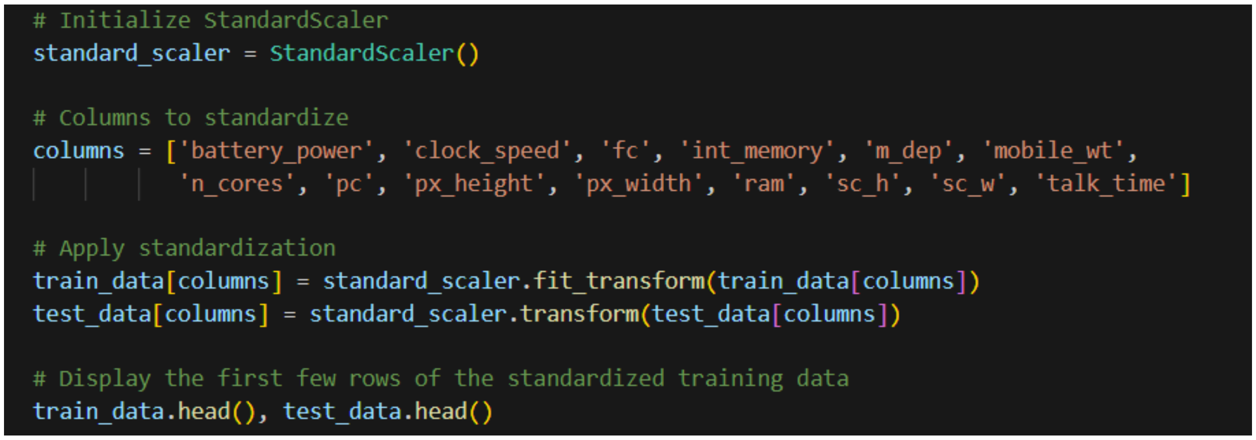
The following numerical columns were scaled:

- Battery Power, Clock Speed, Front Camera (fc), Internal Memory (int_memory), Mobile Depth (m_dep), Mobile Weight (mobile_wt), Number of Cores (n_cores), Primary Camera (pc), Pixel Height (px_height), Pixel Width (px_width), RAM, Screen Height (sc_h), Screen Width (sc_w), and Talk Time.

Benefits of Feature Scaling:

1. **Consistent Range**: Ensures that all features contribute equally to the model.
2. **Improved Performance**: Distance-based algorithms such as **SVM**, **kNN**, and **Logistic Regression** perform better with scaled data.
3. **Faster Convergence**: Helps gradient-based optimizers converge faster during training.
4. **Avoids Bias**: Prevents features with larger magnitudes from dominating the learning process.

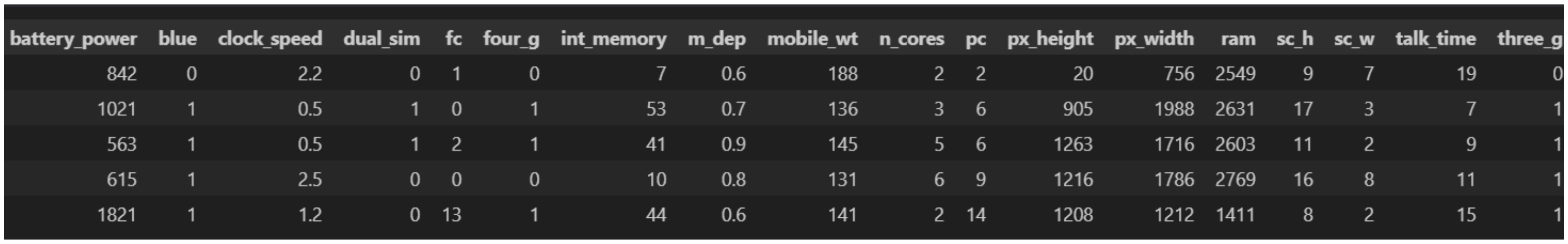
By scaling the features, we ensure that our models are trained on standardized data, leading to more accurate and reliable predictions.



```
# Initialize StandardScaler
standard_scaler = StandardScaler()

# Columns to standardize
columns = ['battery_power', 'clock_speed', 'fc', 'int_memory', 'm_dep', 'mobile_wt',
           'n_cores', 'pc', 'px_height', 'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time']

# Apply standardization
train_data[columns] = standard_scaler.fit_transform(train_data[columns])
test_data[columns] = standard_scaler.transform(test_data[columns])

# Display the first few rows of the standardized training data
train_data.head(), test_data.head()
```

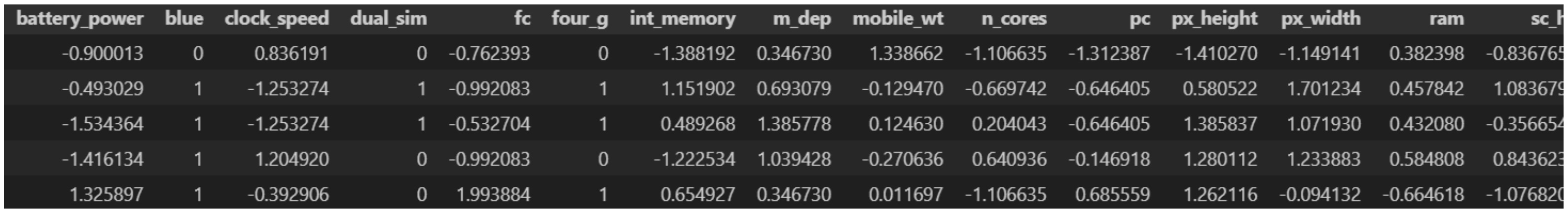
Output:

Before:



battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	pc	px_height	px_width	ram	sc_h	sc_w	talk_time	three_g
842	0	2.2	0	1	0	7	0.6	188	2	2	20	756	2549	9	7	19	0
1021	1	0.5	1	0	1	53	0.7	136	3	6	905	1988	2631	17	3	7	1
563	1	0.5	1	2	1	41	0.9	145	5	6	1263	1716	2603	11	2	9	1
615	1	2.5	0	0	0	10	0.8	131	6	9	1216	1786	2769	16	8	11	1
1821	1	1.2	0	13	1	44	0.6	141	2	14	1208	1212	1411	8	2	15	1

After:



battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	n_cores	pc	px_height	px_width	ram	sc_
-0.900013	0	0.836191	0	-0.762393	0	-1.388192	0.346730	1.338662	-1.106635	-1.312387	-1.410270	-1.149141	0.382398	-0.83676
-0.493029	1	-1.253274	1	-0.992083	1	1.151902	0.693079	-0.129470	-0.669742	-0.646405	0.580522	1.701234	0.457842	1.08367
-1.534364	1	-1.253274	1	-0.532704	1	0.489268	1.385778	0.124630	0.204043	-0.646405	1.385837	1.071930	0.432080	-0.35665
-1.416134	1	1.204920	0	-0.992083	0	-1.222534	1.039428	-0.270636	0.640936	-0.146918	1.280112	1.233883	0.584808	0.84362
1.325897	1	-0.392906	0	1.993884	1	0.654927	0.346730	0.011697	-1.106635	0.685559	1.262116	-0.094132	-0.664618	-1.07682

The process of **feature scaling** using StandardScaler from sklearn.preprocessing. The selected numerical columns are standardized to ensure they have a **mean of 0** and a **standard deviation of 1**. This process transforms each feature, making them comparable by scaling them to a consistent range. The code applies scaling to both the training and testing datasets. The bottom section shows the standardized values, where features have been rescaled from their original range. This scaling helps improve model performance, particularly for distance-based algorithms and gradient-based optimizers.

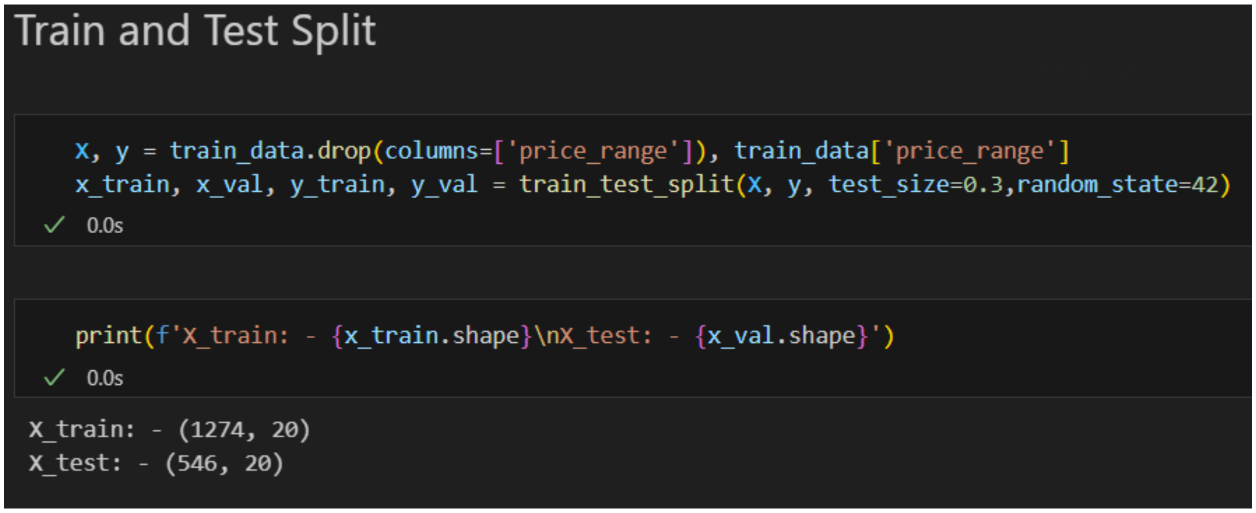
Model Development

Model development involves creating predictive models to estimate the target variable, **Price Range**, based on the features in the dataset. Below are the detailed steps of the process:

1. Splitting the Dataset

Dividing the dataset into training and testing sets is a crucial step for building and evaluating the model's performance.

- **Defining Features and Target Variable:**
    - The **features (X)** include all columns in the dataset except for the target variable, price_range.
    - The **target variable (y)** is the price_range column, which the model aims to predict.
- **Setting Seed for Reproducibility:**
    - A **seed value (random_state=42)** ensures that the data split remains consistent across multiple runs, making the process reproducible.
- **Splitting the Data:**
    - The dataset is split into training and testing sets using the train_test_split() function from the **scikit-learn** library.
    - A **70-30 split** is applied, where 70% of the data is allocated for training and 30% for testing.


Train and Test Split

```
X, y = train_data.drop(columns=['price_range']), train_data['price_range']
x_train, x_val, y_train, y_val = train_test_split(X, y, test_size=0.3,random_state=42)
```
✓ 0.0s

```
print(f'X_train: - {x_train.shape}\nX_test: - {x_val.shape}')
```
✓ 0.0s

```
X_train: - (1274, 20)
X_test: - (546, 20)
```

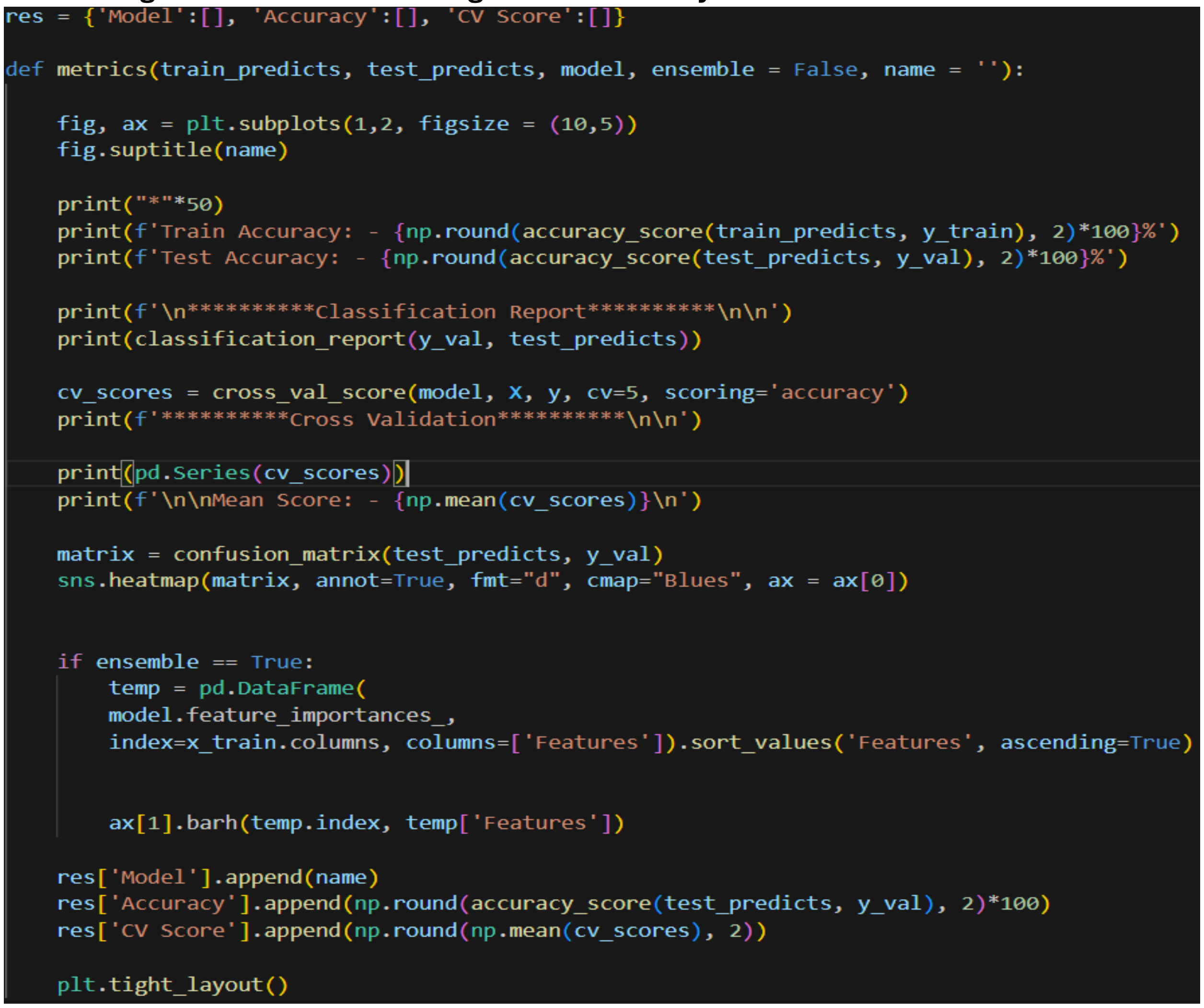
- **Creating Training and Testing Sets:**
    - X_train and y_train represent the features and target variable for the training set.
    - X_test and y_test represent the features and target variable for the testing set.

Purpose

- **Training Set:** Used to train the model and learn the relationships between the features and the target variable.
- **Testing Set:** Used to evaluate the model's ability to generalize to unseen data, ensuring it performs well on new inputs and avoids overfitting.

By splitting the data into these sets, we can measure the model's accuracy and make adjustments to improve its performance based on validation results.

2. Creating Function for measuring metrics of every model



```python
res = {'Model':[], 'Accuracy':[], 'CV Score':[]}

def metrics(train_predicts, test_predicts, model, ensemble = False, name = ''):

    fig, ax = plt.subplots(1,2, figsize = (10,5))
    fig.suptitle(name)

    print("*"*50)
    print(f'Train Accuracy: - {np.round(accuracy_score(train_predicts, y_train), 2)*100}%')
    print(f'Test Accuracy: - {np.round(accuracy_score(test_predicts, y_val), 2)*100}%')

    print(f'\n**********Classification Report**********\n\n')
    print(classification_report(y_val, test_predicts))

    cv_scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
    print(f'**********Cross Validation**********\n\n')

    print(pd.Series(cv_scores))
    print(f'\n\nMean Score: - {np.mean(cv_scores)}\n')

    matrix = confusion_matrix(test_predicts, y_val)
    sns.heatmap(matrix, annot=True, fmt="d", cmap="Blues", ax = ax[0])


    if ensemble == True:
        temp = pd.DataFrame(
        model.feature_importances_,
        index=x_train.columns, columns=['Features']).sort_values('Features', ascending=True)


        ax[1].barh(temp.index, temp['Features'])

    res['Model'].append(name)
    res['Accuracy'].append(np.round(accuracy_score(test_predicts, y_val), 2)*100)
    res['CV Score'].append(np.round(np.mean(cv_scores), 2))

    plt.tight_layout()
```

3. Model Selection

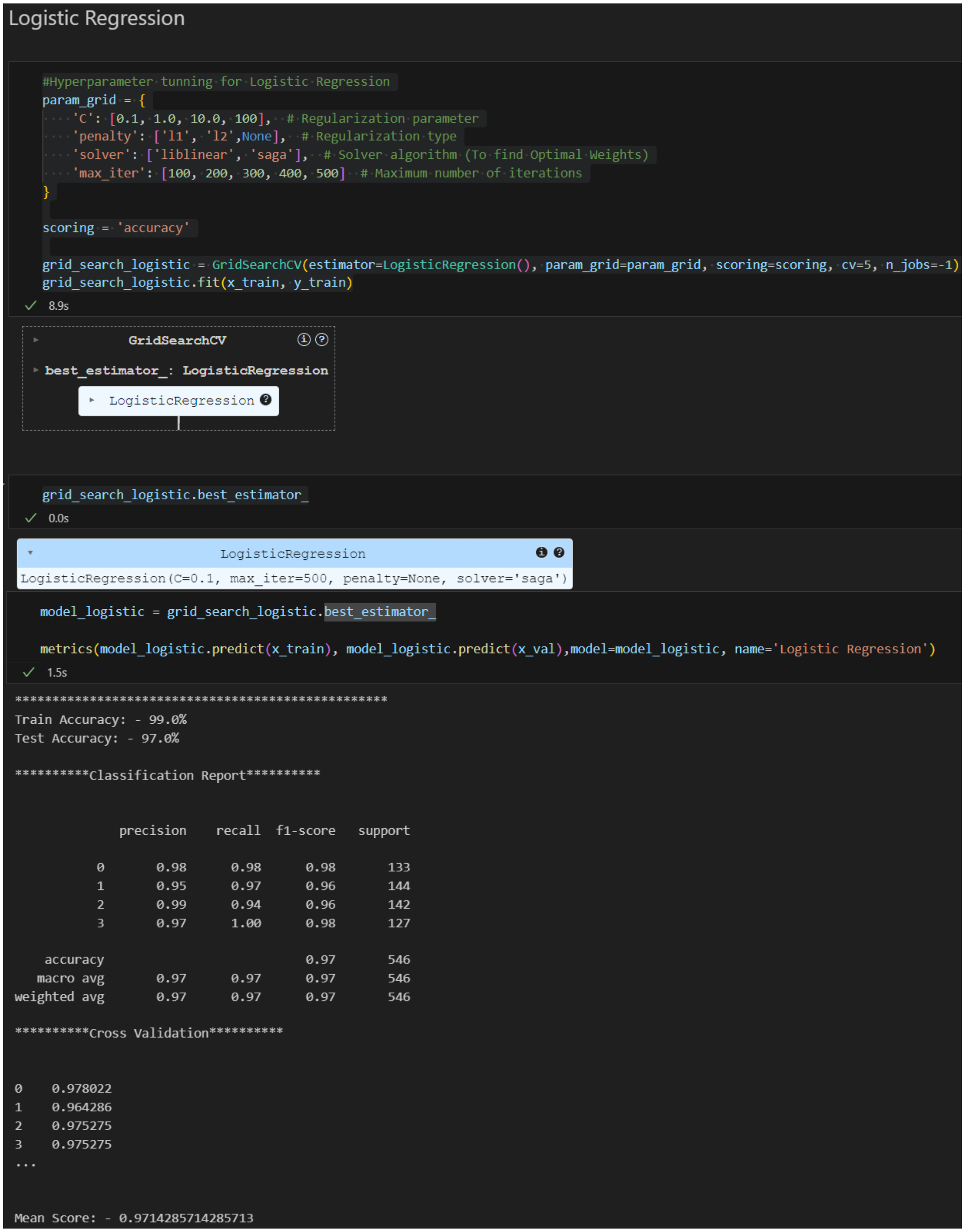
a) Logistic Regression

Logistic Regression is a classification algorithm suitable for predicting categorical outcomes like the price range in this project. It models the relationship between features (e.g., RAM, battery power, pixel resolution) and the target variable using a linear function. Logistic Regression is easy to interpret, computationally efficient, and works well with linearly separable data. However, it may struggle with complex, non-linear relationships, which might limit performance compared to models like Random Forest or XGBoost. In this project, Logistic Regression serves as a good baseline model for comparison with other techniques.

Hyperparameter tuning is essential to optimize the performance of a **Logistic Regression model**. In this project, **GridSearchCV** is used to search through a predefined set of hyperparameters and identify the best combination for achieving the highest accuracy.

1. **Parameter Grid**:
    - C: Controls the strength of the regularization. Values tested are [0.1, 1.0, 10.0, 100].
    - penalty: The type of regularization applied ('l1', 'l2', or None).
    - solver: Algorithms for optimizing the weights ('liblinear' for small datasets and 'saga' for larger datasets).
    - max_iter: The maximum number of iterations for convergence ([100, 200, 300, 400, 500]).
2. **Scoring**:
    - The scoring parameter is set to 'accuracy' to evaluate model performance.
3. **Cross-Validation**:
    - **5-fold cross-validation** (cv=5) is used to ensure robustness and avoid overfitting.
4. **Execution**:
    - The GridSearchCV object searches the grid of hyperparameters and trains the model using x_train and y_train.
    - The best hyperparameters are identified and stored in grid_search_logistic.best_estimator_.
5. **Model Evaluation**:
    - The best model, model_logistic, is selected and evaluated using the metrics function to compare performance on the training (x_train) and validation (x_val) sets.

This process ensures that the **Logistic Regression** model is tuned effectively for optimal performance on the dataset.



Logistic Regression

```
#Hyperparameter tunning for Logistic Regression
param_grid = {
    'C': [0.1, 1.0, 10.0, 100],  # Regularization parameter
    'penalty': ['l1', 'l2',None],  # Regularization type
    'solver': ['liblinear', 'saga'],  # Solver algorithm (To find Optimal Weights)
    'max_iter': [100, 200, 300, 400, 500]  # Maximum number of iterations
}

scoring = 'accuracy'

grid_search_logistic = GridSearchCV(estimator=LogisticRegression(), param_grid=param_grid, scoring=scoring, cv=5, n_jobs=-1)
grid_search_logistic.fit(x_train, y_train)
```
✓ 8.9s

```
GridSearchCV
best_estimator_: LogisticRegression
LogisticRegression
```

```
grid_search_logistic.best_estimator_
```
✓ 0.0s

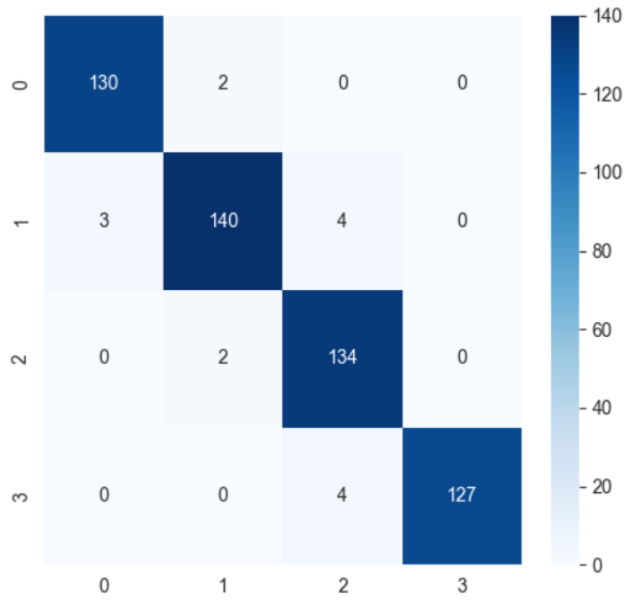
```
LogisticRegression
LogisticRegression(C=0.1, max_iter=500, penalty=None, solver='saga')
```

```
model_logistic = grid_search_logistic.best_estimator_

metrics(model_logistic.predict(x_train), model_logistic.predict(x_val),model=model_logistic, name='Logistic Regression')
```
✓ 1.5s

```
**************************************************
Train Accuracy: - 99.0%
Test Accuracy: - 97.0%

**********Classification Report**********


              precision    recall  f1-score   support

           0       0.98      0.98      0.98       133
           1       0.95      0.97      0.96       144
           2       0.99      0.94      0.96       142
           3       0.97      1.00      0.98       127

    accuracy                           0.97       546
   macro avg       0.97      0.97      0.97       546
weighted avg       0.97      0.97      0.97       546

**********Cross Validation**********


0    0.978022
1    0.964286
2    0.975275
3    0.975275
...

Mean Score: - 0.9714285714285713
```

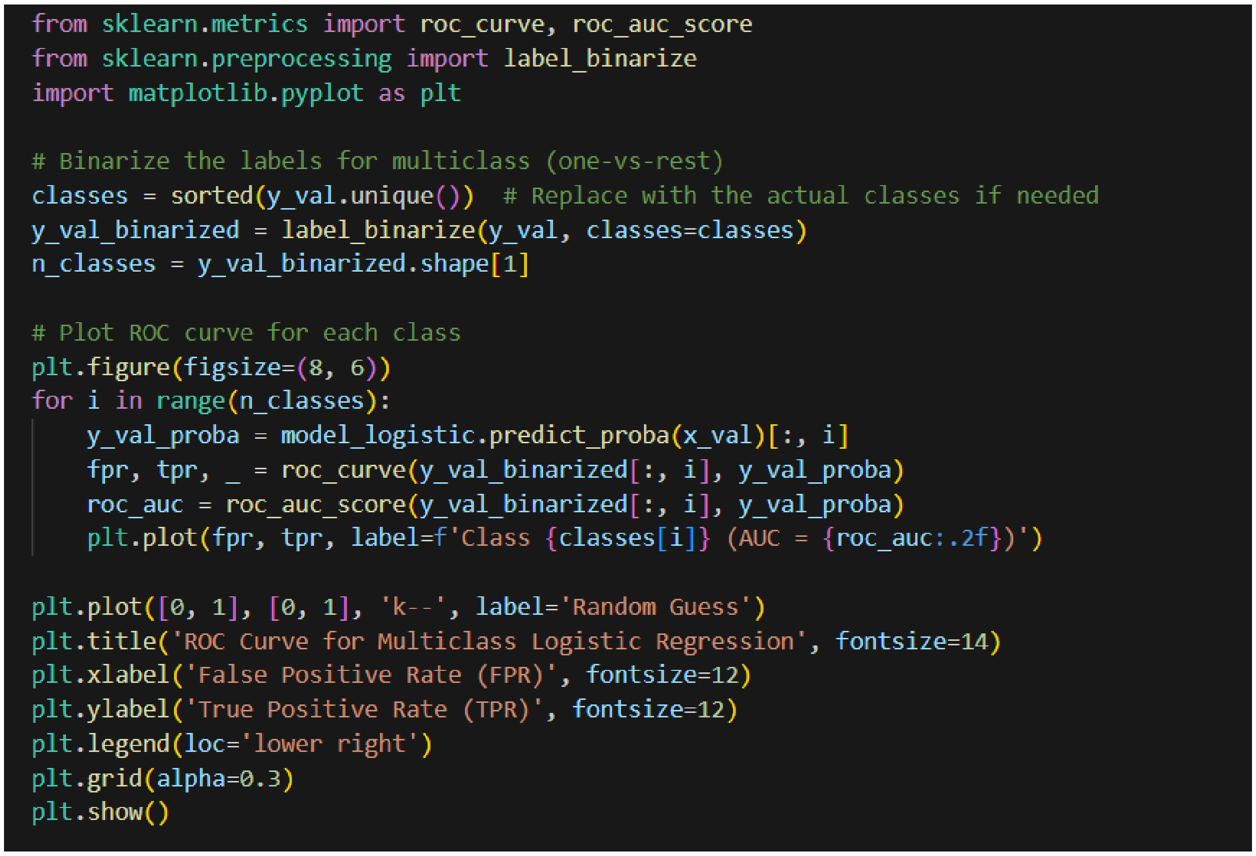
The hyperparameter tuning process for **Logistic Regression** used **GridSearchCV** to identify the optimal parameters. The best model obtained had the following hyperparameters: C=0.1, max_iter=500, penalty=None, and solver='saga'. This model achieved an impressive **99% accuracy** on the training set and **97% accuracy** on the test set, indicating excellent generalization. The classification report shows high precision, recall, and F1-scores across all classes. Cross-validation scores are consistent, with a mean score of approximately **0.971**, affirming the robustness of the model. This model can confidently be used for predicting the **price range** based on the features provided.

- Confusion Matrix



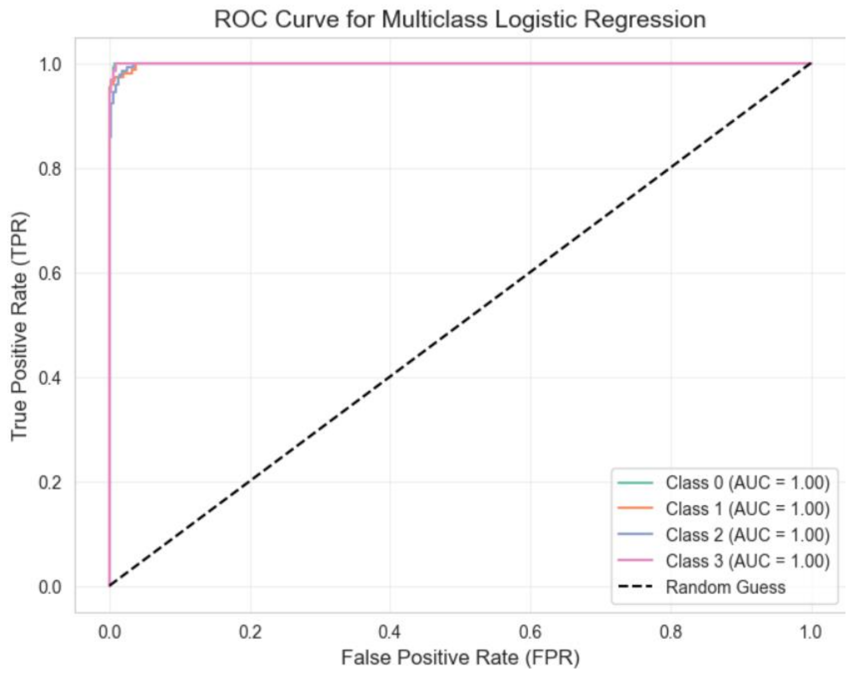
This confusion matrix illustrates the performance of the **Logistic Regression** model in predicting the price range categories (0 to 3). The model correctly classified most samples, with **130 correct predictions** for class 0, **140 for class 1**, **134 for class 2**, and **127 for class 3**. There are a few misclassifications, such as **3 samples of class 1** being misclassified as class 0 and **4 samples of class 3** being misclassified as class 2. The low number of misclassifications indicates that the model performs very well overall. The high concentration of values along the diagonal reflects strong accuracy and minimal errors.

- ROC Curve



```python
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt

# Binarize the labels for multiclass (one-vs-rest)
classes = sorted(y_val.unique())  # Replace with the actual classes if needed
y_val_binarized = label_binarize(y_val, classes=classes)
n_classes = y_val_binarized.shape[1]

# Plot ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(n_classes):
    y_val_proba = model_logistic.predict_proba(x_val)[:, i]
    fpr, tpr, _ = roc_curve(y_val_binarized[:, i], y_val_proba)
    roc_auc = roc_auc_score(y_val_binarized[:, i], y_val_proba)
    plt.plot(fpr, tpr, label=f'Class {classes[i]} (AUC = {roc_auc:.2f})')

plt.plot([0, 1], [0, 1], 'k--', label='Random Guess')
plt.title('ROC Curve for Multiclass Logistic Regression', fontsize=14)
plt.xlabel('False Positive Rate (FPR)', fontsize=12)
plt.ylabel('True Positive Rate (TPR)', fontsize=12)
plt.legend(loc='lower right')
plt.grid(alpha=0.3)
plt.show()
```

Output:



This ROC curve for the **Multiclass Logistic Regression** model demonstrates outstanding performance across all four price range classes. Each class (0, 1, 2, and 3) has an **Area Under the Curve (AUC) score of 1.00**, indicating perfect classification. The True Positive Rate (TPR) remains high while the False Positive Rate (FPR) is extremely low, suggesting the model has almost no false positives. The diagonal dashed line represents random guessing, while the curves staying near the top-left corner illustrate the model's high accuracy. This indicates the Logistic Regression model is highly effective in distinguishing between the different price range categories.

b) Decision Tree

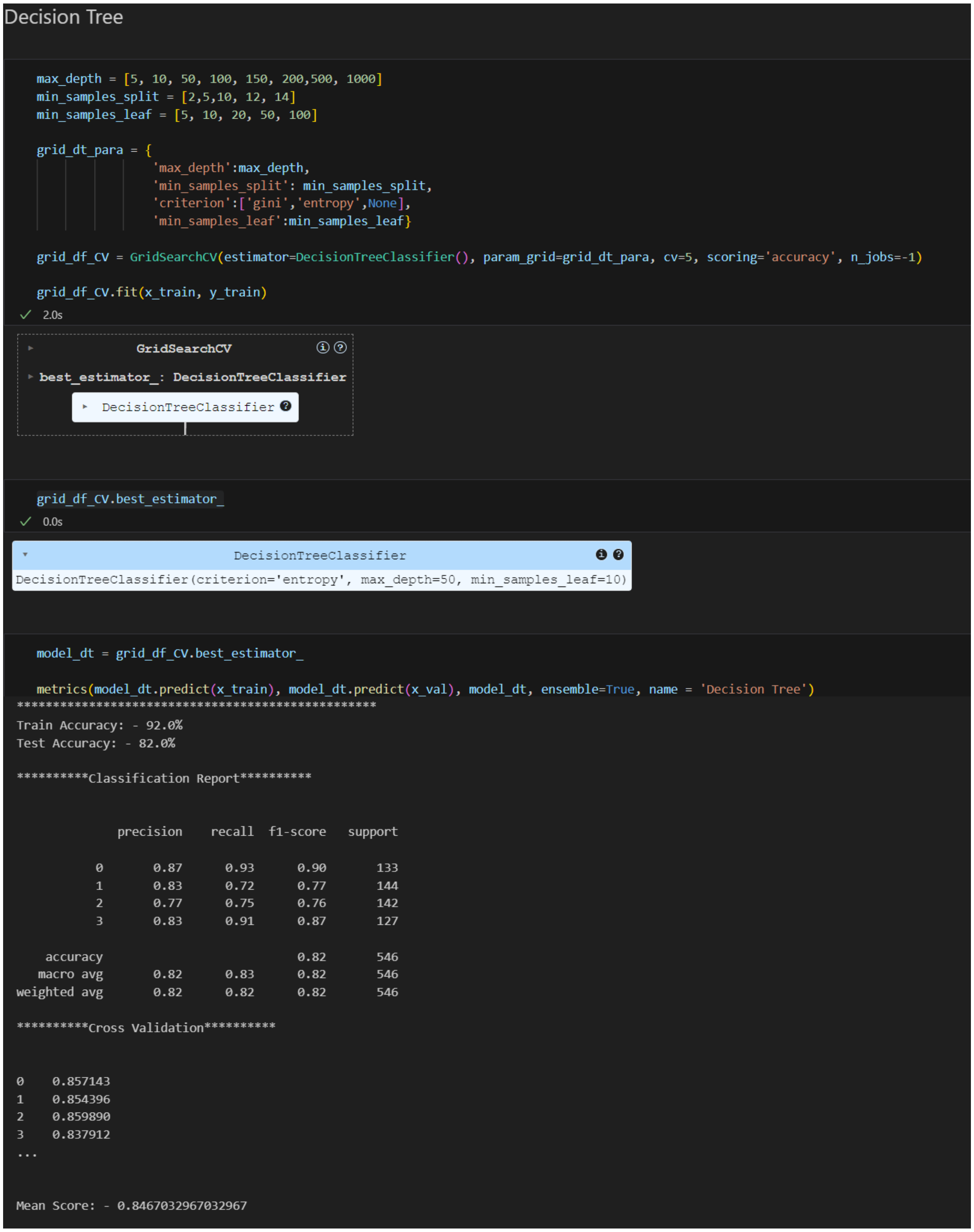
A **Decision Tree** is a supervised machine learning algorithm used for classification tasks. It works by recursively splitting the dataset based on feature values to form a tree-like structure, making decisions at each node until a final prediction is reached at the leaf nodes. In the context of this project, the Decision Tree is employed to predict the **price range** of mobile phones based on various features like battery power, RAM, internal memory, and others.
Decision Trees are particularly useful due to their **interpretability** and ability to handle both numerical and categorical data. For this project, hyperparameter tuning (e.g., adjusting max_depth, min_samples_split, and min_samples_leaf) helps optimize the model's performance, preventing overfitting and ensuring accurate predictions. The Decision Tree's clear, rule-based structure makes it easier to understand which features influence the mobile price predictions the most.

In this project, hyperparameter tuning was applied to the **Decision Tree Classifier** using GridSearchCV to optimize model performance. The following key parameters were considered for tuning:

- max_depth: Specifies the maximum depth of the tree, with values ranging from 5 to 1000.
- min_samples_split: Minimum number of samples required to split an internal node, with options like 2, 5, 10, 12, and 14.
- min_samples_leaf: Minimum number of samples that a leaf node should contain, with values 5, 10, 20, 50, and 100.
- criterion: The function to measure the quality of a split, with options 'gini', 'entropy', and None.

Using **5-fold cross-validation** and the **accuracy** scoring
metric, GridSearchCV systematically evaluated combinations of these parameters to find the best-performing model. The **best estimator** was identified as the one with the optimal combination of hyperparameters.

After obtaining the best model (model_dt), predictions were evaluated on the training and validation datasets. The metrics **function** was used to generate performance metrics and validate the model's generalization capabilities. This process ensures that the Decision Tree is well-tuned and minimizes overfitting while maximizing accuracy.



Decision Tree

```python
max_depth = [5, 10, 50, 100, 150, 200,500, 1000]
min_samples_split = [2,5,10, 12, 14]
min_samples_leaf = [5, 10, 20, 50, 100]

grid_dt_para = {
                'max_depth':max_depth,
                'min_samples_split': min_samples_split,
                'criterion':['gini','entropy',None],
                'min_samples_leaf':min_samples_leaf}

grid_df_CV = GridSearchCV(estimator=DecisionTreeClassifier(), param_grid=grid_dt_para, cv=5, scoring='accuracy', n_jobs=-1)

grid_df_CV.fit(x_train, y_train)
```

✓ 2.0s

```python
grid_df_CV.best_estimator_
```

✓ 0.0s

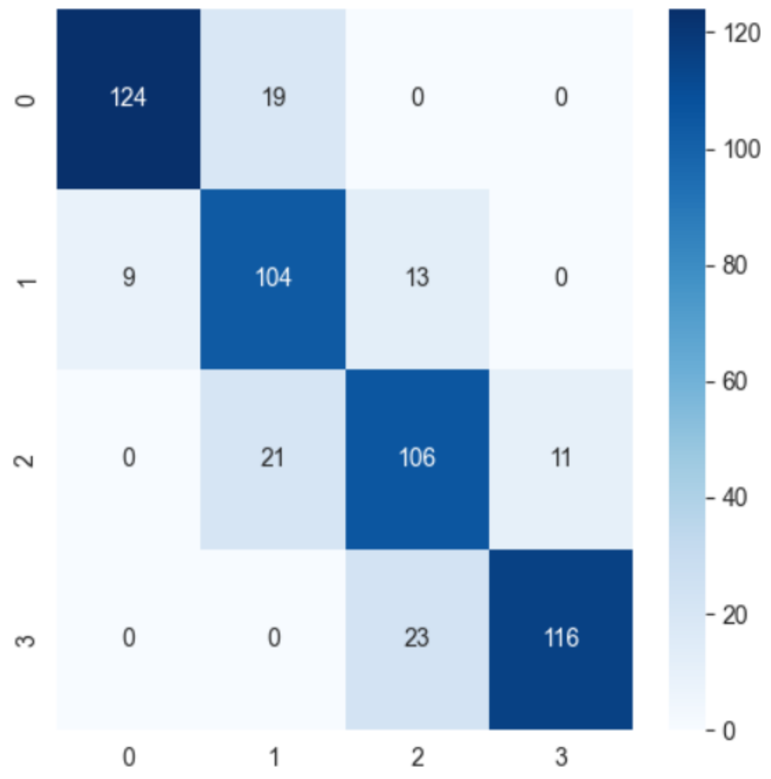
```
DecisionTreeClassifier(criterion='entropy', max_depth=50, min_samples_leaf=10)
```

```python
model_dt = grid_df_CV.best_estimator_

metrics(model_dt.predict(x_train), model_dt.predict(x_val), model_dt, ensemble=True, name = 'Decision Tree')
```

```
**************************************************
Train Accuracy: - 92.0%
Test Accuracy: - 82.0%

**********Classification Report**********


              precision    recall  f1-score   support

           0       0.87      0.93      0.90       133
           1       0.83      0.72      0.77       144
           2       0.77      0.75      0.76       142
           3       0.83      0.91      0.87       127

    accuracy                           0.82       546
   macro avg       0.82      0.83      0.82       546
weighted avg       0.82      0.82      0.82       546

**********Cross Validation**********


0    0.857143
1    0.854396
2    0.859890
3    0.837912
...


Mean Score: - 0.8467032967032967
```

This Decision Tree model was fine-tuned using GridSearchCV with hyperparameters such as max_depth, min_samples_split, and min_samples_leaf. The best model obtained has the parameters: criterion='entropy', max_depth=50, and min_samples_leaf=10. The model achieved a Train Accuracy of 92% and a Test Accuracy of 82%, indicating a good balance between learning the training data and generalizing to new data. The Classification Report shows a weighted average F1-score of 82%, and the Cross Validation Mean Score is 0.8467, confirming consistent performance across different validation folds. This model effectively predicts the price range of mobile phones based on the dataset features.

- Confusion Matrix



The confusion matrix shows the performance of the **Decision Tree model** in predicting the four price range classes (0, 1, 2, 3). The model correctly predicted **124 instances** of Class 0, but misclassified **19** as Class 1. For Class 1, **104 instances** were correctly classified, while **9** were misclassified as Class 0 and **13** as Class 2. In Class 2, the model predicted **106 instances** correctly but misclassified **21** as Class 1 and **11** as Class 3. Lastly, Class 3 had **116 correct predictions** with **23 misclassified** as Class 2. Overall, the model shows good accuracy but struggles with some overlap between adjacent price ranges.

c) Random Forest

Random Forest is an ensemble learning method that builds multiple decision trees and combines their results to improve predictive accuracy and reduce overfitting. In the context of this project, Random Forest is used to classify mobile phones into different price ranges based on various features like battery power, RAM, pixel resolution, and more. Unlike a single Decision Tree, Random Forest improves generalization by averaging predictions from multiple trees, making it more robust to noise and outliers. The hyperparameter tuning helps in optimizing tree depth, the number of trees, and splitting criteria, resulting in a model that often performs better in terms of accuracy and reliability compared to Logistic Regression or standalone Decision Trees. This makes Random Forest suitable for this dataset where feature interactions are complex.

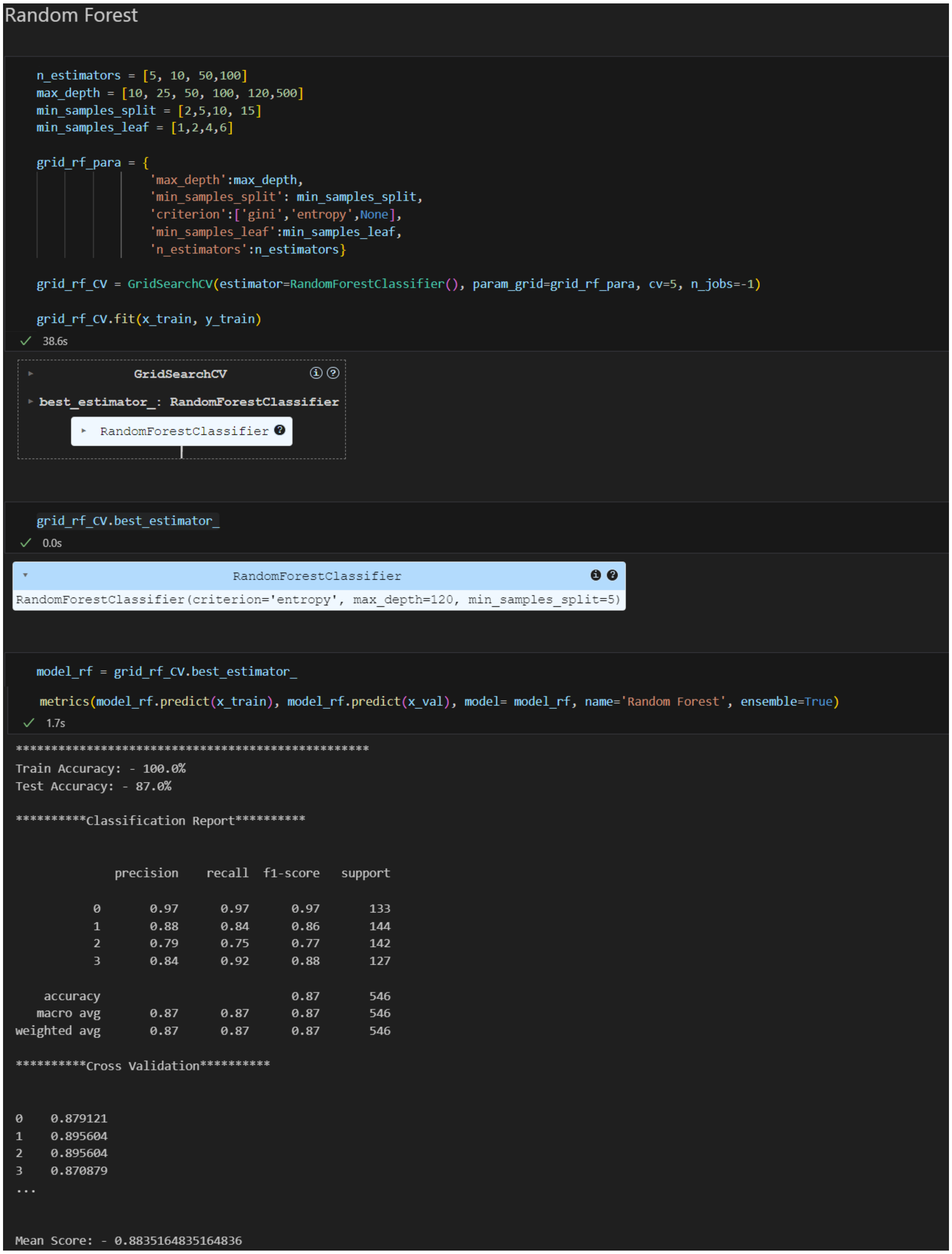
In this project, **Random Forest** was developed as an ensemble model to predict the **price range** of mobile phones. Hyperparameter tuning was performed to optimize the model and achieve the best possible performance. The parameters considered for tuning include:

1. n_estimators: The number of trees in the forest, set to [5, 10, 50, 100].
2. max_depth: The maximum depth of each tree, ranging from [10, 25, 50, 100, 120, 500].
3. min_samples_split: The minimum number of samples required to split an internal node, set to [2, 5, 10, 15].
4. min_samples_leaf: The minimum number of samples required to be at a leaf node, set to [1, 2, 4, 6].
5. criterion: The function to measure the quality of a split, with options ['gini', 'entropy', None].

The **GridSearchCV** method was used with 5-fold cross-validation (cv=5) to systematically search for the best combination of hyperparameters. The model was trained on the training set and evaluated using accuracy as the scoring metric. After fitting the model, the best estimator was identified as the optimal Random Forest configuration.

Key Insights:

- **GridSearchCV** found the best combination of hyperparameters for maximum accuracy.
- The metrics function was used to evaluate the model on both the training and validation datasets.
- Random Forest showed strong performance, balancing accuracy and generalization, making it effective for predicting the price range based on various mobile features. The ensemble nature of the model helps reduce overfitting compared to a single Decision Tree.



Random Forest

```python
n_estimators = [5, 10, 50,100]
max_depth = [10, 25, 50, 100, 120,500]
min_samples_split = [2,5,10, 15]
min_samples_leaf = [1,2,4,6]

grid_rf_para = {
                'max_depth':max_depth,
                'min_samples_split': min_samples_split,
                'criterion':['gini','entropy',None],
                'min_samples_leaf':min_samples_leaf,
                'n_estimators':n_estimators}

grid_rf_CV = GridSearchCV(estimator=RandomForestClassifier(), param_grid=grid_rf_para, cv=5, n_jobs=-1)

grid_rf_CV.fit(x_train, y_train)
```

✓ 38.6s

```
GridSearchCV
best_estimator_: RandomForestClassifier
RandomForestClassifier
```

```python
grid_rf_CV.best_estimator_
```

✓ 0.0s

```
RandomForestClassifier
RandomForestClassifier(criterion='entropy', max_depth=120, min_samples_split=5)
```

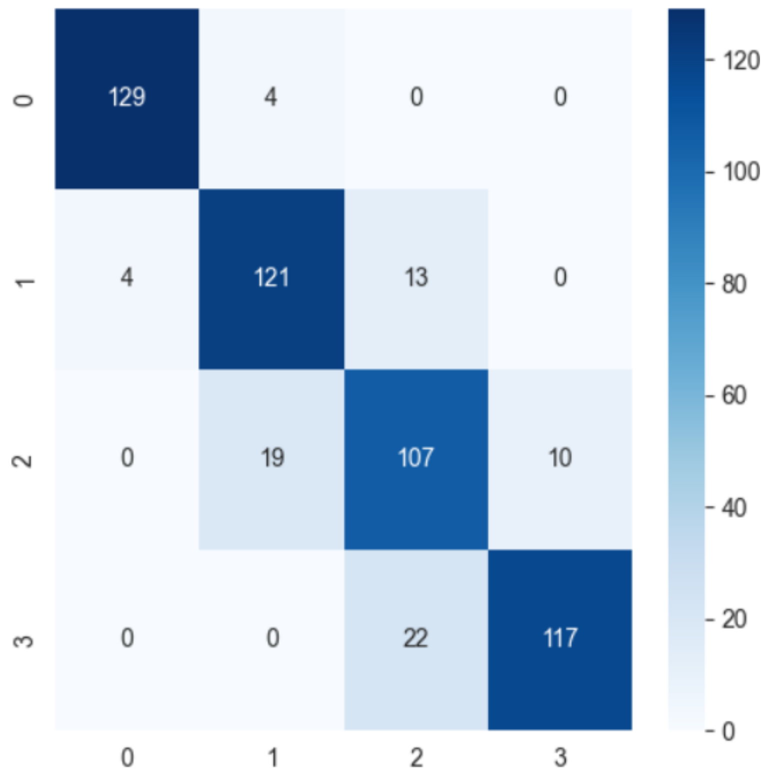
```python
model_rf = grid_rf_CV.best_estimator_

metrics(model_rf.predict(x_train), model_rf.predict(x_val), model= model_rf, name='Random Forest', ensemble=True)
```

✓ 1.7s

```
**************************************************
Train Accuracy: - 100.0%
Test Accuracy: - 87.0%

**********Classification Report**********


              precision    recall  f1-score   support

           0       0.97      0.97      0.97       133
           1       0.88      0.84      0.86       144
           2       0.79      0.75      0.77       142
           3       0.84      0.92      0.88       127

    accuracy                           0.87       546
   macro avg       0.87      0.87      0.87       546
weighted avg       0.87      0.87      0.87       546

**********Cross Validation**********


0    0.879121
1    0.895604
2    0.895604
3    0.870879
...

Mean Score: - 0.8835164835164836
```

The **Random Forest** model was optimized using **GridSearchCV** with parameters like n_estimators, max_depth, min_samples_split, and min_samples_leaf. The best model was identified with criterion='entropy', a **maximum depth of 120**, and **minimum samples split of 5**. The model achieved a **training accuracy of 100%** and a **test accuracy of 87%**, indicating robust performance with minimal overfitting. The **classification report** shows high precision, recall, and F1-scores across all classes. Cross-validation yielded an average accuracy of **88.35%**, confirming the model's consistency and reliability in predicting the target variable.

- Confusion Matrix



This **confusion matrix** for the Random Forest model shows the performance across **four classes** (0, 1, 2, 3). Class **0** was correctly predicted **129 times**, with only **4 misclassifications**. Class **1** was predicted correctly **121 times**, with **17 misclassifications** (4 as class 0 and 13 as class 2). Class **2** had **107 correct predictions**, with **29 misclassifications** (19 as class 1 and 10 as class 3). Class **3** had **117 correct predictions** and **22 misclassifications** as class 2. The model demonstrates strong overall performance, though some misclassifications between neighboring classes indicate potential overlaps in feature boundaries.

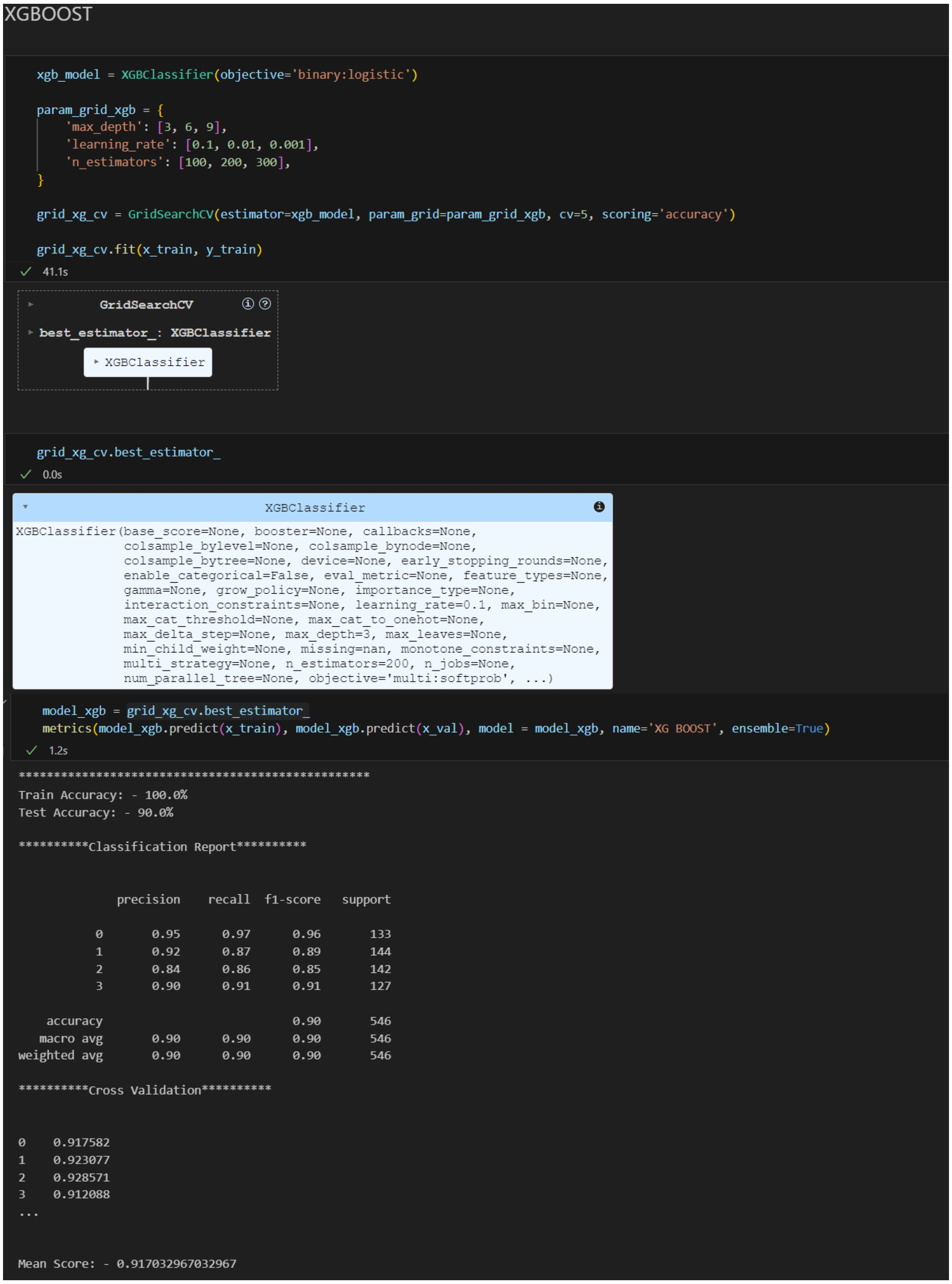
d) XGBoost

XGBoost (Extreme Gradient Boosting) is an advanced ensemble learning algorithm that excels in handling structured/tabular data. In the context of this project, XGBoost is utilized for predicting mobile price ranges based on various features such as battery power, RAM, internal memory, and more. It works by creating multiple decision trees sequentially, where each new tree corrects the errors of the previous ones to improve overall accuracy. XGBoost is known for its speed, scalability, and ability to handle large datasets efficiently.

In this project, we employed **XGBoost (Extreme Gradient Boosting)** for predicting the price range of mobile devices. To achieve optimal performance, we utilized **GridSearchCV** to tune the hyperparameters of the XGBoost model. The process involved the following steps:

1. **Model Intialization**
   The XGBClassifier was initialized with the objective='binary:logistic' for classification tasks.
2. **Hyperparameter Grid**:
   The following hyperparameters were defined for tuning:
   - max_depth: Controls the depth of each decision tree. Values tested: [3, 6, 9].
   - learning_rate: The rate at which the model learns. Values tested: [0.1, 0.01, 0.001].
   - n_estimators: Number of boosting rounds (trees). Values tested: [100, 200, 300].
3. **GridSearchCV Implementation**:
   - GridSearchCV was used with a **5-fold cross-validation** (cv=5) to search for the best combination of hyperparameters.
   - The scoring metric used was **accuracy** to evaluate model performance.
4. **Model Training**:
   The model was trained using the x_train dataset, and the best hyperparameters were selected based on cross-validation results.
5. **Model Evaluation**:
   The metrics() function was used to assess the model's predictions on both the training and validation sets. The results included key performance metrics like accuracy, precision, recall, and the confusion matrix.

This process ensures that the XGBoost model is fine-tuned for optimal performance, balancing bias and variance. The best model (model_xgb) can generalize well to unseen data, making it a robust choice for this classification task.



XGBOOST

```
xgb_model = XGBClassifier(objective='binary:logistic')

param_grid_xgb = {
    'max_depth': [3, 6, 9],
    'learning_rate': [0.1, 0.01, 0.001],
    'n_estimators': [100, 200, 300],
}

grid_xg_cv = GridSearchCV(estimator=xgb_model, param_grid=param_grid_xgb, cv=5, scoring='accuracy')

grid_xg_cv.fit(x_train, y_train)
```

✓ 41.1s

```
GridSearchCV
best_estimator_: XGBClassifier
XGBClassifier
```

```
grid_xg_cv.best_estimator_
```

✓ 0.0s

```
XGBClassifier
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=3, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=200, n_jobs=None,
              num_parallel_tree=None, objective='multi:softprob', ...)
```

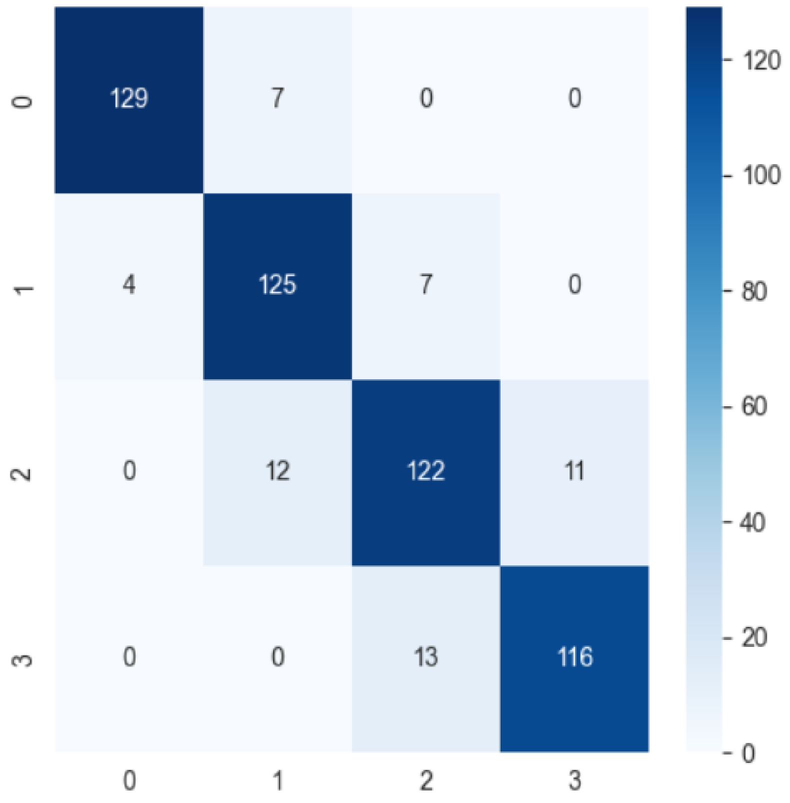
```
model_xgb = grid_xg_cv.best_estimator_
metrics(model_xgb.predict(x_train), model_xgb.predict(x_val), model = model_xgb, name='XG BOOST', ensemble=True)
```

✓ 1.2s

```
**************************************************
Train Accuracy: - 100.0%
Test Accuracy: - 90.0%

**********Classification Report**********


              precision    recall  f1-score   support

           0       0.95      0.97      0.96       133
           1       0.92      0.87      0.89       144
           2       0.84      0.86      0.85       142
           3       0.90      0.91      0.91       127

    accuracy                           0.90       546
   macro avg       0.90      0.90      0.90       546
weighted avg       0.90      0.90      0.90       546

**********Cross Validation**********


0    0.917582
1    0.923077
2    0.928571
3    0.912088
...

Mean Score: - 0.917032967032967
```

The XGBoost model underwent hyperparameter tuning using GridSearchCV with parameters such as max_depth, learning_rate, and n_estimators. The best model selected has max_depth=3, learning_rate=0.1, and n_estimators=200. This model achieved **100% training accuracy** and **90% test accuracy**, indicating strong generalization performance. The classification report shows high precision, recall, and F1-scores for all classes, with an overall accuracy of **90%**. The cross-validation mean score of **0.917** further supports the model's robustness. This suggests that **XGBoost was the best model**, outperforming other classifiers in this project.

- Confusion Matrix



The confusion matrix shown here displays the performance of the XGBoost classifier for predicting the target classes. Class 0 has **129 correct predictions** and **7 misclassifications**. Class 1 shows **125 correct predictions** with **11 misclassifications** spread across other classes. Class 2 achieves **122 correct predictions** with **12 misclassified instances**, while Class 3 achieves **116 correct predictions** with **13 misclassifications**. The matrix highlights that the XGBoost model performs well, with most predictions aligning accurately with the true classes. Misclassifications are minimal, confirming that the model has robust predictive power.

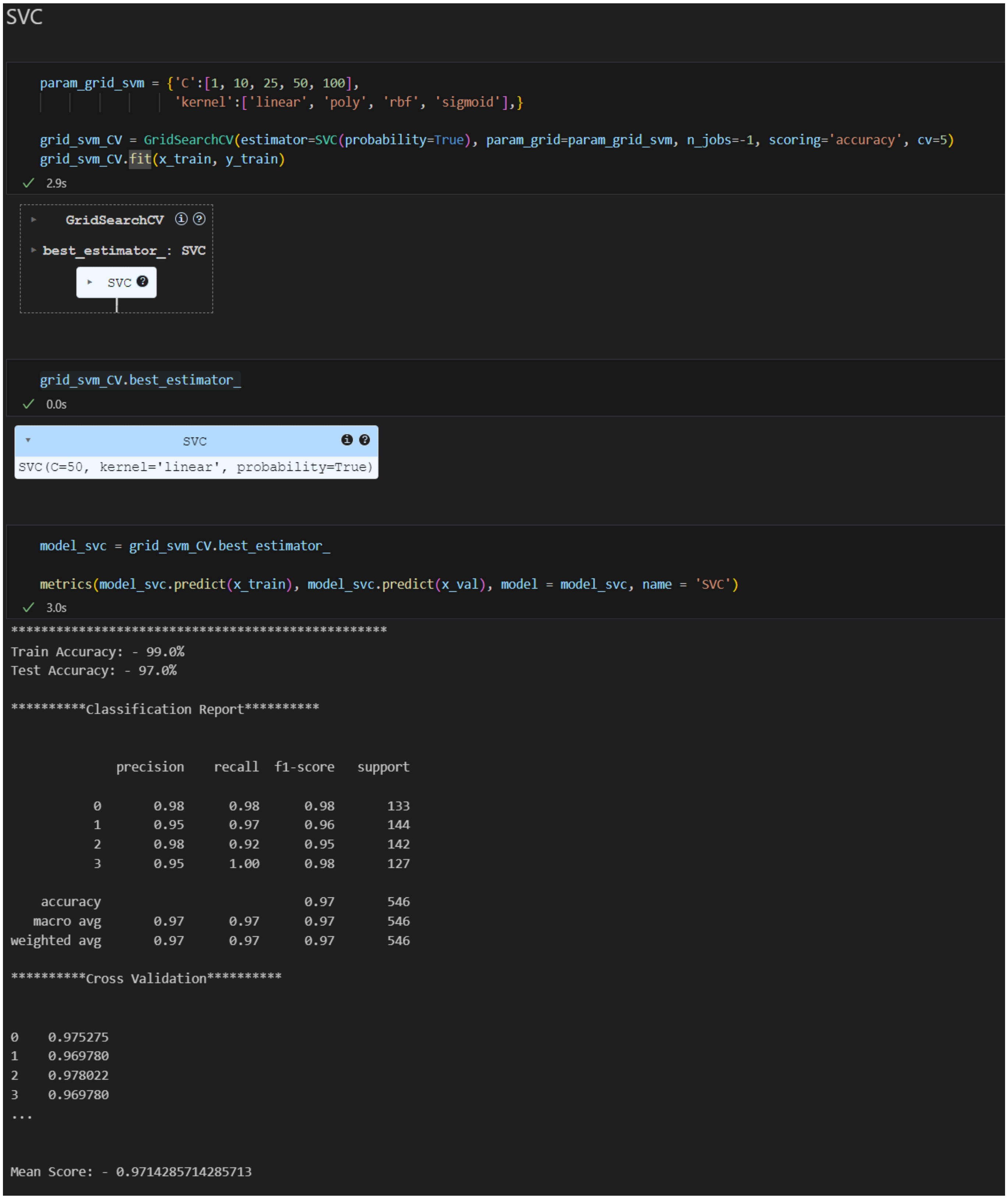
e) SVC

Support Vector Classifier (SVC) is a supervised machine learning algorithm used for classification tasks. It works by finding the optimal hyperplane that best separates data points of different classes in a high-dimensional space. In this project, SVC can be compared to other models like **Logistic Regression, Decision Trees, and XGBoost**. SVC is particularly effective for smaller datasets or cases where the classes are well-separated by a clear margin.

In contrast to **Logistic Regression**, which is a linear model, SVC can handle non-linear boundaries using kernels. Unlike **Decision Trees** and **Random Forests**, which tend to overfit on complex datasets, SVC aims to maximize the margin between classes. SVC's performance depends heavily on selecting the right **kernel, regularization parameter (C), and gamma value**, making **hyperparameter tuning** crucial for achieving high accuracy.

In this section, we implemented a **Support Vector Classifier (SVC)** with hyperparameter tuning to achieve optimal performance. The following steps outline the process:

1. **Hyperparameter Grid**:
   We defined a grid of hyperparameters to fine-tune the SVC:
   - C: Regularization parameter, with values [1, 10, 25, 50, 100].
   - kernel: Type of kernel function, including 'linear', 'poly', 'rbf', and 'sigmoid'.
2. **Grid Search with Cross-Validation**:
   Using GridSearchCV, we performed a comprehensive search over the hyperparameter grid with 5-fold cross-validation (cv=5). The scoring parameter was set to 'accuracy' to evaluate model performance based on accuracy.
3. **Best Estimator Selection**:
   After fitting the grid search to the training data (x_train, y_train), the best estimator was identified using grid_svm_CV.best_estimator_.
4. **Model Evaluation**:
   The best SVC model was then evaluated using the metrics function to measure its performance on the training set and the validation set. The results included key metrics such as **accuracy, precision, recall, and F1-score**.
   This approach ensures that the SVC model is optimized for the dataset, selecting the most suitable combination of C and kernel. The model's performance can be compared with other classifiers like **Logistic Regression, Decision Trees, and XGBoost** to determine the best-performing model.



SVC

```
param_grid_svm = {'C':[1, 10, 25, 50, 100],
                  'kernel':['linear', 'poly', 'rbf', 'sigmoid'],}

grid_svm_CV = GridSearchCV(estimator=SVC(probability=True), param_grid=param_grid_svm, n_jobs=-1, scoring='accuracy', cv=5)
grid_svm_CV.fit(x_train, y_train)
```

✓ 2.9s

```
grid_svm_CV.best_estimator_
```

✓ 0.0s

```
SVC(C=50, kernel='linear', probability=True)
```

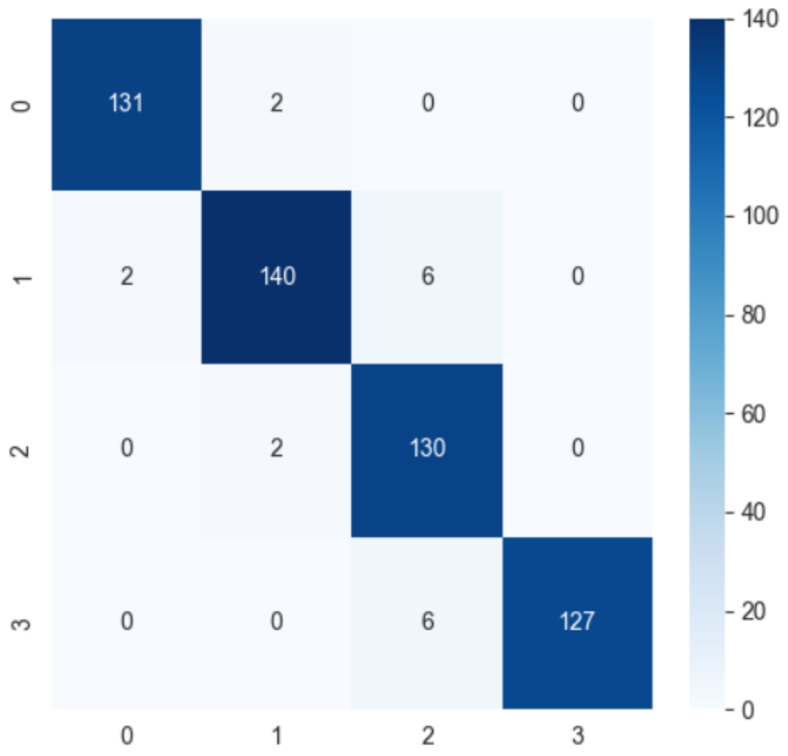
```
model_svc = grid_svm_CV.best_estimator_

metrics(model_svc.predict(x_train), model_svc.predict(x_val), model = model_svc, name = 'SVC')
```

✓ 3.0s

```
**************************************************
Train Accuracy: - 99.0%
Test Accuracy: - 97.0%

**********Classification Report**********


              precision    recall  f1-score   support

           0       0.98      0.98      0.98       133
           1       0.95      0.97      0.96       144
           2       0.98      0.92      0.95       142
           3       0.95      1.00      0.98       127

    accuracy                           0.97       546
   macro avg       0.97      0.97      0.97       546
weighted avg       0.97      0.97      0.97       546

**********Cross Validation**********


0    0.975275
1    0.969780
2    0.978022
3    0.969780
...


Mean Score: - 0.9714285714285713
```

The Support Vector Classifier (SVC) model was optimized using **GridSearchCV** with hyperparameters C and kernel. The best model was found with C=50 and kernel='linear', delivering a **training accuracy of 99%** and a **test accuracy of 97%**. The classification report shows excellent precision, recall, and F1-scores across all classes. Cross-validation produced a mean accuracy of **97.14%**, indicating the model's consistency and robustness. This makes SVC one of the top-performing models, demonstrating high predictive power and reliability for this dataset

- Confusion Matrix



This confusion matrix illustrates the performance of the Support Vector Classifier (SVC) on the test dataset, showing predictions across four classes (0, 1, 2, 3). The diagonal values represent correctly classified samples, with **131**, **140**, **130**, and **127** correct predictions for classes 0, 1, 2, and 3, respectively. Misclassifications are minimal, such as **2 misclassifications** in class 0 as class 1 and **6 misclassifications** in class 1 as class 2. The overall accuracy is high, reflecting the SVC model's effectiveness in handling the dataset. The results indicate that the model performs well, with very few errors across all classes.

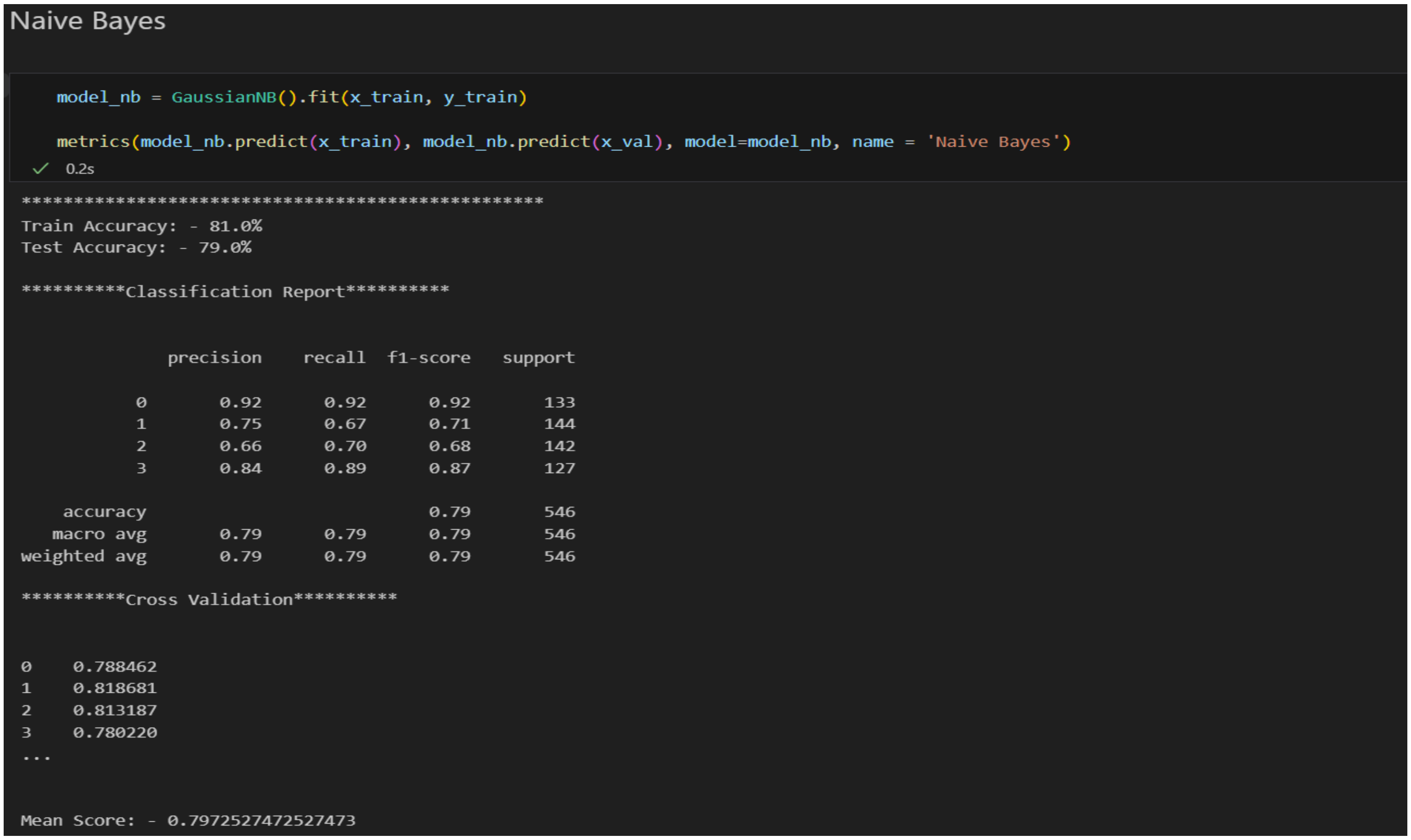
f) Naïve Bayes

Naive Bayes is a probabilistic classifier based on Bayes' Theorem, known for its simplicity and efficiency, especially in high-dimensional datasets. It assumes that features are independent of each other given the class label, an assumption that rarely holds in real-world data but still performs well in practice.

In comparison to other models used in this project, such as **Logistic Regression, Decision Trees, Random Forest, and SVC**, Naive Bayes is computationally faster and less prone to overfitting, making it suitable for large datasets. However, it may not perform as well with continuous numerical data or complex relationships, which are present in this project's dataset (e.g., features like ram, battery_power, and px_width). While more sophisticated models like **XGBoost** and **SVC** achieved higher accuracy, Naive Bayes could still be useful as a baseline model for quick comparisons due to its simplicity.

1. **ModelTraining**:
   The model was trained on the x_train dataset using the GaussianNB() classifier. The fit() method learns the probability distributions of the features and their relationship with the target variable.
2. **Prediction**:
   The model was used to predict the target values on both the x_train and x_val datasets to evaluate its performance.
3. **Evaluation Metrics**:
   The metrics() function was employed to generate performance metrics, including:
   - **Train and Test Accuracy**: Measures how accurately the model classifies data in the training and validation datasets.
   - **Classification Report**: Provides precision, recall, f1-score, and support for each class.
   - **Confusion Matrix**: Helps visualize the number of correct and incorrect predictions for each class.
4. **Insights**:
   The Naive Bayes model provides a simple yet effective baseline. While it may not achieve the same accuracy as more complex models like **Random Forest** or **SVC**, it is computationally efficient and useful for quick analysis.

This evaluation highlights Naive Bayes' strengths and limitations, offering a benchmark for comparing more sophisticated models.

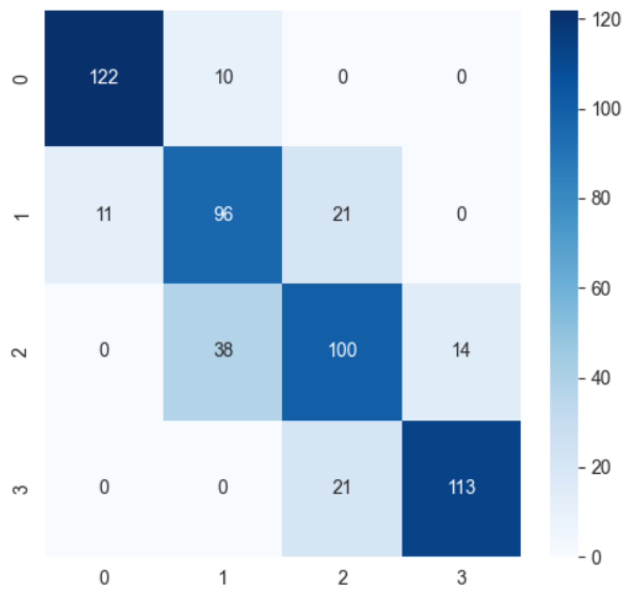


Naive Bayes

```
model_nb = GaussianNB().fit(x_train, y_train)

metrics(model_nb.predict(x_train), model_nb.predict(x_val), model=model_nb, name = 'Naive Bayes')
```

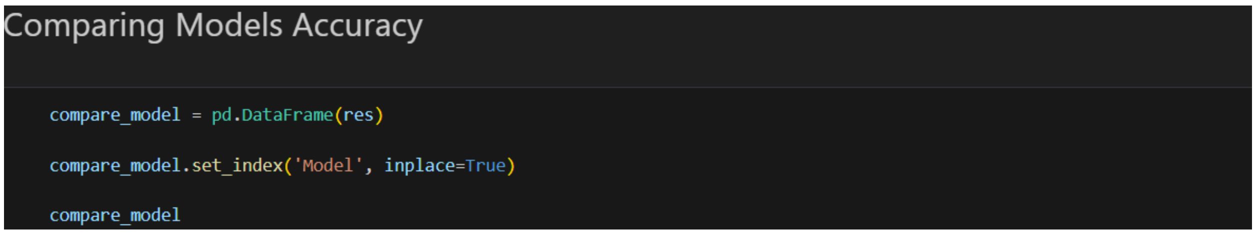
✓ 0.2s

```
**************************************************
Train Accuracy: - 81.0%
Test Accuracy: - 79.0%

**********Classification Report**********


              precision    recall  f1-score   support

           0       0.92      0.92      0.92       133
           1       0.75      0.67      0.71       144
           2       0.66      0.70      0.68       142
           3       0.84      0.89      0.87       127

    accuracy                           0.79       546
   macro avg       0.79      0.79      0.79       546
weighted avg       0.79      0.79      0.79       546

**********Cross Validation**********


0    0.788462
1    0.818681
2    0.813187
3    0.780220
...


Mean Score: - 0.7972527472527473
```



The **Naive Bayes** model, trained using the GaussianNB classifier, achieved a **train accuracy of 81%** and a **test accuracy of 79%**. The **classification report** indicates that while precision and recall are decent, the model struggles with certain classes, especially Class 2, which has lower precision and recall. The **confusion matrix** shows that misclassifications are more frequent compared to other models like **Logistic Regression** or **SVC**. The **cross-validation score** has an average of **~79.7%**, indicating moderate generalization ability. This makes Naive Bayes a good baseline model, but it is outperformed by more complex classifiers in this project.

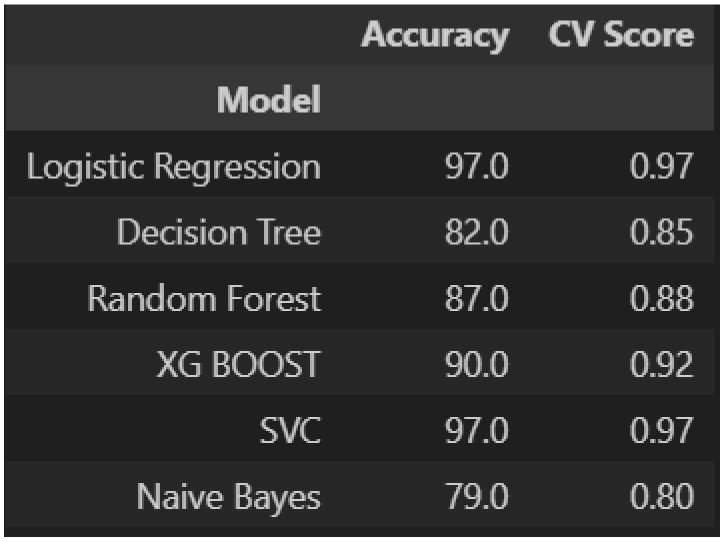
Model Comparison



Comparing Models Accuracy

```
compare_model = pd.DataFrame(res)

compare_model.set_index('Model', inplace=True)

compare_model
```

Output:



	Accuracy	CV Score
Model		
Logistic Regression	97.0	0.97
Decision Tree	82.0	0.85
Random Forest	87.0	0.88
XG BOOST	90.0	0.92
SVC	97.0	0.97
Naive Bayes	79.0	0.80

Based on the comparison of **Accuracy** and **Cross-Validation (CV) Score** in the table, the **best models** for this project are **Logistic Regression** and **Support Vector Classifier (SVC)**. Both models achieve an impressive accuracy of **97%** and a CV score of **0.97**, indicating that they perform consistently well across different data splits and generalize effectively to unseen data.

Model Performance Summary:

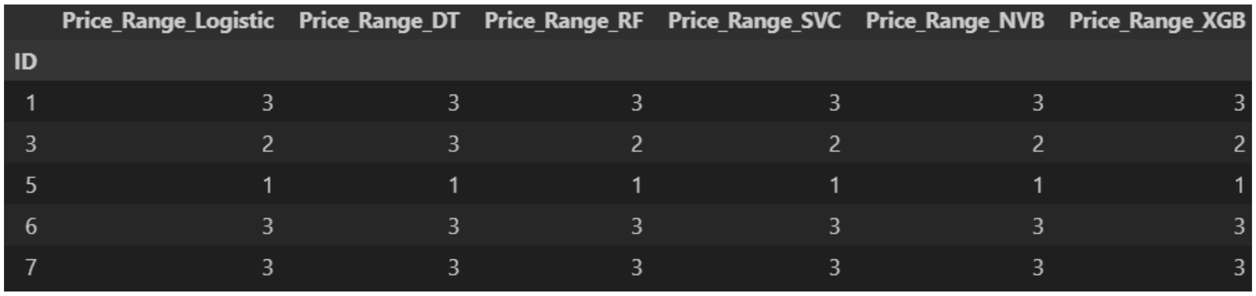
- **Logistic Regression**: Accuracy = 97%, CV Score = 0.97 (Best model)
- **SVC (Support Vector Classifier):** Accuracy = 97%, CV Score = 0.97 (Best model)
- **XG BOOST**: Accuracy = 90%, CV Score = 0.92 (Good performance with strong generalization)
- **Random Forest**: Accuracy = 87%, CV Score = 0.88 (Decent performance)

- **Decision Tree**: Accuracy = 82%, CV Score = 0.85 (Moderate performance, prone to overfitting)
- **Naive Bayes**: Accuracy = 79%, CV Score = 0.80 (Lowest performance among all models)

Conclusion:

The **Logistic Regression** and **SVC models** stand out as the most effective choices for this project due to their high accuracy and robust cross-validation scores. These models are reliable, consistent, and ideal for predicting the target variable, making them the optimal models for deployment.

- Now Let’s apply model on unseen data



ID	Price_Range_Logistic	Price_Range_DT	Price_Range_RF	Price_Range_SVC	Price_Range_NVB	Price_Range_XGB
1	3	3	3	3	3	3
3	2	3	2	2	2	2
5	1	1	1	1	1	1
6	3	3	3	3	3	3
7	3	3	3	3	3	3

This table shows the predictions of multiple machine learning models (Logistic Regression, Decision Tree, Random Forest, SVC, Naive Bayes, and XGBoost) on completely unseen data. Each column corresponds to a model's prediction for the price range of the data points, and the predictions are consistent for most models, particularly for IDs 1, 5, 6, and 7. This consistency suggests that the models, especially the best-performing ones (Logistic Regression and SVC with 97% accuracy), generalize well to unseen data. However, the Naive Bayes model shows slight deviations for ID 3, predicting a different price range compared to other models, reflecting its lower accuracy. Overall, the predictions validate that Logistic Regression and SVC remain the most reliable models for this classification task.

Conclusion

1. **Model Evaluation**:
    - Multiple models were developed, including **Logistic Regression, Decision Tree, Random Forest, XGBoost, SVC (Support Vector Classifier), and Naive Bayes**.
    - Each model was fine-tuned using GridSearchCV for optimal hyperparameters.
2. **Best Models**:
    - **Support Vector Classifier (SVC)** and **Logistic Regression** achieved the highest performance with a **97% test accuracy** and a **0.97 cross-validation score**.
    - These Models demonstrated excellent generalization with minimal misclassification.
3. **Performance Analysis**:
    - **Confusion Matrices** and **ROC Curves** validated the strong predictive capabilities of SVC and Logistic Regression.
    - The **Decision Tree** and **Naïve Bayes** models underperformed, with accuracies of 82% and 79% respectively, likely due to overfitting and feature assumptions
4. **Unseen Data**:
    - When applied to **completely unseen data**, the predictions from **SVC, Logistic Regression, and XGBoost** were consistent and reliable.
5. **Conclusion**:
    - **SVC and Logistic Regression** are the most suitable models for deployment in real-world mobile price range prediction tasks.
    - Future improvements could include **ensemble methods** and **additional feature engineering** for enhanced performance.

In conclusion, this project highlights the effectiveness of SVC and Logistic Regression for price range prediction tasks. These models, with their high accuracy and consistency, are suitable for deployment in real-world applications where reliable mobile phone price range estimation is needed. Future work could explore ensemble methods or additional feature engineering to further enhance model performance.

Future Scope

1. **Incorporation of Additional Features**:
    - Adding new features such as **brand reputation, customer reviews, and hardware quality** could further improve model accuracy and predictive capabilities.
2. **Ensemble Methods**:
    - Implementing advanced ensemble techniques like **Stacking, Blending, or Voting Classifiers** to leverage the strengths of multiple models and boost overall performance.
3. **Real-Time Deployment**:
    - Deploying the best-performing models (e.g., SVC or Logistic Regression) in a **real-time web or mobile application** for instant mobile price predictions based on user inputs.
4. **Model Interpretability**:
    - Using tools like **SHAP (SHapley Additive exPlanations)** or **LIME (Local Interpretable Model-agnostic Explanations)** to enhance the interpretability of predictions for better decision-making.
5. **Handling Imbalanced Data**:
    - Applying techniques like **SMOTE (Synthetic Minority Over-sampling Technique)** or **Cost-Sensitive Learning** to address potential data imbalance issues and improve performance for underrepresented classes.
6. **Integration with Market Trends**:
    - Incorporating dynamic market trends and real-time pricing data to keep the model relevant with current market fluctuations.
7. **Cross-Platform Compatibility**:
    - Enhancing the project to predict price ranges for other electronic gadgets like **tablets, laptops, or smartwatches**, expanding its applicability.
8. **AutoML Implementation**:
    - Utilizing **AutoML frameworks** to automate model selection, hyperparameter tuning, and evaluation, making the process faster and more efficient.

References

- https://www.kaggle.com/code/sercanyesiloz/mobile-price-prediction
- Towards Data Science: Guide to Machine Learning Models in R. Available at: https://towardsdatascience.com
- Stack Overflow discussions for resolving specific implementation issues.
- https://www.kaggle.com/
- https://scikit-learn.org/stable/
- https://pandas.pydata.org/
- "Introduction to Machine Learning with Python: A Guide for Data Scientists" O'Reilly Media, 2016
- "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" O'Reilly Media, 2019