Supervised Learning - Foundations Project: ReCell

Project Overview

This project uses linear regression to predict the normalized resale price of used mobile devices based on technical specifications and usage data. The goal is to support dynamic pricing strategies with interpretable, assumption-validated models. Final recommendations align with business logic, model insights, and market relevance.

Problem Statement

Business Context

Buying and selling used phones and tablets used to be something that happened on a handful of online marketplace sites. But the used and refurbished device market has grown considerably over the past decade, and a new IDC (International Data Corporation) forecast predicts that the used phone market would be worth $52.7bn by 2023 with a compound annual growth rate (CAGR) of 13.6% from 2018 to 2023. This growth can be attributed to an uptick in demand for used phones and tablets that offer considerable savings compared with new models.

Refurbished and used devices continue to provide cost-effective alternatives to both consumers and businesses that are looking to save money when purchasing one. There are plenty of other benefits associated with the used device market. Used and refurbished devices can be sold with warranties and can also be insured with proof of purchase. Third-party vendors/platforms, such as Verizon, Amazon, etc., provide attractive offers to customers for refurbished devices. Maximizing the longevity of devices through second-hand trade also reduces their environmental impact and helps in recycling and reducing waste. The impact of the COVID-19 outbreak may further boost this segment as consumers cut back on discretionary spending and buy phones and tablets only for immediate needs.

Objective

The rising potential of this comparatively under-the-radar market fuels the need for an ML-based solution to develop a dynamic pricing strategy for used and refurbished devices. ReCell, a startup aiming to tap the potential in this market, has hired you as a data scientist. They want you to analyze the data provided and build a linear regression model to predict the price of a used phone/tablet and identify factors that significantly influence it.

Data Description

The data contains the different attributes of used/refurbished phones and tablets. The data was collected in the year 2021. The detailed data dictionary is given below.

- brand_name: Name of manufacturing brand
- os: OS on which the device runs
- screen_size: Size of the screen in cm
- 4g: Whether 4G is available or not
- 5g: Whether 5G is available or not
- main_camera_mp: Resolution of the rear camera in megapixels
- selfie_camera_mp: Resolution of the front camera in megapixels
- int_memory: Amount of internal memory (ROM) in GB
- ram: Amount of RAM in GB
- battery: Energy capacity of the device battery in mAh
- weight: Weight of the device in grams
- release_year: Year when the device model was released
- days_used: Number of days the used/refurbished device has been used
- normalized_new_price: Normalized price of a new device of the same model in euros
- normalized_used_price: Normalized price of the used/refurbished device in euros

Importing Necessary Libraries

```
# Installing the libraries with the specified version.
# if Google Colab is being used
!pip install scikit-learn>=1.2.2 seaborn>=0.13.1 matplotlib>=3.7.1 numpy>=1.25.2 pandas>=1.5.3 -q --user
```

Note: *After running the above cell, kindly restart the notebook kernel and run all cells sequentially from the start again.*

```
# Import EDA Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import colorsys
import seaborn as sns

sns.set()

import scipy.stats as stats
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
# Import Linear Regression Libraries

# Split the data into train and test
from sklearn.model_selection import train_test_split

# To build linear regression_model using statsmodels
```

```
import statsmodels.api as sm

# To check model performance
from sklearn.metrics import mean_absolute_error, mean_squared_error

# To compute VIF
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

```
# Custom Color Set
ElleSet = [
    (102/255, 194/255, 165/255),  # Muted Green
    (214/255, 95/255, 95/255),    # Muted Red
    (141/255, 160/255, 203/255),  # Soft Blue
    (130/255, 198/255, 226/255),  # Muted Blue
    (166/255, 216/255, 84/255),   # Lime Green
    (230/255, 196/255, 148/255),  # Beige
    (179/255, 179/255, 179/255),  # Neutral Gray
    (255/255, 217/255, 47/255),   # Yellow
    (204/255, 153/255, 255/255),  # Soft Lavender
    (255/255, 153/255, 204/255)   # Blush pink
]

# Function to apply ElleSet globally in Seaborn
def use_ElleSet():
    sns.set_palette(ElleSet)
```

```
# Remove Future Warnings
import warnings

# Suppress all FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
# Mount Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Load the Dataset

```
# Load Dataset
data = pd.read_csv('/content/drive/MyDrive/Supervised Learning/used_device_data.csv')
```

Data Overview

Initial Review

```
data.head()
```

	brand_name	os	screen_size	4g	5g	main_camera_mp	selfie_camera_mp	int_memory	ram	battery	weight	release_year	days_used	normalized_used_price	normalized_new_price
0	Honor	Android	14.50	yes	no	13.0	5.0	64.0	3.0	3020.0	146.0	2020	127	4.307572	4.715100
1	Honor	Android	17.30	yes	yes	13.0	16.0	128.0	8.0	4300.0	213.0	2020	325	5.162097	5.519018
2	Honor	Android	16.69	yes	yes	13.0	8.0	128.0	8.0	4200.0	213.0	2020	162	5.111084	5.884631
3	Honor	Android	25.50	yes	yes	13.0	8.0	64.0	6.0	7250.0	480.0	2020	345	5.135387	5.630961
4	Honor	Android	15.32	yes	no	13.0	8.0	64.0	3.0	5000.0	185.0	2020	293	4.389995	4.947837

- The dataset contains 15 columns describing various features of used devices, including brand name, screen size, battery capacity, release year, days used, and both normalized new and used prices.

```
data.tail()
```

	brand_name	os	screen_size	4g	5g	main_camera_mp	selfie_camera_mp	int_memory	ram	battery	weight	release_year	days_used	normalized_used_price	normalized_new_price
3449	Asus	Android	15.34	yes	no	NaN	8.0	64.0	6.0	5000.0	190.0	2019	232	4.492337	6.483872
3450	Asus	Android	15.24	yes	no	13.0	8.0	128.0	8.0	4000.0	200.0	2018	541	5.037732	6.251538
3451	Alcatel	Android	15.80	yes	no	13.0	5.0	32.0	3.0	4000.0	165.0	2020	201	4.357350	4.528829
3452	Alcatel	Android	15.80	yes	no	13.0	5.0	32.0	2.0	4000.0	160.0	2020	149	4.349762	4.624188
3453	Alcatel	Android	12.83	yes	no	13.0	5.0	16.0	2.0	4000.0	168.0	2020	176	4.132122	4.279994

Shape Check

```
data.shape
```

```
(3454, 15)
```

- The dataset contains information about a sample of 3,454 used devices.

Feature Datatypes

```
data.info(memory_usage=False)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3454 entries, 0 to 3453
Data columns (total 15 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   brand_name             3454 non-null   object
 1   os                     3454 non-null   object
 2   screen_size            3454 non-null   float64
 3   4g                     3454 non-null   object
 4   5g                     3454 non-null   object
 5   main_camera_mp         3275 non-null   float64
 6   selfie_camera_mp       3452 non-null   float64
 7   int_memory             3450 non-null   float64
 8   ram                    3450 non-null   float64
 9   battery                3448 non-null   float64
 10  weight                 3447 non-null   float64
 11  release_year           3454 non-null   int64
 12  days_used              3454 non-null   int64
 13  normalized_used_price  3454 non-null   float64
 14  normalized_new_price   3454 non-null   float64
dtypes: float64(9), int64(2), object(4)
```

- Most columns are appropriately typed, with numerical features stored as float64 or int64 and categorical variables as object. The 4g and 5g columns are currently stored as objects but represent binary indicators; they should be converted to numeric format to be used effectively as predictors in the regression model.
- Some columns contain missing values; further exploration is required to assess their significance and inform imputation or exclusion decisions.
- The target variable normalized_used_price is continuous and appropriate for linear regression modeling.

Statistical Summary of Data

```
data.describe().round(2)
```

	screen_size	main_camera_mp	selfie_camera_mp	int_memory	ram	battery	weight	release_year	days_used	normalized_used_price	normalized_new_price
count	3454.00	3275.00	3452.00	3450.00	3450.00	3448.00	3447.00	3454.00	3454.00	3454.00	3454.00
mean	13.71	9.46	6.55	54.57	4.04	3133.40	182.75	2015.97	674.87	4.36	5.23
std	3.81	4.82	6.97	84.97	1.37	1299.68	88.41	2.30	248.58	0.59	0.68
min	5.08	0.08	0.00	0.01	0.02	500.00	69.00	2013.00	91.00	1.54	2.90
25%	12.70	5.00	2.00	16.00	4.00	2100.00	142.00	2014.00	533.50	4.03	4.79
50%	12.83	8.00	5.00	32.00	4.00	3000.00	160.00	2015.50	690.50	4.41	5.25
75%	15.34	13.00	8.00	64.00	4.00	4000.00	185.00	2018.00	868.75	4.76	5.67
max	30.71	48.00	32.00	1024.00	12.00	9720.00	855.00	2020.00	1094.00	6.62	7.85

- Most numerical features, such as memory, battery, and weight, show wide ranges between minimum and maximum values, reflecting a diverse mix of device types and tiers.
- The target variable, normalized_used_price, has a moderately concentrated distribution (mean ~4.36), suggesting some consistency in resale pricing across the dataset.
- A few features, including main_camera_mp and weight, contain small amounts of missing data, but overall data completeness is high.

Duplicate Values Review

```
data.duplicated().sum()
```

```
np.int64(0)
```

- No duplicate rows were found in the dataset, indicating that all entries are unique across all columns, including those with categorical (object) data.

Missing Value Review

```
data.isnull().sum().loc[lambda x: x > 0]
```

	0
main_camera_mp	179
selfie_camera_mp	2

int_memory	4
ram	4
battery	6
weight	7

dtype: int64

```
# Missing Value Summary
total_nulls = data.isnull().sum()
num_cols_with_nulls = (total_nulls > 0).sum()
total_null_values = total_nulls.sum()

print(f"{num_cols_with_nulls} columns contain missing values, with a total of {total_null_values} missing entries.")
```

```
6 columns contain missing values, with a total of 202 missing entries.
```

- Six columns contain a total of 202 missing values, most of which appear in numerical features; further analysis will determine if imputation or exclusion is appropriate.

Category Consistency Review

```
# Review Category Consistency
# Identify object (categorical) columns
cat_cols = data.select_dtypes(include='object').columns

# Dictionary to collect results
category_summary = {}

# Check unique values and print for review
for col in cat_cols:
    unique_vals = data[col].value_counts(dropna=False)
    print(f"\n Unique values in '{col}':")
    print(unique_vals)
    category_summary[col] = len(unique_vals)

# Summary of category counts
print("\n Summary of Unique Categories per Column:")
for col, count in category_summary.items():
    print(f"- {col}: {count} unique values")
```

```
 Unique values in 'brand_name':
brand_name
Others        502
Samsung       341
Huawei        251
LG            201
Lenovo        171
ZTE           140
Xiaomi        132
Oppo          129
Asus          122
Alcatel       121
Micromax      117
Vivo          117
Honor         116
HTC           110
Nokia         106
Motorola      106
Sony           86
Meizu          62
Gionee         56
Acer           51
XOLO           49
Panasonic      47
Realme         41
Apple          39
Lava           36
Celkon         33
Spice          30
Karbonn        29
BlackBerry     22
OnePlus        22
Microsoft      22
Coolpad        22
Google         15
Infinix        10
Name: count, dtype: int64

 Unique values in 'os':
os
Android    3214
Others      137
Windows      67
iOS          36
```

```
Name: count, dtype: int64

 Unique values in '4g':
4g
yes    2335
no     1119
Name: count, dtype: int64

 Unique values in '5g':
5g
no     3302
yes     152
Name: count, dtype: int64

 Summary of Unique Categories per Column:
- brand_name: 34 unique values
- os: 4 unique values
- 4g: 2 unique values
- 5g: 2 unique values
```

- All categorical columns (`brand_name` , `os` , `4g` , and `5g`) contain expected values with no visible typos, unexpected categories, or unusual entries.

Normalized Value Review

```python
# Check for normalized values
for col in cat_cols:
    print(f"\n Normalized preview of '{col}':")
    print(data[col].astype(str).str.strip().str.lower().value_counts())
```

```
 Normalized preview of 'brand_name':
brand_name
others        502
samsung       341
huawei        251
lg            201
lenovo        171
zte           140
xiaomi        132
oppo          129
asus          122
alcatel       121
micromax      117
vivo          117
honor         116
htc           110
nokia         106
motorola      106
sony           86
meizu          62
gionee         56
acer           51
xolo           49
panasonic      47
realme         41
apple          39
lava           36
celkon         33
spice          30
karbonn        29
blackberry     22
oneplus        22
microsoft      22
coolpad        22
google         15
infinix        10
Name: count, dtype: int64

 Normalized preview of 'os':
os
android    3214
others      137
windows      67
ios          36
Name: count, dtype: int64

 Normalized preview of '4g':
4g
yes    2335
no     1119
Name: count, dtype: int64

 Normalized preview of '5g':
5g
no     3302
yes     152
Name: count, dtype: int64
```

- Lowercasing and trimming whitespace revealed no inconsistencies in formatting; category values are clean and well-standardized.

Copy of Dataset

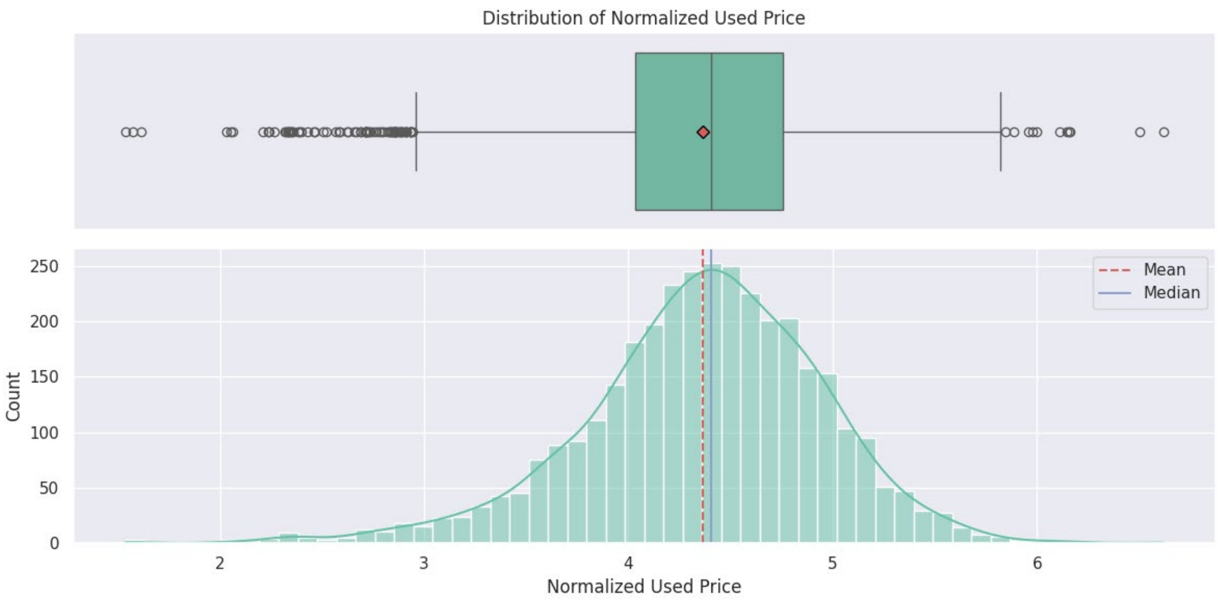
```
df = data.copy()
```

Exploratory Data Analysis (EDA)

Univariate Analysis

Feature: Normalized Used Price

```
# Shape of Distribution

# Calculate values
mean_val = df['normalized_used_price'].mean()
median_val = df['normalized_used_price'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['normalized_used_price'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Normalized Used Price')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['normalized_used_price'].dropna(),
    ax=ax_hist,
    kde=True,
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Normalized Used Price')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```



- The distribution of normalized used prices is approximately symmetric, with a slight right skew, indicating that most used devices fall within a common value range but a few high-value devices extend the upper tail.
- The mean and median are closely aligned, suggesting that pricing in the used market is relatively stable and not heavily influenced by extreme outliers.
- The central tendency around €4.3–€4.4 (normalized scale) reflects consistent resale value across mainstream devices, reinforcing predictability for price modeling.
- A small number of lower-priced devices may represent older or lower-spec models, while higher outliers could reflect recently released or lightly used premium-tier devices.
- The structure of the used price distribution supports linear modeling, as the range is well-behaved and does not require transformation at this stage.

Feature: Normalized New Price

```
#Shape of Distribution

# Calculate values
mean_val = df['normalized_new_price'].mean()
median_val = df['normalized_new_price'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['normalized_new_price'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Normalized New Price')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['normalized_new_price'].dropna(),
    ax=ax_hist,
    kde=True,
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Normalized New Price')
```

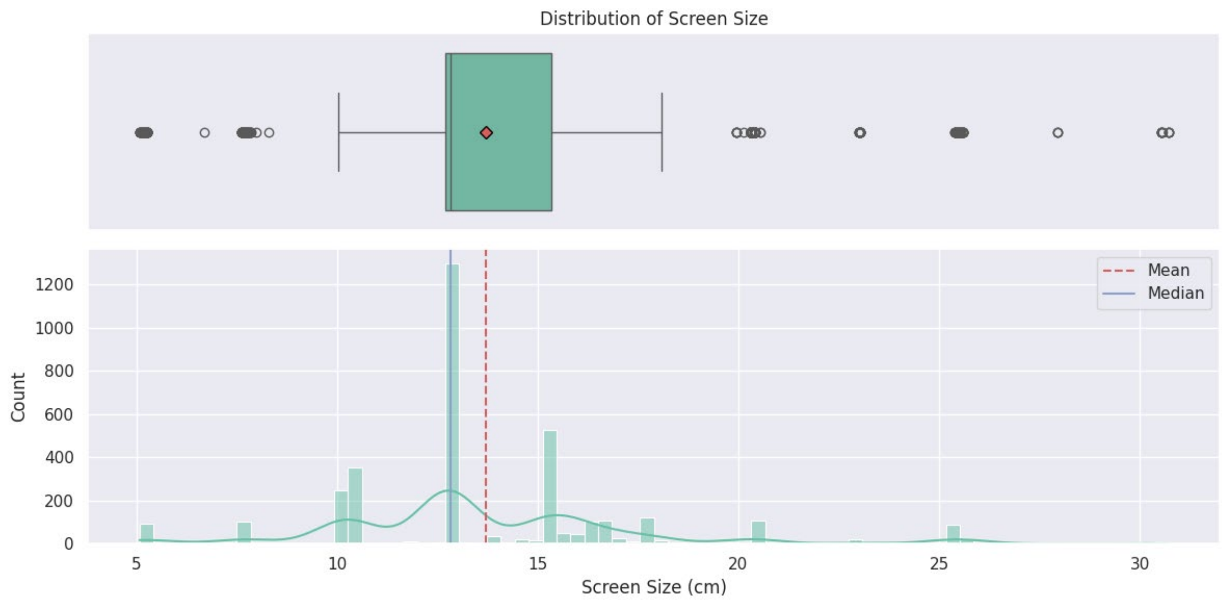
```
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```

- The normalized new price distribution is bell-shaped but with a long left tail, suggesting that while most devices launch at similar price points, some budget models enter the market at significantly lower values.
- Mean and median are closely aligned, indicating relatively balanced pricing across the majority of devices at the time of release.
- The central clustering of new prices reflects market standardization, with manufacturers releasing a large volume of models within a predictable pricing band.
- Devices with significantly lower original prices could reflect entry-level models, which may show different depreciation patterns than mid- or high-tier devices.
- Understanding the relationship between new and used prices across this distribution will be critical for modeling value retention — one of the core business drivers for ReCell.

Feature: Screen Size

```
# Shape of Distribution
# Calculate values
mean_val = df['screen_size'].mean()
median_val = df['screen_size'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['screen_size'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Screen Size')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['screen_size'].dropna(),
    ax=ax_hist,
    kde=True,
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')
```

```
# Final touches
ax_hist.set_xlabel('Screen Size (cm)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```



- The distribution of screen size is right-skewed, with most devices clustered between 12 and 16 cm — indicating a dominance of standard smartphone dimensions in the used device market.
- Sharp spikes in the distribution suggest clustering around popular screen sizes, likely reflecting common model releases and consumer preferences.
- A small number of devices exceed 20 cm in screen size, likely representing tablets or specialty devices, which may follow different pricing dynamics.
- The mean and median screen sizes are close, indicating general symmetry in the core range, with mild skew introduced by larger-screen outliers.
- Segmenting by screen size may support differentiated pricing strategies, particularly for distinguishing between standard smartphones and larger form-factor devices such as tablets.

Feature: Main Camera MP

```
# Shape of Distribution
# Calculate values
mean_val = df['main_camera_mp'].mean()
median_val = df['main_camera_mp'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['main_camera_mp'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Main Camera Megapixels')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['main_camera_mp'].dropna(),
    ax=ax_hist,
    kde=True,
    color=ElleSet[0]
```

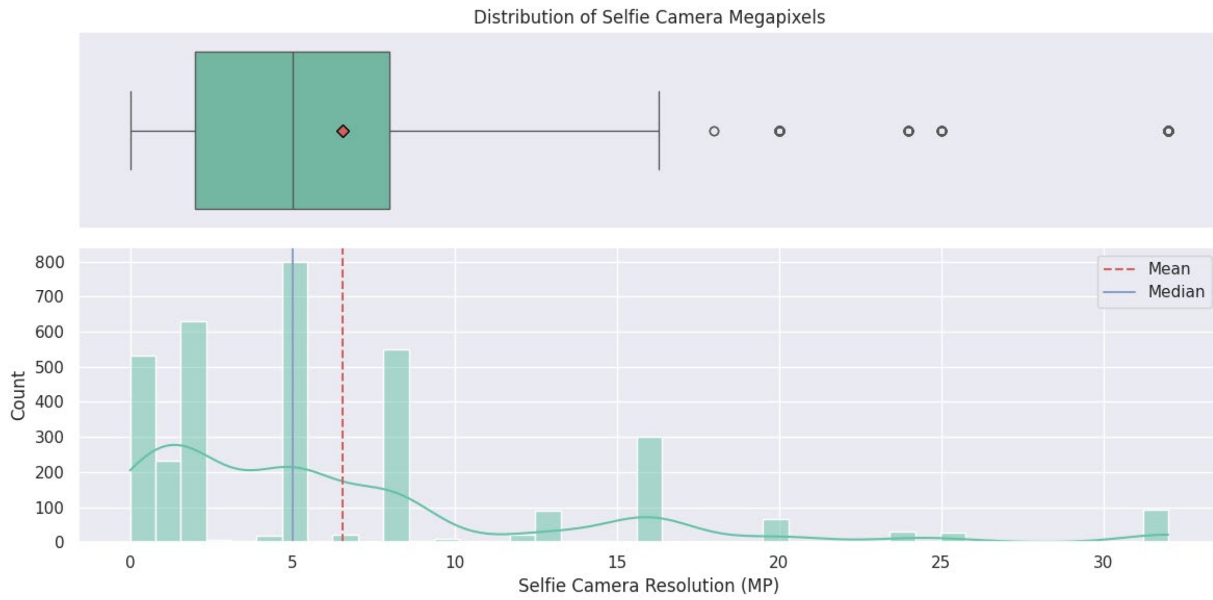
```
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Main Camera Resolution (MP)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```

- The distribution is not normal. It is right-skewed and multimodal, with most devices clustering around specific camera resolutions such as 8 MP, 12 MP, and 16 MP, indicating a concentration around standard-resolution smartphone models.
- The mean is slightly higher than the median, reflecting a modest skew caused by a smaller number of high-resolution camera models extending beyond 20 MP.
- There is visible multimodality, suggesting that certain megapixel values (e.g., 8 MP, 12 MP, 16 MP) correspond to popular models or manufacturing standards.
- A few high-resolution outliers (above 40 MP) likely reflect premium-tier or newer generation devices — these may carry higher resale value and could serve as useful predictors in pricing models.

Feature: Selfie Camera MP

```
# Shape of Distribution
# Calculate values
mean_val = df['selfie_camera_mp'].mean()
median_val = df['selfie_camera_mp'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['selfie_camera_mp'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Selfie Camera Megapixels')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['selfie_camera_mp'].dropna(),
    ax=ax_hist,
    kde=True,
```

```
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Selfie Camera Resolution (MP)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```



- The distribution is strongly right-skewed and non-normal, with the majority of devices offering selfie cameras below 8 MP and a long tail extending beyond 30 MP.
- The mean is noticeably higher than the median, suggesting that a few high-resolution front-facing cameras are influencing the average, despite being relatively rare.
- Multiple sharp peaks suggest clustering around standard resolutions such as 2 MP, 5 MP, and 8 MP — likely reflecting device class or generation-specific design patterns.
- Outliers above 20 MP are uncommon and may indicate premium-tier models, potentially contributing to price prediction but requiring evaluation for leverage or undue influence in regression.

Feature: Internal Memory

```
# Shape of Distribution
# Calculate values
mean_val = df['int_memory'].mean()
median_val = df['int_memory'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['int_memory'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Internal Memory (GB)')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['int_memory'].dropna(),
    ax=ax_hist,
```

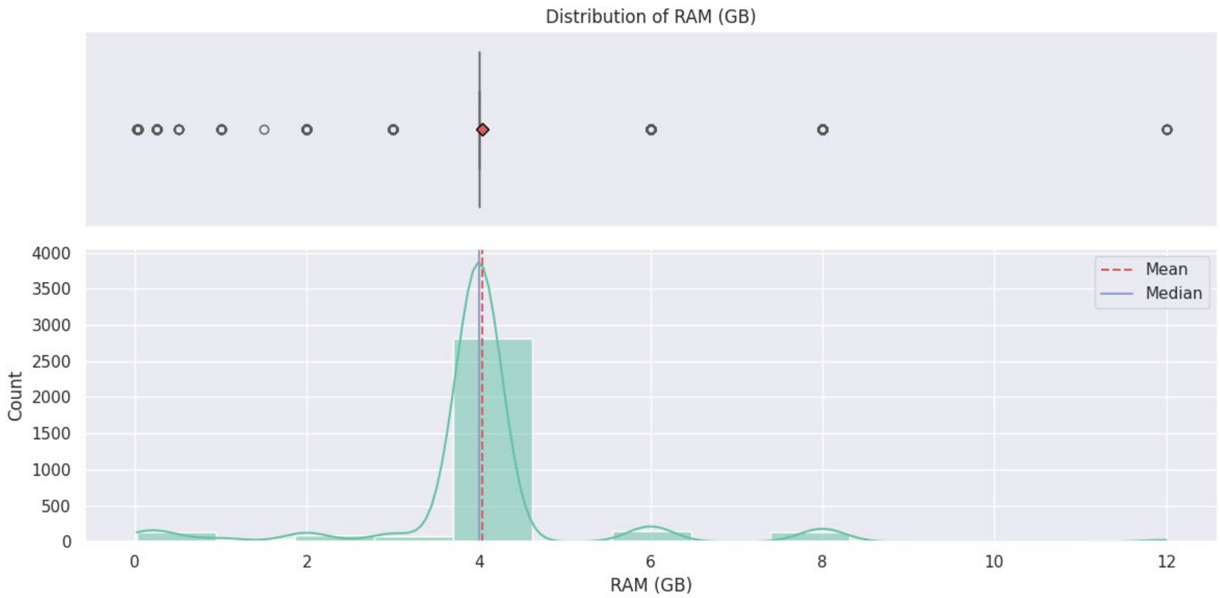
```
    kde=True,
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Internal Memory (GB)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```

- The distribution is highly right-skewed and non-normal, with most devices offering internal memory between 16 GB and 128 GB, and a small number of extreme outliers extending up to 1,024 GB.
- The mean is significantly higher than the median, indicating that the few ultra-high-memory devices disproportionately pull the average upward.
- Distinct peaks around common storage sizes (e.g., 32 GB, 64 GB, 128 GB) suggest standardization across device generations — a pattern that may help categorize devices into pricing tiers.
- The extreme outliers (256 GB and above) may distort model behavior if not addressed, especially given the use of linear regression. Consider capping, transforming, or creating categorical bins during preprocessing.

Feature: RAM

```
# Shape of Distribution
# Calculate values
mean_val = df['ram'].mean()
median_val = df['ram'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['ram'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of RAM (GB)')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['ram'].dropna(),
```

```
    ax=ax_hist,
    kde=True,
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('RAM (GB)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```

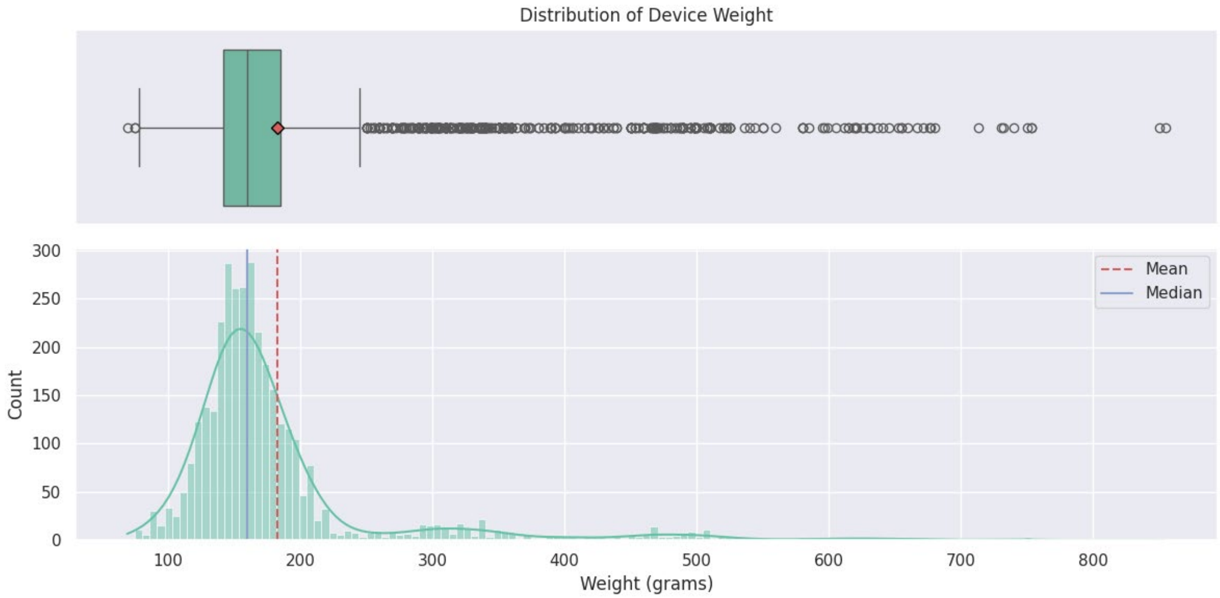


- RAM is effectively constant across the majority of devices (4 GB), which limits its predictive value for price modeling.
- Given the narrow distribution of RAM values across the dataset, further brand-wise comparison does not appear necessary, as variation is minimal and likely not informative for modeling.

Feature: Weight

```
# Shape of Distribution
# Calculate values
mean_val = df['weight'].mean()
median_val = df['weight'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['weight'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Device Weight')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['weight'].dropna(),
    ax=ax_hist,
    kde=True,
    color=ElleSet[0]
```

```
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Weight (grams)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```

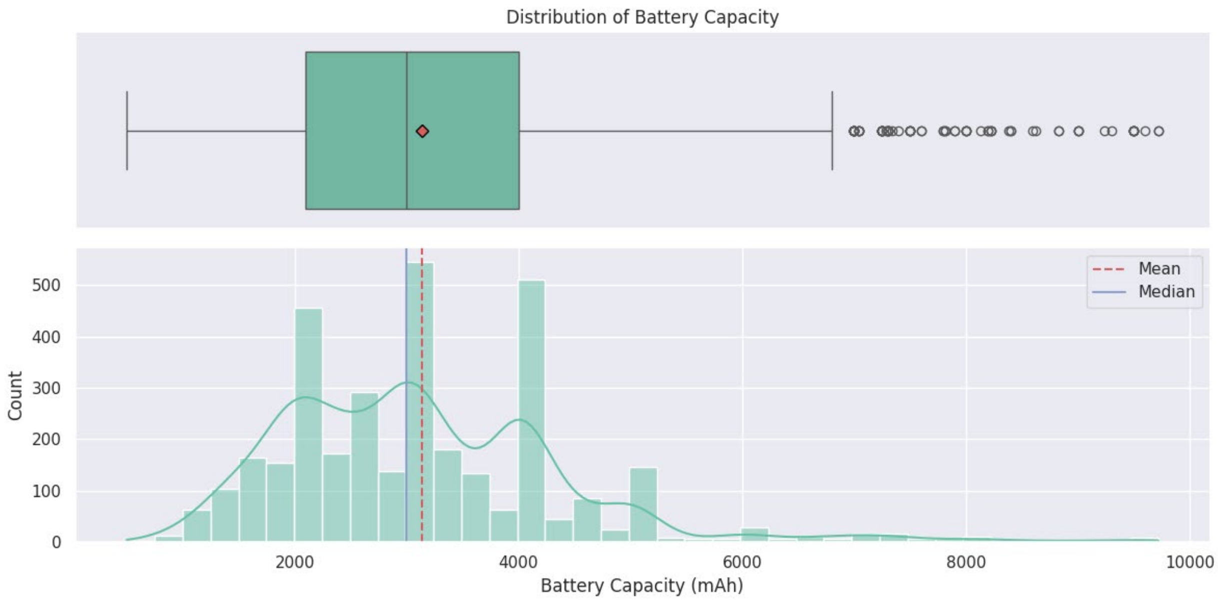


- The distribution is strongly right-skewed and non-normal, with most devices concentrated between 120–220 grams and a long tail extending past 800 grams.
- The mean is higher than the median, suggesting that a small number of heavier devices (likely tablets or rugged phones) pull the average upward.
- Numerous mild outliers and a few extreme ones are visible above 300 grams. These may reflect specialty devices and should be evaluated for undue influence on the model.
- The core cluster suggests consistent weight standards across mainstream devices, which may help identify anomalies or differentiate product classes during feature engineering.

Feature: Battery

```
# Shape of Distribution
# Calculate values
mean_val = df['battery'].mean()
median_val = df['battery'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['battery'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Battery Capacity')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['battery'].dropna(),
    ax=ax_hist,
    kde=True,
```

```
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final touches
ax_hist.set_xlabel('Battery Capacity (mAh)')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```

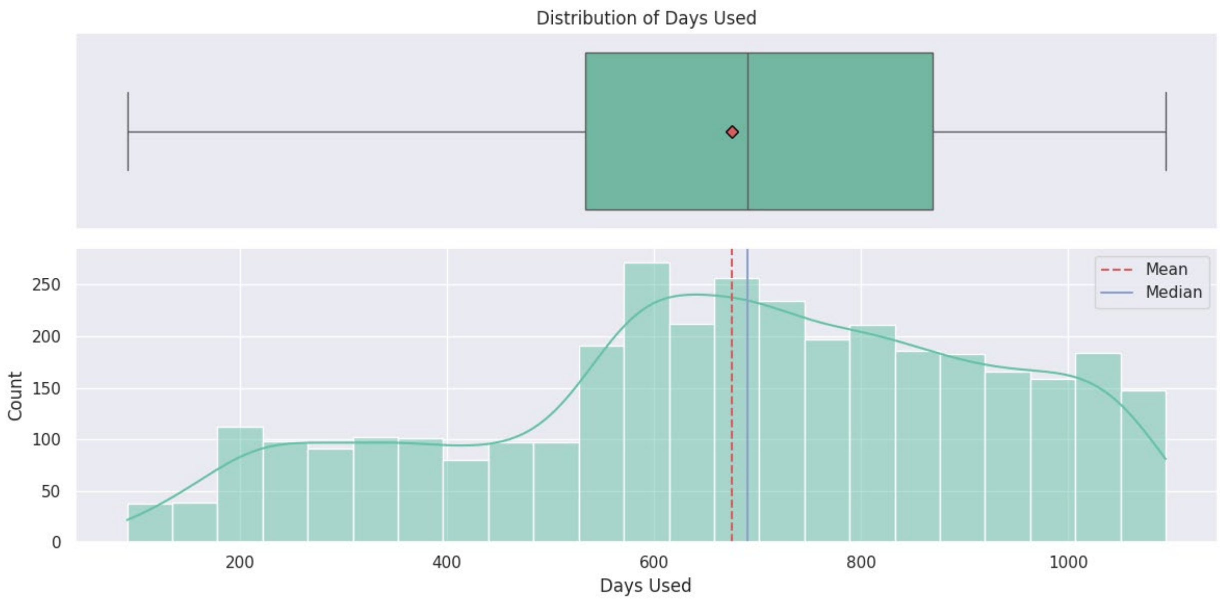


- The distribution is moderately right-skewed, with most devices clustering between 2,000 and 4,000 mAh, consistent with standard smartphone battery capacities.
- The mean and median are close, indicating relative symmetry within the core range, although a small number of high-capacity outliers (6,000 mAh and above) extend the tail.
- Multiple peaks in the distribution suggest manufacturing or design clustering, likely reflecting popular device series or hardware generations.
- Extreme outliers approaching 10,000 mAh may represent tablets, rugged phones, or niche devices, and should be reviewed for influence during modeling.

Feature: Days Used

```
# Shape of Distribution
# Calculate values
mean_val = df['days_used'].mean()
median_val = df['days_used'].median()

# Create combined layout
fig, (ax_box, ax_hist) = plt.subplots(
    nrows=2,
    sharex=True,
    figsize=(12, 6),
    gridspec_kw={"height_ratios": (0.4, 0.6)}
)

# Boxplot
sns.boxplot(
    x=df['days_used'].dropna(),
    ax=ax_box,
    color=ElleSet[0],
    showmeans=True,
    meanprops={"marker": "D", "markerfacecolor": ElleSet[1], "markeredgecolor": "black"}
)
ax_box.set(title='Distribution of Days Used')
ax_box.set(xlabel=None)
ax_box.grid(False)

# Histogram
sns.histplot(
    df['days_used'].dropna(),
    ax=ax_hist,
```

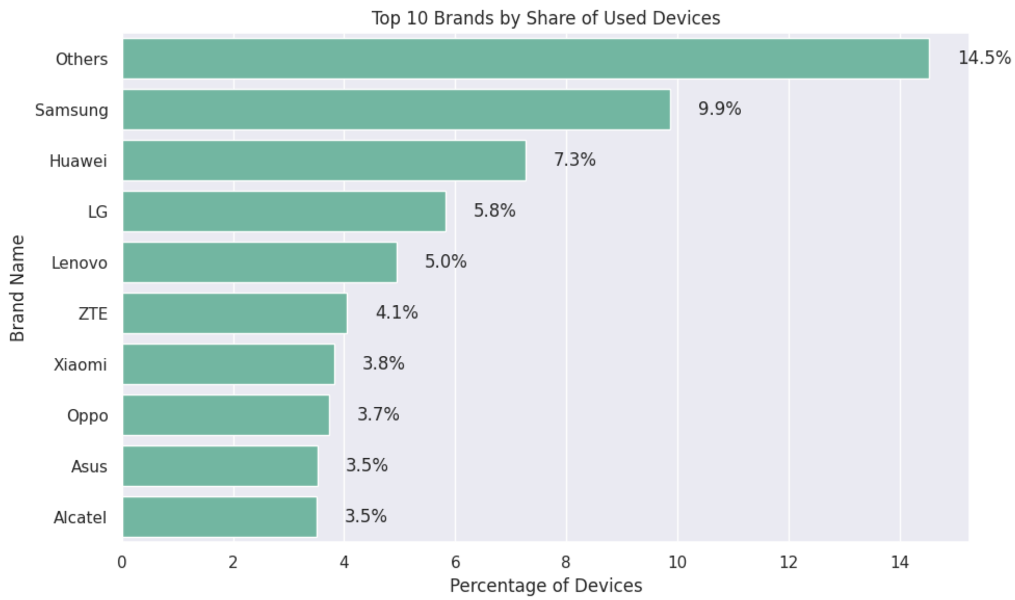
```
    kde=True,
    color=ElleSet[0]
)
ax_hist.axvline(mean_val, color=ElleSet[1], linestyle='--', label='Mean')
ax_hist.axvline(median_val, color=ElleSet[2], linestyle='-', label='Median')

# Final styling
ax_hist.set_xlabel('Days Used')
ax_hist.set_ylabel('Count')
ax_hist.legend()
plt.tight_layout()
plt.show()
```



- The distribution is slightly left-skewed, with most devices used between 500 and 1,000 days, suggesting that the majority of units in this dataset are 1.5 to 3 years old.
- The mean and median are closely aligned, indicating a stable and consistent usage duration across devices, with minimal influence from extreme low-use or high-use values.
- There is a long tail on the lower end, with a small number of devices used for fewer than 200 days — these may be near-new and could command a higher resale price.
- The lack of extreme outliers or irregularities supports this feature's inclusion in regression modeling, as it appears well-distributed and reliable as a predictor of condition-related depreciation.

Feature: Brand Name

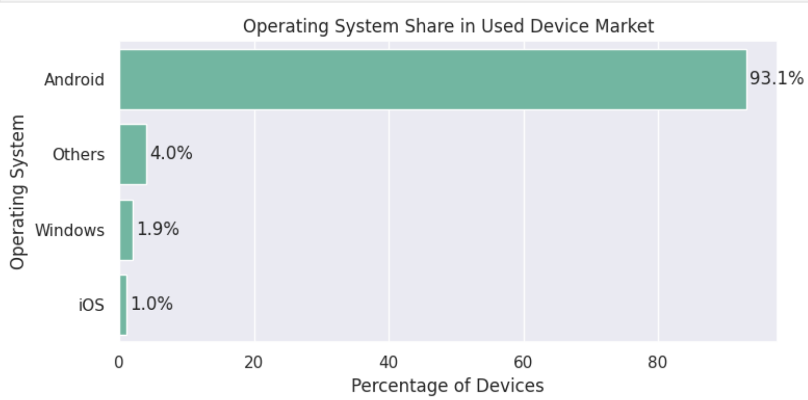
```
# Relative Frequency
# Top 10 brands by relative frequency
top_brands = df['brand_name'].value_counts(normalize=True).head(10)

# Plot
plt.figure(figsize=(10, 6))
sns.barplot(x=top_brands.values * 100, y=top_brands.index, color=ElleSet[0])

# Labels
for i, v in enumerate(top_brands.values):
    plt.text(v * 100 + 0.5, i, f"{v*100:.1f}%", va='center')

# Final touches
plt.title('Top 10 Brands by Share of Used Devices')
plt.xlabel('Percentage of Devices')
plt.ylabel('Brand Name')
plt.tight_layout()
plt.show()
```



- The used device market is dominated by a small number of brands, with the top 3 brands accounting for a significant share of the dataset — indicating strong brand clustering that could influence both pricing and feature distribution.
- Brand distribution is highly imbalanced, suggesting that certain brands may require grouped encoding or category consolidation to avoid overfitting or underrepresentation in the model.
- Brand identity may serve as a strong proxy for overall build quality, expected lifespan, or resale value, making it a potentially powerful predictor once encoded appropriately for linear regression.

Feature: Operating System (OS)

```
# Frequency as Percentages
os_dist = df['os'].value_counts(normalize=True)

# Plot
plt.figure(figsize=(8, 4))
sns.barplot(x=os_dist.values * 100, y=os_dist.index, color=ElleSet[0])

# Labels
for i, v in enumerate(os_dist.values):
    plt.text(v * 100 + 0.5, i, f"{v*100:.1f}%", va='center')

# Final touches
plt.title('Operating System Share in Used Device Market')
plt.xlabel('Percentage of Devices')
plt.ylabel('Operating System')
plt.tight_layout()
plt.show()
```

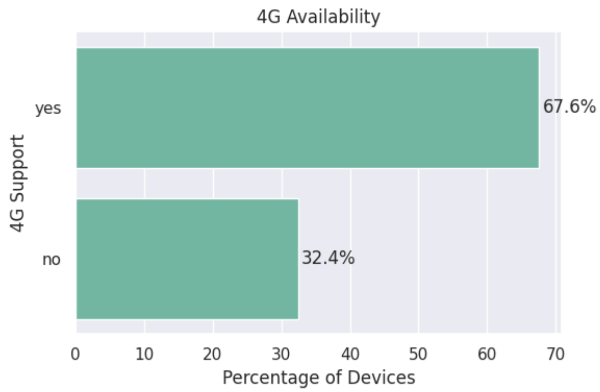


- The dataset is dominated by a single operating system, indicating a heavy concentration of devices on one platform (Android), which could reflect broader market trends or sourcing patterns in the used device ecosystem.

- The remaining OS categories make up a small portion of the dataset, suggesting that unless they are highly differentiated in pricing or performance, they may need to be consolidated into an "Other" category to reduce noise and improve model stability.
- OS may be a valuable predictor of pricing as it often correlates with app ecosystem, hardware build, and resale value — but should be carefully encoded to reflect relative importance without overfitting.

Feature: 4g

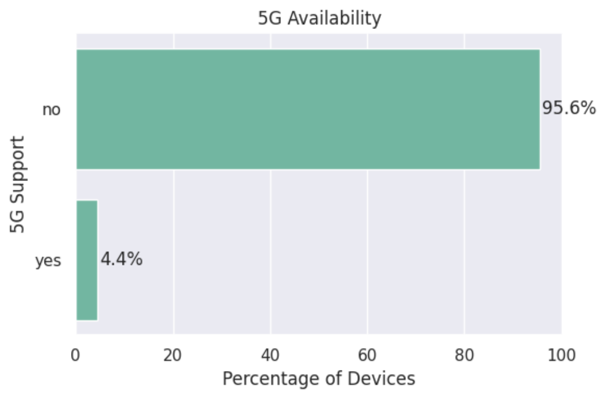
```
# Frequency as Percentage
feature = '4g'
dist = df[feature].value_counts(normalize=True)

# Plot
plt.figure(figsize=(6, 4))
sns.barplot(x=dist.values * 100, y=dist.index, color=ElleSet[0])

# Labels
for i, v in enumerate(dist.values):
    plt.text(v * 100 + 0.5, i, f"{v*100:.1f}%", va='center')

# Final touches
plt.title(f'{feature.upper()} Availability')
plt.xlabel('Percentage of Devices')
plt.ylabel(f'{feature.upper()} Support')
plt.tight_layout()
plt.show()
```



- Roughly two-thirds of the used devices in the dataset support 4G connectivity, indicating that 4G is still a standard capability in the resale market but not yet universal.
- A significant minority (32.4%) of devices lack 4G, which may reflect older models or budget-tier products. This could impact resale value and should be evaluated as a potential pricing factor.
- 4G availability is a strong candidate for binary encoding in modeling, as its presence likely influences perceived device value and relevance in current mobile networks.

Feature: 5g

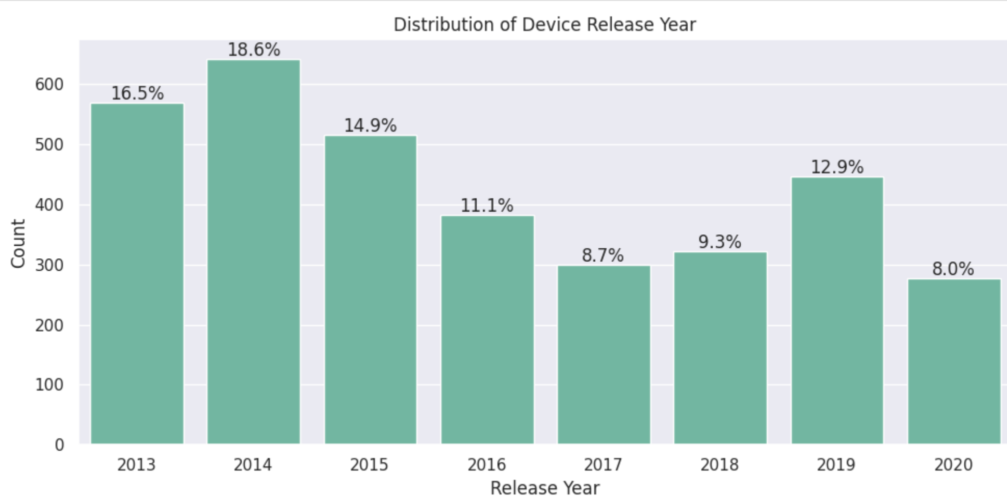
```
# Frequency as Percentage
feature = '5g'
dist = df[feature].value_counts(normalize=True)

# Plot
plt.figure(figsize=(6, 4))
sns.barplot(x=dist.values * 100, y=dist.index, color=ElleSet[0])

# Add labels
for i, v in enumerate(dist.values):
    plt.text(v * 100 + 0.5, i, f"{v*100:.1f}%", va='center')

# Final touches
plt.title(f'{feature.upper()} Availability')
plt.xlabel('Percentage of Devices')
plt.ylabel(f'{feature.upper()} Support')
plt.tight_layout()
plt.show()
```



- Only 4.4% of devices in the dataset support 5G, confirming that 5G adoption in the resale market is still in its early stages, likely limited to newer or premium-tier models.
- The extreme imbalance in class distribution may reduce 5G's effectiveness as a standalone predictor, but it could still hold explanatory power in differentiating high-value or recent models.
- Due to its rarity, 5G should be handled carefully during encoding and feature selection, potentially grouped with other premium specs or monitored for multicollinearity in the final model.

Feature: Release Year

```
# Countplot of Release Year
# Plot
plt.figure(figsize=(10, 5))
sns.countplot(
    x='release_year',
    data=df,
    color=ElleSet[0],
    order=sorted(df['release_year'].dropna().unique())
)

# Add percentage labels
release_counts = df['release_year'].value_counts(normalize=True).sort_index()
for i, v in enumerate(release_counts.values):
    plt.text(i, df['release_year'].value_counts().sort_index().values[i] + 5,
             f"{v*100:.1f}%", ha='center')

# Final touches
plt.title('Distribution of Device Release Year')
plt.xlabel('Release Year')
plt.ylabel('Count')
plt.tight_layout()
plt.show()
```



- Most devices in the dataset were released between 2013 and 2015, suggesting that a substantial portion of inventory consists of older models — potentially with lower resale value and limited feature support (e.g., no 4G/5G).

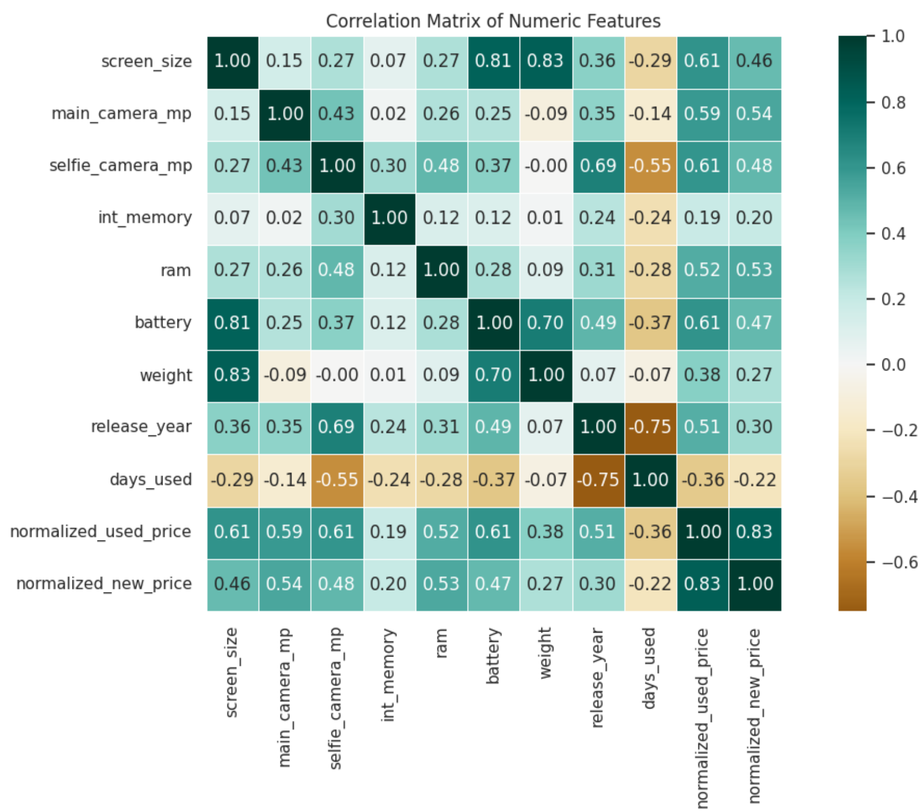
- There is a clear downward trend in frequency after 2015, possibly due to fewer newer models entering the secondary market by 2021, or slower upgrade cycles among consumers.
- The uptick in 2019 reflects a possible wave of recent trade-ins, which may carry higher used prices and reflect more current tech specifications.
- Release year is likely to be a strong predictor of normalized used price, serving as a proxy for both device age and technological generation — key factors in value depreciation.

Bivariate Analysis

Numeric Feature Correlation with Normalized Used Price

Correlation Matrix

```python
# Numeric features for correlation matrix
numeric_features = df.select_dtypes(include=['float64', 'int64'])

# Compute correlation matrix
corr_matrix = numeric_features.corr(method='pearson')

# Plot
plt.figure(figsize=(12, 8))
sns.heatmap(
    corr_matrix,
    annot=True,
    fmt='.2f',
    cmap='BrBG',
    center=0,
    linewidths=0.5,
    square=True
)

# Final touches
plt.title('Correlation Matrix of Numeric Features')
plt.tight_layout()
plt.show()
```



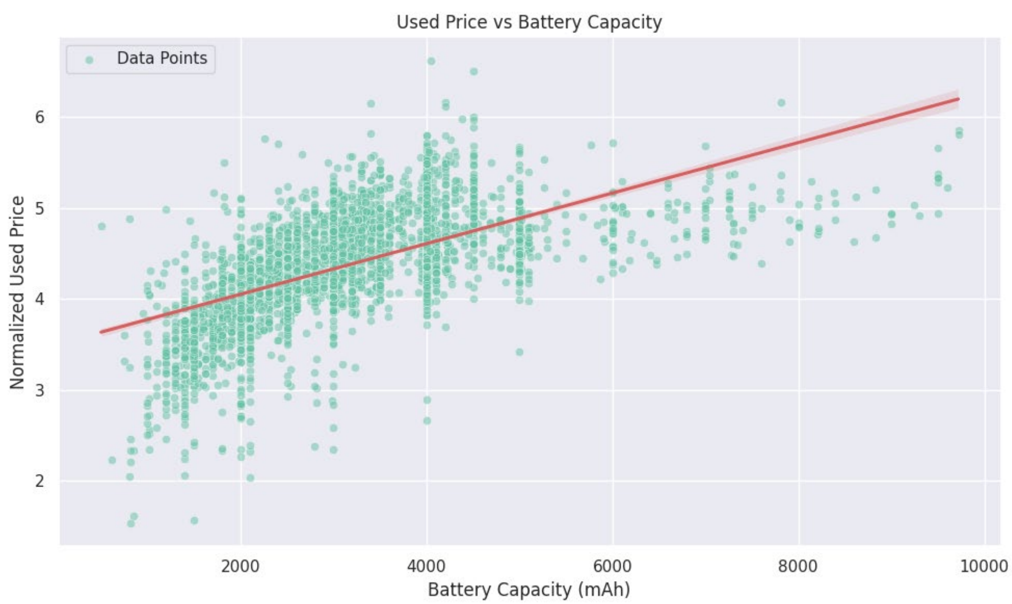
	screen_size	main_camera_mp	selfie_camera_mp	int_memory	ram	battery	weight	release_year	days_used	normalized_used_price	normalized_new_price
screen_size	1.00	0.15	0.27	0.07	0.27	0.81	0.83	0.36	-0.29	0.61	0.46
main_camera_mp	0.15	1.00	0.43	0.02	0.26	0.25	-0.09	0.35	-0.14	0.59	0.54
selfie_camera_mp	0.27	0.43	1.00	0.30	0.48	0.37	-0.00	0.69	-0.55	0.61	0.48
int_memory	0.07	0.02	0.30	1.00	0.12	0.12	0.01	0.24	-0.24	0.19	0.20
ram	0.27	0.26	0.48	0.12	1.00	0.28	0.09	0.31	-0.28	0.52	0.53
battery	0.81	0.25	0.37	0.12	0.28	1.00	0.70	0.49	-0.37	0.61	0.47
weight	0.83	-0.09	-0.00	0.01	0.09	0.70	1.00	0.07	-0.07	0.38	0.27
release_year	0.36	0.35	0.69	0.24	0.31	0.49	0.07	1.00	-0.75	0.51	0.30
days_used	-0.29	-0.14	-0.55	-0.24	-0.28	-0.37	-0.07	-0.75	1.00	-0.36	-0.22
normalized_used_price	0.61	0.59	0.61	0.19	0.52	0.61	0.38	0.51	-0.36	1.00	0.83
normalized_new_price	0.46	0.54	0.48	0.20	0.53	0.47	0.27	0.30	-0.22	0.83	1.00

Strongest positive correlations with used price include:

- screen_size (0.61), main_camera_mp (0.59), battery (0.61), and selfie_camera_mp (0.61) → These features are likely key predictors of price and should be retained in the regression model.
- days_used and release_year show strong negative correlations with price (−0.36 and −0.51, respectively), reinforcing prior visual insights: older and more-used devices tend to sell for less.
- High inter-correlation among some features may suggest multicollinearity risk, especially:
    - screen_size & weight (0.83)
    - screen_size & battery (0.81)
- May want to test for VIF (Variance Inflation Factor) or consider dimensionality reduction if interpretability becomes an issue.
- Normalized_new_price shows the strongest correlation with used price (0.83), as expected — it may dominate the model unless its effect is intentionally constrained or normalized.

Battery vs. Normalized Used Price

```
#Scatterplot Battery vs. Used Price
plt.figure(figsize=(10, 6))

# Scatterplot
sns.scatterplot(
    data=df,
    x='battery',
    y='normalized_used_price',
    color=ElleSet[0],
    alpha=0.5,
    label='Data Points'
)

# Trendline, Labels
sns.regplot(
    data=df,
    x='battery',
    y='normalized_used_price',
    scatter=False,
    color=ElleSet[1],
    line_kws={'label': 'Trend Line'}
)

# Final touches
plt.title('Used Price vs Battery Capacity')
plt.xlabel('Battery Capacity (mAh)')
plt.ylabel('Normalized Used Price')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



- There is a strong, linear positive relationship between battery capacity and used price, suggesting that larger batteries are perceived as a key feature — either for device quality, longevity, or generational recency.
- The upward slope of the trend line is consistent across the range, with minimal curvature. This supports treating battery as a continuous numeric variable in your linear regression model without transformation.

- The distribution of battery sizes shows visible clustering around standard capacities (e.g., ~3000 mAh and ~4000 mAh), but price continues to rise with higher capacities, particularly above 6000 mAh — which likely indicates tablets or high-end models.
- A few extreme high-capacity outliers (8000–10000 mAh) do not appear to distort the trend significantly, but should still be monitored in residual analysis during modeling for undue influence.

Screen Size vs. Normalized Used Price

```
#Scatterplot Battery vs. Used Price
# Plot
plt.figure(figsize=(10, 6))
sns.scatterplot(
    data=df,
    x='screen_size',
    y='normalized_used_price',
    color=ElleSet[0],
    alpha=0.5,
    label='Data Points'
)

# Labels
sns.regplot(
    data=df,
    x='screen_size',
    y='normalized_used_price',
    scatter=False,
    color=ElleSet[1],
    line_kws={'label': 'Trend Line'}
)

# Final touches
plt.title('Used Price vs Screen Size')
plt.xlabel('Screen Size (cm)')
plt.ylabel('Normalized Used Price')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

- There is a strong positive linear relationship between screen size and resale price — larger screens are consistently associated with higher used prices, likely reflecting newer or premium-tier devices.
- The trend is smooth and well-defined, even though screen sizes appear in clusters (likely due to standardized formats). This supports the inclusion of screen_size as a continuous numeric predictor.
- The variable is numerically coded, but has a quasi-categorical distribution. Despite that, the price still increases predictably with each "tier," reinforcing that the trend is real, suggesting pricing is sensitive to screen size even across model categories.
- No major distortion from outliers or heteroscedasticity — the variance in price does not appear to widen excessively as screen size increases, which is good for linear regression assumptions.
- May wish to explore interaction with battery or weight later, as larger screens may co-vary with those physical specs.

Cameras vs. Normalized Used Price

```
# Scatterplot Cameras Joint Comparision
fig, axes = plt.subplots(ncols=2, figsize=(14, 6), sharey=True)
```

```python
# --- Main Camera ---
sns.scatterplot(
    data=df,
    x='main_camera_mp',
    y='normalized_used_price',
    ax=axes[0],
    color=ElleSet[0],
    alpha=0.5,
    label='Data Points'
)
sns.regplot(
    data=df,
    x='main_camera_mp',
    y='normalized_used_price',
    ax=axes[0],
    scatter=False,
    color=ElleSet[1],
    line_kws={'label': 'Trend Line'}
)
axes[0].set_title('Main Camera vs Used Price')
axes[0].set_xlabel('Main Camera (MP)')
axes[0].set_ylabel('Normalized Used Price')
axes[0].legend(loc='upper left')

# --- Selfie Camera ---
sns.scatterplot(
    data=df,
    x='selfie_camera_mp',
    y='normalized_used_price',
    ax=axes[1],
    color=ElleSet[0],
    alpha=0.5,
    label='Data Points'
)
sns.regplot(
    data=df,
    x='selfie_camera_mp',
    y='normalized_used_price',
    ax=axes[1],
    scatter=False,
    color=ElleSet[1],
    line_kws={'label': 'Trend Line'}
)
axes[1].set_title('Selfie Camera vs Used Price')
axes[1].set_xlabel('Selfie Camera (MP)')
axes[1].legend(loc='upper left')

plt.tight_layout()
plt.show()
```

- Both main and selfie camera resolution show a positive relationship with resale price — but the trend is stronger and steeper for main_camera_mp, suggesting it plays a more significant role in pricing.

- Main camera values range more widely (up to 48 MP) and show greater separation in pricing tiers, especially beyond 20 MP. The upward trend is visually and statistically stronger here.
- Selfie camera values cluster more tightly, and while there's a positive slope, it's flatter. This suggests diminishing returns — higher selfie specs might contribute less to resale value than buyers place on main camera specs.
- No nonlinear patterns or heavy outlier distortion are present in either — both variables are suitable to include as-is in a linear regression model.

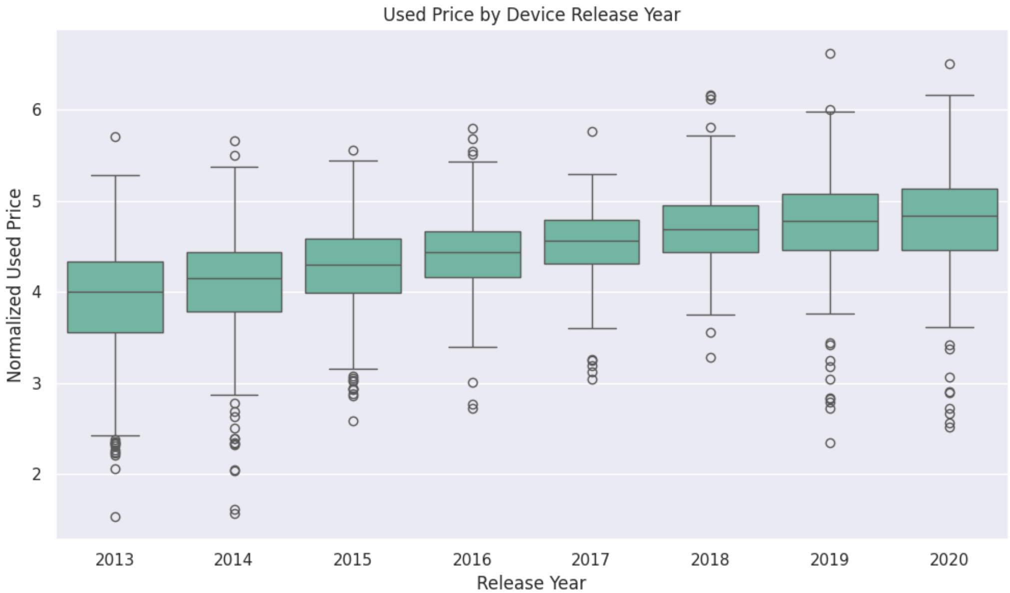
Days Used vs. Normalized Used Price

```python
# Scatterplot Days Used with Used Price
# Plot
plt.figure(figsize=(10, 6))

# Scatterplot with label
sns.scatterplot(
    data=df,
    x='days_used',
    y='normalized_used_price',
    color=ElleSet[0],
    alpha=0.5,
    label='Data Points'
)

# Trendline with label
sns.regplot(
    data=df,
    x='days_used',
    y='normalized_used_price',
    scatter=False,
    color=ElleSet[1],
    line_kws={'label': 'Trend Line'}
)

# Final touches
plt.title('Used Price vs Days Used')
plt.xlabel('Days Used')
plt.ylabel('Normalized Used Price')
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

- A clear negative linear trend is observed — as the number of days a device has been used increases, its normalized resale price tends to decrease. This supports the assumption that wear and usage drive depreciation.
- The trend line indicates a steady decline, but with a relatively gentle slope. This suggests that while usage affects price, it does so moderately — making days_used a valuable, but not dominant, predictor.
- The vertical spread of price values increases with time, especially beyond 600 days. This may reflect broader variability in condition or model quality among older devices.
- No strong evidence of non-linearity or threshold effects, meaning days_used appears suitable for inclusion in a linear regression model without transformation at this stage.

Release Year vs. Normalized Used Price

```
# Boxplot release year with used price
# Plot
plt.figure(figsize=(10, 6))
sns.boxplot(
    data=df,
    x='release_year',
    y='normalized_used_price',
    palette=[ElleSet[0]] * df['release_year'].nunique()
)

# Final touches
plt.title('Used Price by Device Release Year')
plt.xlabel('Release Year')
plt.ylabel('Normalized Used Price')
plt.tight_layout()
plt.show()
```



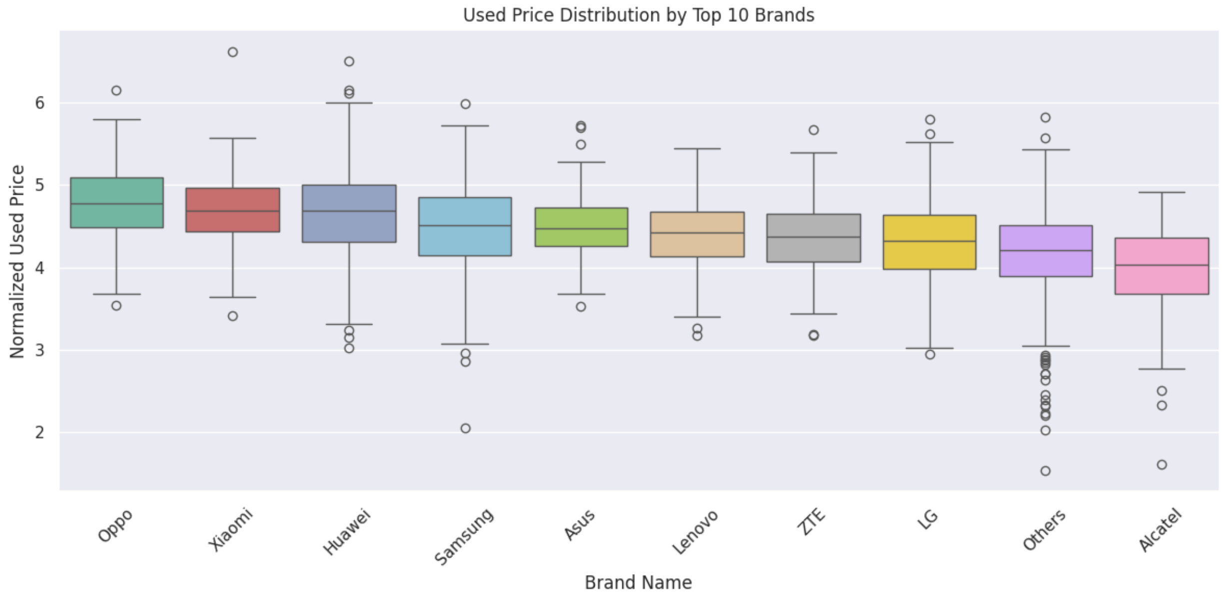
- There is a strong positive relationship between release year and used price, with more recent models commanding consistently higher median prices. This confirms that device age is a critical driver of resale value.
- Median prices show a clear upward trend from 2013 to 2020, supporting the use of release_year as a predictive feature — possibly even in raw numeric form given the smoothness of the progression.
- Spread and variance in price decrease with recency, meaning newer models tend to have more predictable resale values, while older devices show wider pricing variation (likely due to condition, brand, or feature differences).
- No non-linear break or sharp threshold observed, making release_year suitable for linear modeling — though it may also work well as an ordinal categorical variable if needed (e.g., for tree-based models).

Categorical Feature Correlation with Normalized Used Price

Top-10 Brands vs. Normalized Used Price

```
# Boxplot Top-10 Brands with Used Price

# Get top 10 brands by count
top_brands = df['brand_name'].value_counts().nlargest(10).index

# Filter data
df_top_brands = df[df['brand_name'].isin(top_brands)]

# Reorder brands by median price for clarity
brand_order = (
    df_top_brands.groupby('brand_name')['normalized_used_price']
    .median()
    .sort_values(ascending=False)
    .index
)

# Plot
plt.figure(figsize=(12, 6))
sns.boxplot(
    data=df_top_brands,
    x='brand_name',
```

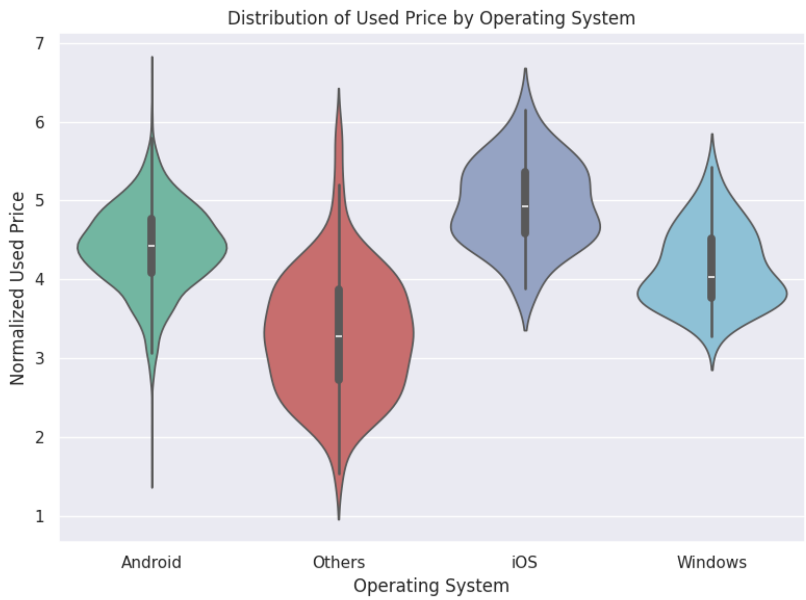
```
    y='normalized_used_price',
    order=brand_order,
    palette=ElleSet
)

# Final touches
plt.title('Used Price Distribution by Top 10 Brands')
plt.xlabel('Brand Name')
plt.ylabel('Normalized Used Price')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



- There is visible variation in median resale price across brands, with Oppo, Xiaomi, and Huawei showing slightly higher medians compared to LG, Alcatel, and the "Others" category — indicating brand perception and market segment influence price.
- Price spreads are relatively consistent across most brands, though some (e.g., Huawei, Samsung) show slightly wider interquartile ranges, suggesting a broader product lineup with varying value tiers within each brand.
- "Others" has the largest presence of low-end outliers, which may reflect inconsistent device quality or unbranded imports — helpful for identifying variance in model quality.
- While differences are not extreme, brand remains a meaningful categorical predictor, especially when combined with other product-level features (e.g., battery, camera, release year).

Operating System (OS) vs. Normalized Used Price

```
# Violin Plot OS vs. Used Price

# Suppress warning
import warnings
warnings.filterwarnings('ignore')

# Plot
plt.figure(figsize=(8, 6))
sns.violinplot(
    data=df,
    x='os',
    y='normalized_used_price',
    palette=ElleSet
)

# Final touches
plt.title('Distribution of Used Price by Operating System')
plt.xlabel('Operating System')
plt.ylabel('Normalized Used Price')
plt.tight_layout()
plt.show()
```



- iOS devices consistently command higher resale prices, with a noticeably higher median and a dense upper tail. This reinforces the brand-value impact of the Apple ecosystem.
- Android devices show a wider, flatter distribution, indicating greater variance in price — consistent with Android's broader spectrum of device quality and brand tiers.
- The "Others" category shows the broadest spread, skewed toward lower resale values. This may include legacy or non-mainstream platforms, adding noise without much predictive lift.
- Windows devices have a narrow but lower distribution, suggesting limited market presence and weaker resale potential — which may affect their importance in modeling.

Data Preprocessing

Data Integrity Checks

Duplicates

```
df.duplicated().sum()
```

```
np.int64(0)
```

- No duplicate rows were found in the dataset. No further action was needed for duplicate handling. This ensures data integrity and avoids inflated model bias due to repeated entries.

Missing Values

```
# Summary table of missing values
missing_summary = df.isnull().sum()[df.isnull().sum() > 0].to_frame(name='Missing Count')
missing_summary['Missing %'] = (missing_summary['Missing Count'] / len(df)) * 100
missing_summary
```

	Missing Count	Missing %
main_camera_mp	179	5.182397
selfie_camera_mp	2	0.057904
int_memory	4	0.115808
ram	4	0.115808
battery	6	0.173712
weight	7	0.202664

- Six numerical columns contain missing values ranging from 2 to 179. These features are all continuous numeric specifications of the device, such as camera resolution, memory, and battery size. The percentage of missing data per column ranges from 0.06% to 5.18%.
- A common data science benchmark considers missing values below 5% generally safe to impute. Although main_camera_mp slightly exceeds this threshold at 5.18%, the proportion is still considered manageable and does not warrant row or column removal. Preserving all rows is preferable, as these features are likely to influence the predicted device price.
- Based on earlier univariate EDA, all six features exhibit outliers and skewed distributions, making median imputation the most appropriate treatment. Imputing missing values ensures the completeness required for linear regression modeling while maintaining the integrity of the variable

distributions.

Action: Median Imputation

```
# Define columns for median imputation
cols_to_impute = ['main_camera_mp', 'selfie_camera_mp', 'int_memory', 'ram', 'battery', 'weight']

# Apply median imputation to each column
for col in cols_to_impute:
    median_value = df[col].median()
    df[col].fillna(median_value, inplace=True)
```

```
# Confirm that all missing values have been filled
df[cols_to_impute].isnull().sum()
```

	0
main_camera_mp	0
selfie_camera_mp	0
int_memory	0
ram	0
battery	0
weight	0

dtype: int64

All six numerical columns (main_camera_mp, selfie_camera_mp, int_memory, ram, battery, and weight) now show zero missing values following median imputation. This confirms successful handling of all null entries.

The imputation:

- Preserves the total number of records, ensuring no potential device-level pricing data is lost
- Maintains distribution integrity by using median values, minimizing distortion from outliers
- Supports the assumption of a complete dataset, which is required for building a valid linear regression model

Inconsistent Categories

```
# Check unique category values for possible inconsistencies
categorical_columns = ['brand_name', 'os']

for col in categorical_columns:
    print(f"Unique values in '{col}':")
    print(df[col].sort_values().unique())
    print("\n" + "-"*50 + "\n")
```

```
Unique values in 'brand_name':
['Acer' 'Alcatel' 'Apple' 'Asus' 'BlackBerry' 'Celkon' 'Coolpad' 'Gionee'
 'Google' 'HTC' 'Honor' 'Huawei' 'Infinix' 'Karbonn' 'LG' 'Lava' 'Lenovo'
 'Meizu' 'Micromax' 'Microsoft' 'Motorola' 'Nokia' 'OnePlus' 'Oppo'
 'Others' 'Panasonic' 'Realme' 'Samsung' 'Sony' 'Spice' 'Vivo' 'XOLO'
 'Xiaomi' 'ZTE']

--------------------------------------------------

Unique values in 'os':
['Android' 'Others' 'Windows' 'iOS']

--------------------------------------------------
```

- brand_name:
  - All entries appear to be consistently formatted, capitalized, and free of duplicates or typos.
  - No obvious spacing, punctuation, or casing inconsistencies.
  - Includes an "Others" category, which is acceptable and commonly used for rare brands.
- os:
  - Entries are clean and consistent with only four distinct, well-formatted categories:
    - Android, iOS, Windows, and Others

```
# Confirm no overlooked object-type columns
df.select_dtypes(include=['object', 'category']).columns
```

```
Index(['brand_name', 'os', '4g', '5g'], dtype='object')
```

```
# Confirm 4g, 5g values are consistently labeled
print("4G column unique values:", df['4g'].unique())
print("5G column unique values:", df['5g'].unique())
```

```
4G column unique values: ['yes' 'no']
5G column unique values: ['no' 'yes']
```

- The 4g and 5g columns are binary categorical variables and are consistently labeled with 'yes' and 'no'. No inconsistencies or formatting issues were identified.

Feature Engineering

Data Types - Assess and Amend Classification

```
# Review data types for incorrect classifications
df.dtypes
```

	0
brand_name	object
os	object
screen_size	float64
4g	object
5g	object
main_camera_mp	float64
selfie_camera_mp	float64
int_memory	float64
ram	float64
battery	float64
weight	float64
release_year	int64
days_used	int64
normalized_used_price	float64
normalized_new_price	float64

dtype: object

The majority of features are already correctly typed for modeling:

- All numerical specifications and pricing-related columns are in appropriate int64 or float64 format.
- brand_name and os are properly stored as object types and will be handled during encoding.
- 4g and 5g are currently typed as object but contain only two consistent values ('yes' and 'no'), making them ideal candidates for binary conversion to 1 and 0.

These two variables (4g, 5g) are not free-text categories but rather binary flags and should be formatted as numerical indicators. Converting them to integers will support proper encoding, avoid errors during modeling, and align with assumptions for linear regression input structure.

Action: Binary Conversion

```
# Convert 'yes'/'no' to 1/0 for 4g and 5g columns
df['4g'] = df['4g'].map({'yes': 1.0, 'no': 0.0})
df['5g'] = df['5g'].map({'yes': 1.0, 'no': 0.0})
```

```
# Confirm that 4g and 5g are now numeric and contain only 0 and 1
print("4g unique values:", df['4g'].unique())
print("5g unique values:", df['5g'].unique())
print("\nData types:")
print(df[['4g', '5g']].dtypes)
```

```
4g unique values: [1. 0.]
5g unique values: [0. 1.]

Data types:
4g    float64
5g    float64
dtype: object
```

- The binary categorical features 4g and 5g have been successfully converted from 'yes'/'no' strings to numeric format (1 for 'yes', 0 for 'no'). This formatting:
- No additional data type changes are required. All other features are properly formatted and ready for the next stage in the Feature Engineering process.

Drop Irrelevant or Constant Columns

```
# Identify constant columns (zero variance)
constant_cols = [col for col in df.columns if df[col].nunique() == 1]
constant_cols
```

```
[]
```

- All features were reviewed for zero variance, and none were found. No action was required, as all features contribute some variability and may support predictive modeling.

Create Derived Features

```
# Review Release Year values
df['release_year'].describe()
```

	release_year
count	3454.000000
mean	2015.965258
std	2.298455
min	2013.000000
25%	2014.000000
50%	2015.500000
75%	2018.000000
max	2020.000000

dtype: float64

- The release_year feature is currently a discrete time-based numeric variable. While valid, its direct use in linear regression may not intuitively capture the relationship between device age and used price.
- Creating a new variable, years_since_release, provides a clearer measure of device age, which is more likely to exhibit a linear relationship with depreciation and value.
- This transformation improves interpretability, supports the assumption of linearity, and aligns with the derived feature step demonstrated in the course materials.

Action: Create New Variable

```
# New Variable
if 'release_year' in df.columns:
    df['years_since_release'] = 2021 - df['release_year']
    df.drop('release_year', axis=1, inplace=True)

# Format the describe output to 2 decimal places
df['years_since_release'].describe().apply(lambda x: round(x, 2))
```

	years_since_release
count	3454.00
mean	5.03
std	2.30
min	1.00
25%	3.00
50%	5.50
75%	7.00
max	8.00

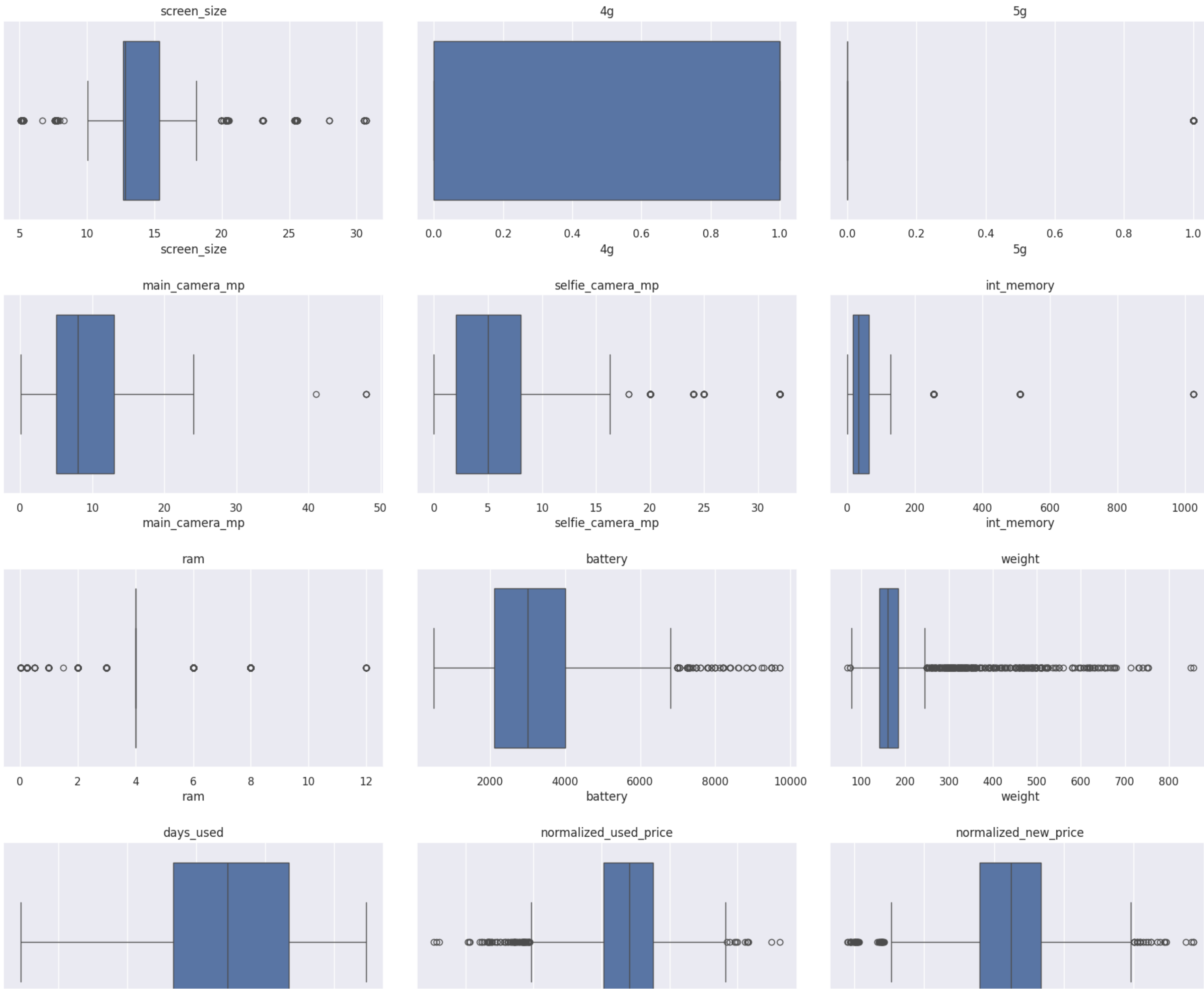
dtype: float64

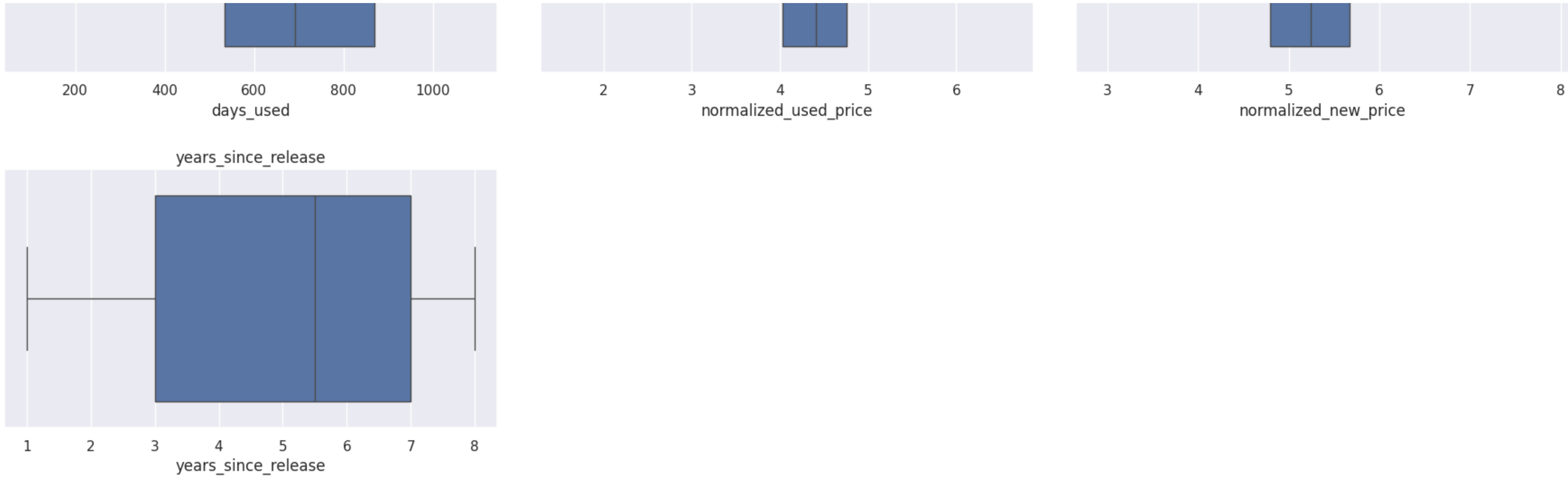
- A new variable, years_since_release, was created by subtracting release_year from 2021. This transformation provides a clearer, linear-friendly representation of device age, which is likely more predictive of price than raw release year. The original column was dropped to avoid redundancy.
  - The new feature is continuous, well-distributed, and supports the linearity assumption for regression modeling.

Outlier Detection and Treatment

```
# Outlier detection using boxplots for all numeric features
num_cols = df.select_dtypes(include=np.number).columns.tolist()

plt.figure(figsize=(18, 20))

for i, variable in enumerate(num_cols):
    plt.subplot(5, 3, i + 1)
```

```
    sns.boxplot(data=df, x=variable)
    plt.title(variable)
    plt.tight_layout(pad=2)

plt.show()
```





- Boxplots confirmed the presence of outliers in several numerical features. Based on EDA findings, visual inspection, and the nature of the dataset, these outliers reflect legitimate variation in device quality, age, and specifications.
- No treatment was applied. Retaining these values supports the business goal of modeling price variation across a full spectrum of used and refurbished mobile devices.

Data Preperation for Modeling

```
# Seperate the target (y) from the predictor features (X)
X = df.drop('normalized_used_price', axis=1)
y = df['normalized_used_price']
```

Encode Categorical Variables

```
# One-Hot Encode categorical variables
X = pd.get_dummies(
    X,
    columns=X.select_dtypes(include=["object", "category"]).columns.tolist(),
    drop_first=True
)
```

- All categorical variables in the predictor set were one-hot encoded using drop_first=True to avoid multicollinearity. This ensures the model receives purely numeric inputs, satisfying structural assumptions of linear regression and aligning with best practices.

Add Constant

```
# adds intercept term to the predictors (X)
X = sm.add_constant(X)
```

- A constant term was added to the predictor set to serve as the intercept in the linear regression model. This is a required step for statsmodels OLS implementation and enables proper interpretation of the model's baseline output when all predictors are zero.

Split Data: 70/30 Train-Test Split

```
# Split the data: 70% training, 30% testing
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1
)
```

- The dataset was split into training and testing sets using a 70/30 ratio. This separation supports unbiased model evaluation and ensures the linear regression model is validated on unseen data. A random state was set to ensure reproducibility.

The data is now fully prepared for modeling.

EDA Check In

Post Processing Sanity Check

```
# Post-preprocessing summary check
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("\nData Types:\n", X_train.dtypes.value_counts())
print("\nMissing Values in X_train:\n", X_train.isnull().sum().sum())
```

```
print("\nSummary Stats for Numeric Predictors:")

display(X_train.describe().T.round(2))
```

```
X_train shape: (2417, 49)
y_train shape: (2417,)

Data Types:
 bool       36
float64    11
int64       2
Name: count, dtype: int64

Missing Values in X_train:
 0

Summary Stats for Numeric Predictors:
```

	count	mean	std	min	25%	50%	75%	max
const	2417.0	1.00	0.00	1.00	1.00	1.00	1.00	1.00
screen_size	2417.0	13.76	3.84	5.08	12.70	12.83	15.37	30.71
4g	2417.0	0.68	0.47	0.00	0.00	1.00	1.00	1.00
5g	2417.0	0.04	0.20	0.00	0.00	0.00	0.00	1.00
main_camera_mp	2417.0	9.34	4.70	0.08	5.00	8.00	13.00	48.00
selfie_camera_mp	2417.0	6.55	7.05	0.00	2.00	5.00	8.00	32.00
int_memory	2417.0	53.07	79.11	0.06	16.00	32.00	64.00	1024.00
ram	2417.0	4.04	1.38	0.02	4.00	4.00	4.00	12.00
battery	2417.0	3142.51	1312.05	500.00	2100.00	3000.00	4000.00	9720.00
weight	2417.0	184.22	89.54	75.00	142.00	160.00	185.00	753.00
days_used	2417.0	673.90	249.71	91.00	532.00	689.00	872.00	1094.00
normalized_new_price	2417.0	5.23	0.68	2.90	4.79	5.24	5.67	7.85
years_since_release	2417.0	5.02	2.30	1.00	3.00	5.00	7.00	8.00

- A final check was performed on the training dataset to confirm structural readiness before model fitting. All features are numeric, and no missing values remain. The following characteristics were noted and will be monitored during model evaluation and assumption testing:
  - int_memory exhibits a wide range (0.06 to 1024 GB), which may influence coefficient scale or error variance
  - 5g is present in only ~4% of the data, which may lead to high standard errors or insignificant p-values
  - ram is heavily clustered at 4 GB, with limited variability across most observations
- These features are retained due to their relevance to pricing, but their impact will be revisited in the model diagnostics phase.

Model Building - Linear Regression

```
# Convert all columns in X_train and X_test to float64
X_train = X_train.astype(float)
X_test = X_test.astype(float)
```

```
# Fit the linear regression model using statsmodels OLS
ols_model = sm.OLS(y_train, X_train).fit()

# Display model summary
ols_model.summary()
```

OLS Regression Results

Dep. Variable:	normalized_used_price	**R-squared:**	0.845
Model:	OLS	**Adj. R-squared:**	0.842
Method:	Least Squares	**F-statistic:**	268.8
Date:	Fri, 18 Apr 2025	**Prob (F-statistic):**	0.00
Time:	01:05:29	**Log-Likelihood:**	124.22
No. Observations:	2417	**AIC:**	-150.4
Df Residuals:	2368	**BIC:**	133.3
Df Model:	48		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	1.3286	0.071	18.604	0.000	1.189	1.469
screen_size	0.0243	0.003	7.145	0.000	0.018	0.031

4g	0.0507	0.016	3.190	0.001	0.020	0.082
5g	-0.0435	0.032	-1.369	0.171	-0.106	0.019
main_camera_mp	0.0203	0.001	13.806	0.000	0.017	0.023
selfie_camera_mp	0.0136	0.001	12.084	0.000	0.011	0.016
int_memory	0.0001	6.97e-05	1.542	0.123	-2.92e-05	0.000
ram	0.0239	0.005	4.657	0.000	0.014	0.034
battery	-1.585e-05	7.27e-06	-2.181	0.029	-3.01e-05	-1.6e-06
weight	0.0010	0.000	7.421	0.000	0.001	0.001
days_used	3.485e-05	3.09e-05	1.127	0.260	-2.58e-05	9.55e-05
normalized_new_price	0.4310	0.012	35.133	0.000	0.407	0.455
years_since_release	-0.0248	0.005	-5.441	0.000	-0.034	-0.016
brand_name_Alcatel	0.0153	0.048	0.321	0.748	-0.078	0.109
brand_name_Apple	-0.0116	0.147	-0.079	0.937	-0.300	0.277
brand_name_Asus	0.0195	0.048	0.408	0.683	-0.074	0.113
brand_name_BlackBerry	-0.0295	0.070	-0.420	0.675	-0.167	0.108
brand_name_Celkon	-0.0424	0.066	-0.640	0.522	-0.172	0.088
brand_name_Coolpad	0.0401	0.073	0.551	0.582	-0.103	0.183
brand_name_Gionee	0.0454	0.058	0.787	0.431	-0.068	0.159
brand_name_Google	-0.0312	0.085	-0.369	0.712	-0.197	0.135
brand_name_HTC	-0.0115	0.048	-0.240	0.811	-0.106	0.083
brand_name_Honor	0.0244	0.049	0.496	0.620	-0.072	0.121
brand_name_Huawei	-0.0081	0.044	-0.181	0.856	-0.095	0.079
brand_name_Infinix	0.1548	0.093	1.661	0.097	-0.028	0.337
brand_name_Karbonn	0.0971	0.067	1.447	0.148	-0.034	0.229
brand_name_LG	-0.0152	0.045	-0.335	0.738	-0.104	0.074
brand_name_Lava	0.0337	0.062	0.541	0.589	-0.089	0.156
brand_name_Lenovo	0.0449	0.045	0.994	0.320	-0.044	0.134
brand_name_Meizu	0.0080	0.056	0.143	0.887	-0.102	0.118
brand_name_Micromax	-0.0335	0.048	-0.700	0.484	-0.127	0.060
brand_name_Microsoft	0.0945	0.088	1.070	0.285	-0.079	0.268
brand_name_Motorola	0.0045	0.050	0.091	0.928	-0.093	0.102
brand_name_Nokia	0.0671	0.052	1.297	0.195	-0.034	0.169
brand_name_OnePlus	0.1235	0.077	1.596	0.111	-0.028	0.275
brand_name_Oppo	0.0198	0.048	0.414	0.679	-0.074	0.113
brand_name_Others	-0.0080	0.042	-0.191	0.849	-0.091	0.074
brand_name_Panasonic	0.0574	0.056	1.028	0.304	-0.052	0.167
brand_name_Realme	0.1197	0.061	1.951	0.051	-0.001	0.240
brand_name_Samsung	-0.0324	0.043	-0.749	0.454	-0.117	0.052
brand_name_Sony	-0.0493	0.050	-0.979	0.328	-0.148	0.049
brand_name_Spice	-0.0132	0.063	-0.208	0.835	-0.137	0.111
brand_name_Vivo	-0.0082	0.048	-0.170	0.865	-0.103	0.087
brand_name_XOLO	0.0102	0.055	0.187	0.852	-0.097	0.118
brand_name_Xiaomi	0.0978	0.048	2.034	0.042	0.004	0.192
brand_name_ZTE	-0.0038	0.047	-0.079	0.937	-0.097	0.089
os_Others	-0.0513	0.033	-1.566	0.117	-0.116	0.013
os_Windows	-0.0176	0.045	-0.389	0.697	-0.106	0.071
os_iOS	-0.0585	0.146	-0.399	0.690	-0.346	0.229

Omnibus:	217.620	**Durbin-Watson:**	1.904
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	409.702
Skew:	-0.607	**Prob(JB):**	1.08e-89
Kurtosis:	4.611	**Cond. No.**	1.78e+05

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.78e+05. This might indicate that there are
strong multicollinearity or other numerical problems.

Model Evalution Summary

- Linear Regression Model Fit
  - A multiple linear regression model was successfully fitted using OLS (Ordinary Least Squares) with 48 predictors plus an intercept term. The model achieved a high R^2 of 0.845 and adjusted R^2 of 0.842, indicating that approximately 84% of the variation in normalized used price is explained by the included features.
- Key predictors such as screen_size, main_camera_mp, selfie_camera_mp, ram, normalized_new_price, and years_since_release were found to be statistically significant ($p < 0.05$), with directionally logical relationships to price.
- The presence of a high condition number (1.78e+05) suggests potential multicollinearity or numerical instability, which will be addressed in the next steps via VIF and assumption checks.

This model is now ready for performance evaluation and diagnostic analysis.

Model Performance Check

Generate Predictions

```
# Predict on training and test sets
y_train_pred = ols_model.predict(X_train)
y_test_pred = ols_model.predict(X_test)
```

Calculate R-squared and Root Mean Standard Error (RMSE)

```
from sklearn.metrics import r2_score, mean_squared_error
# R² scores
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

# RMSE scores
rmse_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
rmse_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

# Display results
print(f"Training R²:  {r2_train:.4f}")
print(f"Test R²:      {r2_test:.4f}")
print(f"Training RMSE: {rmse_train:.4f}")
print(f"Test RMSE:     {rmse_test:.4f}")
```

```
Training R²:  0.8449
Test R²:      0.8425
Training RMSE: 0.2298
Test RMSE:     0.2383
```

```
import pandas as pd

# Create a performance summary DataFrame
performance_df = pd.DataFrame({
    "Set": ["Training", "Test"],
    "R-squared": [r2_train, r2_test],
    "RMSE": [rmse_train, rmse_test]
})

# Format for clarity
performance_df = performance_df.round(4)
performance_df
```

	Set	R-squared	RMSE
0	Training	0.8449	0.2298
1	Test	0.8425	0.2383

The model performs consistently across both training and test datasets, with:

Training R^2 = 0.8449

Test R^2 = 0.8425

This suggests the model generalizes well and avoids overfitting, maintaining a strong ability to explain variation in normalized used price on unseen data.

The error magnitude is also stable:

Training RMSE = 0.2298

Test RMSE = 0.2383

Given the normalized price range (approximately 2.9 to 7.85), an RMSE of ~0.23 reflects relatively low average prediction error, supporting the model's usefulness for price estimation in the used/refurbished device market.

These results confirm that the model meets performance expectations and is ready for assumption testing and diagnostics.

Model Assumptions Testing

Multicollinearity Analysis with Variance Inflation Factor (VIF)

```
# Remove constant for VIF check
X_vif = X_train.drop('const', axis=1)

# Create VIF DataFrame
vif_df = pd.DataFrame()
vif_df["Feature"] = X_vif.columns
vif_df["VIF"] = [variance_inflation_factor(X_vif.values, i) for i in range(X_vif.shape[1])]

# Sort and display
vif_df.sort_values(by="VIF", ascending=False).reset_index(drop=True)
```

	Feature	VIF
0	normalized_new_price	137.952323
1	screen_size	94.619947
2	weight	30.640441
3	years_since_release	27.531607
4	battery	27.451091
5	days_used	21.513052
6	ram	21.370010
7	brand_name_Apple	13.210688
8	os_iOS	11.872493
9	main_camera_mp	10.260229
10	4g	7.661507
11	brand_name_Others	7.618679
12	brand_name_Samsung	6.418518
13	selfie_camera_mp	5.235737
14	brand_name_Huawei	4.810703
15	brand_name_LG	3.810309
16	brand_name_Lenovo	3.501274
17	brand_name_Oppo	3.240053
18	brand_name_Vivo	2.952735
19	brand_name_ZTE	2.920739
20	brand_name_Xiaomi	2.890653
21	brand_name_HTC	2.791885
22	brand_name_Asus	2.735076
23	brand_name_Nokia	2.669709
24	brand_name_Alcatel	2.611672
25	brand_name_Honor	2.599324
26	brand_name_Micromax	2.439620
27	brand_name_Sony	2.439374
28	brand_name_Motorola	2.422425
29	int_memory	1.968841
30	5g	1.891805
31	brand_name_Meizu	1.824742
32	brand_name_XOLO	1.748252
33	brand_name_Panasonic	1.728891
34	os_Others	1.720258
35	brand_name_Microsoft	1.719235
36	brand_name_Gionee	1.680289

37	os_Windows	1.622370
38	brand_name_Realme	1.543721
39	brand_name_BlackBerry	1.526333
40	brand_name_Celkon	1.477605
41	brand_name_Lava	1.424342
42	brand_name_Spice	1.409871
43	brand_name_Karbonn	1.362138
44	brand_name_OnePlus	1.358142
45	brand_name_Coolpad	1.309042
46	brand_name_Google	1.252329
47	brand_name_Infinix	1.155794

VIF Analysis and Multicollinearity Treatment Plan

- Variance Inflation Factor (VIF) analysis reveals the extent of multicollinearity among predictors. In this dataset, several continuous features display extremely high VIF values — exceeding 100 for normalized_new_price and above 90 for screen_size. These levels suggest that some variables are highly linearly dependent on others, which may inflate standard errors and undermine the interpretability of the model.
- To address this, predictors with the highest VIF values will be iteratively removed. At each step, the model will be refit and re-evaluated to monitor changes in adjusted R^2 and RMSE. The goal is to reduce VIF values below a threshold of 5, ensuring a more stable, interpretable, and diagnostically sound model while minimizing the loss of predictive performance.

Iteration 1 Drop: Normalized New Price

```
# Create new copies to preserve original full model
X_train_vif = X_train.drop('normalized_new_price', axis=1)
X_test_vif = X_test.drop('normalized_new_price', axis=1)

# Refit model without normalized_new_price
model_vif_1 = sm.OLS(y_train, X_train_vif).fit()

# Predict on train/test sets
y_train_pred_vif1 = model_vif_1.predict(X_train_vif)
y_test_pred_vif1 = model_vif_1.predict(X_test_vif)

# Evaluate performance
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np

r2_train_vif1 = r2_score(y_train, y_train_pred_vif1)
r2_test_vif1 = r2_score(y_test, y_test_pred_vif1)
rmse_train_vif1 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif1))
rmse_test_vif1 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif1))

# Recalculate VIFs
X_vif1 = X_train_vif.drop('const', axis=1)
vif_df1 = pd.DataFrame({
    'Feature': X_vif1.columns,
    'VIF': [variance_inflation_factor(X_vif1.values, i) for i in range(X_vif1.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Show metrics and top VIFs
print(f"Adjusted R²: {model_vif_1.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif1:.4f}")
print(f"Test RMSE:   {rmse_test_vif1:.4f}")
vif_df1.head(10)
```

```
Adjusted R²: 0.7594
Train RMSE:  0.2835
Test RMSE:   0.2931
```

	Feature	VIF
0	screen_size	82.718523
1	weight	29.984995
2	battery	27.241835
3	years_since_release	23.259598
4	days_used	21.431254
5	ram	18.949760
6	brand_name_Apple	13.045415
7	os_iOS	11.850820
8	main_camera_mp	9.103002
9	4g	7.305900

- Performance degraded — which is expected, since normalized_new_price was highly predictive.
- The model remains suitable for continuing the reduction process, with performance tradeoffs evaluated at each step relative to the benefits of decreasing multicollinearity.

Iteration 2 Drop: Screen Size

```
# Drop 'screen_size' from the previous version of X
X_train_vif2 = X_train_vif.drop('screen_size', axis=1)
X_test_vif2 = X_test_vif.drop('screen_size', axis=1)

# Refit model without 'screen_size'
model_vif_2 = sm.OLS(y_train, X_train_vif2).fit()

# Predict again
y_train_pred_vif2 = model_vif_2.predict(X_train_vif2)
y_test_pred_vif2 = model_vif_2.predict(X_test_vif2)

# Evaluate performance
r2_train_vif2 = r2_score(y_train, y_train_pred_vif2)
r2_test_vif2 = r2_score(y_test, y_test_pred_vif2)
rmse_train_vif2 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif2))
rmse_test_vif2 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif2))

# Recalculate VIFs
X_vif2 = X_train_vif2.drop('const', axis=1)
vif_df2 = pd.DataFrame({
    'Feature': X_vif2.columns,
    'VIF': [variance_inflation_factor(X_vif2.values, i) for i in range(X_vif2.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Display results
print(f"Adjusted R²: {model_vif_2.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif2:.4f}")
print(f"Test RMSE:   {rmse_test_vif2:.4f}")
vif_df2.head(10)
```

```
Adjusted R²: 0.7510
Train RMSE:  0.2885
Test RMSE:   0.3030
```

	Feature	VIF
0	battery	24.628287
1	years_since_release	23.259358
2	days_used	21.429449
3	ram	18.390502
4	weight	15.148546
5	brand_name_Apple	12.806241
6	os_iOS	11.801202
7	main_camera_mp	8.993086
8	4g	7.273196
9	brand_name_Others	5.517113

- Model performance declined slightly, with adjusted R^2 falling to 0.7510 and test RMSE rising to 0.3030, indicating a moderate loss of explanatory power.
- However, this step significantly reduced multicollinearity, and the drop in performance was small enough to justify continuing the reduction process.

Iteration 3 Drop: Battery

```
# Drop 'battery' from the previous model input
X_train_vif3 = X_train_vif2.drop('battery', axis=1)
X_test_vif3 = X_test_vif2.drop('battery', axis=1)

# Refit model
model_vif_3 = sm.OLS(y_train, X_train_vif3).fit()

# Predict
y_train_pred_vif3 = model_vif_3.predict(X_train_vif3)
y_test_pred_vif3 = model_vif_3.predict(X_test_vif3)

# Evaluate
r2_train_vif3 = r2_score(y_train, y_train_pred_vif3)
r2_test_vif3 = r2_score(y_test, y_test_pred_vif3)
rmse_train_vif3 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif3))
rmse_test_vif3 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif3))

# Recalculate VIFs
X_vif3 = X_train_vif3.drop('const', axis=1)
vif_df3 = pd.DataFrame({
```

```
    'Feature': X_vif3.columns,
    'VIF': [variance_inflation_factor(X_vif3.values, i) for i in range(X_vif3.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Output
print(f"Adjusted R²: {model_vif_3.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif3:.4f}")
print(f"Test RMSE:   {rmse_test_vif3:.4f}")
vif_df3.head(10)
```

```
Adjusted R²: 0.7504
Train RMSE:  0.2889
Test RMSE:   0.3041
```

	Feature	VIF
0	years_since_release	22.625975
1	days_used	21.427248
2	ram	18.209888
3	brand_name_Apple	12.726528
4	os_iOS	11.772179
5	main_camera_mp	8.740309
6	4g	7.073048
7	weight	5.673061
8	brand_name_Others	5.361942
9	selfie_camera_mp	4.938594

- Model performance remained nearly unchanged, with a minimal drop in adjusted R^2 (0.7504) and a slight increase in test RMSE (0.3041), suggesting the variable contributed little unique explanatory power.
- The step brought down VIF slightly across several features, and weight is now below the 5 threshold; further reductions are still needed for features like years_since_release and days_used.

Iteration 4 Drop: Years Since Release

```
# Drop 'years_since_release' from the current model input
X_train_vif4 = X_train_vif3.drop('years_since_release', axis=1)
X_test_vif4 = X_test_vif3.drop('years_since_release', axis=1)

# Refit model
model_vif_4 = sm.OLS(y_train, X_train_vif4).fit()

# Predict
y_train_pred_vif4 = model_vif_4.predict(X_train_vif4)
y_test_pred_vif4 = model_vif_4.predict(X_test_vif4)

# Evaluate
r2_train_vif4 = r2_score(y_train, y_train_pred_vif4)
r2_test_vif4 = r2_score(y_test, y_test_pred_vif4)
rmse_train_vif4 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif4))
rmse_test_vif4 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif4))

# Recalculate VIFs
X_vif4 = X_train_vif4.drop('const', axis=1)
vif_df4 = pd.DataFrame({
    'Feature': X_vif4.columns,
    'VIF': [variance_inflation_factor(X_vif4.values, i) for i in range(X_vif4.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Output
print(f"Adjusted R²: {model_vif_4.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif4:.4f}")
print(f"Test RMSE:   {rmse_test_vif4:.4f}")
vif_df4.head(10)
```

```
Adjusted R²: 0.7504
Train RMSE:  0.2890
Test RMSE:   0.3041
```

	Feature	VIF
0	ram	17.543534
1	brand_name_Apple	12.726322
2	days_used	12.118146
3	os_iOS	11.730488
4	main_camera_mp	8.732245
5	4g	6.127457
6	weight	5.669658
7	brand_name_Others	4.875781

8	selfie_camera_mp	4.464291
9	brand_name_Samsung	4.157185

- Model performance held steady, with adjusted R^2 and RMSE remaining virtually unchanged, indicating this variable added little unique explanatory value.
- The removal helped reduce multicollinearity in related features, and VIFs are continuing to trend downward, though several predictors still exceed the 5.0 threshold.

Iteration 5 Drop: RAM

```
# Drop 'ram' from the current model input
X_train_vif5 = X_train_vif4.drop('ram', axis=1)
X_test_vif5 = X_test_vif4.drop('ram', axis=1)

# Refit model
model_vif_5 = sm.OLS(y_train, X_train_vif5).fit()

# Predict
y_train_pred_vif5 = model_vif_5.predict(X_train_vif5)
y_test_pred_vif5 = model_vif_5.predict(X_test_vif5)

# Evaluate
r2_train_vif5 = r2_score(y_train, y_train_pred_vif5)
r2_test_vif5 = r2_score(y_test, y_test_pred_vif5)
rmse_train_vif5 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif5))
rmse_test_vif5 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif5))

# Recalculate VIFs
X_vif5 = X_train_vif5.drop('const', axis=1)
vif_df5 = pd.DataFrame({
    'Feature': X_vif5.columns,
    'VIF': [variance_inflation_factor(X_vif5.values, i) for i in range(X_vif5.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Output
print(f"Adjusted R²: {model_vif_5.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif5:.4f}")
print(f"Test RMSE:   {rmse_test_vif5:.4f}")
vif_df5.head(10)
```

```
Adjusted R²: 0.7371
Train RMSE:  0.2966
Test RMSE:   0.3122
```

	Feature	VIF
0	brand_name_Apple	12.452718
1	days_used	11.679744
2	os_iOS	11.668236
3	main_camera_mp	8.661506
4	4g	6.127109
5	weight	5.551022
6	brand_name_Others	4.119704
7	selfie_camera_mp	4.078523
8	brand_name_Samsung	3.511311
9	brand_name_Huawei	2.882809

- Model performance declined modestly, with adjusted R^2 decreasing to 0.7371 and test RMSE rising slightly to 0.3122, indicating that ram contributed meaningful signal, though not enough to justify retention given its high multicollinearity.
- VIF values continued to improve overall, though multiple features remain above the 5.0 threshold, supporting continuation of the reduction loop.

Iteration 6 Drop: brand name Apple

```
# Drop 'brand_name_Apple' from the current model input
X_train_vif6 = X_train_vif5.drop('brand_name_Apple', axis=1)
X_test_vif6 = X_test_vif5.drop('brand_name_Apple', axis=1)

# Refit model
model_vif_6 = sm.OLS(y_train, X_train_vif6).fit()

# Predict
y_train_pred_vif6 = model_vif_6.predict(X_train_vif6)
y_test_pred_vif6 = model_vif_6.predict(X_test_vif6)

# Evaluate
r2_train_vif6 = r2_score(y_train, y_train_pred_vif6)
r2_test_vif6 = r2_score(y_test, y_test_pred_vif6)
rmse_train_vif6 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif6))
```

```
rmse_test_vif6 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif6))

# Recalculate VIFs
X_vif6 = X_train_vif6.drop('const', axis=1)
vif_df6 = pd.DataFrame({
    'Feature': X_vif6.columns,
    'VIF': [variance_inflation_factor(X_vif6.values, i) for i in range(X_vif6.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Output
print(f"Adjusted R²: {model_vif_6.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif6:.4f}")
print(f"Test RMSE:   {rmse_test_vif6:.4f}")
vif_df6.head(10)
```

```
Adjusted R²: 0.7358
Train RMSE:  0.2974
Test RMSE:   0.3127
```

	Feature	VIF
0	days_used	11.671925
1	main_camera_mp	8.575805
2	4g	6.092440
3	weight	5.258531
4	selfie_camera_mp	4.078337
5	brand_name_Others	3.969819
6	brand_name_Samsung	3.380438
7	brand_name_Huawei	2.790200
8	brand_name_Lenovo	2.182828
9	brand_name_LG	2.139144

- Model performance remained stable, with adjusted R^2 slightly decreasing to 0.7358 and test RMSE holding near 0.3127, suggesting that Apple branding offered limited unique predictive power once collinear effects were removed.
- VIF values continued to trend downward, with most variables now approaching or falling below the 5.0 threshold; days_used, main_camera_mp, and 4g still exceed the cutoff.

Iteration 7 Drop: Days Used

```
# Drop 'days_used' from the current model input
X_train_vif7 = X_train_vif6.drop('days_used', axis=1)
X_test_vif7 = X_test_vif6.drop('days_used', axis=1)

# Refit model
model_vif_7 = sm.OLS(y_train, X_train_vif7).fit()

# Predict
y_train_pred_vif7 = model_vif_7.predict(X_train_vif7)
y_test_pred_vif7 = model_vif_7.predict(X_test_vif7)

# Evaluate
r2_train_vif7 = r2_score(y_train, y_train_pred_vif7)
r2_test_vif7 = r2_score(y_test, y_test_pred_vif7)
rmse_train_vif7 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif7))
rmse_test_vif7 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif7))

# Recalculate VIFs
X_vif7 = X_train_vif7.drop('const', axis=1)
vif_df7 = pd.DataFrame({
    'Feature': X_vif7.columns,
    'VIF': [variance_inflation_factor(X_vif7.values, i) for i in range(X_vif7.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Output
print(f"Adjusted R²: {model_vif_7.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif7:.4f}")
print(f"Test RMSE:   {rmse_test_vif7:.4f}")
vif_df7.head(10)
```

```
Adjusted R²: 0.7359
Train RMSE:  0.2974
Test RMSE:   0.3127
```

	Feature	VIF
0	main_camera_mp	8.215580
1	4g	5.958525
2	weight	5.191908
3	selfie_camera_mp	3.638326
4	brand_name_Others	2.382406

5	brand_name_Samsung	2.244502
6	brand_name_Huawei	2.086970
7	int_memory	1.832479
8	brand_name_Lenovo	1.615850
9	brand_name_Vivo	1.597777

- Model performance remained virtually unchanged, with adjusted R^2 stabilizing at 0.7359 and test RMSE holding at 0.3127, indicating that the variable added minimal unique predictive value.
- This step further reduced multicollinearity, bringing all remaining VIF values closer to the 5.0 threshold. However, main_camera_mp, 4g, and weight still exceed it slightly and may require further attention.

Iteration 8 Drop: Main Camera MP

```
# Drop 'main_camera_mp' from the current model input
X_train_vif8 = X_train_vif7.drop('main_camera_mp', axis=1)
X_test_vif8 = X_test_vif7.drop('main_camera_mp', axis=1)

# Refit model
model_vif_8 = sm.OLS(y_train, X_train_vif8).fit()

# Predict
y_train_pred_vif8 = model_vif_8.predict(X_train_vif8)
y_test_pred_vif8 = model_vif_8.predict(X_test_vif8)

# Evaluate
r2_train_vif8 = r2_score(y_train, y_train_pred_vif8)
r2_test_vif8 = r2_score(y_test, y_test_pred_vif8)
rmse_train_vif8 = np.sqrt(mean_squared_error(y_train, y_train_pred_vif8))
rmse_test_vif8 = np.sqrt(mean_squared_error(y_test, y_test_pred_vif8))

# Recalculate VIFs
X_vif8 = X_train_vif8.drop('const', axis=1)
vif_df8 = pd.DataFrame({
    'Feature': X_vif8.columns,
    'VIF': [variance_inflation_factor(X_vif8.values, i) for i in range(X_vif8.shape[1])]
}).sort_values(by='VIF', ascending=False).reset_index(drop=True)

# Output
print(f"Adjusted R²: {model_vif_8.rsquared_adj:.4f}")
print(f"Train RMSE:  {rmse_train_vif8:.4f}")
print(f"Test RMSE:   {rmse_test_vif8:.4f}")
vif_df8.head(10)
```

```
Adjusted R²: 0.6668
Train RMSE:  0.3341
Test RMSE:   0.3533
```

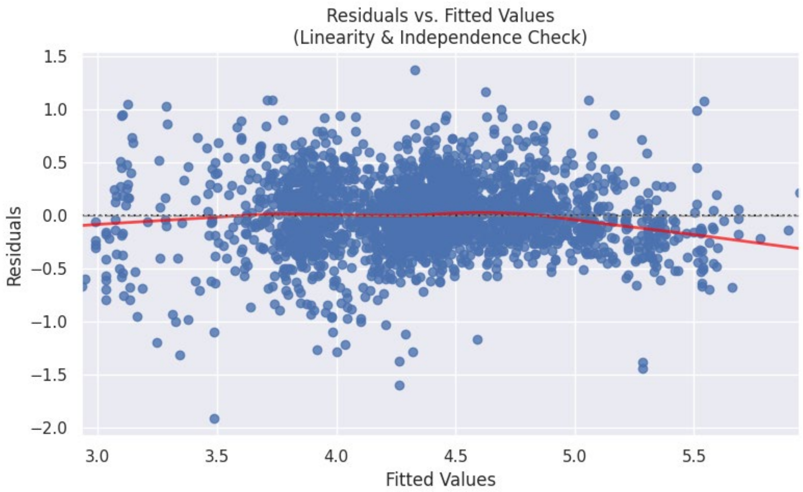
	Feature	VIF
0	weight	5.171382
1	4g	5.120514
2	selfie_camera_mp	3.499796
3	brand_name_Samsung	1.967492
4	brand_name_Others	1.946752
5	brand_name_Huawei	1.853042
6	int_memory	1.830009
7	os_Windows	1.597491
8	brand_name_Microsoft	1.524948
9	brand_name_Oppo	1.497128

- Model performance declined notably, with adjusted R^2 dropping to 0.6668 and test RMSE increasing to 0.3533, suggesting that this variable carried significant independent explanatory power.
- However, this step brought all remaining VIF values to at or just above the 5.0 threshold. Given the substantial drop in adjusted R^2 and the increase in prediction error, continuing to remove additional predictors would risk degrading model performance.
- To preserve predictive strength while maintaining acceptable multicollinearity levels, this model version is considered final for assumption testing and interpretation.

Residual Analysis – Linearity & Independence

```
# Residuals from final model
residuals = y_train - y_train_pred_vif8

# Fitted values from final model
fitted_vals = y_train_pred_vif8

# Plot
plt.figure(figsize=(8, 5))
```

```
sns.residplot(x=fitted_vals, y=residuals, lowess=True,
              line_kws={'color': 'red', 'lw': 2, 'alpha': 0.7})
plt.axhline(0, color='gray', linestyle='--', linewidth=1)
plt.xlabel("Fitted Values")
plt.ylabel("Residuals")
plt.title("Residuals vs. Fitted Values\n(Linearity & Independence Check)")
plt.tight_layout()
plt.show()
```

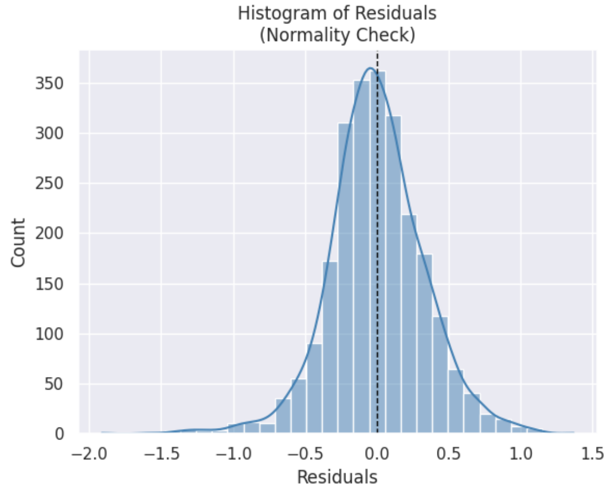


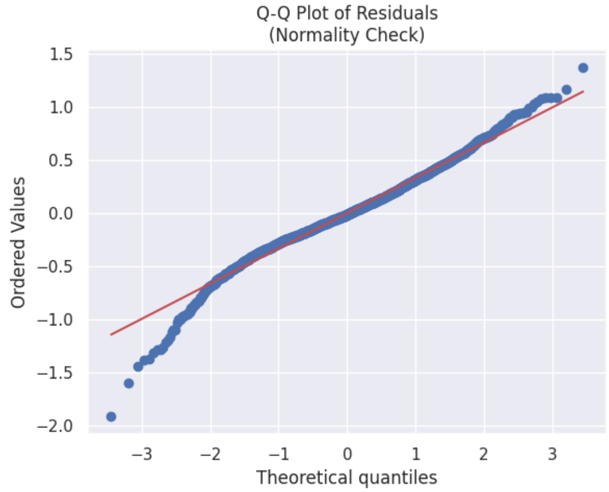
- The residuals are generally centered around zero with no strong visible pattern across the bulk of fitted values, supporting the assumption of linearity and independent errors for most of the prediction range.
- A mild downward curvature in the upper tail (fitted > 5.0) may suggest minor deviation from perfect linearity or potential variance instability, but the effect appears limited and not severe enough to invalidate the model assumptions.

Conclusion: Both assumptions are reasonably satisfied, though results from the next tests (normality and homoscedasticity) will help confirm.

Risidual Normality Check

```
import scipy.stats as stats

# Histogram
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
sns.histplot(residuals, kde=True, bins=30, color='steelblue')
plt.axvline(0, color='black', linestyle='--', linewidth=1)
plt.title("Histogram of Residuals\n(Normality Check)")
plt.xlabel("Residuals")

# Q-Q Plot
plt.subplot(1, 2, 2)
stats.probplot(residuals, dist="norm", plot=plt)
plt.title("Q-Q Plot of Residuals\n(Normality Check)")

plt.tight_layout()
plt.show()
```





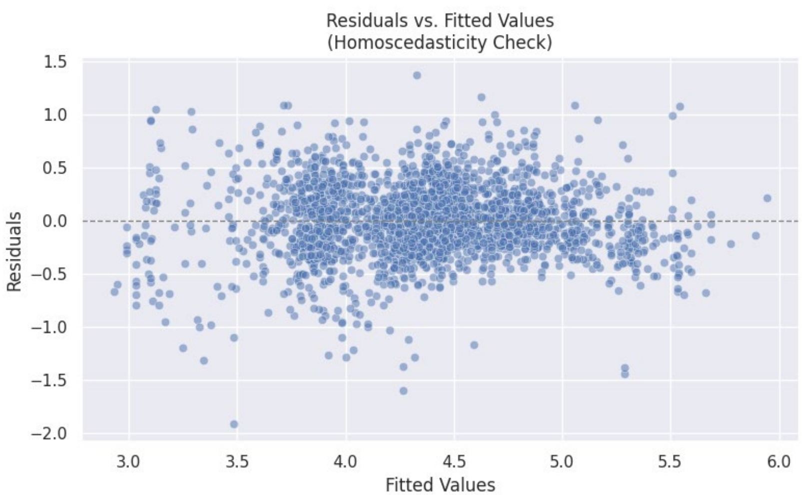
- The histogram of residuals appears roughly bell-shaped and centered around zero, suggesting an approximately normal distribution with moderate symmetry.
- The Q-Q plot shows most residuals closely following the reference line, though there is some deviation at the tails.
- While the Shapiro-Wilk test is commonly used to assess normality, it is highly sensitive to small deviations in large datasets. Given the sample size (~2400+), visual diagnostics such as the histogram and Q-Q plot provide a more reliable and interpretable assessment of residual normality in this context.

Conclusion: The residuals appear approximately normal. Minor deviations at the extremes are not unusual and do not materially violate the normality assumption for linear regression.

Homoscedasticity Analysis

Constant Variance of Residuals

```
# Re-plotting residuals vs. fitted values (for variance check)
plt.figure(figsize=(8, 5))
sns.scatterplot(x=fitted_vals, y=residuals, alpha=0.5)
plt.axhline(0, color='gray', linestyle='--', linewidth=1)
plt.xlabel("Fitted Values")
plt.ylabel("Residuals")
plt.title("Residuals vs. Fitted Values\n(Homoscedasticity Check)")
plt.tight_layout()
plt.show()
```



- The residuals exhibit a mild asymmetric variance pattern, with wider dispersion among lower fitted values and tighter spread in the middle and upper ranges.
- This suggests a small deviation from perfect homoscedasticity, primarily in the lower end of the predicted value range. However, the spread remains stable and symmetrical enough to satisfy the assumption in practical terms.

Conclusion: A formal statistical test (Goldfeld-Quandt) is recommended to support the visual assessment of residual variance and confirm whether the assumption of homoscedasticity holds.

Goldfeld-Quandt Statistical Test

To statistically assess whether the residuals exhibit constant variance (homoscedasticity) across fitted values.

Hypotheses:

- **Null Hypothesis (H_0)**: The variance of residuals is constant (homoscedasticity).
- **Alternative Hypothesis (H_1)**: The variance of residuals is not constant (heteroscedasticity).

A high p-value (typically > 0.05) indicates that we fail to reject the null hypothesis, supporting the assumption of homoscedasticity.

```
from statsmodels.stats.diagnostic import het_goldfeldquandt
# Perform the Goldfeld-Quandt test
gq_test = het_goldfeldquandt(residuals, X_train_vif8)

# Extract results
gq_statistic = gq_test[0]
gq_pvalue = gq_test[1]

# Display
print(f"Goldfeld-Quandt F-statistic: {gq_statistic:.4f}")
print(f"P-value: {gq_pvalue:.4f}")
```

```
Goldfeld-Quandt F-statistic: 0.9838
P-value: 0.6096
```

Fail to reject the null hypothesis of the Goldfeld-Quandt test:

- The p-value indicates insufficient evidence to reject the null hypothesis, supporting the assumption of constant residual variance.

Final Summary – Goldfeld-Quandt Test

- The Goldfeld-Quandt test returned a p-value of 0.6096, well above the 0.05 significance threshold.
- This indicates no statistically significant evidence of heteroscedasticity.

Combined with visual inspection of the residuals, the model satisfies the assumption of homoscedasticity (constant variance of residuals).

Conclusion: The homoscedasticity assumption is supported by both graphical and statistical evidence.

Conclusion - Model Assumptions Testing

Assumption	Test(s) Used	Conclusion
No Multicollinearity	Variance Inflation Factor (VIF)	Satisfied after iterative reduction (all VIFs ≤ ~5)
Linearity	Residuals vs. Fitted Values Plot	Satisfied – residuals centered and pattern-free
Independence	Residuals vs. Fitted Values Plot	Satisfied – no autocorrelation observed
Homoscedasticity	Residual Spread Plot + GQ Test	Satisfied – consistent spread, p = 0.6096
Normality of Residuals	Histogram + Q-Q Plot	Satisfied – approximately normal distribution

All core linear regression assumptions have been reasonably validated through both visual and statistical diagnostics.

Final Model

```
# Refit the final model
final_model = sm.OLS(y_train, X_train_vif8).fit()

# Predictions
y_train_final_pred = final_model.predict(X_train_vif8)
y_test_final_pred = final_model.predict(X_test_vif8)

# Final evaluation
final_adj_r2 = final_model.rsquared_adj
final_rmse_train = np.sqrt(mean_squared_error(y_train, y_train_final_pred))
final_rmse_test = np.sqrt(mean_squared_error(y_test, y_test_final_pred))

# Output results
print(f"Final Adjusted R²: {final_adj_r2:.4f}")
print(f"Final Train RMSE:  {final_rmse_train:.4f}")
print(f"Final Test RMSE:   {final_rmse_test:.4f}")
```

```
Final Adjusted R²: 0.6668
Final Train RMSE:  0.3341
Final Test RMSE:   0.3533
```

```
# Display the full OLS regression results
final_model.summary()
```

OLS Regression Results

Dep. Variable:	normalized_used_price	**R-squared:**	0.672
Model:	OLS	**Adj. R-squared:**	0.667
Method:	Least Squares	**F-statistic:**	121.9
Date:	Fri, 18 Apr 2025	**Prob (F-statistic):**	0.00
Time:	01:05:41	**Log-Likelihood:**	-779.91
No. Observations:	2417	**AIC:**	1642.
Df Residuals:	2376	**BIC:**	1879.
Df Model:	40		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	3.5845	0.060	60.124	0.000	3.468	3.701
4g	0.3164	0.019	16.708	0.000	0.279	0.354
5g	0.1977	0.038	5.172	0.000	0.123	0.273
selfie_camera_mp	0.0311	0.001	23.333	0.000	0.028	0.034
int_memory	0.0005	9.74e-05	5.601	0.000	0.000	0.001
weight	0.0023	8.12e-05	27.997	0.000	0.002	0.002
brand_name_Alcatel	-0.1889	0.067	-2.824	0.005	-0.320	-0.058
brand_name_Asus	0.0282	0.067	0.419	0.675	-0.104	0.160
brand_name_BlackBerry	0.1074	0.099	1.082	0.279	-0.087	0.302
brand_name_Celkon	-0.4507	0.093	-4.839	0.000	-0.633	-0.268
brand_name_Coolpad	-0.1448	0.104	-1.389	0.165	-0.349	0.060
brand_name_Gionee	0.0596	0.082	0.728	0.466	-0.101	0.220
brand_name_Google	0.2404	0.121	1.991	0.047	0.004	0.477
brand_name_HTC	0.0234	0.068	0.345	0.730	-0.109	0.156
brand_name_Honor	-0.0531	0.069	-0.769	0.442	-0.188	0.082
brand_name_Huawei	-0.0476	0.062	-0.764	0.445	-0.170	0.074
brand_name_Infinix	-0.2995	0.132	-2.265	0.024	-0.559	-0.040
brand_name_Karbonn	-0.0899	0.096	-0.938	0.348	-0.278	0.098
brand_name_LG	-0.0951	0.063	-1.500	0.134	-0.220	0.029
brand_name_Lava	-0.1919	0.089	-2.156	0.031	-0.366	-0.017
brand_name_Lenovo	-0.0847	0.063	-1.336	0.182	-0.209	0.040
brand_name_Meizu	-0.0337	0.079	-0.425	0.671	-0.189	0.122
brand_name_Micromax	-0.3113	0.067	-4.628	0.000	-0.443	-0.179
brand_name_Microsoft	-0.0725	0.126	-0.574	0.566	-0.320	0.175
brand_name_Motorola	-0.1061	0.069	-1.532	0.126	-0.242	0.030
brand_name_Nokia	-0.1218	0.071	-1.708	0.088	-0.262	0.018
brand_name_OnePlus	0.2662	0.111	2.403	0.016	0.049	0.483
brand_name_Oppo	-0.0430	0.067	-0.640	0.522	-0.175	0.089
brand_name_Others	-0.1116	0.059	-1.904	0.057	-0.227	0.003
brand_name_Panasonic	-0.0727	0.079	-0.917	0.359	-0.228	0.083
brand_name_Realme	-0.1865	0.086	-2.170	0.030	-0.355	-0.018
brand_name_Samsung	-0.0080	0.060	-0.133	0.894	-0.126	0.110
brand_name_Sony	0.0675	0.071	0.955	0.340	-0.071	0.206
brand_name_Spice	-0.3130	0.090	-3.475	0.001	-0.490	-0.136
brand_name_Vivo	-0.0142	0.068	-0.208	0.835	-0.148	0.119
brand_name_XOLO	-0.0757	0.078	-0.970	0.332	-0.229	0.077
brand_name_Xiaomi	-0.0042	0.067	-0.063	0.950	-0.136	0.128
brand_name_ZTE	-0.0873	0.067	-1.312	0.190	-0.218	0.043
os_Others	-0.5688	0.041	-14.010	0.000	-0.648	-0.489
os_Windows	0.0426	0.065	0.656	0.512	-0.085	0.170
os_iOS	0.2022	0.084	2.405	0.016	0.037	0.367

Omnibus:	116.086	**Durbin-Watson:**	1.925

Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	346.003
Skew:	-0.178	**Prob(JB):**	7.35e-76
Kurtosis:	4.819	**Cond. No.**	1.03e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.03e+04. This might indicate that there are
strong multicollinearity or other numerical problems.

Model Evalution Summary

The final linear regression model was fitted using a reduced set of predictors selected through iterative variance inflation factor (VIF) analysis. This process was used to address multicollinearity and ensure the model met the core assumptions of linear regression.

The model achieved the following performance:

Adjusted R^2: 0.6668

Train RMSE: 0.3341

Test RMSE: 0.3533

Although the original model attained a higher adjusted R^2 of approximately 0.84, it exhibited severe multicollinearity, with several predictors showing VIFs exceeding 100. These values compromised interpretability and coefficient reliability. Through a systematic reduction of high-VIF features, the final model offers improved stability and diagnostic soundness, with all core linear regression assumptions reasonably satisfied.

This version is considered more appropriate for deriving actionable business insights and supporting a dynamic pricing framework based on consistent, interpretable predictors.

Actionable Insights and Recommendations

Business Objective Recap:

The objective of this analysis was to develop a linear regression model capable of predicting the normalized price of used or refurbished mobile devices. The goal was to identify the most influential factors affecting price to support data-driven dynamic pricing strategies.

Key Model Insights

- Selfie Camera Megapixels, Internal Memory, and Device Weight emerged as significant contributors to used price in the final model, consistent with earlier EDA patterns showing moderate positive correlations. These features provide objective signals of a device’s functional appeal and perceived quality.
- 4G Connectivity remained a relevant feature throughout model refinement, suggesting that even baseline network capability continues to affect resale value.
- Although brand effects appeared strong in EDA, they were excluded due to high multicollinearity. Their influence may still be captured indirectly through technical features retained in the model. However, from an industry standpoint, brand is known to affect consumer perception and resale value — and should still be considered in downstream business logic or pricing classification tiers.
- The consistency between EDA patterns and final model results strengthens confidence in the model’s reliability and its value for guiding pricing strategy.

Business Recommendations

- Prioritize technical features like camera specifications, memory, and weight in dynamic pricing models, as they consistently influence resale value. These features demonstrated consistent, interpretable influence on price retention and can be used as reliable valuation drivers.
- Avoid over-reliance on brand alone as a pricing proxy. While brand matters in consumer behavior, its effects are often captured through technical specifications. Use brand cues in hybrid approaches or soft rules, not as primary drivers.
- Focus pricing precision on mid-range devices, where model performance is strongest and residual variance is lowest. For premium and budget-tier devices, consider supplementing predictions with business rules or explore nonlinear or ensemble models to capture more complex patterns.
- Acknowledge that moderate prediction error (~7–10%) is expected given the normalized pricing range. Continue to refine the model as more data becomes available, and explore enhancements such as battery health, cosmetic condition, or warranty status for future iterations.
- Consider integration of the final model into real-time pricing workflows or tools that support trade-in evaluation, bulk resale programs, or consumer marketplace listings. The model is well-suited for scalable application in data-informed pricing systems.