Smartphone Price Prediction

Vishakha Joshi (22070126132)
Yash Chandak (22070126134)
Girish Mahale (23070126504)

GitHub Link - https://github.com/girishmahale786/smartphone-price-prediction
Deployment Link - https://smartphone-price-prediction.streamlit.app

Data Aquisition

Importing Libraries
In this cell, we start by importing the necessary libraries for our project. We use BeautifulSoup for parsing HTML, Pandas for data manipulation, Requests for making HTTP requests, and other standard Python libraries.

In [ ]:
```
from bs4 import BeautifulSoup
import pandas as pd
import requests
import time
import os
```

Function Definition - get_soup
In this cell, we define a custom function called `get_soup(url)`. This function takes a URL as its input and returns a BeautifulSoup object that we can use to parse the content of a webpage. We've also added error handling to manage request exceptions and implemented a retry mechanism in case of issues.

In [ ]:
```
def get_soup(url):
    while True:
        try:
            response = requests.get(url)
            response.raise_for_status()
            return BeautifulSoup(response.content, 'html.parser')
        except requests.exceptions.RequestException as e:
            time.sleep(2)
            continue
```

Function Definition - scrape_phone_details
Here, we define another function named `scrape_phone_details(phone_url)`. This function is responsible for extracting details of a mobile phone from a given URL. It collects information such as the phone's name, price, brand, rating, and specifications. All of these details are stored in a dictionary and returned.

In [ ]:
```
def scrape_phone_details(phone_url):
    phone_soup = get_soup(phone_url)

    name = phone_soup.find('div', class_='aMaAEs').find('span', class_='B_NuCI')
```

```
    disc_price = phone_soup.find('div', class_='_30jeq3').text.strip()

    price = disc_price
    if phone_soup.find('div', class_='_3I9_wc'):
        price = phone_soup.find('div', class_='_3I9_wc').text.strip()

    brand = None
    if phone_soup.find('div', class_='_1MR4o5'):
        brand = phone_soup.find('div', class_='_1MR4o5').find_all('a')[3].text.s

    rating = None
    if phone_soup.find('div', class_='_3LWZlK'):
        rating = phone_soup.find('div', class_='_3LWZlK').text.strip()

    phone = {
        'Name': name,
        'Brand': brand,
        'Price': price,
        'Discounted Price': disc_price,
        'Rating': rating
    }

    specs_table = phone_soup.find_all('table', class_='_14cfVK')
    for spec in specs_table:
        for tr in spec.find_all('tr'):
            td = tr.contents
            if len(td) > 1:
                phone[td[0].text.strip()] = td[1].text.strip()

    return phone
```

Function Definition - scrape_flipkart_data

In this cell, we define a function called `scrape_flipkart_data(base_url, brand_urls)`. This function is the core of our web scraping project. It scrapes data from Flipkart for various brands of mobile phones. It takes a base URL and a dictionary of brand URLs as inputs, iterates through the brand URLs, and collects information from each page. The data is then stored in a nested dictionary structure for further analysis.

In [ ]:
```
def scrape_flipkart_data(base_url, brand_urls):
    phones = {}
    for brand, url in brand_urls.items():
        phones[brand] = []

        brand_soup = get_soup(url)
        page_count = 0
        if brand_soup.find('div', class_='_2MImiq'):
            page_count = int(brand_soup.find('div', class_='_2MImiq').span.text.

        for page in range(0, page_count + 1):
            page_url = f'{url}&page={page + 1}'
            page_soup = get_soup(page_url)
            phones_list = page_soup.find_all('div', class_='_13oc-S')

            for phone in phones_list:
                phone_url = f"{base_url}{phone.find('a')['href']}"
                phone_specs = scrape_phone_details(phone_url)
                phones[brand].append(phone_specs)
```

```
    return phones
```

Define Base URL and Brand URLs

Here, we set the base URL to 'https://www.flipkart.com' and define the URLs for various smartphone brands on Flipkart. Each brand URL is specified, allowing us to focus on collecting data for specific brands.

In [ ]:
```
base_url = 'https://www.flipkart.com'
search = f'{base_url}/search?sid=tyy%2C4io&otracker=CLP_Filters&p%5B%5D=facets.p
apple = f'{search}&p%5B%5D=facets.brand%255B%255D%3DAPPLE'
samsung = f'{search}&p%5B%5D=facets.brand%255B%255D%3DSAMSUNG'
google = f'{search}&p%5B%5D=facets.brand%255B%255D%3DGoogle'
nothing = f'{search}&p%5B%5D=facets.brand%255B%255D%3DNothing'
asus = f'{search}&p%5B%5D=facets.brand%255B%255D%3DASUS'
oneplus = f'{search}&p%5B%5D=facets.brand%255B%255D%3DOnePlus'
oppo = f'{search}&p%5B%5D=facets.brand%255B%255D%3DOPPO'
vivo = f'{search}&p%5B%5D=facets.brand%255B%255D%3Dvivo'
mi = f'{search}&p%5B%5D=facets.brand%255B%255D%3DMi'
redmi = f'{search}&p%5B%5D=facets.brand%255B%255D%3DREDMI'
realme = f'{search}&p%5B%5D=facets.brand%255B%255D%3Drealme'
poco = f'{search}&p%5B%5D=facets.brand%255B%255D%3DPOCO'
iqoo = f'{search}&p%5B%5D=facets.brand%255B%255D%3DIQOO'
motorola = f'{search}&p%5B%5D=facets.brand%255B%255D%3DMOTOROLA'

brand_urls = {
    'apple': apple,
    'samsung': samsung,
    'google': google,
    'nothing': nothing,
    'asus': asus,
    'oneplus': oneplus,
    'oppo': oppo,
    'vivo': vivo,
    'mi': mi,
    'redmi': redmi,
    'realme': realme,
    'poco': poco,
    'iqoo': iqoo,
    'motorola': motorola,
}
```

Scraping Data

This cell is where the actual scraping happens. We call the `scrape_flipkart_data` function, passing in the base URL and the dictionary of brand URLs. The code then iterates through each brand, scrapes information from their respective pages, and stores the data in separate CSV files, one for each brand.

In [ ]:
```
phones = scrape_flipkart_data(base_url, brand_urls)
for brand in brand_urls.keys():
    df = pd.DataFrame(phones[brand])
    df.to_csv(f'data/{brand}.csv', index=False)
```

Combining Data from CSV Files

In this final cell, we bring all the data together. We read the CSV files for each brand into Pandas DataFrames and merge them into one comprehensive DataFrame named

'phones_df.' This combined dataset is saved as a 'phones.csv' file, which we can use for further analysis, research, or visualization.

In [ ]:
```
all_df = []
for file in os.listdir('data/'):
    df = pd.read_csv(f'data/{file}')
    all_df.append(df)

phones_df = pd.concat(all_df)
phones_df.to_csv('data/phones.csv', index=False)
```

Importing Libraries

We start by importing the necessary libraries, including scikit-learn for preprocessing tasks and Pandas for data manipulation.

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
from sklearn.impute import KNNImputer
import pandas as pd
import numpy as np
import re
```

Dataset Description

We load the dataset 'smartphones.csv' and display the first few rows, shape, description, and information about the dataset. This helps us understand the data's structure and identify any missing values.

In [ ]:
```
df = pd.read_csv('smartphones.csv')
df.head()
```

Out[ ]:

	Name	Brand	Price	Color	SIM Type	Hybrid Sim Slot	Display Size	Resolution	Resolution Type
0	APPLE iPhone 11 (Black, 128 GB)	APPLE Mobiles	₹48,900	Black	Dual Sim	No	15.49 cm (6.1 inch)	1792 x 828 Pixels	Liquic Retina HD Display
1	APPLE iPhone 11 (Black, 64 GB)	APPLE Mobiles	₹43,900	Black	Dual Sim	No	15.49 cm (6.1 inch)	1792 x 828 Pixels	Liquic Retina HD Display
2	APPLE iPhone 11 (White, 128 GB)	APPLE Mobiles	₹48,900	White	Dual Sim	No	15.49 cm (6.1 inch)	1792 x 828 Pixels	Liquic Retina HD Display
3	APPLE iPhone 13 (Midnight, 128 GB)	APPLE Mobiles	₹69,900	Midnight	Dual Sim	No	15.49 cm (6.1 inch)	2532 x 1170 Pixels	Super Retina XDR Display
4	APPLE iPhone 13 (Green, 128 GB)	APPLE Mobiles	₹69,900	Green	Dual Sim	No	15.49 cm (6.1 inch)	2532 x 1170 Pixels	Super Retina XDR Display

5 rows × 27 columns

In [ ]:
```
df.shape
```

Out[ ]: (3296, 27)

In [ ]:
```
df.describe()
```

Out[ ]:

	Name	Brand	Price	Color	SIM Type	Hybrid Sim Slot	Display Size	Resolution	Resolut T
count	3296	3284	3296	3296	3296	3208	3296	3296	2
unique	2914	13	442	768	4	2	72	286	
top	OnePlus 10R 5G (Sierra Black, 256 GB) (12 GB ...	SAMSUNG Mobiles	₹17,999	Black	Dual Sim	No	16.51 cm (6.5 inch)	2400 x 1080 Pixels	Full H
freq	6	563	100	166	3057	2251	367	633	1

4 rows × 27 columns

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3296 entries, 0 to 3295
Data columns (total 27 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   Name                 3296 non-null   object
 1   Brand                3284 non-null   object
 2   Price                3296 non-null   object
 3   Color                3296 non-null   object
 4   SIM Type             3296 non-null   object
 5   Hybrid Sim Slot      3208 non-null   object
 6   Display Size         3296 non-null   object
 7   Resolution           3296 non-null   object
 8   Resolution Type      2470 non-null   object
 9   Display Type         2273 non-null   object
 10  Processor Type       2550 non-null   object
 11  Internal Storage     3296 non-null   object
 12  Primary Camera       3293 non-null   object
 13  Secondary Camera     2645 non-null   object
 14  Network Type         3295 non-null   object
 15  Bluetooth Version    2440 non-null   object
 16  Battery Capacity     2963 non-null   object
 17  Width                2564 non-null   object
 18  Height               2568 non-null   object
 19  Depth                2562 non-null   object
 20  Weight               2535 non-null   object
 21  Quick Charging       1068 non-null   object
 22  Processor Core       2972 non-null   object
 23  Primary Clock Speed  2890 non-null   object
 24  Audio Jack           1674 non-null   object
 25  RAM                  2864 non-null   object
 26  Expandable Storage   1431 non-null   object
dtypes: object(27)
memory usage: 695.4+ KB
```

In [ ]:
```
df.isna().sum().sum()
```

Out[ ]:
```
14371
```

In [ ]:
```
df.isna().sum()
```

Out[ ]:
```
Name                    0
Brand                  12
Price                   0
Color                   0
SIM Type                0
Hybrid Sim Slot        88
Display Size            0
Resolution              0
Resolution Type       826
Display Type         1023
Processor Type        746
Internal Storage        0
Primary Camera          3
Secondary Camera      651
Network Type            1
Bluetooth Version     856
Battery Capacity      333
Width                 732
Height                728
Depth                 734
Weight                761
Quick Charging       2228
Processor Core        324
Primary Clock Speed   406
Audio Jack           1622
RAM                   432
Expandable Storage   1865
dtype: int64
```

Data Preprocessing

We begin the data preprocessing phase. We perform various transformations to make the data more suitable for machine learning. We start by converting strings in object columns to lowercase for consistency.

In [ ]:
```
object_cols = df.select_dtypes('object')
df[object_cols.columns] = object_cols.apply(lambda col: col.str.lower())
```

Brand Preprocessing

- We address missing values in the 'Brand' column by extracting the brand from the 'Name' column where it's missing.
- We encode the 'Brand' column using Label Encoding to convert brand names into numerical categories.

In [ ]:
```
df['Brand'].isna().sum()
```

Out[ ]:
```
12
```

In [ ]:
```
df['Brand'].value_counts()
```

Out[ ]:
```
Brand
samsung mobiles     563
apple mobiles       432
realme mobiles      392
vivo mobiles        308
redmi mobiles       277
oppo mobiles        275
mi mobiles          264
oneplus mobiles     215
iqoo mobiles        159
poco mobiles        137
asus mobiles        135
motorola mobiles    113
google mobiles       14
Name: count, dtype: int64
```

In [ ]:
```
df['Brand'] = df['Brand'].str.replace('mobiles', '').str.strip()
df['Brand'].value_counts()
```

Out[ ]:
```
Brand
samsung     563
apple       432
realme      392
vivo        308
redmi       277
oppo        275
mi          264
oneplus     215
iqoo        159
poco        137
asus        135
motorola    113
google       14
Name: count, dtype: int64
```

In [ ]:
```
df[['Name', 'Brand']][df['Brand'].isna()]
```

Out[ ]:

	Name	Brand
1117	nothing phone (2) (white, 512 gb) (12 gb ram)	NaN
1118	nothing phone (2) (dark grey, 128 gb) (8 gb ram)	NaN
1119	nothing phone (1) (white, 256 gb) (8 gb ram)	NaN
1120	nothing phone (1) (black, 256 gb) (8 gb ram)	NaN
1121	nothing phone (1) (black, 128 gb) (8 gb ram)	NaN
1122	nothing phone (2) (dark grey, 256 gb) (12 gb ...	NaN
1123	nothing phone (2) (white, 256 gb) (12 gb ram)	NaN
1124	nothing phone (2) (dark grey, 512 gb) (12 gb ...	NaN
1125	nothing phone (1) (black, 256 gb) (12 gb ram)	NaN
1126	nothing phone (1) (white, 256 gb) (12 gb ram)	NaN
2361	redmi note 9 pro max (interstellar black, 128 ...	NaN
2362	redmi note 9 pro max (interstellar black, 128 ...	NaN

In [ ]:
```
def get_brand(name):
    return name.split(' ')[0]

new_brand = df['Name'].apply(get_brand)
df['Brand'].fillna(new_brand, inplace=True)
df.drop(columns=['Name'], inplace=True)
df['Brand'].isna().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
le = LabelEncoder()
df['Brand'] = le.fit_transform(df['Brand'])
df['Brand'] = df['Brand'].astype('category')
df['Brand'].value_counts()
```

Out[ ]:
```
Brand
12    563
0     432
10    392
13    308
11    279
8     275
4     264
7     215
3     159
9     137
1     135
5     113
2      14
6      10
Name: count, dtype: int64
```

Price Preprocessing

We clean the 'Price' column by removing currency symbols and commas and converting it to an integer.

In [ ]:
```
def clean_price(price_str):
    return price_str.replace('₹', '').replace(',', '')

df['Price'] = df['Price'].apply(clean_price).astype(int)
```

Color Preprocessing

- We clean the 'Color' column by categorizing similar colors and handling values that occur less frequently.
- We encode the 'Color' column using Label Encoding.

In [ ]:
```
df['Color'].value_counts()
```

Out[ ]:
```
Color
black              166
gold               109
blue                82
silver              81
white               69
                  ...
silk white           1
?stardust brown      1
bronze               1
stardust silver      1
racing black         1
Name: count, Length: 734, dtype: int64
```

In [ ]:
```
def clean_var(choices, df, var):
    for choice in choices:
        idx = df[var][df[var].str.contains(choice)].index
        df.loc[idx, var] = choice

colors = ['black', 'blue', 'green', 'white', 'gold', 'silver', 'red', 'grey', 'g
clean_var(colors, df, 'Color')
df['Color'].value_counts()
```

Out[ ]:
```
Color
black             817
blue              658
green             261
white             244
gold              204
                 ...
bronze              1
viva magneta        1
satin maroon        1
radiant mist        1
marble odyssey      1
Name: count, Length: 131, dtype: int64
```

In [ ]:
```
color_groups = {
    'blue': ['sky', 'night'],
    'red': ['coral', 'pink', 'sunset'],
    'green': ['mint'],
```

```
    'orange': ['copper', 'brown'],
    'silver': ['diamond', 'starlight'],
    'gray': ['grey', 'graphite'],
    'purple': ['voilet', 'lavender']
}

for color_group, colors in color_groups.items():
    for color in colors:
        df['Color'].replace(color, color_group, inplace=True)
```

In [ ]:
```
extra_colors = df['Color'].value_counts()[df['Color'].value_counts() < 30].index
for color in extra_colors:
    idx = df['Color'][df['Color'].str.contains(color)].index
    df.loc[idx, 'Color'] = 'others'
df['Color'].value_counts()
```

Out[ ]:
```
Color
black     817
blue      726
green     282
others    279
white     244
silver    220
gold      204
gray      178
red       160
purple     96
orange     49
yellow     41
Name: count, dtype: int64
```

In [ ]:
```
le = LabelEncoder()
df['Color'] = le.fit_transform(df['Color'])
df['Color'] = df['Color'].astype('category')
df['Color'].value_counts()
```

Out[ ]:
```
Color
0     817
1     726
4     282
6     279
10    244
9     220
2     204
3     178
8     160
7      96
5      49
11     41
Name: count, dtype: int64
```

SIM Type Preprocessing

We encode the 'SIM Type' column using Label Encoding.

In [ ]:
```
le = LabelEncoder()
df['SIM Type'] = le.fit_transform(df['SIM Type'])
```

```
df['SIM Type'] = df['SIM Type'].astype('category')
df['SIM Type'].value_counts()
```

Out[ ]:
```
SIM Type
0    3057
3     140
1      94
2       5
Name: count, dtype: int64
```

Hybrid Sim Slot Preprocessing

- We handle missing values in the 'Hybrid Sim Slot' column based on the 'SIM Type.'
- We encode the 'Hybrid Sim Slot' column using Label Encoding.

In [ ]:
```
df['Hybrid Sim Slot'].isna().sum()
```

Out[ ]:
```
88
```

In [ ]:
```
df['Hybrid Sim Slot'].value_counts()
```

Out[ ]:
```
Hybrid Sim Slot
no     2251
yes     957
Name: count, dtype: int64
```

In [ ]:
```
no_hsim = df['Hybrid Sim Slot'][df['Hybrid Sim Slot'].isna()][df['SIM Type'] ==
df.loc[no_hsim, 'Hybrid Sim Slot'] = 'no'
```

In [ ]:
```
df['Hybrid Sim Slot'] = df['Hybrid Sim Slot'].apply(lambda x: 1 if x == 'yes' el
df['Hybrid Sim Slot'].value_counts()
```

Out[ ]:
```
Hybrid Sim Slot
0.0    2251
1.0     957
Name: count, dtype: int64
```

In [ ]:
```
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[['Hybrid Sim Slot']])
imputed_data = pd.Series([l[0] for l in list(imputed_data)], name='Hybrid Sim Sl
df['Hybrid Sim Slot'] = imputed_data.astype(int).astype('category')
df['Hybrid Sim Slot'].isna().sum()
```

Out[ ]:
```
0
```

Display Size Preprocessing

We extract and rename the 'Display Size' column to 'Display Size (cm).'

In [ ]:
```
df['Display Size'] = df['Display Size'].str.extract(r'(\d+\.*\d*)\s*cm').astype(
df.rename(columns={'Display Size': 'Display Size (cm)'}, inplace=True)
```

Display & Resolution Type Preprocessing

- We clean and categorize the 'Display Type' and 'Resolution Type' columns.

- We handle missing values in both columns based on the other.
- We encode both columns using Label Encoding.

Display Type

In [ ]:
```
df['Display Type'].value_counts()
```

Out[ ]:
```
Display Type
full hd+ amoled display                    215
super retina xdr display                   197
full hd+ display                           118
full hd+ super amoled display              103
super amoled                                95
                                          ...
ltps ips with tol                            1
pls tft lcd                                  1
optic amoled                                 1
dynamic amoled 2x - infinity-o display       1
oled fhd+display                             1
Name: count, Length: 189, dtype: int64
```

In [ ]:
```
df['Display Type'] = df['Display Type'].str.replace('display', '').str.strip()
df['Display Type'].fillna('unknown', inplace=True)
choices = ['retina', 'amoled', ' oled', 'oled ', 'lcd', 'ips', 'hd', 'tft']
clean_var(choices, df, 'Display Type')
```

In [ ]:
```
disp_groups = {
    'oled': [' oled', 'oled ']
}

for disp_group, disps in disp_groups.items():
    for disp in disps:
        df['Display Type'].replace(disp, disp_group, inplace=True)
```

In [ ]:
```
extra_disps = df['Display Type'].value_counts()[df['Display Type'].value_counts(
for disp in extra_disps:
    idx = df['Display Type'][df['Display Type'].str.contains(disp)].index
    df.loc[idx, 'Display Type'] = 'others'
df['Display Type'].value_counts()
```

Out[ ]:
```
Display Type
unknown    1023
amoled      804
lcd         633
hd          235
retina      229
ips         196
oled         95
tft          49
others       32
Name: count, dtype: int64
```

In [ ]:
```
le = LabelEncoder()
df['Display Type'] = le.fit_transform(df['Display Type'])
df['Display Type'].value_counts()
```

Out[ ]:
```
Display Type
8    1023
0     804
3     633
1     235
6     229
2     196
4      95
7      49
5      32
Name: count, dtype: int64
```

Resolution Type

In [ ]:
```
df['Resolution Type'].value_counts()
```

Out[ ]:
```
Resolution Type
full hd+                           1217
hd+                                 444
super retina xdr display            221
full hd+ amoled display             135
retina hd display                   125
full hd                             107
hd                                   63
quad hd+                             44
full hd+ super amoled display        36
retina display                       23
super retina hd display              21
liquid retina hd display             12
quad hd                               9
full hd+ e3 super amoled display      7
qxga+                                 3
wvga                                  2
wqhd                                  1
Name: count, dtype: int64
```

In [ ]:
```
df['Resolution Type'] = df['Resolution Type'].str.replace('display', '').str.str
df['Resolution Type'].fillna('unknown', inplace=True)
choices = ['retina', 'amoled', 'hd']
clean_var(choices, df, 'Resolution Type')
```

In [ ]:
```
res_groups = {
    'amoled': ['qxga+', 'wvga']
}

for res_group, ress in res_groups.items():
    for res in ress:
        df['Resolution Type'].replace(res, res_group, inplace=True)
df['Resolution Type'].value_counts()
```

Out[ ]:
```
Resolution Type
hd         1885
unknown     826
retina      402
amoled      183
Name: count, dtype: int64
```

In [ ]:
```
le = LabelEncoder()
df['Resolution Type'] = le.fit_transform(df['Resolution Type'])
```

```
df['Resolution Type'].value_counts()
```

Out[ ]:
```
Resolution Type
1    1885
3     826
2     402
0     183
Name: count, dtype: int64
```

Handling NULL Values

In [ ]:
```
df.loc[df['Display Type'][df['Display Type'] == 8].index, 'Display Type'] = np.n

df.loc[df['Resolution Type'][df['Resolution Type'] == 3].index, 'Resolution Type
```

In [ ]:
```
df['Display Type'].isna().sum()
```

Out[ ]:
```
1023
```

In [ ]:
```
res_to_disp = df['Display Type'].isna().index
df['Display Type'].fillna(df['Resolution Type'][res_to_disp], inplace=True)
df['Display Type'].isna().sum()
```

Out[ ]:
```
725
```

In [ ]:
```
df['Resolution Type'].isna().sum()
```

Out[ ]:
```
826
```

In [ ]:
```
disp_to_res = df['Resolution Type'].isna().index
df['Resolution Type'].fillna(df['Display Type'][disp_to_res], inplace=True)
df['Resolution Type'].isna().sum()
```

Out[ ]:
```
725
```

In [ ]:
```
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[['Display Type', 'Resolution Type']])
imputed_data = pd.DataFrame(imputed_data, columns=['Display Type', 'Resolution T
df['Display Type'] = imputed_data['Display Type'].astype(int).astype('category')
df['Resolution Type'] = imputed_data['Resolution Type'].astype(int).astype('cate
df['Display Type'].isna().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
df['Resolution Type'].isna().sum()
```

Out[ ]:
```
0
```

Resolution Prepocessing

We extract pixel width and height from the 'Resolution' column.

In [ ]:
```
px_w = df['Resolution'].apply(lambda x: re.findall('[0-9]+', x)[0]).astype(int)
px_w.name = 'Pixel Width'
px_h = df['Resolution'].apply(lambda x: re.findall('[0-9]+', x)[1]).astype(int)
px_h.name = 'Pixel Height'
```

```
df = pd.concat([df, px_w, px_h], axis=1)
df.drop(columns=['Resolution'], inplace=True)
```

Processor Type, Core & Primary Clock Speed Preprocessing

- We clean and categorize the 'Processor Type' column.
- We handle missing values in 'Processor Type.'
- We encode the 'Processor Type' column using Label Encoding.
- We clean the 'Processor Core' column.
- We handle missing values in 'Processor Core.'
- We encode the 'Processor Core' column using Label Encoding.
- We convert 'Primary Clock Speed' from GHz to MHz, handle missing values, and impute them.

Processor Type

In [ ]:
```
df['Processor Core'].isna().sum()
```

Out[ ]:
```
324
```

In [ ]:
```
df['Processor Type'].value_counts()
```

Out[ ]:
```
Processor Type
a15 bionic chip
71
mediatek helio p35
67
qualcomm snapdragon 680
60
a14 bionic chip with next generation neural engine
59
a12 bionic chip
47

..
krait cpu  the motorola x8 mobile computing system is comprised of a qualcomm s
napdragon s4 pro family processor, a natural language processor and a contextua
l computing processor     1
qualcomm snapdragon 430
1
qualcomm snapdragon 801 msm8974-ac
1
qualcomm snapdragon octa core 750g 5g processor
1
qualcomm snapdragon 765g
1
Name: count, Length: 326, dtype: int64
```

In [ ]:
```
df['Processor Type'].fillna('unknown', inplace=True)

choices = ['qualcomm', 'snapdragon', 'mediatek', 'dimensity', 'helio', 'apple',
clean_var(choices, df, 'Processor Type')
```

In [ ]:
```
proc_groups = {
    'qualcomm snapdragon': ['qualcomm', 'snapdragon'],
```

```
    'samsung exynos': ['exynos'],
    'mediatek dimensity': ['dimensity'],
    'mediatek helio': ['helio'],
    'google tensor': ['tensor'],
    'apple chip': ['apple', 'bionic chip', 'chip'],
}

for proc_group, procs in proc_groups.items():
    for proc in procs:
        df['Processor Type'].replace(proc, proc_group, inplace=True)
```

In [ ]:
```
df['Processor Type'] = df['Processor Type'].str.replace('(', '').str.replace(')'
extra_procs = df['Processor Type'].value_counts()[df['Processor Type'].value_cou
for proc in extra_procs:
    idx = df['Processor Type'][df['Processor Type'].str.contains(proc)].index
    df.loc[idx, 'Processor Type'] = 'others'
df['Processor Type'].value_counts()
```

Out[ ]:
```
Processor Type
qualcomm snapdragon    1045
unknown                 746
mediatek                661
apple chip              432
samsung exynos          190
mediatek dimensity       73
mediatek helio           34
others                   29
unisoc                   26
octa core                25
intel                    24
google tensor            11
Name: count, dtype: int64
```

In [ ]:
```
le = LabelEncoder()
df['Processor Type'] = le.fit_transform(df['Processor Type'])
df['Processor Type'] = df['Processor Type'].astype('category')
df['Processor Type'].value_counts()
```

Out[ ]:
```
Processor Type
8     1045
11     746
3      661
0      432
9      190
4       73
5       34
7       29
10      26
6       25
2       24
1       11
Name: count, dtype: int64
```

In [ ]:
```
df.loc[df['Processor Type'][df['Processor Type'] == 11].index, 'Processor Type']
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[['Processor Type']])
imputed_data = pd.Series([l[0] for l in list(imputed_data)], name='Processor Typ
df['Processor Type'] = imputed_data.astype(int).astype('category')
df['Processor Type'].isna().sum()
```

Out[ ]:
```
0
```

Processor Core

In [ ]:
```
df['Processor Core'].isna().sum()
```

Out[ ]:
```
324
```

In [ ]:
```
df['Processor Core'].value_counts()
```

Out[ ]:
```
Processor Core
octa core      2553
dual core       193
hexa core       151
quad core        65
deca core         8
single core       2
Name: count, dtype: int64
```

In [ ]:
```
le = LabelEncoder()
df['Processor Core'] = le.fit_transform(df['Processor Core'])
df['Processor Core'] = df['Processor Core'].astype('category')
df['Processor Core'].value_counts()
```

Out[ ]:
```
Processor Core
3    2553
6     324
1     193
2     151
4      65
0       8
5       2
Name: count, dtype: int64
```

In [ ]:
```
df.loc[df['Processor Core'][df['Processor Core'] == 6].index, 'Processor Core']
df['Processor Core'].value_counts()
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[['Processor Core']])
imputed_data = pd.Series([l[0] for l in list(imputed_data)], name='Processor Cor
df['Processor Core'] = imputed_data.astype(int).astype('category')
df['Processor Core'].isna().sum()
```

Out[ ]:
```
0
```

Primary Clock Speed

In [ ]:
```
df['Primary Clock Speed'].isna().sum()
```

Out[ ]:
```
406
```

In [ ]:
```
df['Primary Clock Speed'].value_counts()
```

Out[ ]:

Primary Clock Speed	
2.4 ghz	478
2 ghz	477
2.2 ghz	426
2.3 ghz	361
1.8 ghz	141
3.2 ghz	111
2.5 ghz	70
3 ghz	69
2.6 ghz	68
2.05 ghz	64
2.8 ghz	62
2.84 ghz	58
3.36 ghz	44
1.6 ghz	35
1.84 ghz	32
2.99 ghz	30
1.4 ghz	29
1.5 ghz	25
2.73 ghz	25
1.95 ghz	24
1.7 ghz	23
2.96 ghz	23
2.1 ghz	22
2.85 ghz	20
2.2 mhz	17
3.1 ghz	13
2.9 ghz	12
3.18 ghz	12
3.05 ghz	11
2.7 ghz	10
1.82 ghz	8
1.3 ghz	8
2.4 mhz	8
3.19 ghz	7
2.91 ghz	6
2.42 mhz	6
3.2 mhz	5
2.86 ghz	5
1.2 ghz	4
2.3 mhz	4
90 mhz	4
2 mhz	3
2.995 ghz	3
1.25 ghz	3
1600 mhz	3
90 ghz	2
950 mhz	2
2.39 ghz	2
1.9 ghz	2
2.15 ghz	2
900 mhz	2
2.26 ghz	1
2350 mhz	1
3.05 mhz	1
2.8 mhz	1
3.19 mhz	1
2.649 ghz	1
3.09 ghz	1
0.1 mhz	1

```
2.45 ghz       1
Name: count, dtype: int64
```

In [ ]:
```
def ghz_to_mhz(value):
    if 'ghz' in value:
        value = float(value.replace(' ghz', '')) * 1000
        return f'{value} mhz'
    return value

df['Primary Clock Speed'].fillna('unknown', inplace=True)
df['Primary Clock Speed'] = df['Primary Clock Speed'].apply(ghz_to_mhz).str.repl
```

In [ ]:
```
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[['Primary Clock Speed']])
imputed_data = pd.Series([l[0] for l in list(imputed_data)], name='Primary Clock
df['Primary Clock Speed'] = imputed_data
df['Primary Clock Speed'].isna().sum()
```

Out[ ]:
```
0
```

Primary & Secondary Camera Preprocessing

- We clean and categorize the 'Primary Camera' and 'Secondary Camera' columns.
- We handle missing values and impute them using K-Nearest Neighbors.

Primary Camera

In [ ]:
```
df['Primary Camera'].isna().sum()
```

Out[ ]:
```
3
```

In [ ]:
```
df['Primary Camera'].value_counts()
```

Out[ ]:
```
Primary Camera
50mp rear camera           292
12mp + 12mp                190
13mp rear camera           164
64mp rear camera           153
12mp rear camera           140
                          ...
12mp + 5mp + 4mp             1
48mp + 5mp + 5mp             1
10mp rear camera             1
48mp + 5mp + 16mp            1
48mp + 13mp + 8mp + 8mp      1
Name: count, Length: 180, dtype: int64
```

In [ ]:
```
def clean_cam(value):
    if not isinstance(value, float):
        cam = re.findall('[0-9]+', value)
        if len(cam) != 0:
            return int(cam[0])
    return value
df['Primary Camera'] = df['Primary Camera'].apply(clean_cam)
```

Secondary Camera

In [ ]:
```
df['Secondary Camera'].isna().sum()
```

Out[ ]:
```
651
```

In [ ]:
```
df['Secondary Camera'].value_counts()
```

Out[ ]:
```
Secondary Camera
16mp front camera
645
8mp front camera
443
32mp front camera
276
12mp front camera
265
5mp front camera
262
13mp front camera
212
7mp front camera
124
20mp front camera
99
10mp front camera
45
1.2mp front camera
43
16mp + 16mp dual front camera
24
25mp front camera
23
50mp front camera
22
24mp front camera
21
32mp + 8mp dual front camera
21
44mp front camera
18
20mp + 2mp dual front camera
14
2mp front camera
11
16mp + 8mp dual front camera
11
40mp front camera
10
50mp + 8mp dual front camera
8
48mp + 13mp dual front camera
7
60mp front camera
6
16mp + 2mp dual front camera
6
44mp + 2mp dual front camera
6
10.8mp front camera
6
16mp dual front camera
3
24mp + 5mp dual front camera
3
20mp + 8mp dual front camera
3
8mp + 8mp dual front camera
```

```
2
4mp front camera
2
8mp + 2mp dual front camera
1
32mp + 16mp dual front camera
1
32mp + 2mp dual front camera
1
48mp(f2.0) + 8mp(ultra wide/f2.2) + tof (time-of-flight) 3d-depth rotating came
ra      1
Name: count, dtype: int64
```

In [ ]:
```
df['Secondary Camera'] = df['Secondary Camera'].apply(clean_cam)
```

In [ ]:
```
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[['Primary Camera', 'Secondary Camera']])
imputed_data = pd.DataFrame(imputed_data, columns=['Primary Camera', 'Secondary
df['Primary Camera'] = imputed_data['Primary Camera'].astype(int)
df['Secondary Camera'] = imputed_data['Secondary Camera'].astype(int)
df['Primary Camera'].isna().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
df['Secondary Camera'].isna().sum()
```

Out[ ]:
```
0
```

Network Type Preprocessing

- We clean the 'Network Type' column and handle missing values.
- We encode the 'Network Type' column using Label Encoding.

In [ ]:
```
df['Network Type'].isna().sum()
```

Out[ ]:
```
1
```

In [ ]:
```
df['Network Type'].value_counts()
```

Out[ ]:
```
Network Type
5g, 4g, 3g, 2g                538
4g volte, 4g, 3g, 2g          514
5g                            387
5g, 4g volte, 4g, 3g, 2g      322
4g, 3g, 2g                    311
3g, 4g, 2g                    243
4g volte                      149
3g, 4g volte, 2g              135
3g, 4g volte, 4g, 2g          127
4g                            108
5g, 4g volte                  105
2g, 3g, 4g, 5g                 41
4g volte, 4g                   33
4g volte, 3g                   32
2g, 3g, 4g                     29
5g, 4g volte, 4g               24
3g                             23
5g, 4g, 3g                     18
3g, 2g                         17
3g, 4g volte, 4g               16
3g, 4g                         13
2g, 3g, 4g volte, 5g           12
2g, 3g, 4g, 4g volte           11
4g volte, 4g, 2g, 3g           11
3g, 4g volte                   10
2g, 3g, 4g, 4g volte, 5g        9
4g volte, 3g, 2g                8
4g, 2g, 3g                      8
4g volte, 5g                    7
4g, 3g                          7
5g, 4g, 4g volte, 3g            5
2g                              4
4g volte, 4g, 3g                3
5g, 4g volte, 3g                2
4g, 3g, 4g volte, 2g            2
4g, 4g volte, 3g                2
3g, 4g, 5g                      2
3g, 4g volte, 5g                1
4g, 5g                          1
4g volte, 3g, 4g                1
5g, 4g                          1
5g, 4g volte, 4g, 3g            1
4g volte, 3g, 4g, 2g            1
5g, 4g, 4g volte                1
Name: count, dtype: int64
```

In [ ]:
```python
def clean_ntype(value):
    if not isinstance(value, float):
        return max(set(re.findall('[0-9]+', value)))
    return value

df['Network Type'] = df['Network Type'].apply(clean_ntype).astype(float)
df['Network Type'].value_counts()
```

Out[ ]:
```
Network Type
4.0    1774
5.0    1477
3.0      40
2.0       4
Name: count, dtype: int64
```

In [ ]:
```
df['Network Type'].fillna(df['Network Type'].median(), inplace=True)
df['Network Type'] = df['Network Type'].astype(int).astype('category')
df['Network Type'].isna().sum()
```

Out[ ]:
```
0
```

Bluetooth Version Preprocessing

- We clean the 'Bluetooth Version' column and handle missing values.
- We encode the 'Bluetooth Version' column using Label Encoding.

In [ ]:
```
df['Bluetooth Version'].isna().sum()
```

Out[ ]:
```
856
```

In [ ]:
```
df['Bluetooth Version'].value_counts()
```

Out[ ]:
```
Bluetooth Version
v5.0                                              865
v5.1                                              328
v5.2                                              321
5                                                 276
v5.3                                              206
4.2                                               147
4                                                 119
v4.2                                               79
4.1                                                32
5.1                                                28
5.2                                                 7
2.1, 4.0, 5.0                                       7
v4.1                                                5
v5.0, v4.0, v2.1 + edr                              4
5.2, a2dp, le, aptx hd, aptx adaptive               3
2.1                                                 3
5.3                                                 3
v5.3, ble                                           2
'v5.0                                               2
bluetooth v5.1, bluetooth low energy (ble)          1
5.1, a2dp, le, aptx hd                              1
6                                                   1
Name: count, dtype: int64
```

In [ ]:
```
def clean_blue(value):
    if not isinstance(value, float):
        val = re.findall(r'\d+\.\d+|\d+', value)
        if len(val) != 0:
            return float(max(val))
df['Bluetooth Version'] = df['Bluetooth Version'].apply(clean_blue)
```

In [ ]:
```
df['Bluetooth Version'].fillna(df['Bluetooth Version'].median(), inplace=True)
df['Bluetooth Version'].isna().sum()
```

Out[ ]:
```
0
```

Width, Height, Depth and Weight Preprocessing

- We clean and transform the 'Width,' 'Height,' 'Depth,' and 'Weight' columns.
- We handle missing values using K-Nearest Neighbors imputation.

Width

In [ ]:
```
df['Width'].isna().sum()
```

Out[ ]:
```
732
```

In [ ]:
```
df['Width'].unique()
```

Out[ ]:
```
array(['75.7 mm', '71.5 mm', '67.3 mm', '78.1 mm', '77.6 mm', '64.2 mm',
       '77.4 mm', '77.8 mm', '67.1 mm', '77.9 mm', '71.4 mm', '70.9 mm',
       '67 mm', '58.6 mm', '59.2 mm', '77.2 mm', '77.5 mm', '76 mm',
       '77.38 mm', '73.7 mm', '75.65 mm', '75.44 mm', '84 mm', nan,
       '76.28 mm', '73.98 mm', '76.24 mm', '77 mm', '71.74 mm', '76.7 mm',
       '78.84 mm', '77.25 mm', '68.5 mm', '75.51 mm', '84.3 mm', '78 mm',
       '72.8 mm', '61.44 mm', '76.16 mm', '71.8 mm', '73.2 mm', '72.9 mm',
       '76.6 mm', '68.2 mm', '158 mm', '16.1 mm', '75.2 mm', '7.7 mm',
       '15.9 mm', '16.3 mm', '16.4 mm', '16.2 mm', '21.1 mm', '16 mm',
       '16.5 mm', '75.72 mm', '76.8 mm', '76.4 mm', '75.73 mm', '74.6 mm',
       '76.19 mm', '75.45 mm', '75.41 mm', '75.21 mm', '76.2 mm',
       '75.8 mm', '69.6 mm', '77.07 mm', '75.35 mm', '75.4 mm',
       '76.68 mm', '88.3 mm', '69.2 mm', '77.26 mm', '74.3 mm',
       '71.85 mm', '75.58 mm', '73.6 mm', '74.8 mm', '75.3 mm', '8.35 mm',
       '75.49 mm', '71.68 mm', '73.96 mm', '73.84 mm', '71.99 mm',
       '74.4 mm', '74.236 mm', '73.5 mm', '74.46 mm', '74.94 mm',
       '75.88 mm', '71.2 mm', '75.83 mm', '75.95 mm', '76.08 mm',
       '76.54 mm', '73.87 mm', '75.99 mm', '73.82 mm', '74.95 mm',
       '73.47 mm', '75 mm', '65.3 mm', '74 mm', '72.4 mm', '73.4 mm',
       '73.3 mm', '75.9 mm', '7.4 mm', '74.1 mm', '74.2 mm', '75.1 mm',
       '75.6 mm', '75.98 mm', '73.23 mm', '75.03 mm', '73.8 mm',
       '76.1 mm', '72.1 mm', '75.5 mm', '154.5 mm', '77.35 mm',
       '73.77 mm', '73.1 mm', '19 mm', '70.4 mm', '80.8 mm', '76.21 mm',
       '76.09 mm', '76.03 mm', '77.3 mm', '8 mm', '75.34 mm', '77.1 mm',
       '76.41 mm', '76.9 mm', '76.67 mm', '73.9 mm', '75.16 mm', '126 mm',
       '74.24 mm', '74.7 mm', '75.18 mm', '75.96 mm', '74.29 mm',
       '76.56 mm', '74.5 mm', '76.15 mm', '129.9 mm', '71.9 mm',
       '130.1 mm', '76.3 mm', '72.2 mm', '159 mm', '9 mm', '70.6 mm',
       '70.2 mm', '78.6 mm', '78.8 mm', '68.7 mm', '68.1 mm', '159.3 mm',
       '76.5 mm', '79.2 mm', '128.2 mm', '73.53 mm', '74.92 mm',
       '74.44 mm', '76.17 mm', '75.55 mm', '74.28 mm', '73.79 mm',
       '73.52 mm', '72.42 mm', '75.46 mm', '73.27 mm', '76.32 mm',
       '75.63 mm', '73.21 mm', '75.84 mm', '75.09 mm', '75.01 mm',
       '75.23 mm', '74.53 mm', '75.32 mm', '74.17 mm', '76.77 mm',
       '74.37 mm', '73.91 mm', '75.93 mm', '77.33 mm', '76.46 mm',
       '74.08 mm', '75.39 mm', '75.04 mm', '74.78 mm', '75.74 mm',
       '74.71 mm', '75.19 mm', '75.08 mm', '73.35 mm', '75.24 mm',
       '73.24 mm'], dtype=object)
```

In [ ]:
```
df['Width'] = df['Width'].str.replace(' mm', '').astype(float)
```

Height

In [ ]:
```
df['Height'].isna().sum()
```

Out[ ]:
```
728
```

In [ ]:
```
df['Height'].unique()
```

Out[ ]:
```
array(['150.9 mm', '146.7 mm', '138.4 mm', '160.8 mm', '147.5 mm',
       '160.7 mm', '131.5 mm', '157.5 mm', '158 mm', '138.3 mm',
       '158.2 mm', '144 mm', '143.6 mm', '138.1 mm', '158.1 mm',
       '158.4 mm', '123.8 mm', '124.4 mm', '152.5 mm', '156.5 mm',
       '156 mm', '10.9 m', '149 mm', '152.59 mm', '156.4 mm', '149.5 mm',
       '153 mm', '159.1 mm', '164.55 mm', nan, '158.41 mm', '143.7 mm',
       '146.87 mm', '151.4 mm', '159 mm', '154.3 mm', '141.18 mm',
       '151.45 mm', '158.9 mm', '172.83 mm', '148 mm', '157.96 mm',
       '166.9 mm', '171 mm', '170.99 mm', '109 mm', '148.2 mm',
       '124.42 mm', '158.83 mm', '152.2 mm', '155.6 mm', '152 mm',
       '162.9 mm', '145.6 mm', '76.7 mm', '7.4 mm', '164.8 mm', '16.5 mm',
       '7.5 mm', '158.51 mm', '7.9 mm', '7.6 mm', '7.8 mm', '10.3 mm',
       '160.53 mm', '165.38 mm', '163.7 mm', '163.6 mm', '163.65 mm',
       '158.5 mm', '156.48 mm', '159.21 mm', '165.1 mm', '157.9 mm',
       '151 mm', '155.4 mm', '164.9 mm', '158.58 mm', '158.7 mm',
       '165.75 mm', '173.1 mm', '144.55 mm', '160.73 mm', '156.7 mm',
       '153.48 mm', '158.73 mm', '165.5 mm', '162.58 mm', '158.3 mm',
       '139.2 mm', '151.86 mm', '149.33 mm', '161.83 mm', '161.78 mm',
       '161.42 mm', '158.43 mm', '160 mm', '159.38 mm', '161.76 mm',
       '160.5 mm', '158.48 mm', '160.89 mm', '161.19 mm', '163.95 mm',
       '169.613 mm', '165.21 mm', '160.1 mm', '160.98 mm', '157.6 mm',
       '158.35 mm', '165.89 mm', '166.13 mm', '170.47 mm', '163.26 mm',
       '161.56 mm', '164.19 mm', '160.61 mm', '163 mm', '165.22 mm',
       '162.66 mm', '129.4 mm', '150.2 mm', '140.8 mm', '148.35 mm',
       '162.1 mm', '157.7 mm', '165.3 mm', '162.6 mm', '160.2 mm',
       '152.9 mm', '16.1 mm', '154.2 mm', '162.4 mm', '163.8 mm',
       '164.2 mm', '165.65 mm', '159.9 mm', '160.01 mm', '163.74 mm',
       '167.3 mm', '160.3 mm', '160.6 mm', '164.4 mm', '162.3 mm',
       '164 mm', '165.6 mm', '166.2 mm', '159.7 mm', '158.8 mm',
       '156.8 mm', '163.9 mm', '155.9 mm', '162 mm', '7.38 mm',
       '159.8 mm', '154.4 mm', '161.2 mm', '156.1 mm', '161.8 mm',
       '160.14 mm', '156.2 mm', '148.1 mm', '153.3 mm', '8 mm',
       '142.7 mm', '163.63 mm', '154.5 mm', '150.5 mm', '161.3 mm',
       '168.6 mm', '165.88 mm', '163.99 mm', '162.91 mm', '161.11 mm',
       '155 mm', '161.81 mm', '163.2 mm', '155.5 mm', '163.3 mm',
       '168.76 mm', '161.9 mm', '167.16 mm', '165.7 mm', '161.6 mm',
       '161.5 mm', '164.1 mm', '76 mm', '164.5 mm', '164.74 mm',
       '162.5 mm', '165.2 mm', '161 mm', '160.9 mm', '164.3 mm',
       '159.2 mm', '162.2 mm', '158.96 mm', '160.83 mm', '157 mm',
       '165.66 mm', '163.64 mm', '169.59 mm', '160.46 mm', '161.48 mm',
       '161.7 mm', '165.4 mm', '166.8 mm', '167.7 mm', '154.9 mm',
       '164.7 mm', '157.4 mm', '157.8 mm', '159.6 mm', '163.4 mm',
       '168.5 mm', '146.3 mm', '155.1 mm', '155.7 mm', '161.4 mm',
       '166 mm', '9 mm', '165 mm', '167.2 mm', '151.7 mm', '159.3 mm',
       '146 mm', '149.3 mm', '156.9 mm', '152.4 mm', '147.7 mm',
       '163.5 mm', '148.9 mm', '74.4 mm', '151.2 mm', '142.4 mm',
       '159.5 mm', '164.05 mm', '158.91 mm', '164.42 mm', '164.06 mm',
       '164.26 mm', '160.87 mm', '159.68 mm', '157.2 mm', '164.57 mm',
       '163.96 mm', '159.46 mm', '164.41 mm', '155.11 mm', '159.63 mm',
       '164.95 mm', '164.07 mm', '163.86 mm', '155.97 mm', '154.81 mm',
       '159.01 mm', '159.43 mm', '161.24 mm', '160.63 mm', '161.97 mm',
       '162.39 mm', '155.06 mm', '164.15 mm', '162.04 mm', '159.54 mm',
       '159.64 mm', '159.53 mm', '154.45 mm', '155.87 mm', '157.25 mm',
       '154.6 mm', '159.25 mm', '164.54 mm', '157.91 mm', '158.59 mm',
       '158.46 mm', '155.21 mm'], dtype=object)
```

In [ ]:
```
df['Height'] = df['Height'].str.replace(' mm', '').str.replace(' m', '').astype(
```

Depth

In [ ]:
```
df['Depth'].isna().sum()
```

Out[ ]:
```
734
```

In [ ]:
```
df['Depth'].unique()
```

Out[ ]:
```
array(['8.3 mm', '7.65 mm', '7.8 mm', '7.3 mm', '7.4 mm', '7.85 mm',
       '7.7 mm', '8.1 mm', '7.1 mm', '6.9 mm', '7.5 mm', '7.6 mm',
       '8.97 mm', '10.9 m', '10.8 mm', '10.55 mm', '152.5 mm', '7.9 mm',
       '7.69 mm', '8.55 mm', '9.67 mm', nan, '10.5 mm', '8.46 mm',
       '7.99 mm', '7.95 mm', '11.95 mm', '9.85 mm', '8.9 mm', '8.5 mm',
       '9.9 mm', '8.85 mm', '10.29 mm', '9.78 mm', '10.34 mm', '11.2 mm',
       '8.7 mm', '9 mm', '0.8 mm', '8.8 mm', '0.9 mm', '9.16 mm',
       '6.2 mm', '6.81 mm', '8.16 mm', '8.34 mm', '8.05 mm', '9.4 mm',
       '9.33 mm', '8.26 mm', '8.4 mm', '7.25 mm', '8.475 mm', '8.47 mm',
       '8.96 mm', '8.35 mm', '75.3 mm', '8.75 mm', '8.59 mm', '8.49 mm',
       '8.29 mm', '7.58 mm', '6.79 mm', '8.39 mm', '7.45 mm', '7.49 mm',
       '9.6 mm', '9.18 mm', '9.15 mm', '8.99 mm', '9.09 mm', '8.25 mm',
       '9.89 mm', '9.13 mm', '6.99 mm', '8.89 mm', '9.19 mm', '8.79 mm',
       '8.98 mm', '9.97 mm', '8.6 mm', '8.2 mm', '8 mm', '7.89 mm',
       '7.93 mm', '8.28 mm', '7.67 mm', '7.59 mm', '76 mm', '7.34 mm',
       '8.67 mm', '7.48 mm', '12 mm', '7.35 mm', '9.3 mm', '9.1 mm',
       '9.5 mm', '7.38 mm', '8.17 mm', '8.08 mm', '7.98 mm', '8.12 mm',
       '75 mm', '8.92 mm', '10.08 mm', '8.77 mm', '7.78 mm', '8.43 mm',
       '9.8 mm', '8.65 mm', '9.65 mm', '8.18 mm', '8.51 mm', '8.95 mm',
       '8.87 mm', '8.81 mm', '6.1 mm', '9.7 mm', '6.3 mm', '17.1 mm',
       '8.15 mm', '7.57 mm', '7.36 mm', '8.07 mm', '8.19 mm', '7.41 mm',
       '8.42 mm', '8.38 mm', '8.62 mm', '7.29 mm', '7.55 mm', '7.73 mm',
       '7.39 mm', '8.41 mm', '8.48 mm', '7.79 mm', '9.34 mm', '8.54 mm',
       '7.32 mm', '7.77 mm', '9.11 mm', '7.83 mm', '8.13 mm', '7.37 mm',
       '8.21 mm', '8.68 mm', '8.04 mm'], dtype=object)
```

In [ ]:
```
df['Depth'] = df['Depth'].str.replace(' mm', '').str.replace(' m', '').astype(fl
```

Weight

In [ ]:
```
df['Weight'].isna().sum()
```

Out[ ]:
```
761
```

In [ ]:
```
df['Weight'].unique()
```

Out[ ]:
```
array(['194 g', '173 g', '172 g', '144 g', '203 g', '162 g', '206 g',
       '240 g', '140 g', '148 g', '238 g', '208 g', '226 g', '143 g',
       '187 g', '133 g', '188 g', '192 g', '138 g', '177 g', '129 g',
       '202 g', '112 g', '132 g', '174 g', '170 g', '150 g', '155 g',
       '165 g', '195 g', '190 g', '160 g', '175 g', '180 g', '147 g',
       '120 g', nan, '185 g', '169 g', '196 g', '242 g', '145 g', '116 g',
       '200 g', '178 g', '197 g', '193.5 g', '212 g', '454 g', '214.5 g',
       '158 g', '157 g', '205 g', '204 g', '186 g', '216 g', '154 g',
       '181 g', '130 g', '168 g', '209 g', '0.45 kg', '191 g', '173.8 g',
       '146 g', '149 g', '184 g', '171 g', '168.3 g', '198.5 g', '220 g',
       '176 g', '221 g', '210 g', '179 g', '166.8 g', '163 g', '179.5 g',
       '201.2 g', '183 g', '199 g', '153 g', '189 g', '182 g', '193 g',
       '189.6 g', '164 g', '137 g', '161 g', '152 g', '141 g', '215 g',
       '156 g', '201 g', '213 g', '198 g', '225 g', '189.5 g', '172.5 g',
       '174.5 g', '199.8 g', '207 g', '196.5 g', '194.5 g', '186.5 g',
       '183.7 g', '178.8 g', '253 g', '228 g', '233 g', '263 g', '166 g',
       '211 g', '218 g', '167 g', '271 g', '282 g', '180.5 g', '219 g',
       '180.3 g', '214.85 g', '163.7 g', '192.3 g', '190.5 g', '163.5 g',
       '185.5 gm', '156.2 g', '201.8 g', '162.8 g', '186.7 g', '171.5 g',
       '181.5 g'], dtype=object)
```

In [ ]:
```python
def kg_to_g(value):
    if 'kg' in value:
        value = float(value.replace(' kg', '')) * 1000
        return f'{value} g'
    return value

df['Weight'].fillna('unknown', inplace=True)
df['Weight'] = df['Weight'].apply(kg_to_g).str.replace(' g', '').str.replace(' g
```

Handling NULL Values

In [ ]:
```python
cols = ['Width', 'Height', 'Depth', 'Weight']
imputer = KNNImputer()
imputed_data = imputer.fit_transform(df[cols])
imputed_data = pd.DataFrame(imputed_data, columns=cols)
for col in cols:
    df[col] = imputed_data[col]
    print(f'NULL Values in {col}: {df[col].isna().sum()}')
```

```
NULL Values in Width: 0
NULL Values in Height: 0
NULL Values in Depth: 0
NULL Values in Weight: 0
```

Quick Charging Preprocessing

- We handle missing values in the 'Quick Charging' column.
- We encode the 'Quick Charging' column using Label Encoding.

In [ ]:
```python
df['Quick Charging'].isna().sum()
```

Out[ ]:
```
2228
```

In [ ]:
```python
df['Quick Charging'].value_counts()
```

Out[ ]:
```
Quick Charging
yes    1048
no       20
Name: count, dtype: int64
```

In [ ]:
```
df['Quick Charging'].fillna('no', inplace=True)
df['Quick Charging'].isna().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
df['Quick Charging'] = df['Quick Charging'].apply(lambda x: 1 if x == 'yes' else
df['Quick Charging'].value_counts()
```

Out[ ]:
```
Quick Charging
0    2248
1    1048
Name: count, dtype: int64
```

Audio Jack Preprocessing

- We clean and categorize the 'Audio Jack' column.
- We handle missing values and encode the 'Audio Jack' column using Label Encoding.

In [ ]:
```
df['Audio Jack'].isna().sum()
```

Out[ ]:
```
1622
```

In [ ]:
```
df['Audio Jack'].value_counts()
```

Out[ ]:
```
Audio Jack
3.5mm            1089
3.5 mm            259
type c            125
yes                47
usb type c         35
type-c             31
usb (type c)       31
3.5                22
usb type-c         16
usb (type-c)       10
no                  7
?3.5 mm             1
3.5 mm stereo       1
Name: count, dtype: int64
```

In [ ]:
```
df['Audio Jack'].fillna('no', inplace=True)
choices = ['3.5', 'type c', 'type-c']
clean_var(choices, df, 'Audio Jack')
df['Audio Jack'].value_counts()
```

Out[ ]:
```
Audio Jack
no        1629
3.5       1372
type c     191
type-c      57
yes         47
Name: count, dtype: int64
```

In [ ]:
```
aud_groups = {
    'yes': ['type-c', 'type c', '3.5']
}

for aud_group, auds in aud_groups.items():
    for aud in auds:
        df['Audio Jack'].replace(aud, aud_group, inplace=True)
df['Audio Jack'].value_counts()
```

Out[ ]:
```
Audio Jack
yes    1667
no     1629
Name: count, dtype: int64
```

In [ ]:
```
df['Audio Jack'] = df['Audio Jack'].apply(lambda x: 1 if x == 'yes' else 0).asty
df['Audio Jack'].value_counts()
```

Out[ ]:
```
Audio Jack
1    1667
0    1629
Name: count, dtype: int64
```

Battery Capacity Preprocessing

We clean the 'Battery Capacity' column and handle missing values.

In [ ]:
```
df['Battery Capacity'].isna().sum()
```

Out[ ]:
```
333
```

In [ ]:
```
df['Battery Capacity'].value_counts()
```

Out[ ]:
```
Battery Capacity
5000 mah    1282
4500 mah     308
6000 mah     232
4000 mah     174
3000 mah      81
            ...
3120 mah       1
3430 mah       1
3520 mah       1
3095 mah       1
4315 mah       1
Name: count, Length: 92, dtype: int64
```

In [ ]:
```
df['Battery Capacity'] = df['Battery Capacity'].str.replace(' mah', '').astype(f
df['Battery Capacity'].fillna(int(df['Battery Capacity'].mean()), inplace=True)
df['Battery Capacity'] = df['Battery Capacity'].astype(int)
df['Battery Capacity'].isna().sum()
```

Out[ ]:
```
0
```

Internal Storage & RAM Preprocessing

- We clean and transform the 'Internal Storage' and 'RAM' columns.
- We handle missing values and convert values from TB to GB.

Internal Storage

In [ ]:
```
df['Internal Storage'].isna().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
df['Internal Storage'].value_counts()
```

Out[ ]:
```
Internal Storage
128 gb    1587
64 gb      732
256 gb     557
32 gb      210
512 gb      90
16 gb       72
1 tb        24
8 gb        16
6 gb         7
4 gb         1
Name: count, dtype: int64
```

In [ ]:
```
def tb_to_gb(value):
    if 'tb' in value:
        value = int(value.replace(' tb', '')) * 1024
        return f'{value} gb'
    return value

df['Internal Storage'] = df['Internal Storage'].apply(tb_to_gb).str.replace(' gb
```

RAM Preprocessing

In [ ]:
```
df['RAM'].isna().sum()
```

Out[ ]:
```
432
```

In [ ]:
```
df['RAM'].value_counts()
```

Out[ ]:
```
RAM
8 gb      899
6 gb      711
4 gb      707
12 gb     266
3 gb      193
2 gb       57
16 gb      12
128 gb      7
1 gb        6
18 gb       2
10 gb       2
1.5 gb      2
Name: count, dtype: int64
```

In [ ]:
```
df['RAM'] = df['RAM'].str.replace(' gb', '').astype(float)
df['RAM'].fillna(df['RAM'].median(), inplace=True)
df['RAM'] = df['RAM'].astype(int)
df['RAM'].isna().sum()
```

Out[ ]:
```
0
```

Expandable Storage Preprocessing

We handle missing values in the 'Expandable Storage' column and impute them based on 'Internal Storage.'

In [ ]:
```
df['Expandable Storage'].isna().sum()
```

Out[ ]:
```
1865
```

In [ ]:
```
df['Expandable Storage'].value_counts()
```

Out[ ]:
```
Expandable Storage
1 tb      557
256 gb    430
512 gb    261
128 gb     75
2 tb       74
64 gb      25
32 gb       3
400 gb      3
7 gb        1
200 gb      1
12 gb       1
Name: count, dtype: int64
```

In [ ]:
```
df['Expandable Storage'].fillna('unknown', inplace=True)
df['Expandable Storage'] = df['Expandable Storage'].apply(tb_to_gb).str.replace(
```

In [ ]:
```
df.groupby('Internal Storage')['Expandable Storage'].median()
```

Out[ ]:
```
Internal Storage
4           NaN
6           NaN
8          64.0
16        128.0
32        256.0
64        512.0
128      1024.0
256      1024.0
512       512.0
1024        NaN
Name: Expandable Storage, dtype: float64
```

In [ ]:
```
mapping = dict(df.groupby('Internal Storage')['Expandable Storage'].median())
mapping
```

Out[ ]:
```
{4: nan,
 6: nan,
 8: 64.0,
 16: 128.0,
 32: 256.0,
 64: 512.0,
 128: 1024.0,
 256: 1024.0,
 512: 512.0,
 1024: nan}
```

In [ ]:
```
new_storage = df['Internal Storage'].apply(lambda x: mapping[x])
df['Expandable Storage'].fillna(new_storage, inplace=True)
df['Expandable Storage'].isna().sum()
```

Out[ ]:
```
32
```

In [ ]:
```
df['Expandable Storage'].fillna(df['Expandable Storage'].median(), inplace=True)
df['Expandable Storage'] = df['Expandable Storage'].astype(int)
df['Expandable Storage'].isna().sum()
```

Out[ ]:
```
0
```

Final Data

- We check for any remaining missing values.
- We save the preprocessed dataset as 'prep_smartphones.csv' for further analysis and modeling.

In [ ]:
```
df.isna().sum().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
df.isna().sum()
```

Out[ ]:
```
Brand                0
Price                0
Color                0
SIM Type             0
Hybrid Sim Slot      0
Display Size (cm)    0
Resolution Type      0
Display Type         0
Processor Type       0
Internal Storage     0
Primary Camera       0
Secondary Camera     0
Network Type         0
Bluetooth Version    0
Battery Capacity     0
Width                0
Height               0
Depth                0
Weight               0
Quick Charging       0
Processor Core       0
Primary Clock Speed  0
Audio Jack           0
RAM                  0
Expandable Storage   0
Pixel Width          0
Pixel Height         0
dtype: int64
```

In [ ]:
```
len(df.select_dtypes('object').columns)
```

Out[ ]:
```
0
```

In [ ]:
```
len(df.select_dtypes('number').columns)
```

Out[ ]:
```
16
```

In [ ]:
```
len(df.select_dtypes('category').columns)
```

Out[ ]:
```
11
```

In [ ]:
```
df.to_csv('prep_smartphones.csv', index=False)
```

Importing Libraries

We start by importing necessary libraries and setting up the environment.

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import pickle
```

Dataset Description

Loading the dataset from a CSV file named 'prep_smartphones.csv' and displaying the first few rows to get a glimpse of the data.

In [ ]:
```
df = pd.read_csv('prep_smartphones.csv')
df.head()
```

Out[ ]:

	Brand	Price	Color	SIM Type	Hybrid Sim Slot	Display Size (cm)	Resolution Type	Display Type	Processor Type	Interna Storage
0	0	48900	0	0	0	15.49	2	6	0	128
1	0	43900	0	0	0	15.49	2	2	0	64
2	0	48900	10	0	0	15.49	2	6	0	128
3	0	69900	1	0	0	15.49	2	6	0	128
4	0	69900	4	0	0	15.49	2	6	0	128

5 rows × 27 columns

Displaying the shape of the dataset (number of rows and columns).

In [ ]:
```
df.shape
```

Out[ ]:
```
(3296, 27)
```

Displaying basic statistics of the dataset, such as mean, standard deviation, minimum, and maximum values for numeric columns.

In [ ]:
```
df.describe()
```

Out[ ]:

	Brand	Price	Color	SIM Type	Hybrid Sim Slot	Display Size (cm)
count	3296.000000	3296.000000	3296.000000	3296.000000	3296.000000	3296.000000
mean	7.587379	36866.834951	3.500607	0.158981	0.290352	16.182488
std	4.427834	33262.345798	3.512821	0.625870	0.453994	1.374624
min	0.000000	799.000000	0.000000	0.000000	0.000000	10.160000
25%	4.000000	16999.000000	1.000000	0.000000	0.000000	16.210000
50%	9.000000	23990.000000	2.000000	0.000000	0.000000	16.510000
75%	12.000000	41574.000000	6.000000	0.000000	1.000000	16.810000
max	13.000000	189999.000000	11.000000	3.000000	1.000000	41.940000

8 rows × 27 columns

Getting information about the dataset, including data types and non-null counts for each column.

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3296 entries, 0 to 3295
Data columns (total 27 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Brand                3296 non-null   int64  
 1   Price                3296 non-null   int64  
 2   Color                3296 non-null   int64  
 3   SIM Type             3296 non-null   int64  
 4   Hybrid Sim Slot      3296 non-null   int64  
 5   Display Size (cm)    3296 non-null   float64
 6   Resolution Type      3296 non-null   int64  
 7   Display Type         3296 non-null   int64  
 8   Processor Type       3296 non-null   int64  
 9   Internal Storage     3296 non-null   int64  
 10  Primary Camera       3296 non-null   int64  
 11  Secondary Camera     3296 non-null   int64  
 12  Network Type         3296 non-null   int64  
 13  Bluetooth Version    3296 non-null   float64
 14  Battery Capacity     3296 non-null   int64  
 15  Width                3296 non-null   float64
 16  Height               3296 non-null   float64
 17  Depth                3296 non-null   float64
 18  Weight               3296 non-null   float64
 19  Quick Charging       3296 non-null   int64  
 20  Processor Core       3296 non-null   int64  
 21  Primary Clock Speed  3296 non-null   float64
 22  Audio Jack           3296 non-null   int64  
 23  RAM                  3296 non-null   int64  
 24  Expandable Storage   3296 non-null   int64  
 25  Pixel Width          3296 non-null   int64  
 26  Pixel Height         3296 non-null   int64  
dtypes: float64(7), int64(20)
memory usage: 695.4 KB
```

Checking for missing values in the dataset. The result is a count of missing values for entire dataframe.

In [ ]:
```
df.isna().sum().sum()
```

Out[ ]:
```
0
```

Checking for missing values again but displaying the count of missing values for each column.

In [ ]:
```
df.isna().sum()
```

Out[ ]:
```
Brand                0
Price                0
Color                0
SIM Type             0
Hybrid Sim Slot      0
Display Size (cm)    0
Resolution Type      0
Display Type         0
Processor Type       0
Internal Storage     0
Primary Camera       0
Secondary Camera     0
Network Type         0
Bluetooth Version    0
Battery Capacity     0
Width                0
Height               0
Depth                0
Weight               0
Quick Charging       0
Processor Core       0
Primary Clock Speed  0
Audio Jack           0
RAM                  0
Expandable Storage   0
Pixel Width          0
Pixel Height         0
dtype: int64
```

EDA

Categorizing certain columns as category types for better analysis.

In [ ]:
```
category_cols = ['Brand', 'Color', 'SIM Type', 'Hybrid Sim Slot', 'Resolution Ty
df[category_cols] = df[category_cols].astype('category')
categoric = df.select_dtypes(include='category')
numeric = df.select_dtypes(include='number')
```

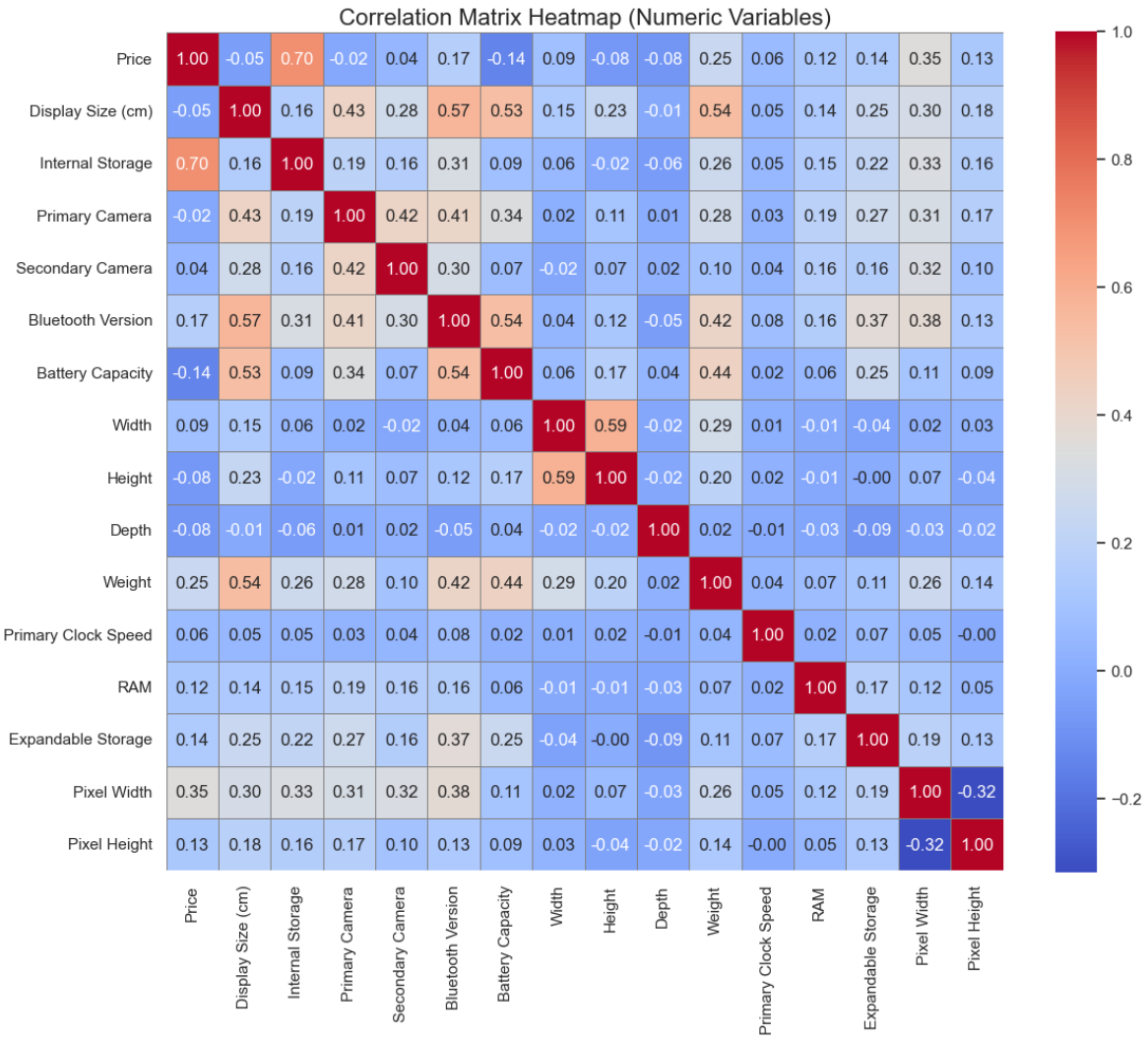
Creating a correlation matrix to visualize the correlation between numeric variables in the dataset.

In [ ]:
```
corr_matrix = numeric.corr()
corr_matrix['Price'].sort_values()
```

Out[ ]:
```
Battery Capacity      -0.141399
Depth                 -0.079805
Height                -0.075750
Display Size (cm)     -0.051669
Primary Camera        -0.015687
Secondary Camera       0.044781
Primary Clock Speed    0.061905
Width                  0.089776
RAM                    0.124198
Pixel Height           0.128298
Expandable Storage     0.142204
Bluetooth Version      0.167440
Weight                 0.245258
Pixel Width            0.347403
Internal Storage       0.701938
Price                  1.000000
Name: Price, dtype: float64
```

Displaying the correlation heatmap using Seaborn.

In [ ]:
```
plt.figure(figsize=(12, 10))
sns.set(font_scale=1)
sns.set_style('whitegrid')
heatmap = sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', square=True, fmt

plt.title('Correlation Matrix Heatmap (Numeric Variables)', fontsize=16)
plt.tight_layout()
plt.show()
```



Inference: The heatmap visually represents the correlation between numeric variables. Higher values on the heatmap indicate stronger positive or negative correlations. For example, the 'Internal Storage' feature is highly positively correlated with 'Price.'

Performing point-biserial correlation analysis between categorical variables and the 'Price' column.

This analysis quantifies the correlation between categorical variables and the 'Price.' It provides the 'H-statistic' and 'p-value' for each categorical variable.

In [ ]:
```
from scipy.stats import pointbiserialr

corrs = []
for col in categoric:
    h_statistic, p_value = pointbiserialr(df[col], df['Price'].to_numpy())
    corr = {
        'Categorical Variable': col,
        'H-statistic': h_statistic,
        'p-value': p_value
    }
    corrs.append(corr)

corr_df = pd.DataFrame(corrs, columns=['Categorical Variable', 'H-statistic', 'p
corr_df_sorted = corr_df.sort_values(by='p-value')
corr_df_sorted
```

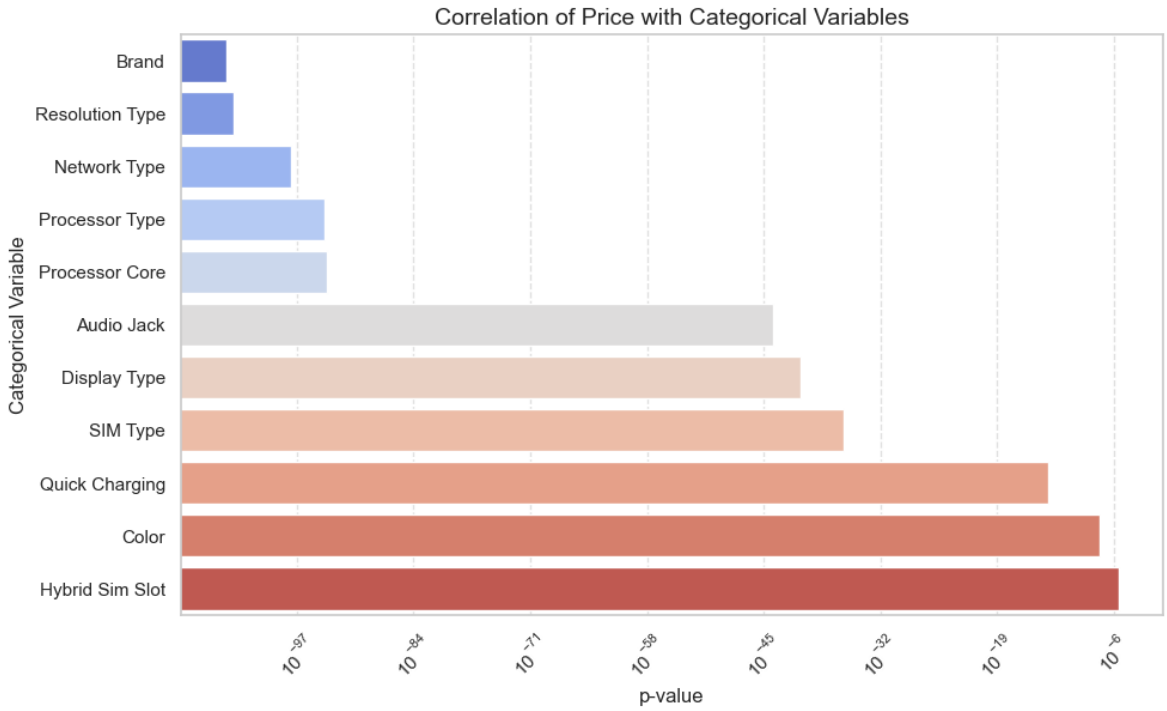
Out[ ]:

	Categorical Variable	H-statistic	p-value
0	Brand	-0.367033	1.202473e-105
4	Resolution Type	0.365707	7.660502e-105
7	Network Type	0.354867	2.079389e-98
6	Processor Type	-0.348224	1.375377e-94
9	Processor Core	-0.347813	2.353500e-94
10	Audio Jack	-0.240866	1.019862e-44
5	Display Type	0.232211	1.338909e-41
2	SIM Type	0.218070	8.946888e-37
8	Quick Charging	0.130972	4.390726e-14
1	Color	0.097037	2.377163e-08
3	Hybrid Sim Slot	-0.080816	3.392067e-06

Creating a bar plot to visualize the p-values from the point-biserial correlation analysis.

In [ ]:
```
plt.figure(figsize=(10, 6))
sns.barplot(x='p-value', y='Categorical Variable', data=corr_df_sorted, palette=
plt.xscale('log')
plt.xlabel('p-value', fontsize=12)
plt.ylabel('Categorical Variable', fontsize=12)
plt.title('Correlation of Price with Categorical Variables', fontsize=14)
plt.tight_layout()
plt.grid(axis='x', linestyle='--', alpha=0.6)
plt.xticks(rotation=45)
plt.show()
```

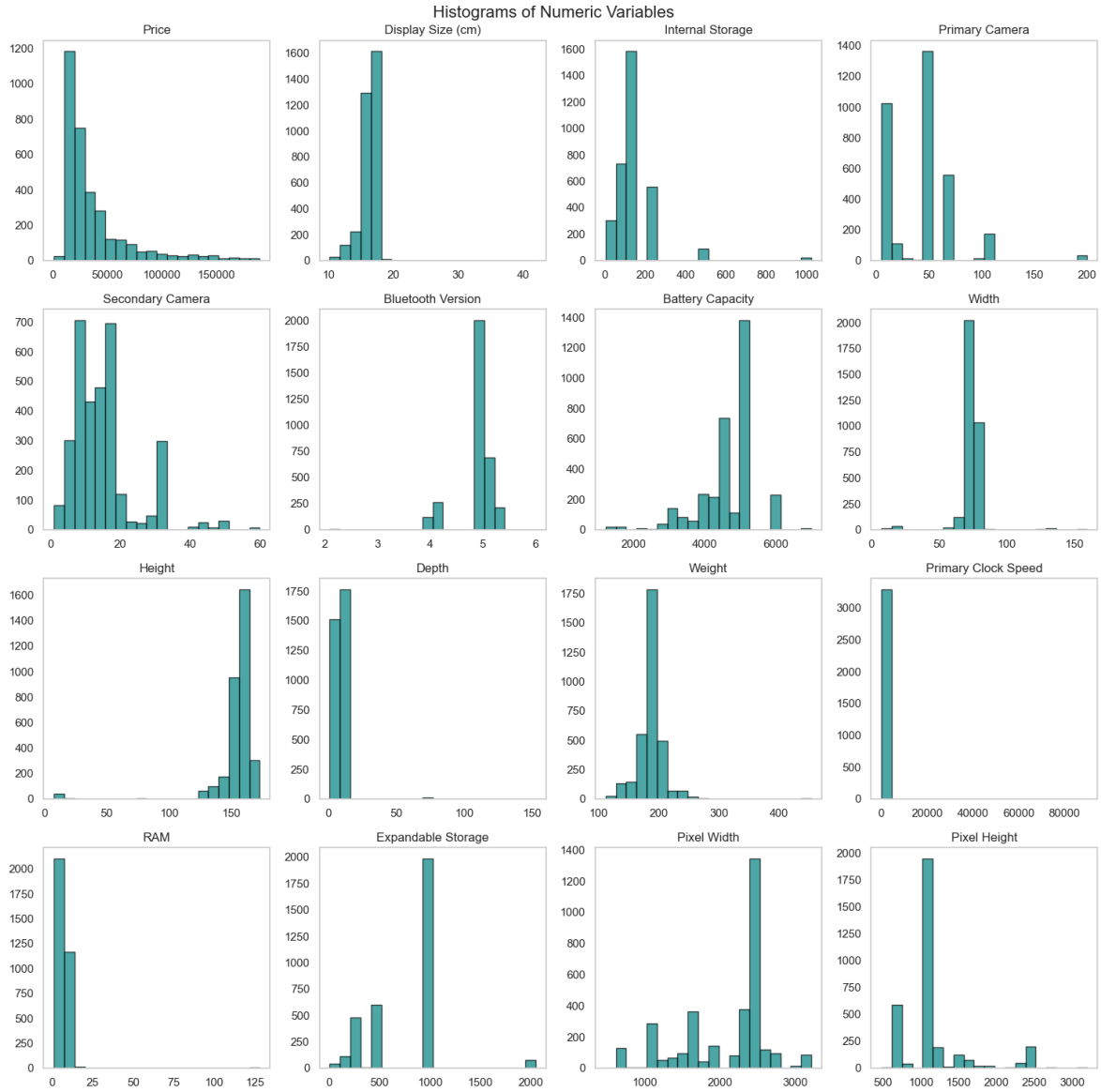
```
C:\Users\Girish\AppData\Local\Temp\ipykernel_11448\1262265084.py:2: FutureWarnin
g:

Passing `palette` without assigning `hue` is deprecated and will be removed in v
0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effe
ct.

  sns.barplot(x='p-value', y='Categorical Variable', data=corr_df_sorted, palette
='coolwarm')
```



Inference: The bar plot helps identify which categorical variables have a significant influence on the 'Price.' Lower p-values indicate stronger correlations.

Generating histograms for numeric variables in the dataset.

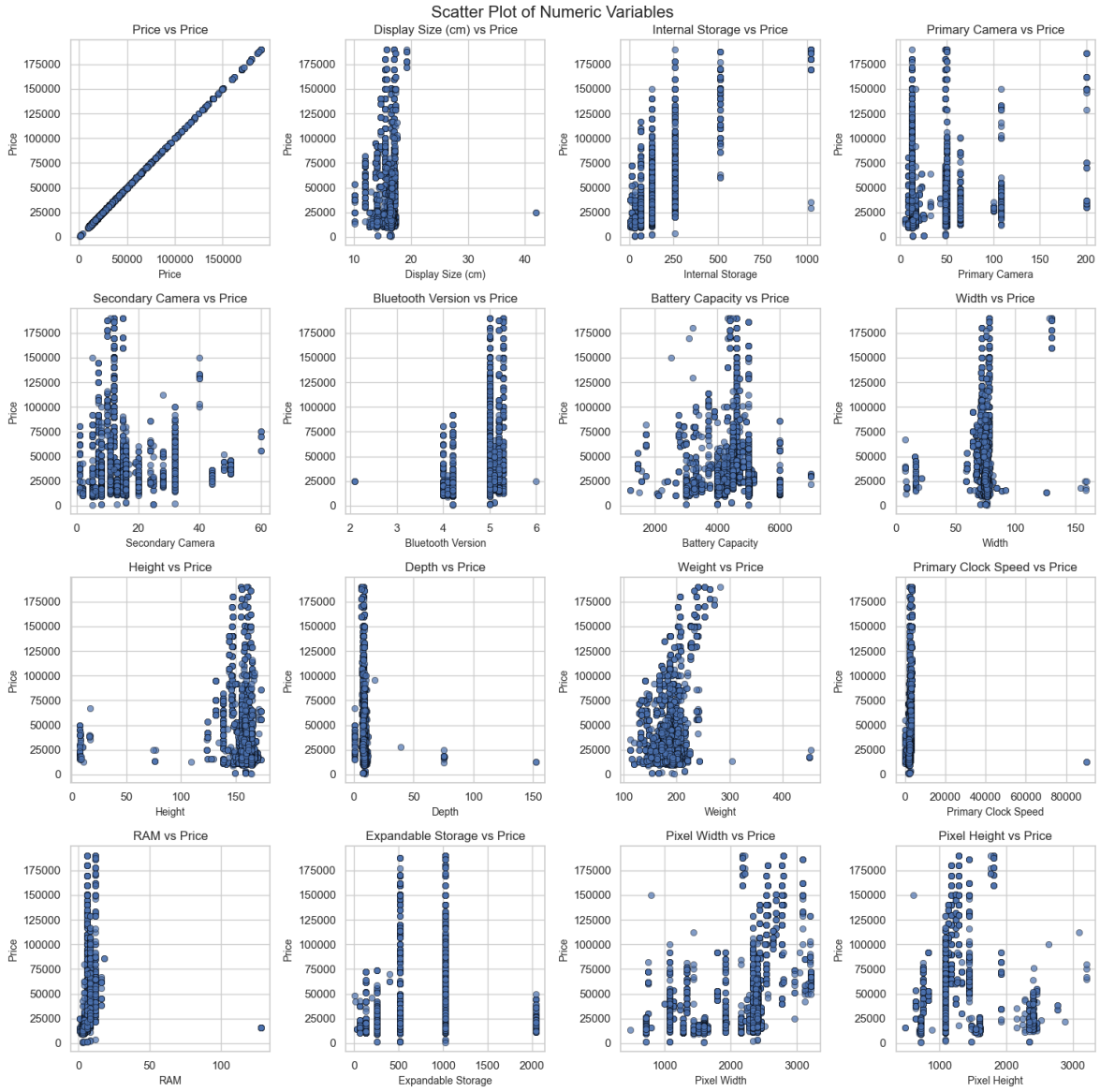
In [ ]:
```
df.hist(bins=20, color='teal', alpha=0.7, edgecolor='black', grid=False, layout=
plt.suptitle('Histograms of Numeric Variables', fontsize=16)
plt.tight_layout()
plt.show()
```



Inference: The histograms provide a visual representation of the distribution of numeric variables. It's evident that the data is not normally distributed and is skewed.

Creating scatter plots for numeric variables against the 'Price' column.

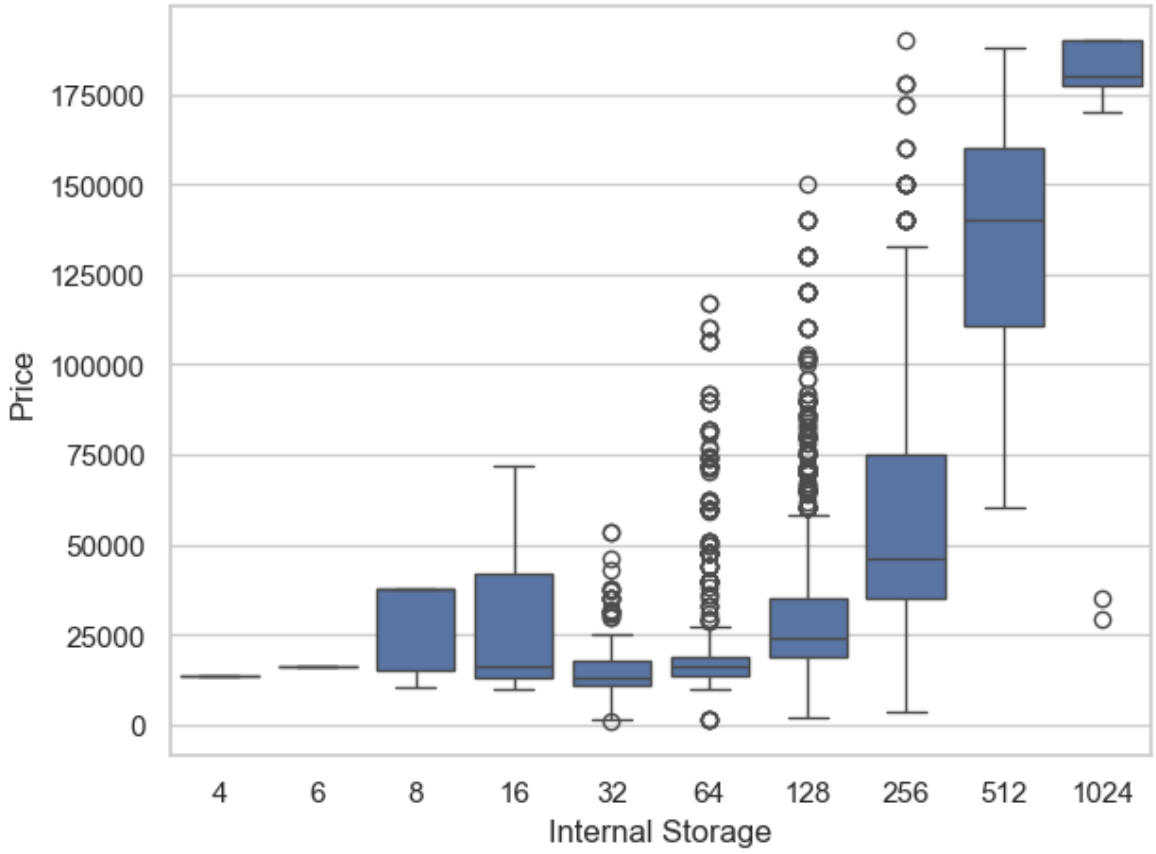
In [ ]:
```
nrows, ncols = 4, 4
fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(15, 15))

for i in range(nrows):
    for j in range(ncols):
        ax = axes[i, j]
        col = numeric.columns[i * ncols + j]
        sns.scatterplot(x=col, y='Price', data=df, ax=ax, alpha=0.7, edgecolor='
        ax.set_xlabel(col, fontsize=10)
        ax.set_ylabel('Price', fontsize=10)
        ax.set_title(f'{col} vs Price', fontsize=12)

plt.suptitle('Scatter Plot of Numeric Variables', fontsize=16)
plt.tight_layout()
plt.show()
```



Inference: These scatter plots show the relationship between numeric variables and the 'Price.' They reveal that the data is dispersed and not homoscedastic.

Generating a box plot for the 'Internal Storage' variable against the 'Price.'

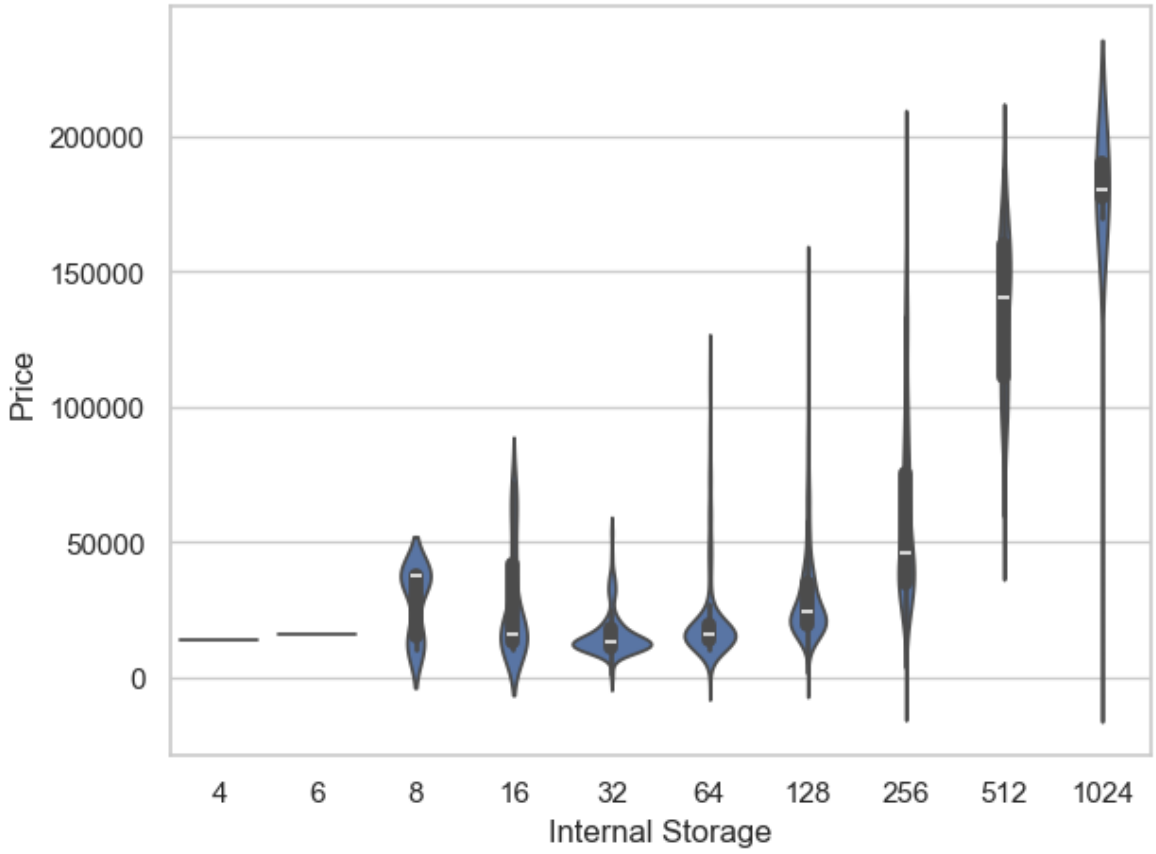
In [ ]:
```
sns.boxplot(x='Internal Storage', y='Price', data=df)
plt.tight_layout()
plt.show()
```



Inference: The box plot shows the distribution of 'Price' concerning 'Internal Storage.' It indicates that the data has many outliers.

Generating a violin plot for the 'Internal Storage' variable against the 'Price.'

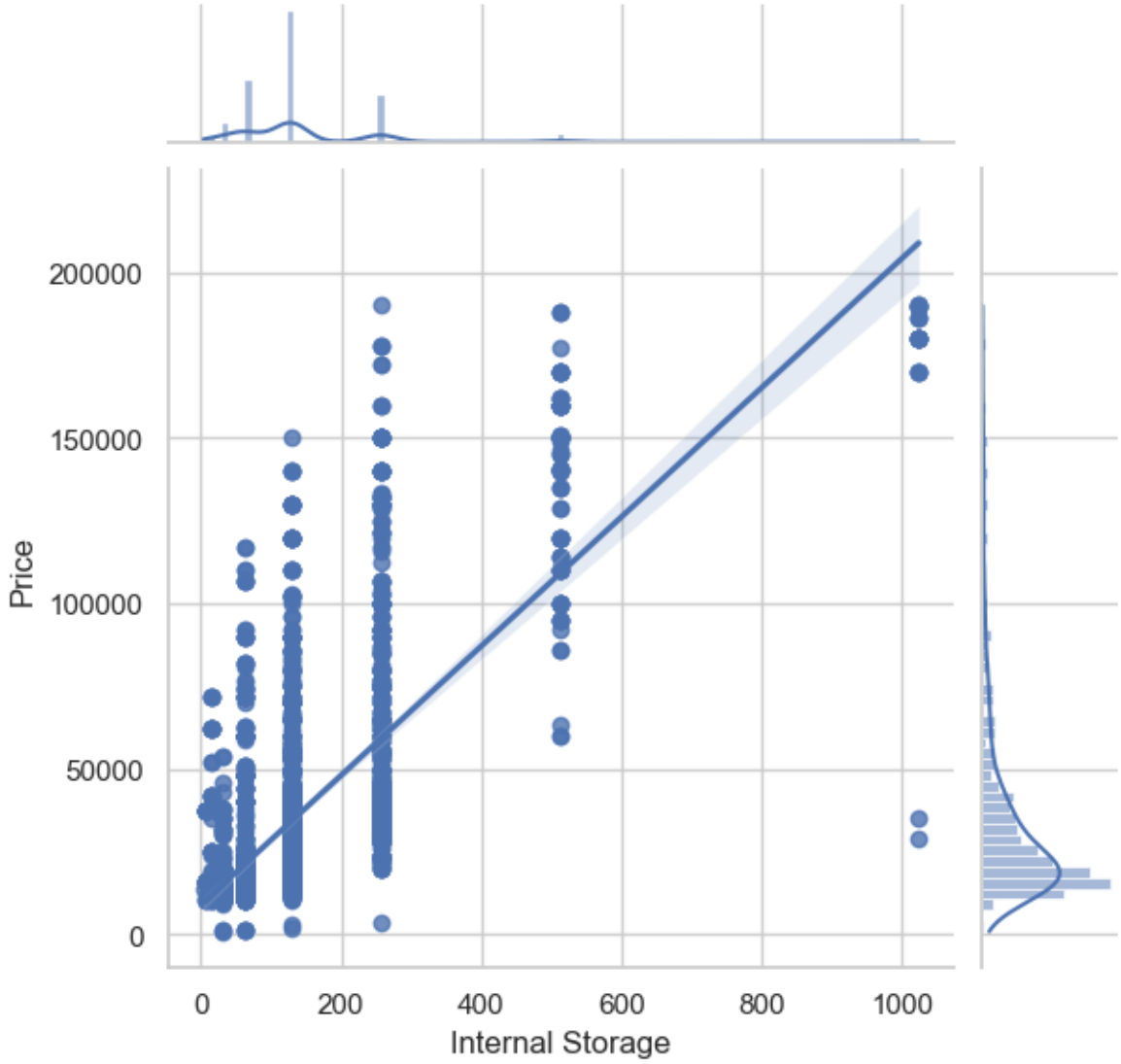
In [ ]:
```
sns.violinplot(x='Internal Storage', y='Price', data=df)
plt.tight_layout()
plt.show()
```



Inference: The violin plot for 'Internal Storage' vs. 'Price' provides information about the data's spread and density.

Creating a joint plot (scatter plot with regression line) for 'Internal Storage' vs. 'Price.'

In [ ]:
```
sns.jointplot(x='Internal Storage', y='Price', data=df, kind='reg')
plt.show()
```



Inference: The joint plot shows the relationship between 'Internal Storage' and 'Price' with a regression line.

Model Building

Preparing the data for model building by selecting specific numeric and categorical features.

In [ ]:
```
numeric_features = ['Price', 'RAM', 'Internal Storage', 'Battery Capacity']
categoric_features = ['Resolution Type', 'Processor Type', 'Processor Core']
features = numeric_features + categoric_features
df = df[features]
numeric = df.select_dtypes(include='number')
categoric = df.select_dtypes(include='category')
```

Applying a log transformation to the selected numeric features and saving the transformer to a file.

The log transformation is used to make the data more suitable for modeling, particularly when the data is skewed.

In [ ]:
```
from sklearn.preprocessing import FunctionTransformer

transform = FunctionTransformer(func=np.log1p)
transformed_df = transform.fit_transform(numeric)
for col in categoric.columns:
    transformed_df[col] = df[col]

transformed_df.head()
```

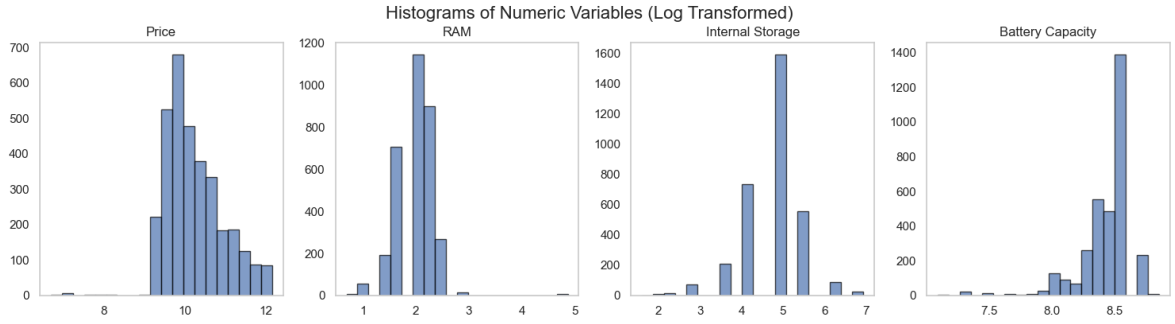
Out[ ]:

	Price	RAM	Internal Storage	Battery Capacity	Resolution Type	Processor Type	Processor Core
0	10.797553	1.94591	4.859812	8.042699	2	0	2
1	10.689692	1.94591	4.174387	8.042699	2	0	2
2	10.797553	1.94591	4.859812	8.042699	2	0	2
3	11.154835	1.94591	4.859812	8.083637	2	0	2
4	11.154835	1.94591	4.859812	8.083637	2	0	2

In [ ]:
```
with open('log_transform.pkl', 'wb') as f:
    pickle.dump(transform, f)
```

Generating histograms of log-transformed numeric variables.

In [ ]:
```
transformed_df.hist(bins=20, alpha=0.7, edgecolor='black', grid=False, layout=(4
plt.suptitle('Histograms of Numeric Variables (Log Transformed)', fontsize=16)
plt.tight_layout()
plt.show()
```



Inference: The histograms now display log-transformed data, which are more normally distributed compared to before.

Scaling the data using Min-Max scaling to bring features within a consistent range.

Scaling the data helps ensure that features are on the same scale, which is important for machine learning models.

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaled = scaler.fit_transform(transformed_df)
scaled_df = pd.DataFrame(scaled, columns=transformed_df.columns)
scaled_df.head()
```

Out[ ]:

	Price	RAM	Internal Storage	Battery Capacity	Resolution Type	Processor Type	Processor Core
0	0.751886	0.300663	0.610627	0.539900	0.285714	0.0	0.4
1	0.732168	0.300663	0.481861	0.539900	0.285714	0.0	0.4
2	0.751886	0.300663	0.610627	0.539900	0.285714	0.0	0.4
3	0.817200	0.300663	0.610627	0.563122	0.285714	0.0	0.4
4	0.817200	0.300663	0.610627	0.563122	0.285714	0.0	0.4

Saving the scaler to a file.

In [ ]:
```
with open('scaler.pkl', 'wb') as f:
    pickle.dump(scaler, f)
```

Splitting the data into training and testing sets for model building.

Preparing a DataFrame with covariates (independent variables) for model evaluation.

In [ ]:
```
from sklearn.model_selection import train_test_split

X = scaled_df.drop('Price', axis=1)
y = scaled_df['Price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

cols = features.copy()
cols.remove('Price')
covariates = pd.DataFrame(X_test, columns=cols)
```

Fitting a Linear Regression model to the training data and making predictions on the test data.

In [ ]:
```
from sklearn.linear_model import LinearRegression

lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
lr_pred = lr_model.predict(X_test)
```

Fitting a Gaussian Process model to the training data and making predictions on the test data.

In [ ]:
```
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, ConstantKernel
from sklearn.gaussian_process import GaussianProcessRegressor

kernel = ConstantKernel(1.0, (1e-3, 1e3)) * RBF(1.0, (1e-2, 1e2)) + WhiteKernel(
gpr_model = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)
gpr_model.fit(X_train, y_train)
gpr_pred = gpr_model.predict(X_test)
```

Defining a function to calculate Theil's U statistics for model evaluation.

In [ ]:
```
def theils_u(y_true, y_pred):
    n = len(y_true)
```

```
    U1 = np.sqrt(np.sum((y_true - y_pred) ** 2)) / np.sqrt(np.sum(y_true ** 2))
    U2 = np.sqrt(np.sum((y_true - y_pred) ** 2)) / np.sqrt(np.sum(y_pred ** 2))
    return U1, U2
```

Defining a function to calculate the Index of Agreement for model evaluation.

In [ ]:
```
def index_of_agreement(y_true, y_pred):
    numerator = np.sum((y_true - y_pred) ** 2)
    denominator = np.sum((np.abs(y_pred - np.mean(y_true)) + np.abs(y_true - np.
    return 1 - (numerator / denominator)
```

Defining a function to perform the Breusch-Pagan test for heteroscedasticity on model residuals.

In [ ]:
```
import statsmodels.api as sm
from statsmodels.stats.diagnostic import het_breuschpagan

def breusch_pagan_test(residuals, covariates):
    covariates = sm.add_constant(covariates)
    min_rows = min(len(residuals), len(covariates))
    residuals = residuals[:min_rows]
    covariates = covariates[:min_rows]
    lm, lm_p_value, fvalue, f_p_value = het_breuschpagan(residuals, covariates)
    return lm, lm_p_value, fvalue, f_p_value
```

Defining a function to perform the Durbin-Watson test for autocorrelation on model residuals.

In [ ]:
```
from statsmodels.stats.stattools import durbin_watson

def durbin_watson_test(residuals):
    dw_statistic = durbin_watson(residuals)
    return dw_statistic
```

Defining a function to plot the histogram of model residuals.

In [ ]:
```
def plot_residual_histogram(residuals, model_name):
    sns.set_style("white")
    plt.hist(residuals, bins=20, edgecolor='k')
    plt.xlabel('Residuals')
    plt.ylabel('Frequency')
    plt.title(f'Histogram of Residuals - {model_name.title()}')
    plt.show()
```

Defining a function to create scatter plots of residuals against covariates.

In [ ]:
```
def plot_residual_vs_covariates(residuals, covariates, model_name):
    num_covariates = len(covariates.columns)
    num_rows = int(num_covariates / 2) + (num_covariates % 2)
    num_cols = 2

    fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 10))
    fig.suptitle(f"{model_name.title()} - Residuals vs. Covariates", fontsize=16
    sns.set_style("white")
    for i, cov_name in enumerate(covariates.columns):
        row = i // num_cols
```

```
        col = i % num_cols
        ax = axes[row, col]
        covariate = covariates[cov_name]
        ax.scatter(covariate, residuals, alpha=0.5)
        ax.set_xlabel(cov_name)
        ax.set_ylabel("Residuals")

    for i in range(num_covariates, num_rows * num_cols):
        fig.delaxes(axes.flatten()[i])

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

Defining a function to create scatter plots of residuals against covariates.

These metrics include R-squared, MAE, MSE, MAPE, Theil's U statistics, and the Index of Agreement. They are used to evaluate model performance.

In [ ]:

```
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

def calculate_metrics(y_true, y_pred, covariates, model_name):
    # Calculate Residuals
    residuals = y_true - y_pred

    # Calculate R-squared
    r_squared = r2_score(y_true, y_pred)

    # Calculate Mean Absolute Error (MAE)
    mae = mean_absolute_error(y_true, y_pred)

    # Calculate Mean Squared Error (MSE)
    mse = mean_squared_error(y_true, y_pred)

    # Calculate Mean Absolute Percentage Error (MAPE)
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    # Calculate Normalized Root Mean Squared Error (nRMSE)
    nrmse = np.sqrt(mse) / (max(y_true) - min(y_true))

    # Calculate Theil's U1 and U2
    u1, u2 = theils_u(y_true, y_pred)

    # Calculate Index of Agreement
    ioa = index_of_agreement(y_true, y_pred)

    # Create a Pandas DataFrame to store the metrics
    metrics_df = pd.DataFrame({
        'Model': [model_name],
        'R-squared': [r_squared],
        'MAE': [mae],
        'MSE': [mse],
        'MAPE': [mape],
        "Theil's U1": [u1],
        "Theil's U2": [u2],
        'Index of Agreement': [ioa]
    })

    # Durbin-Watson test (Test for autocorrelation among residuals)
    dw_stat = durbin_watson_test(residuals)
```

```
    metrics_df['Durbin-Watson Statistic'] = [dw_stat]

    # Calculate Test for heteroscedasticity (Breusch-Pagan Test) - p-value
    lm, lm_p_value, fvalue, f_p_value = breusch_pagan_test(residuals, covariates

    if lm_p_value < 0.05:
        metrics_df['Heteroscedasticity'] = ["Yes"]
    else:
        metrics_df['Heteroscedasticity'] = ["No"]

    metrics_df['Breusch-Pagan LM Statistic'] = [lm]
    metrics_df['Breusch-Pagan LM P-Value'] = [lm_p_value]
    metrics_df['Breusch-Pagan F-Stat'] = [fvalue]
    metrics_df['Breusch-Pagan F P-Value'] = [f_p_value]

    return metrics_df
```

Applying the Linear Regression model to the test data and calculating various metrics.

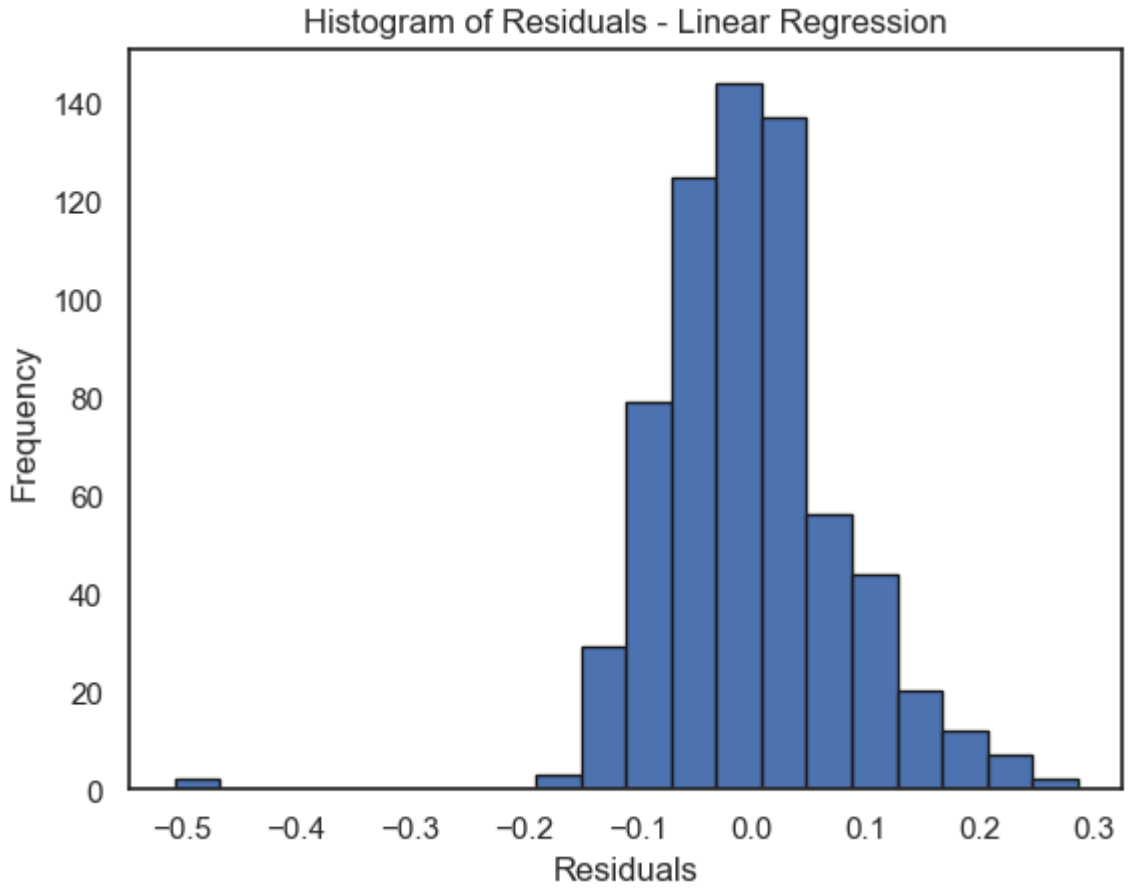
In [ ]:
```
residuals = y_test - lr_pred
model_name = 'Linear Regression'
lr_metrics = calculate_metrics(y_test, lr_pred, covariates, model_name)
lr_metrics.T
```

Out[ ]:

	0
Model	Linear Regression
R-squared	0.632301
MAE	0.059899
MSE	0.006383
MAPE	10.685678
Theil's U1	0.119332
Theil's U2	0.119961
Index of Agreement	0.874461
Durbin-Watson Statistic	2.137027
Heteroscedasticity	Yes
Breusch-Pagan LM Statistic	14.896712
Breusch-Pagan LM P-Value	0.021075
Breusch-Pagan F-Stat	2.513177
Breusch-Pagan F P-Value	0.020624

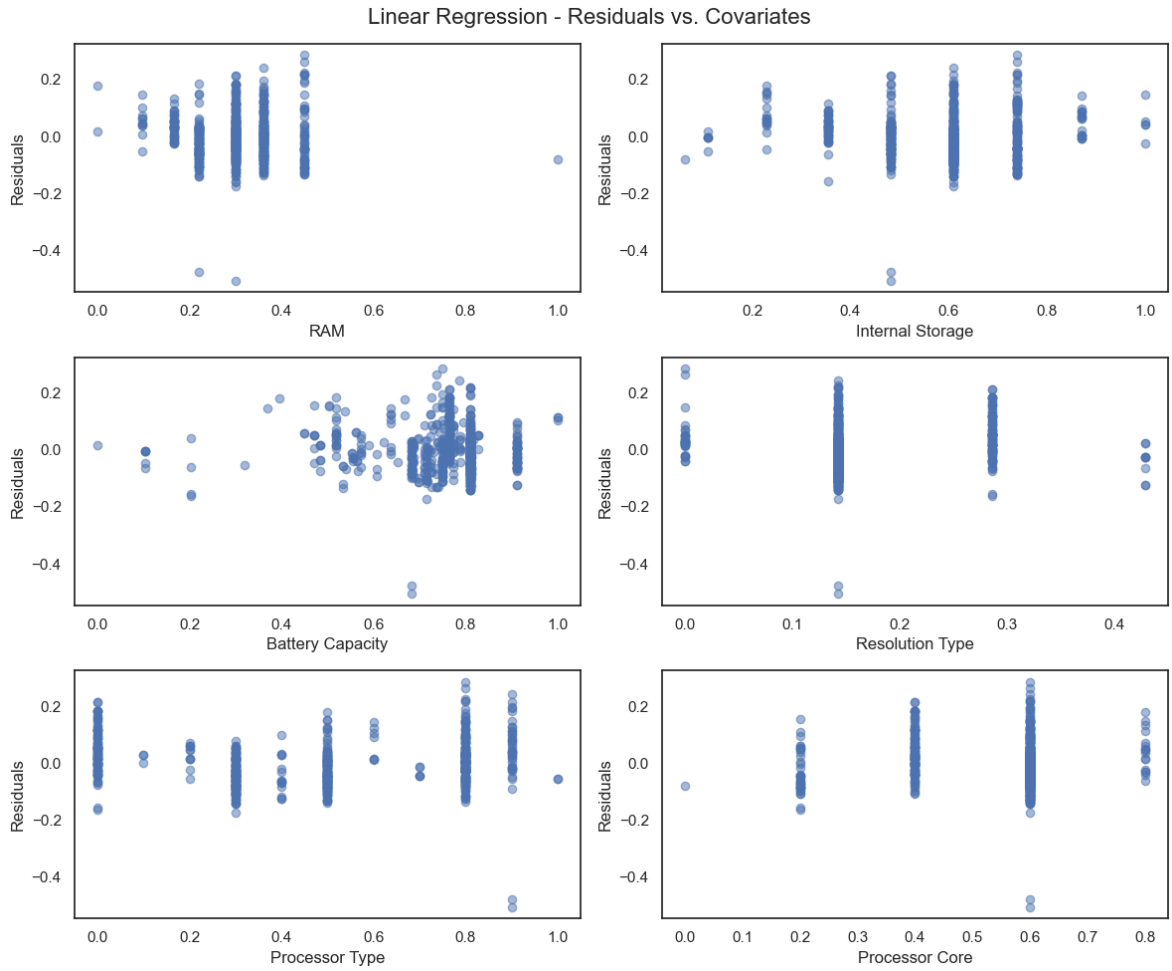
Plotting the histogram of residuals for the Linear Regression model.

In [ ]:
```
plot_residual_histogram(residuals, model_name)
```



Creating scatter plots of residuals against covariates for the Linear Regression model.

In [ ]:
```
plot_residual_vs_covariates(residuals, covariates, model_name)
```



Applying the Gaussian Process model to the test data and calculating various metrics.

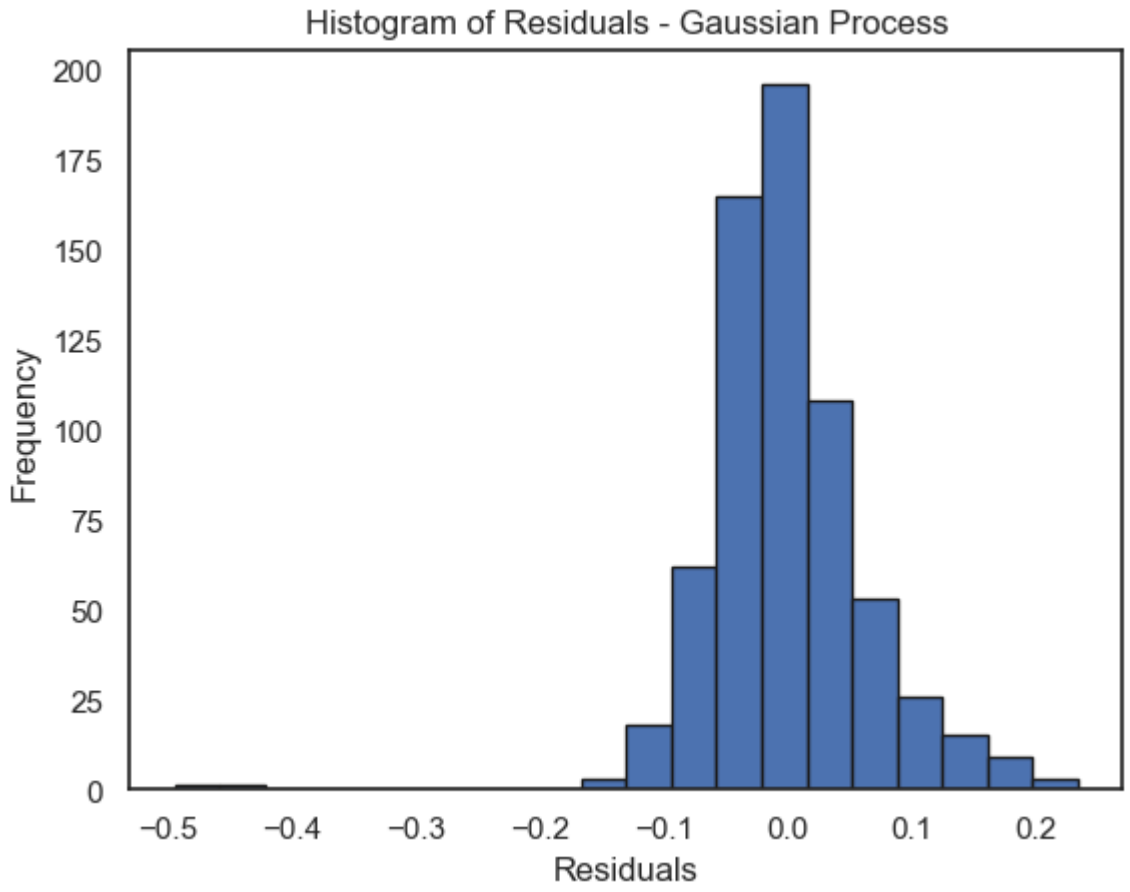
In [ ]:
```
residuals = y_test - gpr_pred
model_name = 'Gaussian Process'
gpr_metrics = calculate_metrics(y_test, gpr_pred, covariates, model_name)
gpr_metrics.T
```

Out[ ]:

	0
Model	Gaussian Process
R-squared	0.763694
MAE	0.045021
MSE	0.004102
MAPE	8.262112
Theil's U1	0.095664
Theil's U2	0.09595
Index of Agreement	0.928707
Durbin-Watson Statistic	2.157801
Heteroscedasticity	Yes
Breusch-Pagan LM Statistic	26.242252
Breusch-Pagan LM P-Value	0.000201
Breusch-Pagan F-Stat	4.506504
Breusch-Pagan F P-Value	0.000174

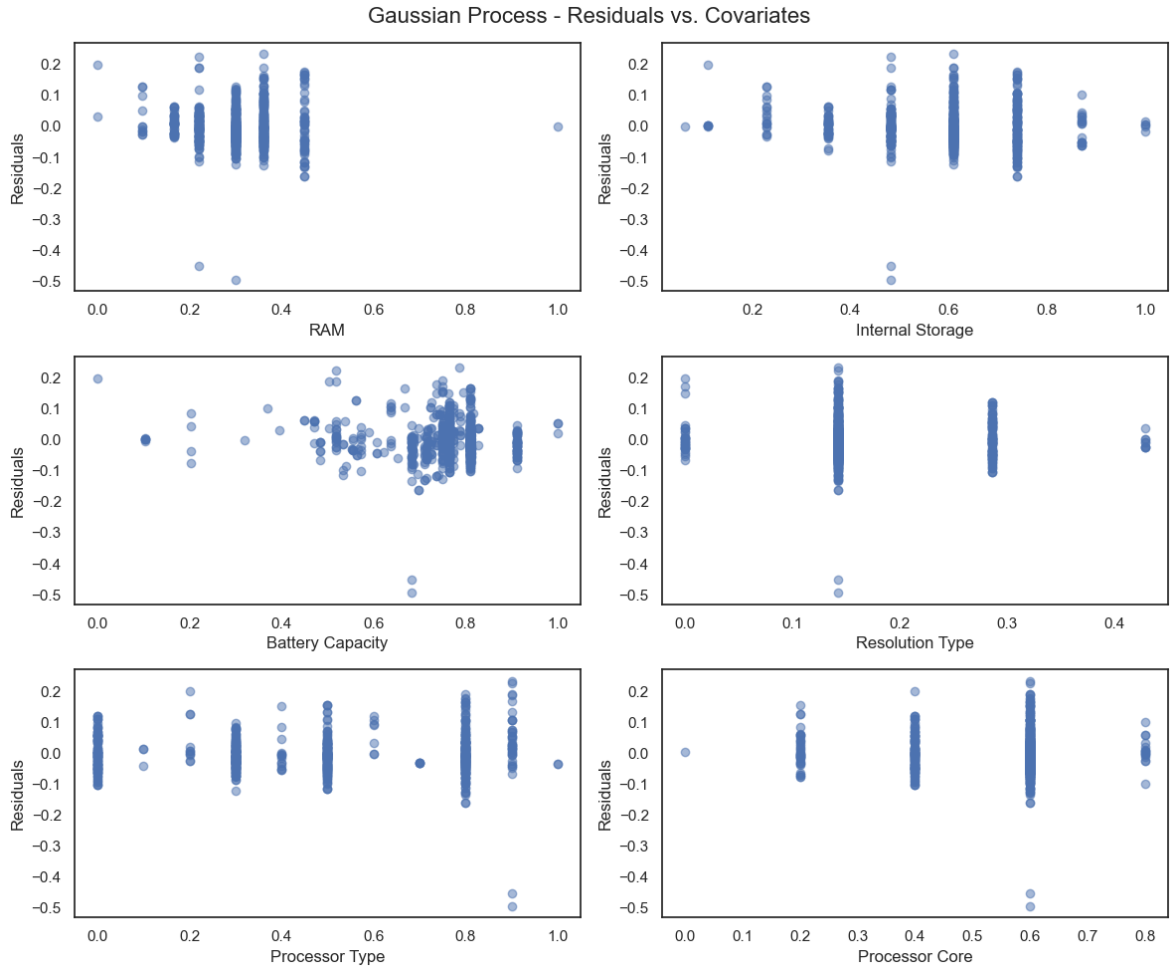
Plotting the histogram of residuals for the Gaussian Process model.

In [ ]:
```
plot_residual_histogram(residuals, model_name)
```



Creating scatter plots of residuals against covariates for the Gaussian Process model.

In [ ]:
```
plot_residual_vs_covariates(residuals, covariates, model_name)
```



Combining the evaluation metrics for both models into a single DataFrame.

The DataFrame provides a summary of model evaluation metrics for the Linear Regression and Gaussian Process models.

In [ ]:
```
combined_metrics = pd.concat([lr_metrics, gpr_metrics], ignore_index=True)
combined_metrics
```

Out[ ]:

	Model	R-squared	MAE	MSE	MAPE	Theil's U1	Theil's U2	Index of Agreement
0	Linear Regression	0.632301	0.059899	0.006383	10.685678	0.119332	0.119961	0.874461
1	Gaussian Process	0.763694	0.045021	0.004102	8.262112	0.095664	0.095950	0.928707

Saving the Gaussian Process model to a file for model deployment.

In [ ]:
```
with open('model.pkl', 'wb') as f:
```

```
    pickle.dump(gpr_model, f)
```