ReCell Data Analysis

Python Foundations : PGP-DSBA

Feb 24,2023

Contents / Agenda

- Background
- Business Problem Overview
- Solution Approach
- EDA Results
- Data Preprocessing
- Model Performance Summary
- Executive Summary
- Recommendations

Background

- ReCell is a company that specializes in refurbishing smartphones. Refurbished smartphones are devices that have been previously used but have been restored to a like-new condition through a process of testing, repair, and cleaning.

- ReCell's refurbishment process typically involves thoroughly inspecting each device to identify any defects or malfunctions, repairing or replacing any faulty components, and cleaning and sanitizing the device. They may also replace the battery if necessary to ensure optimal performance.

Business Problem Overview

- The problem that ReCell, a startup that deals in refurbished smartphones and tablets, is facing is the need for a dynamic pricing strategy for used and refurbished devices. The used and refurbished device market is growing rapidly, and there is a significant potential for growth in the future. However, ReCell needs to determine the optimal pricing for these devices to remain competitive in the market and achieve their business objectives.

Solution Approach

- As a data scientist we are going to approach to the solution for this case by building a linear regression model using machine learning techniques to precit the price of a used phone/tablet and identify the factors that significantly influence it.

EDA Results – Data Overview

```
In [5]: data.info()
```

```
<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)
memory usage: 404.9+ KB
```

- brand_name , os , 4g , and 5g are categorical data types, while the others are numeric data types.
- normalized_used_price is the dependent variable.

```
In [8]: data.isnull().sum().sort_values(ascending= False)

Out[8]: main_camera_mp           179
        weight                     7
        battery                    6
        int_memory                 4
        ram                        4
        selfie_camera_mp           2
        brand_name                 0
        os                         0
        screen_size                0
        4g                         0
        5g                         0
        release_year               0
        days_used                  0
        normalized_used_price      0
        normalized_new_price       0
        dtype: int64
```

- main_camera_mp , selfie_camera_mp , int_memory , ram , battery , and weight contain missing values.
- main_camera_mp contains the most missing values which is 179.

```
In [6]: data.describe(include='all').T ## Complete the code to print the statistical summary of the data
```

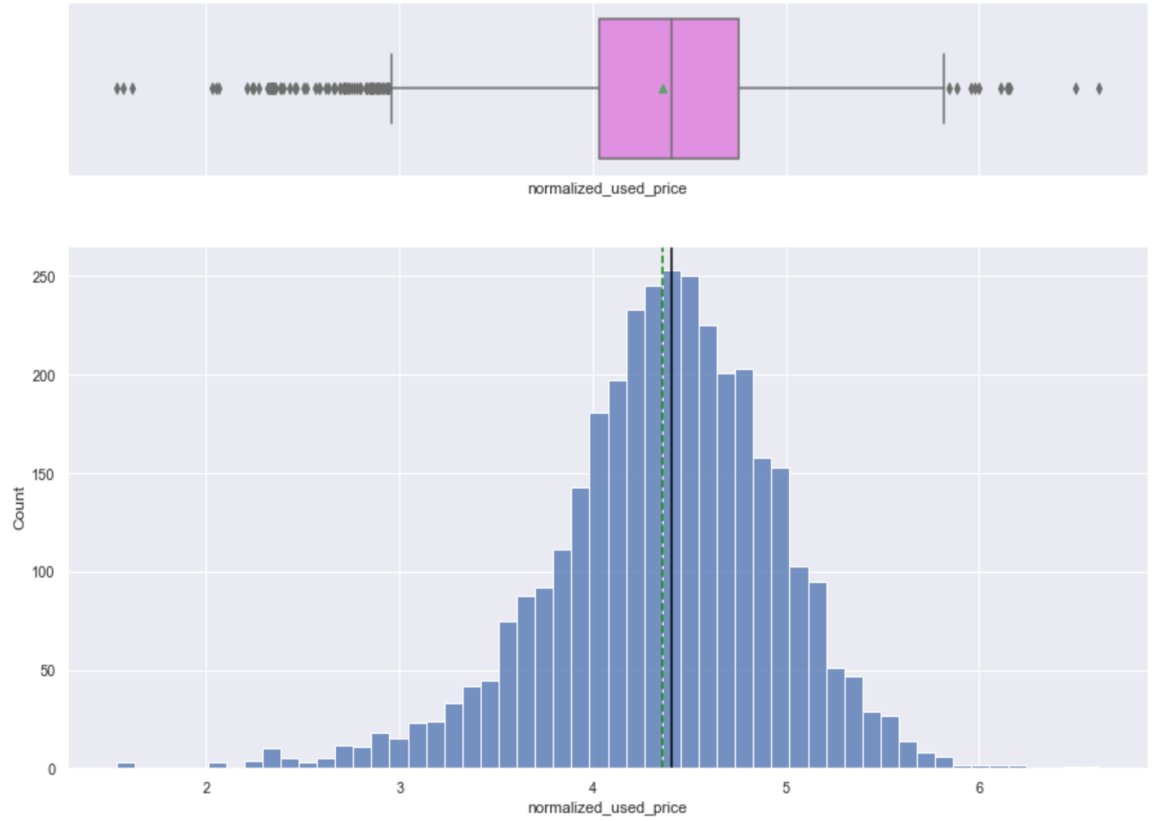
Out[6]:

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
brand_name	3454	34	Others	502	NaN	NaN	NaN	NaN	NaN	NaN	NaN
os	3454	4	Android	3214	NaN	NaN	NaN	NaN	NaN	NaN	NaN
screen_size	3454.0	NaN	NaN	NaN	13.713115	3.80528	5.08	12.7	12.83	15.34	30.71
4g	3454	2	yes	2335	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5g	3454	2	no	3302	NaN	NaN	NaN	NaN	NaN	NaN	NaN
main_camera_mp	3275.0	NaN	NaN	NaN	9.460208	4.815461	0.08	5.0	8.0	13.0	48.0
selfie_camera_mp	3452.0	NaN	NaN	NaN	6.554229	6.970372	0.0	2.0	5.0	8.0	32.0
int_memory	3450.0	NaN	NaN	NaN	54.573099	84.972371	0.01	16.0	32.0	64.0	1024.0
ram	3450.0	NaN	NaN	NaN	4.036122	1.365105	0.02	4.0	4.0	4.0	12.0
battery	3448.0	NaN	NaN	NaN	3133.402697	1299.682844	500.0	2100.0	3000.0	4000.0	9720.0
weight	3447.0	NaN	NaN	NaN	182.751871	88.413228	69.0	142.0	160.0	185.0	855.0
release_year	3454.0	NaN	NaN	NaN	2015.965258	2.298455	2013.0	2014.0	2015.5	2018.0	2020.0
days_used	3454.0	NaN	NaN	NaN	674.869716	248.580166	91.0	533.5	690.5	868.75	1094.0
normalized_used_price	3454.0	NaN	NaN	NaN	4.364712	0.588914	1.536867	4.033931	4.405133	4.7557	6.619433
normalized_new_price	3454.0	NaN	NaN	NaN	5.233107	0.683637	2.901422	4.790342	5.245892	5.673718	7.847841

- There are 34 unique manufacturing brands, four distinct operating systems, and 4g or 5g values of either yes or no for various phone models. 'Android' is the most common operating system, with 3,214 mobile devices using it.
- There are 4 operating systems.
- Only 152 phones support 5G, compared to 2335 that support 4G, with 967 phones not supporting either 4G or 5G.
- There is a maximum screen size of 30cm, indicating that it is obviously a tablet, and a minimum screen size of 5cm, indicating that low-end cell phones or "no smartphones" are possible.
- In the main_camera_mp variable, the average and median values are close.

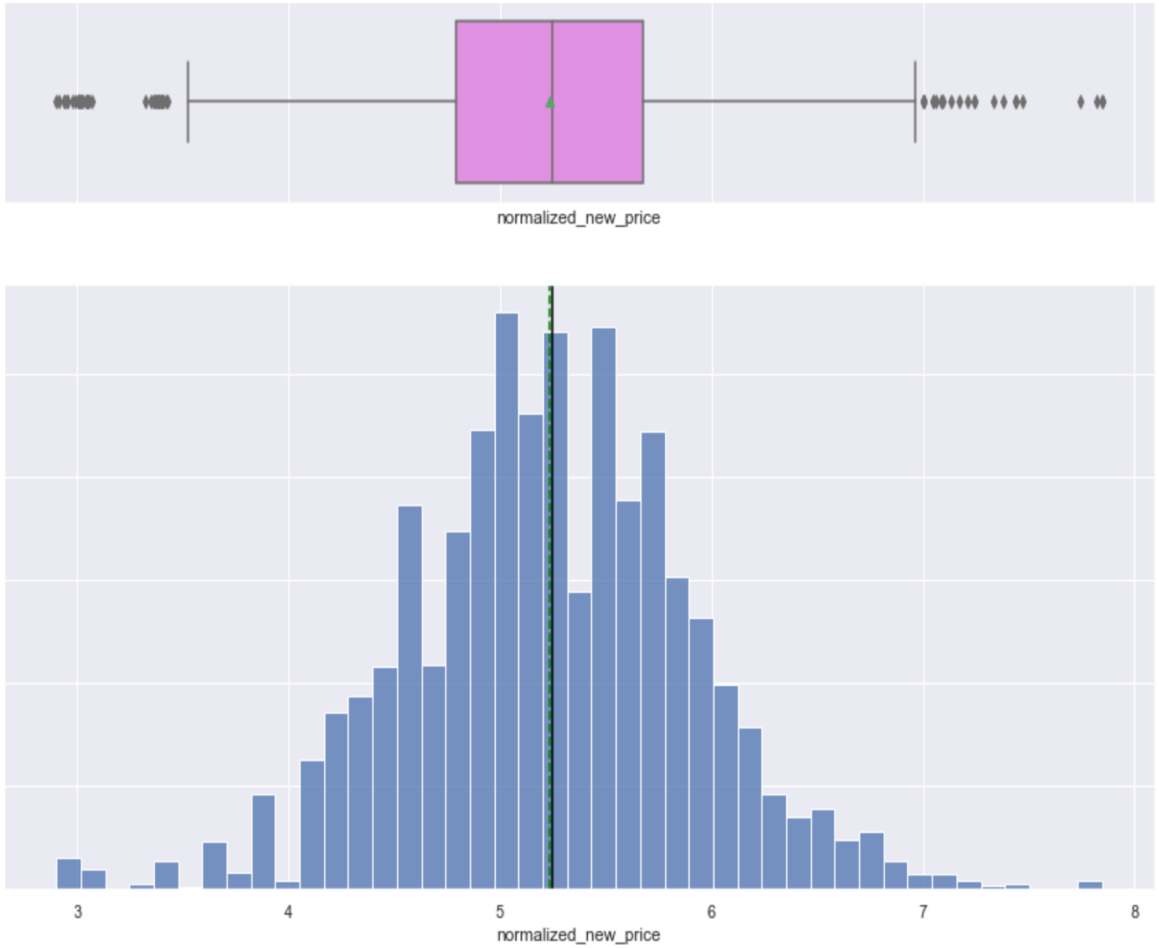
EDA Results – Univariate Analysis

```
histogram_boxplot(df, "normalized_used_price")
```



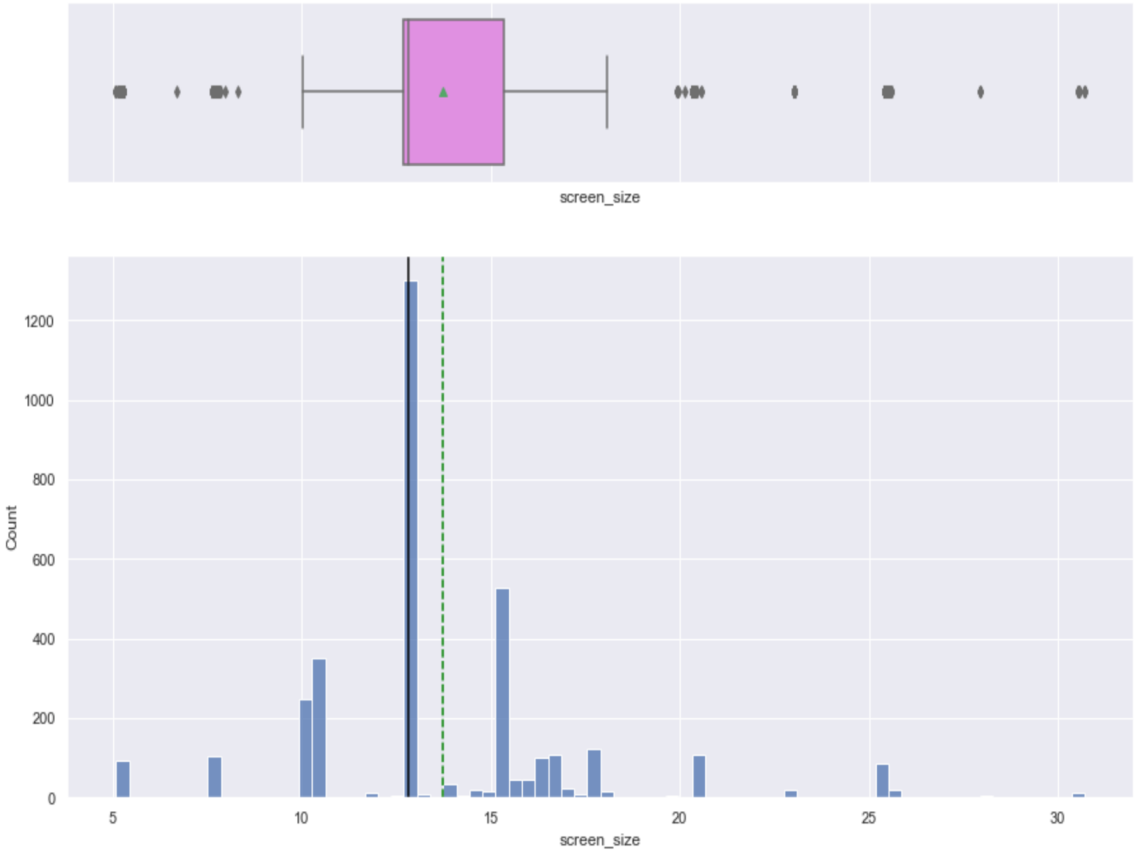
- The normalized_used_price is close to normally distributed.
- The boxplot shows that there are outliers.

```
histogram_boxplot(df,'normalized_new_price')  ## Complete the code to create histogram_boxplot for 'normal
```



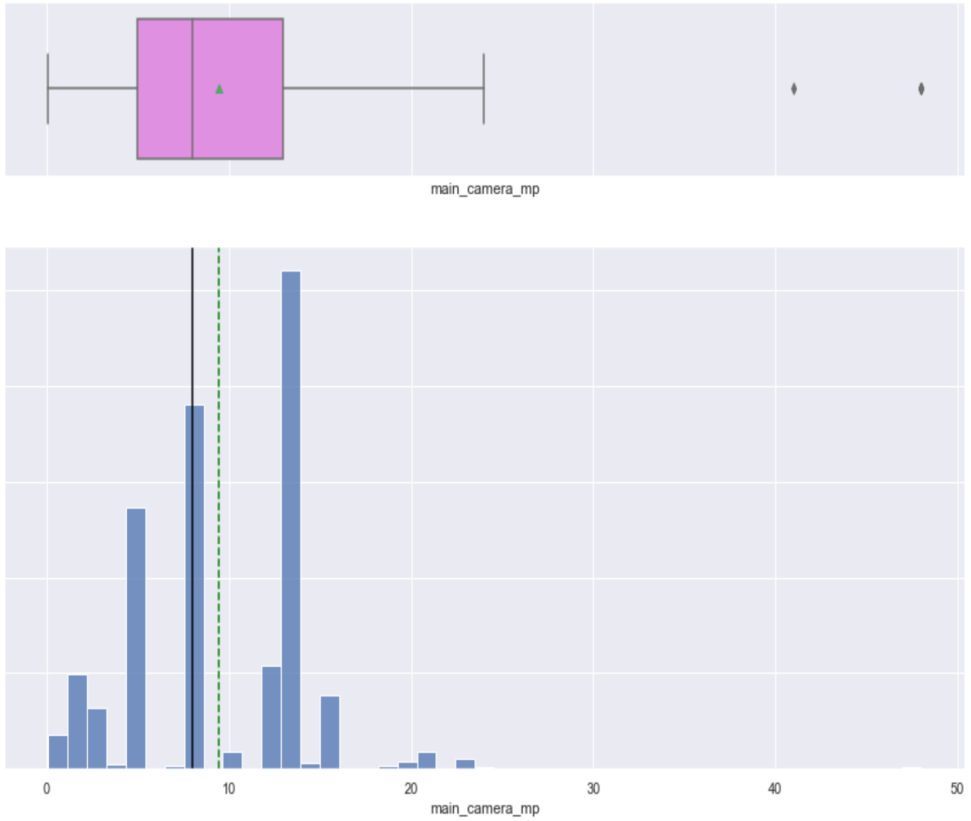
- The normalized_new_price is close to normally distributed.

```
histogram_boxplot(df,'screen_size')  ## Complete the code to create histogram_boxplot for 'screen_size'
```



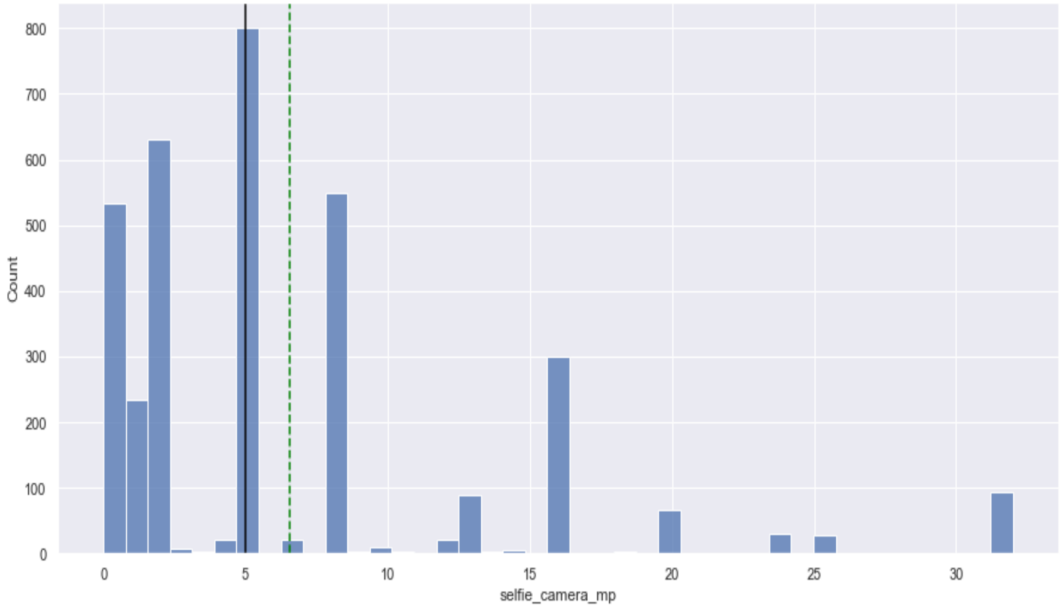
- The screen_size appears to be normally distributed; reducing the number of bins would allow us to better visualize the normal distribution.
- There are outliers above 20cm, indicating Tablets in the data set, but there are also outliers as low as 5cm, which is intriguing.

```
histogram_boxplot(df,'main_camera_mp')  ## Complete the code to create histogram_boxplot for 'main_camera_
```



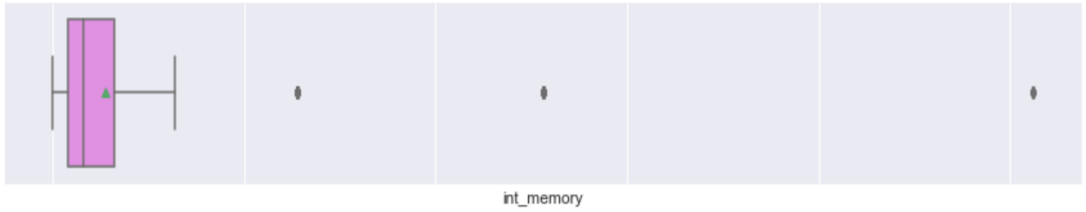
- The main_camera_mp is kind of normally distributed.
- There are outliers in the boxplot.

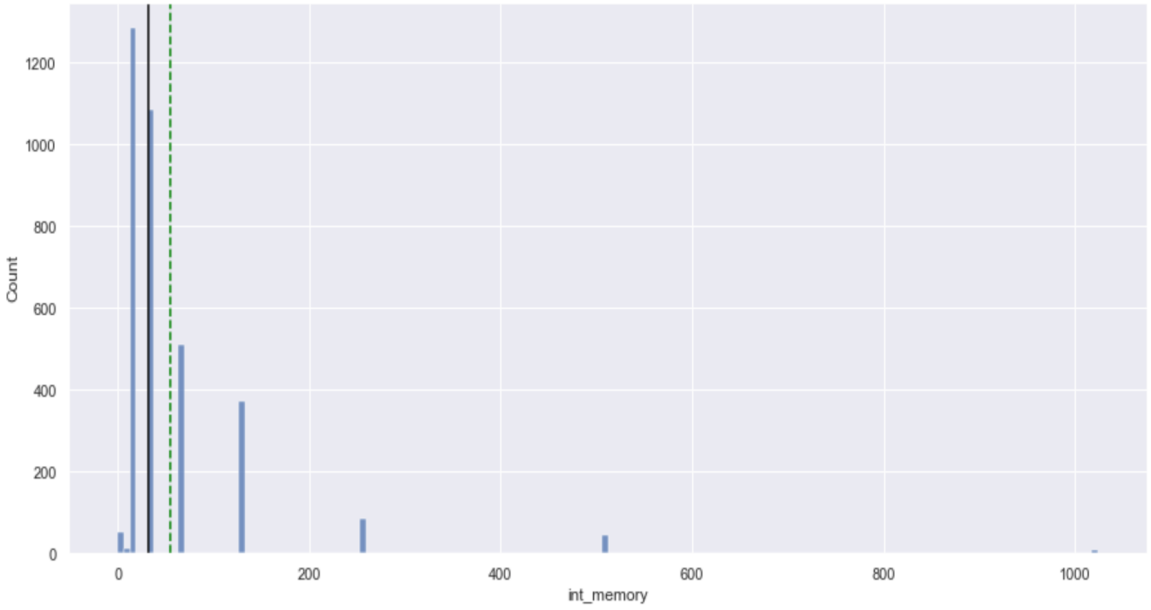
```
histogram_boxplot(df, 'selfie_camera_mp')  ## Complete the code to create histogram_boxplot for 'selfie_ca
```



- The selfie_camera_mp distribution is right skewed, with outliers above 20mp, which is unusual for selfie cameras.

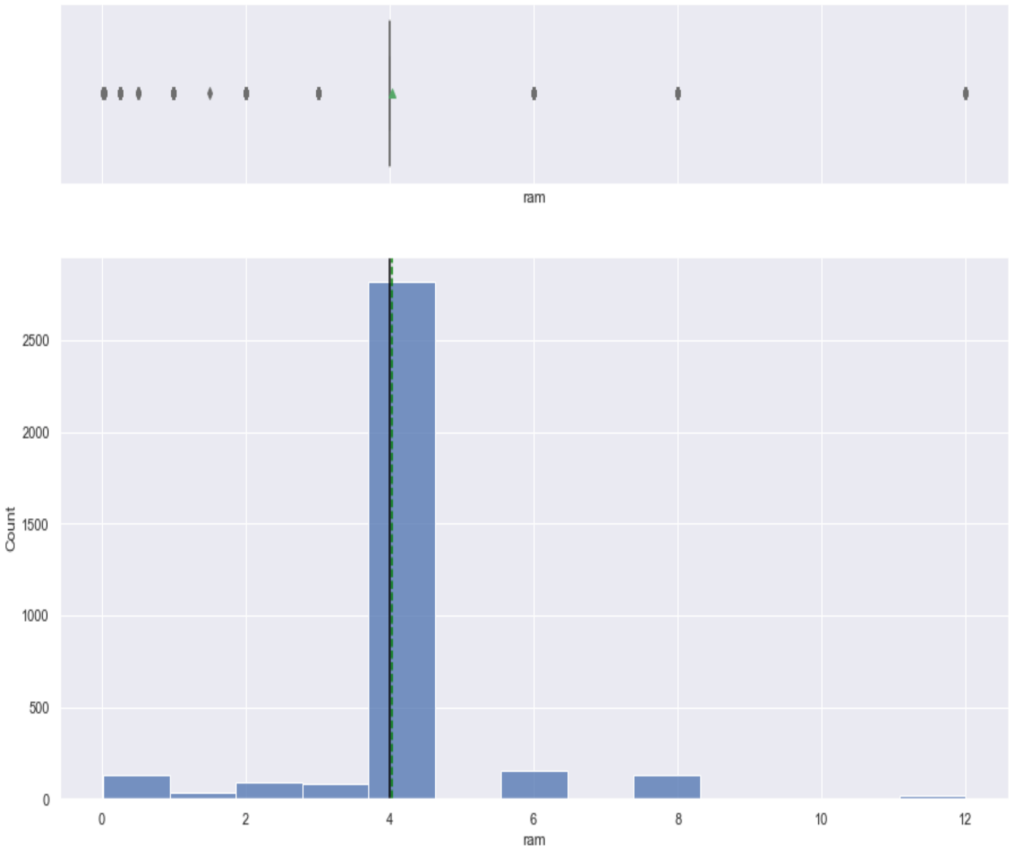
```
histogram_boxplot(df, 'int_memory')  ## Complete the code to create histogram_boxplot for 'int_memory'
```





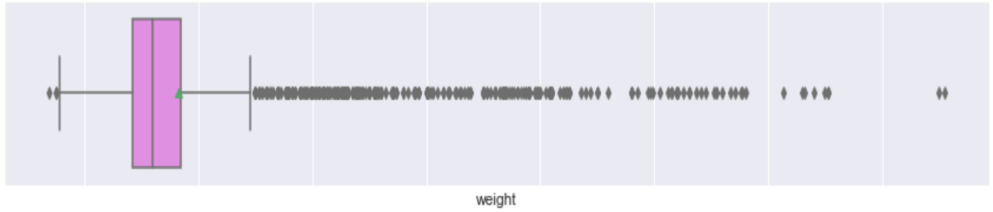
- The int_memory is right skewed
- There are some outliers in the box plot

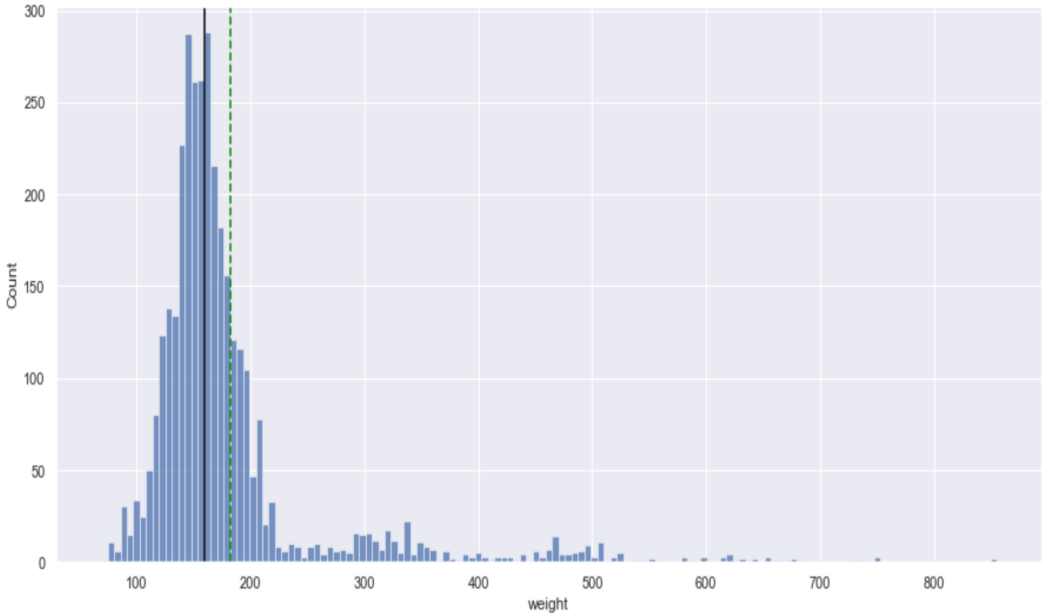
```
histogram_boxplot(df, 'ram')  ## Complete the code to create histogram_boxplot for 'ram'
```



- Ram looks normally distributed

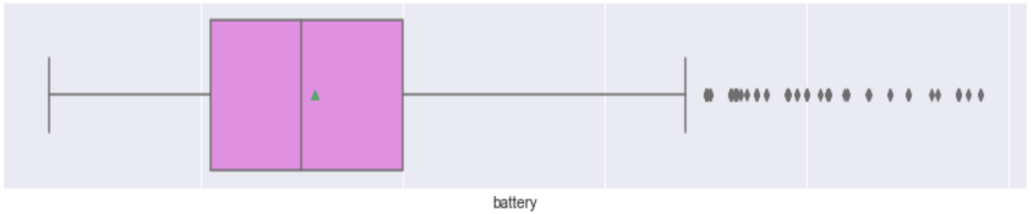
```
histogram_boxplot(df, "weight")  ## Complete the code to create histogram_boxplot for 'weight'
```

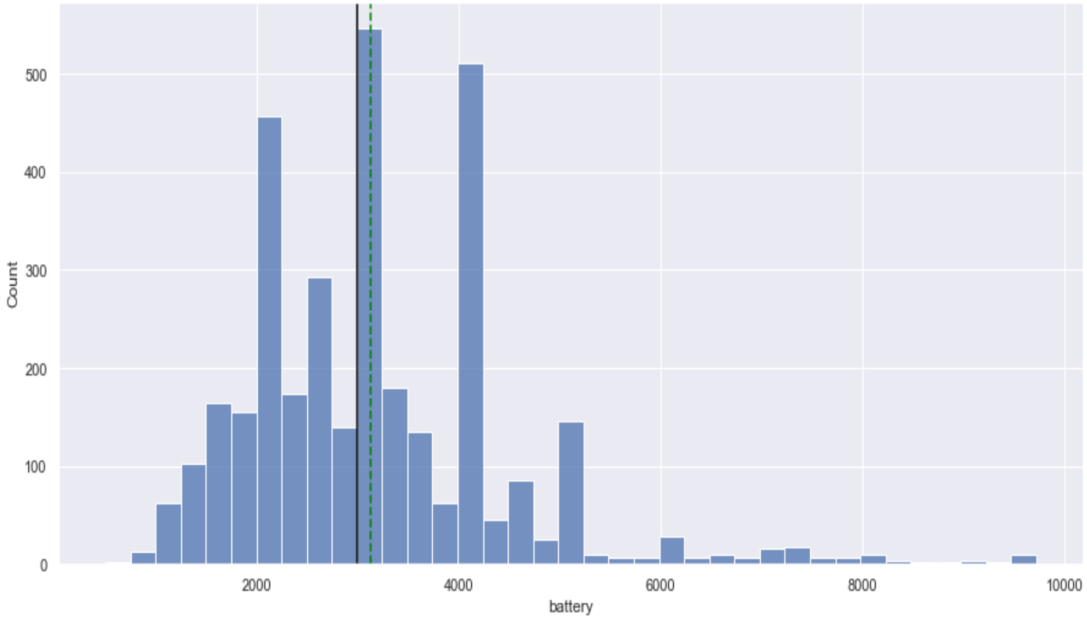




- Weight looks highly right skewed to the right.

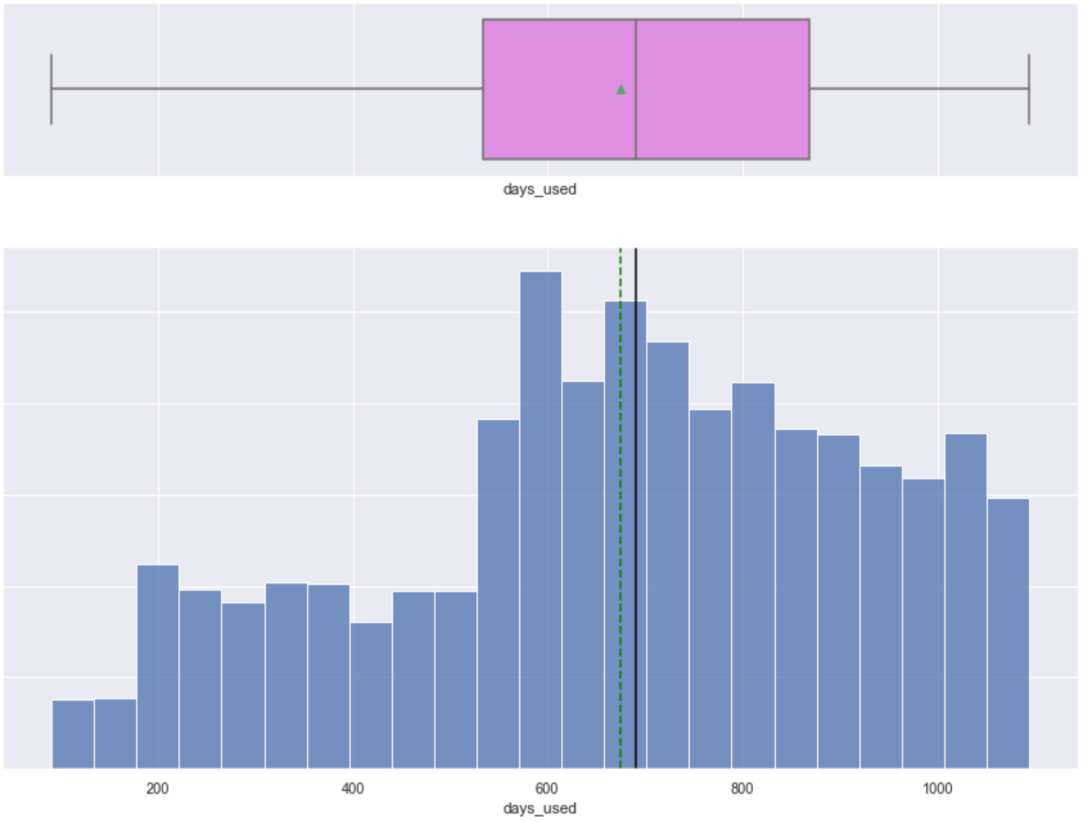
```
histogram_boxplot(df,'battery')  ## Complete the code to create histogram_boxplot for 'battery'
```





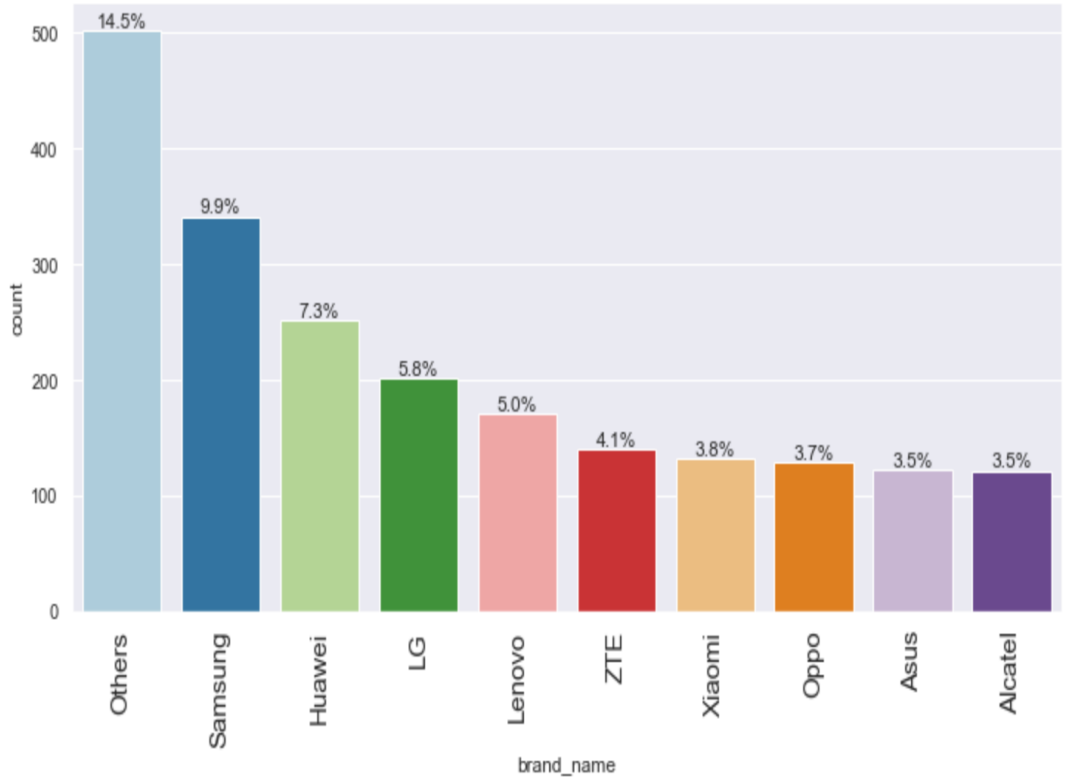
- There is a right skewness in battery.

```
histogram_boxplot(df, 'days_used')  ## Complete the code to create histogram_boxplot for 'days_used'
```



- Days_used is aproximately normally distributed.

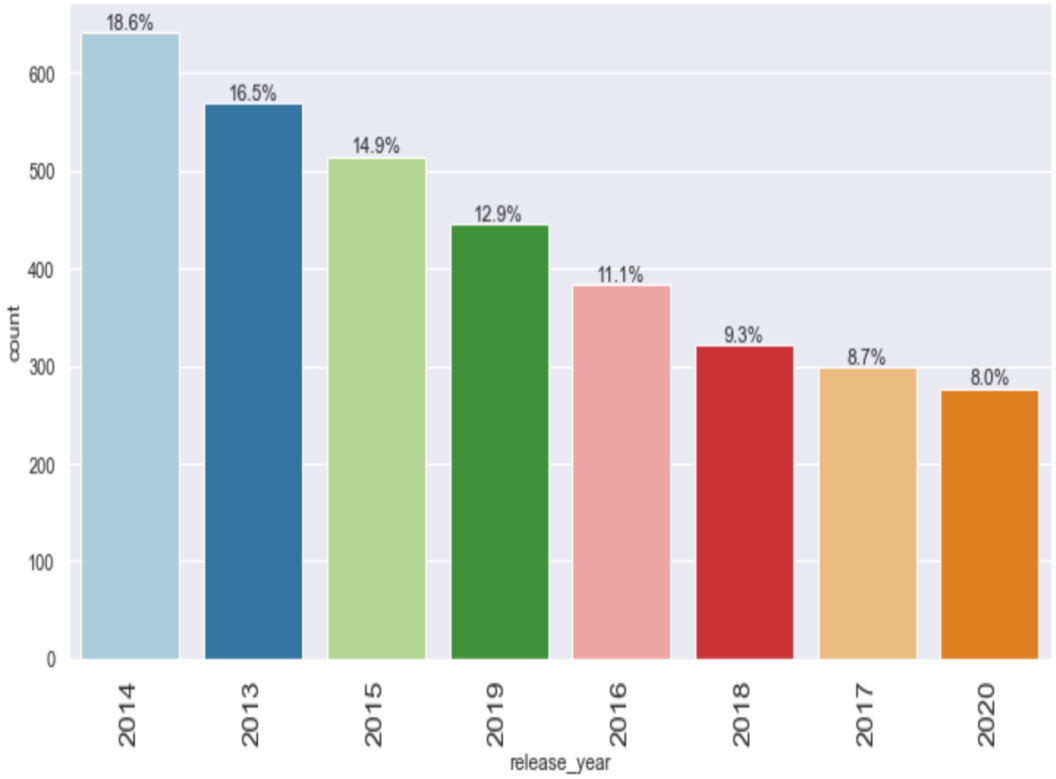
```
In [23]: labeled_barplot(df, "brand_name", perc=True, n=10)
```



- Because the majority of brand names were not provided, they are categorized as "others."

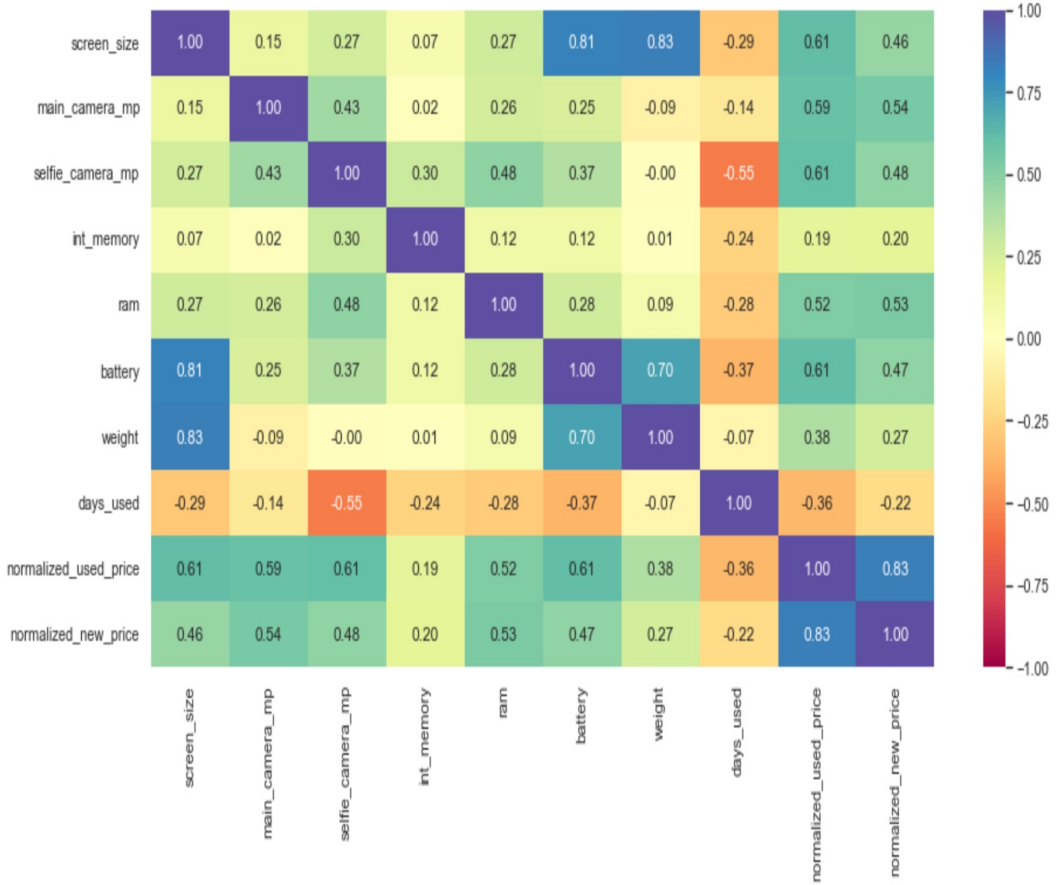
- Samsung has a higher proportion when compared to other companies. Oneplus focuses on mid to high end smartphones and is a relatively new brand; this is because its RAM average is higher; brands with more time in the market, such as Samsung, Huawei, LG, and ZTE, have "not smartphones" devices on the refurbished market; of course, they are also renowned brands, which can influence purchase bias.

```
In [27]: labeled_barplot(df, "release_year", perc=True, n=10) ## Complete the code to create lab
```



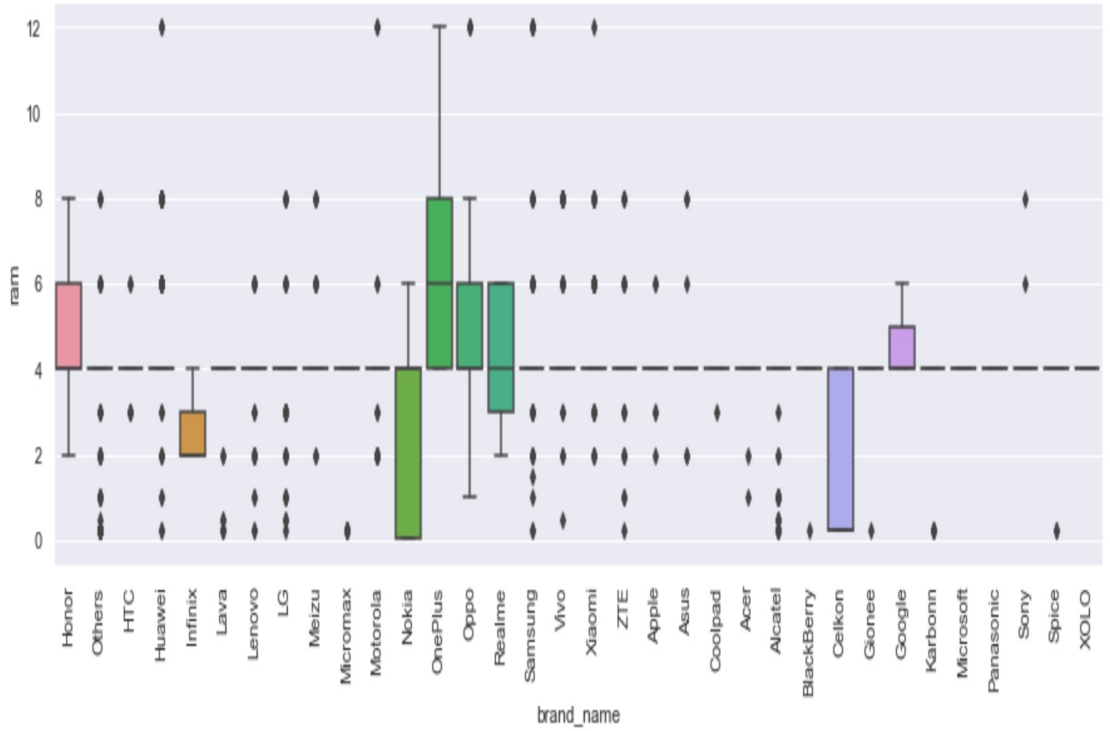
- The most refurbished devices were those released in 2014.

EDA results – Bivariate Analysis



- battery and screen_size have a strong correlation, also battery and weight also have a strong correlation, this makes sense as devices with larger screens tend to have larger batteries and thus considerably more weight .
- normalized_new_price and normalized_used_price show strong correlation as well.
- There is a negative correlation between selfie camera and days used, as well as a negative correlation between days used and normalized used price. For this reason, I decided to include release year in the graph, and it can be seen that the highest negative correlation is with release year, which leads us to believe that more the days of usage, the older the cell phone gets, and thus it has lower performance and features like screen size, RAM, battery, and internal storage.

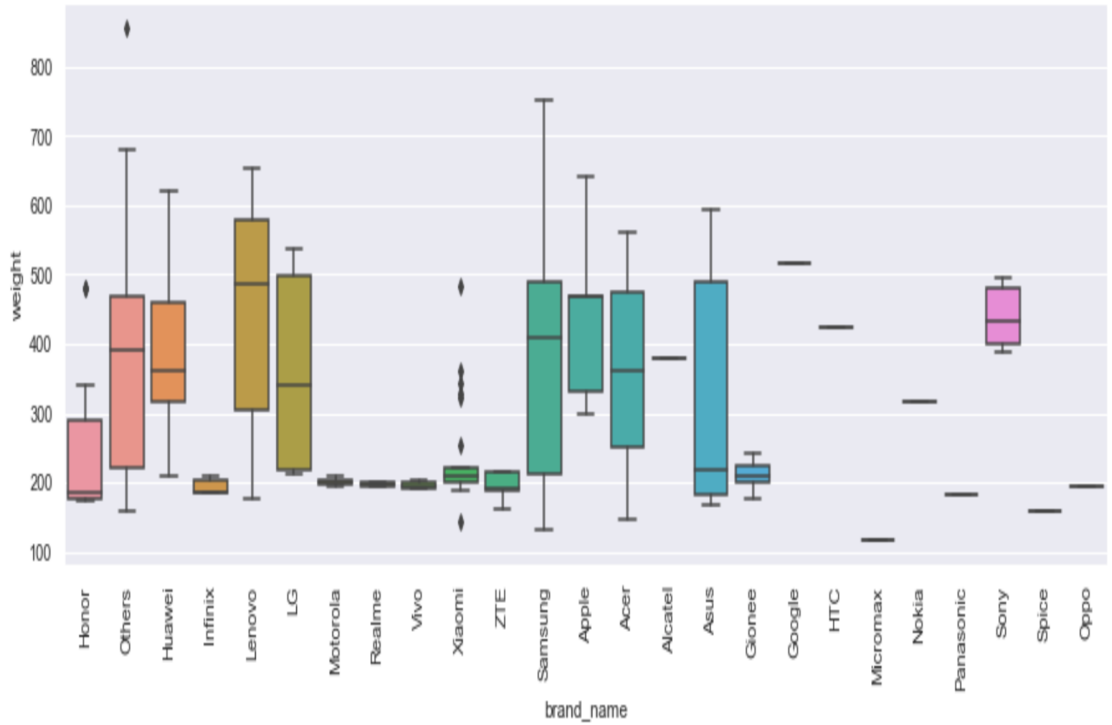
```
plt.figure(figsize=(15, 5))
sns.boxplot(data=df, x="brand_name", y="ram")
plt.xticks(rotation=90)
plt.show()
```



- In the box plot, a wide range of brands show minimum and maximum values of 4Gb, with clear outliers at 6Gb, 8Gb, and 12Gb. This makes sense because these specifications can be found in high-end cell phones or tablets. However, there are outliers in the 2Gb and 3Gb ranges, and for values closer to 0, they are likely simpler models (not smartphones), and possibly part of the same ones that do not appear as 4G or 5G.

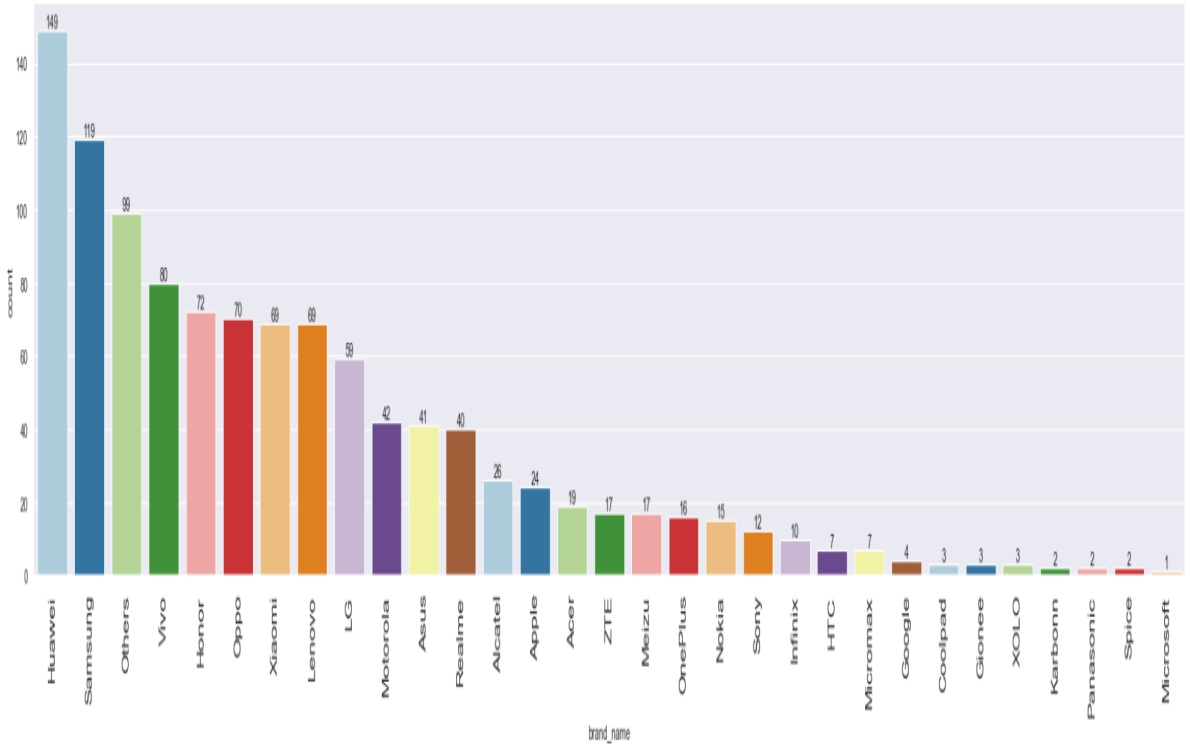
- OnePlus focuses on mid to high end smartphones and is a relatively new brand, which is why its average RAM is higher; brands with more time in the market, such as Samsung, Nokia, Huawei, LG, ZTE, Alcatel, ZTE, will have older equipment and of the non-smartphone type, which reduces their average RAM.

```
plt.figure(figsize=(15, 5))
sns.boxplot(data=df_large_battery, x='brand_name', y='weight') ## Complete the code to create a boxplot fo
plt.xticks(rotation=90)
plt.show()
```



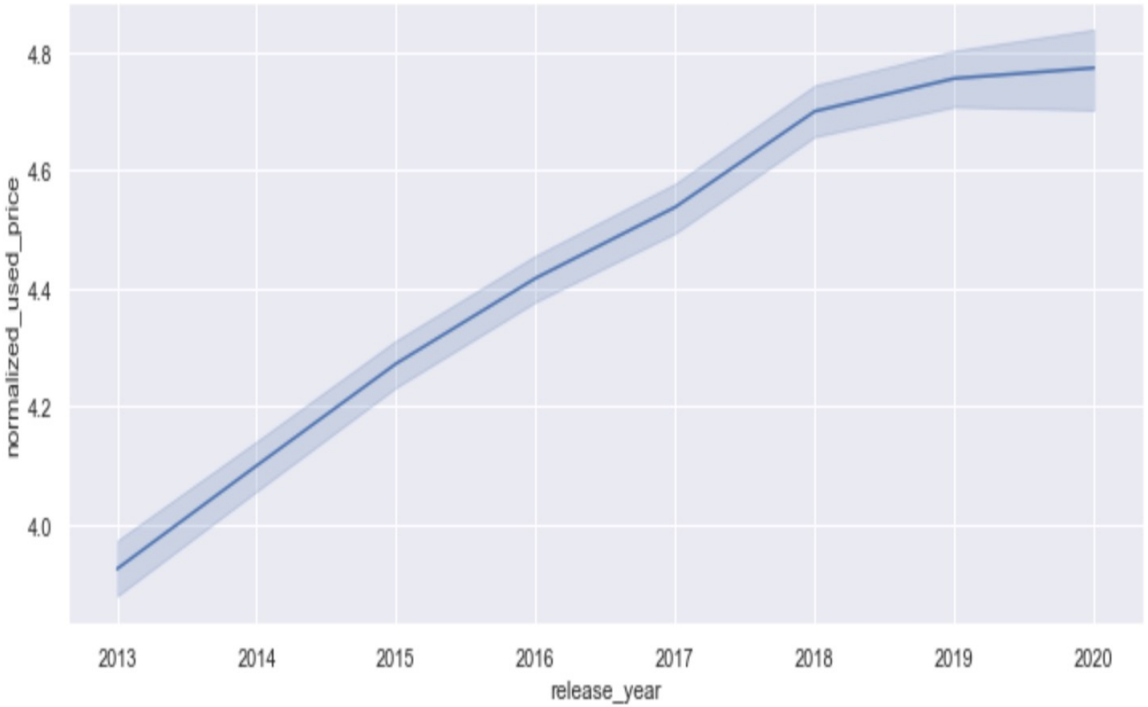
- Most of the samsung phone has more weight . This is because of larger batteries with larger mah.
- As battery capacity increases the weight increases.

```
labeled_barplot(df_large_screen, 'brand_name') ## Complete the code to create labeled_barplot for 'brand_name' in la
```



- Huawei has the most devices with 149 units followed by Samsung with 119 units.

```
plt.figure(figsize=(12, 5))
sns.lineplot(data=df, x="release_year", y='normalized_used_price') ## Complete the co
plt.show()
```



- There is a strong correlation between normalized_new_price and normalized_used_price .

Data Preprocessing – Duplicate Value Check

- There is no duplicate values in the data.

```
In [7]: data.duplicated().sum()## Complete the code to check duplicate entries in the data
```

```
Out[7]: 0
```

Data Preprocessing – Missing Value Treatment

In [44]:
```
cols_impute = [
    "main_camera_mp",
    "selfie_camera_mp",
    "int_memory",
    "ram",
    "battery",
    "weight",
]

for col in cols_impute:
    df1[col] = df1[col].fillna(
        value=df1.groupby(['release_year','brand_name'])[col].transform("median")
    )   ## Complete the code to impute missing values in cols_impute with median b

# checking for missing values
df1.isnull().sum().sort_values(ascending=False) ## Complete the code to check miss
```

Out[44]:
```
main_camera_mp           179
weight                     7
battery                    6
selfie_camera_mp           2
brand_name                 0
os                         0
screen_size                0
4g                         0
5g                         0
int_memory                 0
ram                        0
release_year               0
days_used                  0
normalized_used_price      0
normalized_new_price       0
dtype: int64
```

- We impute the missing values in the data by the column medians grouped by release_year and brand_name.

Data Preprocessing – Missing Value Treatment

In [45]:
```
cols_impute = [
    "main_camera_mp",
    "selfie_camera_mp",
    "battery",
    "weight",
]

for col in cols_impute:
    df1[col] = df1[col].fillna(
        value=df1.groupby(['brand_name'])[col].transform("median")
    ) ## Complete the code to impute the missing values in cols_im

# checking for missing values
df1.isnull().sum().sort_values(ascending= False) ## Complete the c
```

Out[45]:
```
main_camera_mp           10
brand_name                0
os                        0
screen_size               0
4g                        0
5g                        0
selfie_camera_mp          0
int_memory                0
ram                       0
battery                   0
weight                    0
release_year              0
days_used                 0
normalized_used_price     0
normalized_new_price      0
dtype: int64
```

- We impute the remaining missing values in the data by the column medians grouped by brand_name.

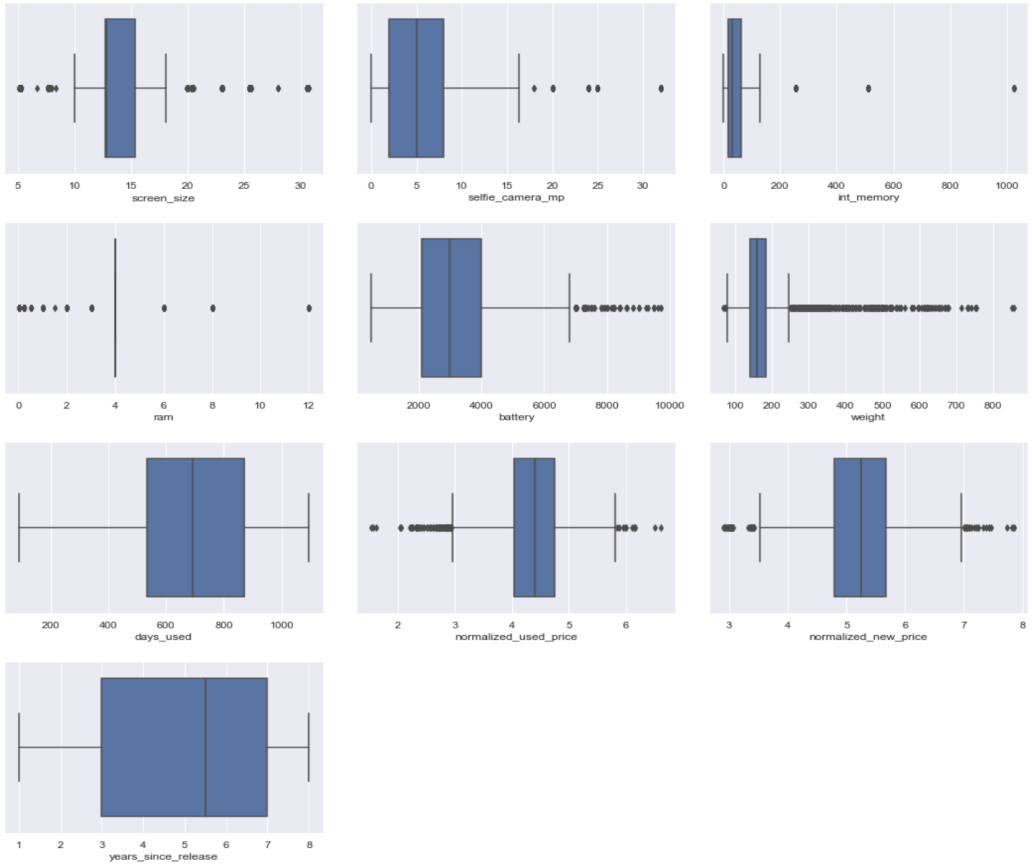
Data Preprocessing – Missing Value Treatment

```
In [46]: df1["main_camera_mp"] = df1["main_camera_mp"].fillna(df1["main_camera_mp"].median)

         # checking for missing values
         df1.isnull().sum().sort_values(ascending= False) ## Complete the code to check mis
```

```
Out[46]: brand_name             0
         os                     0
         screen_size            0
         4g                     0
         5g                     0
         main_camera_mp         0
         selfie_camera_mp       0
         int_memory             0
         ram                    0
         battery                0
         weight                 0
         release_year           0
         days_used              0
         normalized_used_price  0
         normalized_new_price   0
         dtype: int64
```

- We filled the remaining missing values in the main_camera_mp column by the column median.
- All missing values have been treated.

Data Preprocessing – Outlier Check



- There are many outliers in the data but we don’t treat them because they are proper values

Data Preprocessing – Feature Engineering

```
In [47]: df1["years_since_release"] = 2021 - df1["release_year"]
         df1.drop("release_year", axis=1, inplace=True)
         df1["years_since_release"].describe()
```

```
Out[47]: count    3454.000000
         mean        5.034742
         std         2.298455
         min         1.000000
         25%         3.000000
         50%         5.500000
         75%         7.000000
         max         8.000000
         Name: years_since_release, dtype: float64
```

- The mean value of the "years_since_release" variable is 5.03, which suggests that the average number of years since release is slightly above 5 years.
- The standard deviation of the " years_since_release " variable is 2.30, which suggests that the data is somewhat spread out around the mean.
- The minimum value of the "years_since_release" variable is 1 year, indicating that at least one observation in the dataset is just 1 year since release.
- The maximum value of the "years_since_release" variable is 8 years, indicating that at least one observation in the dataset is 8 years since release.
- The 25th percentile of the "years_since_release" variable is 3 years, meaning that 25% of the observations in the dataset have been released within the last 3 years.
- The 50th percentile (or median) of the "years_since_release" variable is 5.5 years, indicating that half of the observations have been released within the last 5.5 years.
- The 75th percentile of the "years_since_release" variable is 7 years, meaning that 75% of the observations in the dataset have been released within the last 7 years.

Data Preprocessing – Data Preparation For Modeling

In [49]:
```
## Complete the code to define the dependent and independent variables
X = df1.drop(['normalized_used_price'],axis=1)
y = df1['normalized_used_price']

print(X.head())
print()
print(y.head())
```

```
  brand_name       os  screen_size   4g   5g main_camera_mp  selfie_camera_mp  \
0      Honor  Android        14.50  yes   no           13.0               5.0
1      Honor  Android        17.30  yes  yes           13.0              16.0
2      Honor  Android        16.69  yes  yes           13.0               8.0
3      Honor  Android        25.50  yes  yes           13.0               8.0
4      Honor  Android        15.32  yes   no           13.0               8.0

   int_memory  ram  battery  weight  days_used  normalized_new_price  \
0        64.0  3.0   3020.0   146.0        127              4.715100
1       128.0  8.0   4300.0   213.0        325              5.519018
2       128.0  8.0   4200.0   213.0        162              5.884631
3        64.0  6.0   7250.0   480.0        345              5.630961
4        64.0  3.0   5000.0   185.0        293              4.947837

   years_since_release
0                    1
1                    1
2                    1
3                    1
4                    1

0    4.307572
1    5.162097
2    5.111084
3    5.135387
4    4.389995
Name: normalized_used_price, dtype: float64
```

In [50]:
```
# let's add the intercept to data
X = sm.add_constant(X)
```

In [51]:
```
# creating dummy variables
X = pd.get_dummies(
    X,
    columns=X.select_dtypes(include=["object", "category"]).columns.tolist(),
    drop_first=True,
)  ## Complete the code to create dummies for independent features

X.head()
```

Out[51]:

	const	screen_size	selfie_camera_mp	int_memory	ram	battery	weight	days_used	normalized_new_price	years_since_release	...	main_camera_mp_20.1
0	1.0	14.50	5.0	64.0	3.0	3020.0	146.0	127	4.715100	1	...	0
1	1.0	17.30	16.0	128.0	8.0	4300.0	213.0	325	5.519018	1	...	0
2	1.0	16.69	8.0	128.0	8.0	4200.0	213.0	162	5.884631	1	...	0
3	1.0	25.50	8.0	64.0	6.0	7250.0	480.0	345	5.630961	1	...	0
4	1.0	15.32	8.0	64.0	3.0	5000.0	185.0	293	4.947837	1	...	0

5 rows × 89 columns

In [52]:
```
# splitting the data in 70:30 ratio for train to test data

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1) ## Complete the code
```

In [53]:
```
print("Number of rows in train data =", x_train.shape[0])
print("Number of rows in test data =", x_test.shape[0])
```

```
Number of rows in train data = 2417
Number of rows in test data = 1037
```

- This how you prepare data for modeling.

Model Performance Summary

```
In [54]: olsmodel1 = sm.OLS(y_train,x_train).fit() ## Complete the code to fit OLS model
         print(olsmodel1.summary())
```

```
                                OLS Regression Results
==============================================================================
Dep. Variable:      normalized_used_price   R-squared:                       0.850
Model:                              OLS   Adj. R-squared:                  0.845
Method:                   Least Squares   F-statistic:                     159.2
Date:                  Wed, 15 Feb 2023   Prob (F-statistic):               0.00
Time:                          12:42:22   Log-Likelihood:                 163.90
No. Observations:                  2417   AIC:                            -159.8
Df Residuals:                      2333   BIC:                             326.6
Df Model:                            83
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const
1.6815       0.080     20.959      0.000       1.524       1.839
screen_size
0.0238       0.003      6.865      0.000       0.017       0.031
selfie_camera_mp
0.0137       0.001     12.146      0.000       0.012       0.016
int_memory
0.0002    7.09e-05      2.259      0.024    2.11e-05       0.000
ram
0.0223       0.005      4.274      0.000       0.012       0.032
battery
-1.753e-05   7.39e-06     -2.373      0.018    -3.2e-05   -3.04e-06
weight
0.0011       0.000      7.982      0.000       0.001       0.001
days_used
2.496e-05   3.11e-05      0.803      0.422    -3.6e-05    8.6e-05
normalized_new_price
0.4146       0.013     32.372      0.000       0.390       0.440
years_since_release
-0.0213      0.005     -4.560      0.000      -0.031      -0.012
brand_name_Alcatel
0.0093       0.047      0.196      0.844      -0.084       0.102
brand_name_Apple
-0.1129      0.147     -0.766      0.444      -0.402       0.176
brand_name_Asus
0.0033       0.048      0.070      0.944      -0.090       0.097
brand_name_BlackBerry
-0.0768      0.072     -1.067      0.286      -0.218       0.064
brand_name_Celkon
-0.0499      0.067     -0.747      0.455      -0.181       0.081
brand_name_Coolpad
0.0089       0.073      0.123      0.902      -0.133       0.151
brand_name_Gionee
0.0425       0.058      0.738      0.461      -0.070       0.155
brand_name_Google
0.0562       0.157      0.359      0.720      -0.251       0.364
brand_name_HTC
-0.0112      0.049     -0.230      0.818      -0.107       0.084
brand_name_Honor
0.0201       0.049      0.407      0.684      -0.076       0.117
brand_name_Huawei
-0.0225      0.045     -0.502      0.616      -0.110       0.065
brand_name_Infinix
0.0192       0.046      0.413      0.680      -0.072       0.110
brand_name_Karbonn
0.0772       0.067      1.155      0.248      -0.054       0.208
brand_name_LG
-0.0206      0.045     -0.455      0.649      -0.110       0.068
brand_name_Lava
0.0197       0.062      0.317      0.752      -0.102       0.142
brand_name_Lenovo
0.0340       0.045      0.751      0.453      -0.055       0.123
brand_name_Meizu
-0.0198      0.056     -0.353      0.724      -0.130       0.090
brand_name_Micromax
-0.0254      0.048     -0.533      0.594      -0.119       0.068
brand_name_Microsoft
0.0642       0.089      0.720      0.472      -0.111       0.239
brand_name_Motorola
-0.0037      0.050     -0.074      0.941      -0.102       0.094
brand_name_Nokia
0.0781       0.052      1.498      0.134      -0.024       0.180
brand_name_OnePlus
0.0680       0.077      0.881      0.378      -0.083       0.219
brand_name_Oppo
0.0051       0.048      0.107      0.915      -0.089       0.099
brand_name_Others
-0.0125      0.042     -0.297      0.767      -0.095       0.070
brand_name_Panasonic
0.0428       0.056      0.768      0.443      -0.067       0.152
brand_name_Realme
0.0156       0.062      0.253      0.800      -0.105       0.136
brand_name_Samsung
-0.0477      0.043     -1.098      0.272      -0.133       0.037
brand_name_Sony
-0.0476      0.055     -0.862      0.389      -0.156       0.061
brand_name_Spice
-0.0184      0.063     -0.291      0.771      -0.142       0.106
brand_name_Vivo
-0.0279      0.049     -0.575      0.565      -0.123       0.067
brand_name_XOLO
-7.906e-05   0.055     -0.001      0.999      -0.107       0.107
brand_name_Xiaomi
0.0650       0.048      1.342      0.180      -0.030       0.160
brand_name_ZTE
-0.0027      0.048     -0.057      0.955      -0.097       0.091
os_Others
0.0241       0.035      0.683      0.495      -0.045       0.093
os_Windows
-2.594e-05   0.048     -0.001      1.000      -0.093       0.093
os_iOS
0.0272       0.146      0.186      0.852      -0.259       0.313
4g_yes
0.0479       0.016      2.975      0.003       0.016       0.080
5g_yes
-0.0665      0.032     -2.065      0.039      -0.130      -0.003
main_camera_mp_8.0
-0.1088      0.015     -7.331      0.000      -0.138      -0.080
main_camera_mp_5.0
-0.1303      0.019     -9.565      0.000      -0.217      -0.143
main_camera_mp_10.5
0.0275       0.055      0.501      0.616      -0.080       0.135
main_camera_mp_3.15
-0.2440      0.032     -7.688      0.000      -0.306      -0.182
main_camera_mp_<bound method NDFrame._add_numeric_operations.<locals>.median of 0
13.0
1       13.0
2       13.0
3       13.0
4       13.0
        ...
3449    13.0
3450    13.0
3451    13.0
3452    13.0
3453    13.0
Name: main_camera_mp, Length: 3454, dtype: float64>    0.0192      0.046      0.413
0.680       -0.072       0.110
main_camera_mp_2.0
-0.2751      0.029     -9.569      0.000      -0.331      -0.219
main_camera_mp_16.0
0.1076       0.025      4.345      0.000       0.059       0.156
main_camera_mp_0.3
-0.4714      0.044    -10.704      0.000      -0.558      -0.385
main_camera_mp_12.0
0.0076       0.024      0.319      0.750      -0.039       0.055
main_camera_mp_14.5
-0.0145      0.078     -0.187      0.852      -0.167       0.138
main_camera_mp_48.0
0.2757       0.118      2.344      0.019       0.045       0.506
main_camera_mp_3.0
-0.1121      0.136     -0.824      0.410      -0.379       0.155
main_camera_mp_21.0
0.0706       0.069      1.028      0.304      -0.064       0.205
main_camera_mp_1.3
-0.4807      0.066     -7.237      0.000      -0.611      -0.350
main_camera_mp_13.1
0.1410       0.165      0.854      0.393      -0.183       0.465
main_camera_mp_24.0
0.0398       0.134      0.296      0.767      -0.224       0.303
main_camera_mp_0.08
-0.4712      0.234     -2.010      0.045      -0.931      -0.011
main_camera_mp_20.7
0.1023       0.083      1.237      0.216      -0.060       0.264
main_camera_mp_23.0
0.2705       0.073      3.720      0.000       0.128       0.413
main_camera_mp_1.0
-1.589e-16   9.33e-17     -1.703      0.089   -3.42e-16    2.4e-17
main_camera_mp_18.0
0.0682       0.238      0.287      0.774      -0.398       0.535
main_camera_mp_12.2
-0.1628      0.175     -0.928      0.354      -0.507       0.181
main_camera_mp_12.3
0.0540       0.108      0.500      0.617      -0.158       0.266
main_camera_mp_20.0
0.1257       0.096      1.303      0.193      -0.064       0.315
main_camera_mp_20.2
0.0289       0.232      0.125      0.901      -0.426       0.484
main_camera_mp_4.0
-0.2934      0.098     -2.981      0.003      -0.486      -0.100
main_camera_mp_12.5
0.1250       0.117      1.067      0.286      -0.105       0.355
main_camera_mp_10.0
-0.2221      0.119     -1.868      0.062      -0.455       0.011
main_camera_mp_6.5
-0.0930      0.164     -0.566      0.572      -0.415       0.229
main_camera_mp_41.0
-1.544e-17   2.62e-17     -0.589      0.556   -6.68e-17    3.59e-17
main_camera_mp_20.1
2.805e-17   4.81e-17      0.583      0.560   -6.63e-17    1.22e-16
main_camera_mp_12.6
-2.974e-17   3.99e-17     -0.745      0.456   -1.08e-16    4.85e-17
main_camera_mp_16.3
0.3742       0.233      1.608      0.108      -0.082       0.830
main_camera_mp_22.6
-0.1479      0.232     -0.639      0.523      -0.602       0.306
main_camera_mp_19.0
-0.0679      0.102     -0.666      0.506      -0.268       0.132
main_camera_mp_21.5
0.1234       0.234      0.527      0.598      -0.335       0.582
main_camera_mp_21.2
0.2065       0.168      1.233      0.218      -0.122       0.535
main_camera_mp_8.1
-0.1930      0.123     -1.572      0.116      -0.434       0.048
main_camera_mp_1.2
-0.1055      0.235     -0.449      0.654      -0.567       0.356
main_camera_mp_22.5
0.1759       0.232      0.759      0.448      -0.279       0.630
==============================================================================
Omnibus:                      205.856   Durbin-Watson:                   1.906
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              389.471
Skew:                          -0.578   Prob(JB):                     2.68e-85
Kurtosis:                       4.591   Cond. No.                     4.21e+19
==============================================================================
```

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The smallest eigenvalue is 1.63e-29. This might indicate that there are strong multicollinearity problems or that the design matrix is singular.

Observations

- The value of adjusted R-squared is 0.84, which is good.
- The y-intercept is equal to the value of the const coefficient which is 1.68
- The coefficients of the different predictor variables is well listed in the model summary for example; The coefficient of
- normalized_new_price is equal to 0.415

Model Performance Summary

In [55]:
```
# function to compute adjusted R-squared
def adj_r2_score(predictors, targets, predictions):
    r2 = r2_score(targets, predictions)
    n = predictors.shape[0]
    k = predictors.shape[1]
    return 1 - ((1 - r2) * (n - 1) / (n - k - 1))


# function to compute MAPE
def mape_score(targets, predictions):
    return np.mean(np.abs(targets - predictions) / targets) * 100


# function to compute different metrics to check performance of a regression model
def model_performance_regression(model, predictors, target):
    """
    Function to compute different metrics to check regression model performance

    model: regressor
    predictors: independent variables
    target: dependent variable
    """

    # predicting using the independent variables
    pred = model.predict(predictors)

    r2 = r2_score(target, pred)  # to compute R-squared
    adjr2 = adj_r2_score(predictors, target, pred)  # to compute adjusted R-squared
    rmse = np.sqrt(mean_squared_error(target, pred))  # to compute RMSE
    mae = mean_absolute_error(target, pred)  # to compute MAE
    mape = mape_score(target, pred)  # to compute MAPE

    # creating a dataframe of metrics
    df_perf = pd.DataFrame(
        {
            "RMSE": rmse,
            "MAE": mae,
            "R-squared": r2,
            "Adj. R-squared": adjr2,
            "MAPE": mape,
        },
        index=[0],
    )

    return df_perf
```

In [56]:
```
# checking model performance on train set (seen 70% data)
print("Training Performance\n")
olsmodel1_train_perf = model_performance_regression(olsmodel1, x_train, y_train)
olsmodel1_train_perf
```

```
Training Performance
```

Out[56]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	0.226107	0.17736	0.849942	0.844202	4.247918

In [57]:
```
# checking model performance on test set (seen 30% data)
print("Test Performance\n")
olsmodel1_test_perf = model_performance_regression(olsmodel1,x_test,y_test) ## Complete the code to check the
olsmodel1_test_perf
```

```
Test Performance
```

Out[57]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	0.239404	0.185272	0.841094	0.82616	4.495925

- The training R2 is 0.84, so the model is not underfitting.
- The train and test RMSE and MAE are comparable, so the model is not overfitting either.
- MAE suggests that the model can predict the price of a used device within a mean error of 0.18 on the test data.
- MAPE of 4.49 on the test data means that we are able to predict within 4.5% of the used device price.

Model Assumptions - Test for Multicollinearity

```
In [58]: #Let's define a function to check VIF.

def checking_vif(predictors):
    vif = pd.DataFrame()
    vif["feature"] = predictors.columns

    # calculating VIF for each feature
    vif["VIF"] = [
        variance_inflation_factor(predictors.values, i)
        for i in range(len(predictors.columns))
    ]
    return vif
```

```
In [61]: checking_vif(x_train).ascending  ## Complete the code to check VIF on train data

/Users/Moafdhal/opt/anaconda3/lib/python3.9/site-packages/statsmodels/stats/outlie
eWarning: divide by zero encountered in double_scalars
  vif = 1. / (1. - r_squared_i)
/Users/Moafdhal/opt/anaconda3/lib/python3.9/site-packages/statsmodels/regression/l
eWarning: invalid value encountered in double_scalars
  return 1 - self.ssr/self.centered_tss
```

Out[61]:

	feature	VIF
0	const	293.713519
1	screen_size	8.072753
2	selfie_camera_mp	2.897750
3	int_memory	1.434004
4	ram	2.375248
5	battery	4.285870
6	weight	6.972076
7	days_used	2.751173
8	normalized_new_price	3.464667
9	years_since_release	5.260309
10	brand_name_Alcatel	3.436328
11	brand_name_Apple	13.339101
12	brand_name_Asus	3.377646
13	brand_name_BlackBerry	1.748348
14	brand_name_Celkon	1.835217

Notes

- We will test for multicollinearity using VIF.
- **General Rule of thumb**:
  - If VIF is 1 then there is no correlation between the *k*kth predictor and the remaining predictor variables.
  - If VIF exceeds 5 or is close to exceeding 5, we say there is moderate multicollinearity.
  - If VIF is 10 or exceeding 10, it shows signs of high multicollinearity.

Observations

- There are two columns with high VIF values: screen_size and weight.
- Multicollinearity is present. We will systematically de-number these columns with VIF greater than 5.

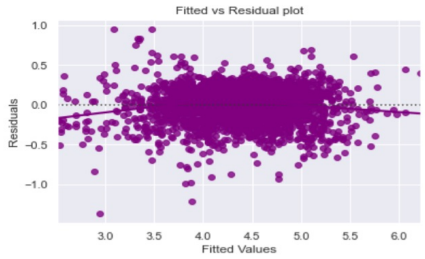
Model Assumptions - Test for Linearity and Independence

In [71]:
```
# let us create a dataframe with actual, fitted and residual values
df_pred = pd.DataFrame()

df_pred["Actual Values"] = y_train  # actual values
df_pred["Fitted Values"] = olsmodel2.fittedvalues  # predicted values
df_pred["Residuals"] = olsmodel2.resid  # residuals

df_pred.head()
```

Out[71]:

	Actual Values	Fitted Values	Residuals
3026	4.087488	3.870343	0.217145
1525	4.448399	4.585538	-0.137139
1128	4.315353	4.293276	0.022077
3003	4.282068	4.260293	0.021775
2907	4.456438	4.456145	0.000293

In [72]:
```
# let's plot the fitted values vs residuals

sns.residplot(
    data=df_pred, x="Fitted Values", y="Residuals", color="purple", lowess=True
)
plt.xlabel("Fitted Values")
plt.ylabel("Residuals")
plt.title("Fitted vs Residual plot")
plt.show()
```



Notes

•We will test for linearity and independence by making a plot of fitted values vs residuals and checking for patterns.

•If there is no pattern, then we say the model is linear and residuals are independent.

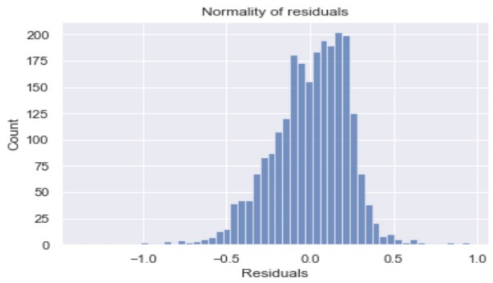
•Otherwise, the model is showing signs of non-linearity and residuals are not independent.

Observations

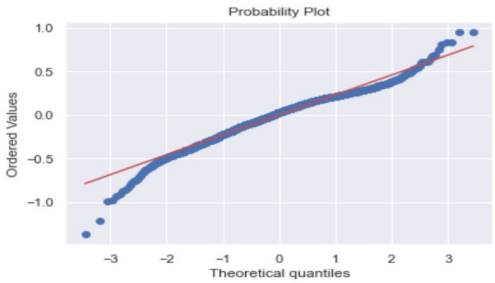
- The scatter plot shows the distribution of residuals (errors) vs fitted values (predicted values).
- We see no pattern in the plot above. Hence, the assumptions of linearity and independence are satisfied.

Model Assumptions - Test for Normality

```
In [73]: sns.histplot(data=df_pred, x='Residuals') ## Complete the code to plot the
         plt.title("Normality of residuals")
         plt.show()
```




```
In [74]: import pylab
         import scipy.stats as stats

         stats.probplot(df_pred['Residuals'], dist="norm", plot=pylab) ## Complete
         plt.show()
```




```
In [75]: stats.shapiro(df_pred['Residuals']) ## Complete the code to apply the Shap
Out[75]: ShapiroResult(statistic=0.974733293056488, pvalue=2.8776883265335313e-20)
```

Notes

•We will test for normality by checking the distribution of residuals, by checking the Q-Q plot of residuals, and by using the Shapiro-Wilk test.

•If the residuals follow a normal distribution, they will make a straight-line plot, otherwise not.

•If the p-value of the Shapiro-Wilk test is greater than 0.05, we can say the residuals are normally distributed.

Observations

- Since p-value < 0.05, the residuals are not normal as per the Shapiro-Wilk test.
- The residuals are not normal, strictly speaking. However, we can accept this distribution as close to normal as an approximation. As a result, the assumption is satisfied.

Model Assumptions - Test for Homoscedasticity

```
In [76]: import statsmodels.stats.api as sms
         from statsmodels.compat import lzip

         name = ["F statistic", "p-value"]
         test = sms.het_goldfeldquandt(df_pred["Residuals"], x_train3) ## Complet
         lzip(name, test)

Out[76]: [('F statistic', 1.0622319654059906), ('p-value', 0.14943725564900745)]
```

Notes

•We will test for homoscedasticity by using the goldfeldquandt test.

•If we get a p-value greater than 0.05, we can say that the residuals are homoscedastic. Otherwise, they are heteroscedastic.

Observations

- Since p-value > 0.05, we can say that the residuals are homoscedastic. So, this assumption is satisfied.

Final Model Summary

```
In [78]: olsmodel_final = sm.OLS(y_train, x_train_final).fit()
         print(olsmodel_final.summary())
```

```
                                OLS Regression Results
==============================================================================
Dep. Variable:     normalized_used_price   R-squared:                       0.843
Model:                               OLS   Adj. R-squared:                  0.841
Method:                    Least Squares   F-statistic:                     557.8
Date:                   Wed, 15 Feb 2023   Prob (F-statistic):               0.00
Time:                           12:46:17   Log-Likelihood:                 107.69
No. Observations:                   2417   AIC:                            -167.4
Df Residuals:                       2393   BIC:                            -28.41
Df Model:                             23
Covariance Type:               nonrobust
==============================================================================
                                                                        [0.025      0.97
5]
------------------------------------------------------------------------------
--
const                          1.5113      0.052     28.786      0.000       1.408       1.6
14
screen_size                    0.0439      0.002     28.724      0.000       0.041       0.0
47
selfie_camera_mp               0.0131      0.001     12.330      0.000       0.011       0.0
15
ram                            0.0156      0.004      3.480      0.001       0.007       0.0
24
normalized_new_price           0.4195      0.011     38.112      0.000       0.398       0.4
41
years_since_release           -0.0098      0.004     -2.780      0.005      -0.017      -0.0
03
brand_name_Nokia               0.1008      0.031      3.235      0.001       0.040       0.1
62
brand_name_Samsung            -0.0395      0.016     -2.402      0.016      -0.072      -0.0
07
brand_name_Xiaomi              0.0723      0.026      2.826      0.005       0.022       0.1
22
4g_yes                         0.0422      0.015      2.800      0.005       0.013       0.0
72
main_camera_mp_8.0            -0.1024      0.014     -7.493      0.000      -0.129      -0.0
76
main_camera_mp_5.0            -0.1589      0.017     -9.200      0.000      -0.193      -0.1
25
main_camera_mp_3.15           -0.2054      0.030     -6.901      0.000      -0.264      -0.1
47
main_camera_mp_2.0            -0.2168      0.026     -8.342      0.000      -0.268      -0.1
66
main_camera_mp_16.0            0.1019      0.024      4.197      0.000       0.054       0.1
49
main_camera_mp_0.3            -0.3934      0.039    -10.037      0.000      -0.470      -0.3
17
main_camera_mp_48.0            0.2836      0.117      2.427      0.015       0.054       0.5
13
main_camera_mp_1.3            -0.3872      0.060     -6.431      0.000      -0.505      -0.2
69
main_camera_mp_23.0            0.2564      0.068      3.789      0.000       0.124       0.3
89
main_camera_mp_4.0            -0.3067      0.096     -3.204      0.001      -0.494      -0.1
19
main_camera_mp_10.0           -0.2505      0.117     -2.133      0.033      -0.481      -0.0
20
main_camera_mp_6.7            -0.2403      0.121     -1.980      0.048      -0.478      -0.0
02
main_camera_mp_16.3            0.5478      0.234      2.344      0.019       0.089       1.0
06
main_camera_mp_8.1            -0.2956      0.118     -2.510      0.012      -0.527      -0.0
65
==============================================================================
Omnibus:                      193.234   Durbin-Watson:                   1.911
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              363.597
Skew:                          -0.550   Prob(JB):                     1.11e-79
Kurtosis:                       4.549   Cond. No.                         889.
==============================================================================
```

click to expand output; double click to hide output

```
In [79]: # checking model performance on train set (seen 70% data)
         print("Training Performance\n")
         olsmodel_final_train_perf = model_performance_regression(o
         olsmodel_final_train_perf
```

Training Performance

Out[79]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	0.231426	0.181502	0.842797	0.84122	4.344607

```
In [80]: # checking model performance on test set (seen 30% data)
         print("Test Performance\n")
         olsmodel_final_test_perf = model_performance_regression(ol
         olsmodel_final_test_perf
```

Test Performance

Out[80]:

	RMSE	MAE	R-squared	Adj. R-squared	MAPE
0	0.240476	0.186263	0.839667	0.835864	4.511603

Executive Summary

- The model can explain 84% of the variation in the data and is within 4.5% of the Used price on the test data, which is satisfactory. This indicates that the model is suitable for both prediction and inference.
- normalized_new_price and normalized_used_price have a strong positive relationship. As a result, the price of an identical device on the refurbished market will rise in proportion to the price of the new device.
- If the selfie_camera_mp increases by one unit, its price rises by 0.0136.
- Likewise, if the screen_size increases by one unit, its price rises by 0.0439.
- screen_size , main_camera_mp, selfie_camera_mp, ram, normalized_new_price, have positive coefficients. So, as they increase, the price of used devices also increases.

Recommendations

- Finally, the following variables have a significant impact on the market price of reconditioned gadgets: - Device display size - Primary camera megapixels - Front-facing camera pixels - Internal memory - RAM of the devices - Price of a new comparable device - Release year - Whether or not the device is 4G/5G - Operating system.

- People want newer devices with better features, and brands were at the upper edge of some of our comparison analyses, with iOS offering the highest prices, despite having fewer devices than Android.

Great Learning

Happy Learning !