New

May 31, 2020

1 Course Project - MATH2319 Machine Learning | Sem 1, 2020

1.1 Honour Code

We solemnly swear that we have not discussed our assignment solutions with anyone in any way and the solutions we are submitting are our own personal work.

Full Name: Ankit Munot (s3764950), Akshay Sunil Salunke (s3730440) | Group 12

1.2 Table of Contents

2 Objective

The objective of this project is to predict mobile phone price range, based on various features a phone has. We predict price for a phone as a price-bucket rather than as a continuos feature, hence the problem at hand is multinomial classification problem. We have performed following steps in this project, data visualization, data preprocessing, feature selection (using f-score and random forest), model evaluation (4 models).

3 Source and description of dataset

3.1 Dataset

The dataset for this project was sourced from kaggle.com. A copy of dataset is included in `/data` folder, as accessed on `26 May 2020`. The dataset originally has 2,000 rows, but we have randomly sampled 1200 rows so that our laptops could cope up. The dataset has 20 features(excluding target), some of which are binary (eg. `wifi`, `bluetooth`, etc.) and denote if a phone has that particular feature, whereas some are continuos features (eg. `batter_power`, `ram`, etc.)

3.2 Target feature

The target feature for this dataset is `price_range`, which has `0,1,2,3` as possible values, which represents phone price category corresponding to `low, mid, high, v.high`

3.3 Descriptive features

The dataset has followinf descriptive features: - `battery_power`: Total energy a battery can store in one time measured in *mAh*. - `bluetooth`: Has bluetooth or not. - `clock_speed`: Speed at which microprocessor executes instructions in *GHz*. - `dual_sim`: Has dual sim support or not. - `front_cam_mp`: Front Camera mega pixels. - `four_g`: Has 4G or not. - `int_memory`: Internal Memory in *Gigabytes*. - `n_cores`: Number of cores of processor. - `back_cam_mp`: Primary Camera in *Megapixels*. - `px_height`: Pixel Resolution Height in *pixels*. - `px_width`: Pixel Resolution Width in *pixels*. - `ram`: Random Access Memory in *Megabytes*. - `sc_h`: Screen Height of mobile in *cm*. - `sc_w`: Screen Width of mobile in *cm*. - `talk_time`: longest time that a single battery charge will last when you are on a call, in *Hours*. - `three_g`: Has 3G or not. - `touch_screen`: Has touch screen or not. - `wifi`: Has wifi or not. - `screen_size`: Screen size diagonally in *inches*.

Some of the features have been transformed into new features, for example, `sc_h` and `sc_w` have been transformed into `screen_size`, which is screen size measured diagonally in inches, which is the industry standard for measuring phone screen sizes.

4 Data preprocessing

4.1 Preliminaries

We import relevant datascience libraries, which we could think off top of our heads. Also, ignore the warning(because they're annoying).

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

```
import numpy as np
from sklearn import preprocessing
import sklearn
import warnings
import seaborn as sns
warnings.filterwarnings('ignore')
%matplotlib inline
```

We now read the data in an pandas dataframe. We randomly sample 1200 rows, with a fixed `random_state` so that results are reproducible.

[2]:
```
df = pd.read_csv("s3730440_data.csv")
df = df.sample(1200, random_state=786)
```

Here's how the original dataset looks like:

[3]:
```
df.head()
```

[3]:
```
      battery_power  blue  clock_speed  dual_sim  fc  four_g  int_memory  \
80             1589     1          0.6         1   0       1          58
880            1554     0          2.7         1   3       1          47
466            1653     0          0.5         1   2       1          37
1781           1876     0          1.3         1   9       1          64
898            1372     1          2.7         0   7       0          34

      m_dep  mobile_wt  n_cores     ...       px_height  px_width   ram  sc_h  \
80      0.9         85        7     ...             319      1206  3464    19
880     0.7        185        5     ...             319      1367   509    12
466     0.9        176        4     ...             447      1785  3955    19
1781    1.0         98        3     ...             600      1211  3132    17
898     0.4        193        4     ...             687       937   725    11

      sc_w  talk_time  three_g  touch_screen  wifi  price_range
80      10          6        1             1     1            3
880      3         19        1             0     0            0
466      4         18        1             1     1            3
1781     0          2        1             1     1            3
898      3         20        1             0     0            0

[5 rows x 21 columns]
```

And here is it's shape: (rows, columns)

[4]:
```
df.shape
```

[4]:
```
(1200, 21)
```

4.2 Data cleaning and transformation

In this section we remove the irrelevant features, transform existing features into new, etc.

We now check for any null values right off the bat, in all `df`.

[5]:
```
df.isnull().any()
```

[5]:
```
battery_power    False
blue             False
clock_speed      False
dual_sim         False
fc               False
four_g           False
int_memory       False
m_dep            False
mobile_wt        False
n_cores          False
pc               False
px_height        False
px_width         False
ram              False
sc_h             False
sc_w             False
talk_time        False
three_g          False
touch_screen     False
wifi             False
price_range      False
dtype: bool
```

But isnull() only detects those values which are `NaN` or `None`. This is not enough and we still need to check the `df` for missing values in other formats, for example `?` or `0`.

We first rename the columns to more readable and understandable names.

[6]:
```
df = df.rename(columns={'blue':'bluetooth', 'fc':'front_cam_mp', 'sc_h':
↪'screen_ht', 'sc_w':'screen_wt', 'pc':'back_cam_mp'})
```

Then, we drop the columns like `mobile_wt` and `m_dep` which we think are not actual deciding factors for a phone price, but more of a byproduct after the phone has been already manufactured. We also dropped `px_height` and `px_width` since it played very little to no role as a deciding factor for price of a phone, for the general consumer, since most of the consumers focus on the physical screen size rather than how many pixels a display has.

[7]:
```
df = df.drop(columns=['m_dep', 'mobile_wt', 'px_height', 'px_width'])
```

Now we perform binning of continuos feature `clock_speed`, into 3-bins namely, `low`, `mid`, `high` which correspond to following buckets `0-1`, `1-2`, `2-3`, in *Ghz*.

[8]:
```
df['clock_speed'] = pd.cut(df['clock_speed'], bins=[0, 1, 2, 3], labels =␣
↪['low', 'mid', 'high'])
level_map = {'low':0, 'mid':1, 'high':2}
df['clock_speed'] = df['clock_speed'].replace(level_map)
df.head()
```

[8]:
```
     battery_power  bluetooth  clock_speed  dual_sim  front_cam_mp  four_g  \
80            1589          1            0         1             0       1
880           1554          0            2         1             3       1
```

```
466         1653          0          0        1         2        1
1781        1876          0          1        1         9        1
898         1372          1          2        0         7        0

      int_memory  n_cores  back_cam_mp   ram  screen_ht  screen_wt  talk_time  \
80            58        7            7  3464         19         10          6
880           47        5           12   509         12          3         19
466           37        4            6  3955         19          4         18
1781          64        3           19  3132         17          0          2
898           34        4           17   725         11          3         20

      three_g  touch_screen  wifi  price_range
80          1             1     1            3
880         1             0     0            0
466         1             1     1            3
1781        1             1     1            3
898         1             0     0            0
```

We also transform the features `screen_wt` and `screen_ht` which are in *cm*, to new feature `screen_size` which is the diagonal screen size in *inches*, which is the industry standard for measuring screen sizes of phones.

[9]:
```python
df[['screen_ht', 'screen_wt']].describe()

# Convert column datatypes to float.
df['screen_ht'] = df['screen_ht'].astype(float)
df['screen_wt'] = df['screen_wt'].astype(float)
```

We first handle the missing values in `screen_wt`. For this, we find the height for missing width, calculate mean width for that height in whole df, then append this mean to missijg value.

[10]:
```python
# Create a temp list of all screen heights whose screen width is 0.
x = df[df['screen_wt']==0]['screen_ht'].value_counts().index.tolist()

# Create a list of all screen heights whose screen width is 0.
arr = []
for d in df.loc[df['screen_wt']==0]['screen_ht']:
    arr.append(d)

# Calculate mean width for a specific screen height.
# We create set out of screen heights list so that width is calculated only for␣
 ↪unique screen heights.
# mean_width stores all unique screen heights, and their mean width.
mean_width = {}
for d in set(arr):
    total = 0
    n = 0
    for width in df.loc[df['screen_ht'] == d]['screen_wt']:
        if width == 0:
```

```
            pass
        total += width
        n += 1
        mean = round(total/n, 2)
    print("Mean width for height", d, "=", mean)
    mean_width[d] = mean

# Append all missing screen widths with the mean screen width for that specific␣
↪height.
for z in x:
    df['screen_wt'] = np.where(((df['screen_wt']==0.0) & (df['screen_ht']==z)),␣
↪mean_width.get(z), df['screen_wt'])
```

```
Mean width for height 5.0 = 2.24
Mean width for height 6.0 = 2.33
Mean width for height 7.0 = 3.36
Mean width for height 8.0 = 3.94
Mean width for height 9.0 = 3.77
Mean width for height 10.0 = 4.04
Mean width for height 11.0 = 5.49
Mean width for height 12.0 = 5.96
Mean width for height 13.0 = 6.08
Mean width for height 14.0 = 6.66
Mean width for height 15.0 = 7.21
Mean width for height 16.0 = 7.19
Mean width for height 17.0 = 8.11
Mean width for height 18.0 = 7.57
Mean width for height 19.0 = 9.05
```

[11]:
```
df.head()
```

[11]:
```
      battery_power  bluetooth  clock_speed  dual_sim  front_cam_mp  four_g  \
80             1589          1            0         1             0       1
880            1554          0            2         1             3       1
466            1653          0            0         1             2       1
1781           1876          0            1         1             9       1
898            1372          1            2         0             7       0

      int_memory  n_cores  back_cam_mp   ram  screen_ht  screen_wt  talk_time  \
80            58        7            7  3464       19.0      10.00          6
880           47        5           12   509       12.0       3.00         19
466           37        4            6  3955       19.0       4.00         18
1781          64        3           19  3132       17.0       8.11          2
898           34        4           17   725       11.0       3.00         20

      three_g  touch_screen  wifi  price_range
80          1             1     1            3
880         1             0     0            0
```

```
466            1              1      1             3
1781           1              1      1             3
898            1              0      0             0
```

We now transform the `screen_height` and `screen_width` features to `screen_size` feature. Using the formula for diagonal of a rectangle = $\sqrt{width^2 + heigth^2}$. Then we convert to *inches* by dividing with 2.54.

[12]:
```
df['screen_size'] = df['screen_ht']**2 + df['screen_wt']**2
df['screen_size'] = np.sqrt(df['screen_size'])
df['screen_size'] = df['screen_size']/2.54
df['screen_size'] = df['screen_size'].round(2)
df.drop(columns=['screen_ht', 'screen_wt'], inplace=True)

p = pd.DataFrame(df['price_range'])
df.drop(columns=['price_range'], inplace=True)
df = df.join(p)
```

We also, seperate out the categorical features, continuos features and the target feature in seperate variables.

[13]:
```
categorical_features = ['bluetooth', 'dual_sim', 'four_g', 'three_g',␣
 ↪'touch_screen', 'wifi']
continuous_features = ['battery_power', 'front_cam_mp', 'int_memory', 'n_cores',␣
 ↪'back_cam_mp', 'ram', 'clock_speed', 'talk_time']
TARGET = ['price_range']
```

4.3 Summary of features

We describe the continuos features, to understand the dataset and draw some insights.

[14]:
```
df[continuous_features].describe(include='all')
```

[14]:
```
       battery_power  front_cam_mp   int_memory      n_cores  back_cam_mp  \
count    1200.000000   1200.000000  1200.000000  1200.000000   1200.00000
mean     1247.585833      4.360833    32.122500     4.527500      9.93500
std       443.048652      4.336624    18.149022     2.308394      6.10001
min       502.000000      0.000000     2.000000     1.000000      0.00000
25%       852.000000      1.000000    16.000000     3.000000      5.00000
50%      1233.000000      3.000000    32.000000     4.000000     10.00000
75%      1640.000000      7.000000    48.000000     7.000000     15.00000
max      1998.000000     18.000000    64.000000     8.000000     20.00000

               ram  clock_speed    talk_time
count  1200.000000  1200.000000  1200.000000
mean   2146.388333     0.941667    11.014167
std    1084.707313     0.816794     5.406322
min     256.000000     0.000000     2.000000
25%    1232.750000     0.000000     6.000000
50%    2172.500000     1.000000    11.000000
```

```
75%     3088.750000      2.000000    16.000000
max     3998.000000      2.000000    20.000000
```

Here is the shape of our df after dropping unrequired columns.

[15]:
```
df.shape
```

[15]:
```
(1200, 16)
```

We now print unique values for all features to find any missing/outlier values.

[16]:
```
df['bluetooth'].unique()
```

[16]:
```
array([1, 0])
```

[17]:
```
df['dual_sim'].value_counts()
```

[17]:
```
1    615
0    585
Name: dual_sim, dtype: int64
```

[18]:
```
df['front_cam_mp'].unique()
```

[18]:
```
array([ 0,  3,  2,  9,  7, 16,  4,  6,  5, 11,  8, 15,  1, 13, 10, 12, 14,
       17, 18])
```

[19]:
```
df['back_cam_mp'].unique()
```

[19]:
```
array([ 7, 12,  6, 19, 17,  8, 14,  3, 16, 11,  9, 15, 18, 10,  1,  0,  2,
        5, 13, 20,  4])
```

We assume the 0 values in from_cam_mp and back_cam_mp are not outliers and instead mean that those phone lack that specific feature.

[20]:
```
df['four_g'].value_counts()
```

[20]:
```
1    604
0    596
Name: four_g, dtype: int64
```

[21]:
```
df['int_memory'].describe()
```

[21]:
```
count    1200.000000
mean       32.122500
std        18.149022
min         2.000000
25%        16.000000
50%        32.000000
75%        48.000000
max        64.000000
Name: int_memory, dtype: float64
```

The extreme values for int_memory are 2Gb and 64Gb, which are both available as internam memory on phones. Hence there are no outliers.

[22]:
```
df['n_cores'].value_counts()
```

[22]:
```
7    162
8    156
4    156
1    150
3    148
5    147
2    147
6    134
Name: n_cores, dtype: int64
```

[23]:
```
df['ram'].describe()
```

[23]:
```
count    1200.000000
mean     2146.388333
std      1084.707313
min       256.000000
25%      1232.750000
50%      2172.500000
75%      3088.750000
max      3998.000000
Name: ram, dtype: float64
```

[24]:
```
df['talk_time'].describe()
```

[24]:
```
count    1200.000000
mean       11.014167
std         5.406322
min         2.000000
25%         6.000000
50%        11.000000
75%        16.000000
max        20.000000
Name: talk_time, dtype: float64
```

[25]:
```
df['three_g'].value_counts()
```

[25]:
```
1    899
0    301
Name: three_g, dtype: int64
```

[26]:
```
df['touch_screen'].value_counts()
```

[26]:
```
1    618
0    582
Name: touch_screen, dtype: int64
```

[27]:
```
df['four_g'].value_counts()
```

[27]:
```
1    604
0    596
Name: four_g, dtype: int64
```

[28]:
```
df['wifi'].value_counts()
```

[28]:
```
1    602
0    598
Name: wifi, dtype: int64
```

[29]:
```
df['price_range'].value_counts()
```

[29]:
```
1    310
3    307
2    303
0    280
Name: price_range, dtype: int64
```

All categorical values have possible values, no missing values or outliers.

[30]:
```
df.head()
```

[30]:
```
      battery_power  bluetooth  clock_speed  dual_sim  front_cam_mp  four_g  \
80             1589          1            0         1             0       1
880            1554          0            2         1             3       1
466            1653          0            0         1             2       1
1781           1876          0            1         1             9       1
898            1372          1            2         0             7       0

      int_memory  n_cores  back_cam_mp   ram  talk_time  three_g  \
80            58        7            7  3464          6        1
880           47        5           12   509         19        1
466           37        4            6  3955         18        1
1781          64        3           19  3132          2        1
898           34        4           17   725         20        1

      touch_screen  wifi  screen_size  price_range
80               1     1         8.45            3
880              0     0         4.87            0
466              1     1         7.64            3
1781             1     1         7.42            3
898              0     0         4.49            0
```

[31]:
```
df.head()
```

[31]:
```
      battery_power  bluetooth  clock_speed  dual_sim  front_cam_mp  four_g  \
80             1589          1            0         1             0       1
880            1554          0            2         1             3       1
466            1653          0            0         1             2       1
1781           1876          0            1         1             9       1
898            1372          1            2         0             7       0

      int_memory  n_cores  back_cam_mp   ram  talk_time  three_g  \
80            58        7            7  3464          6        1
880           47        5           12   509         19        1
466           37        4            6  3955         18        1
1781          64        3           19  3132          2        1
```

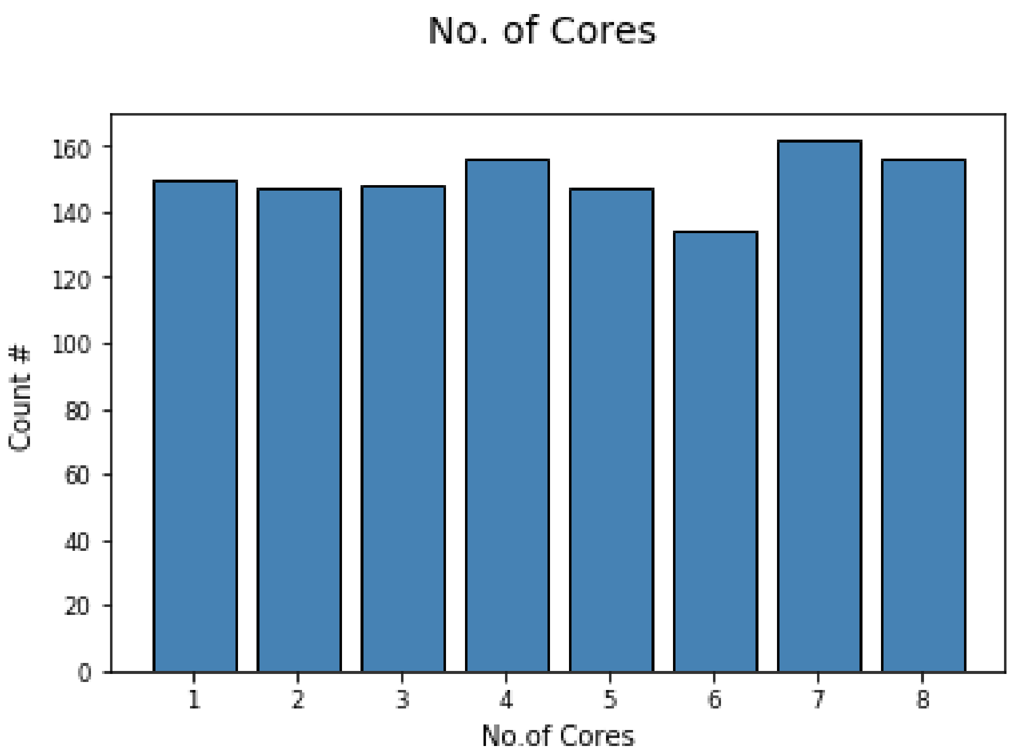
```
898            34         4            17   725          20         1

      touch_screen  wifi  screen_size  price_range
80               1     1         8.45            3
880              0     0         4.87            0
466              1     1         7.64            3
1781             1     1         7.42            3
898              0     0         4.49            0
```

5 Data Exploration and Visualisation

In this section we visualize the data and try to get some insights from it. ## Univariate visualizations In this section we try to visualize and analyze one feature at a time.

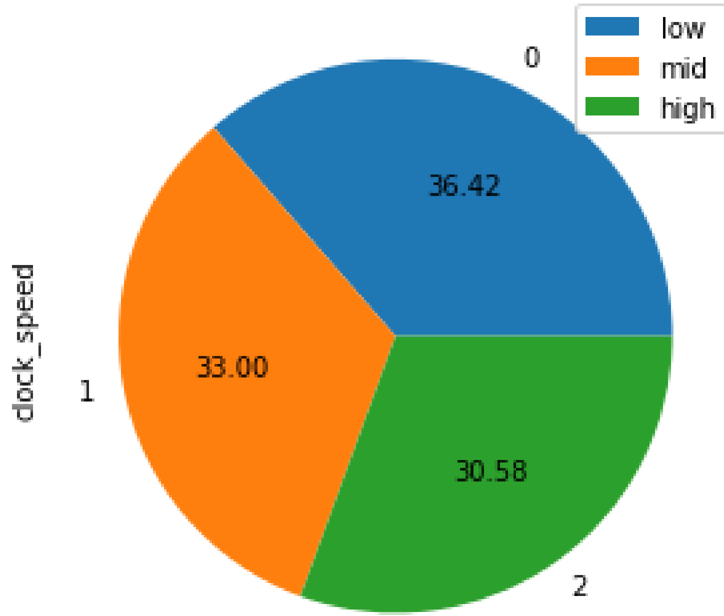
First we plot a bar plot of counts of `n_cores` across the whole dataset. There is no clear pattern visible, but you learn that some number of cores appear more frequently (like `4`, `7`, `8`), denoting that these core sizes occur more frequently, which aligns with the fact that quad and octa core are standard # of cores for many major processor brands (like Qualcomm and Mediatek).

[32]:
```
# Bar Plot
fig = plt.figure(figsize = (6, 4))
title = fig.suptitle("No. of Cores", fontsize = 14)
fig.subplots_adjust(top=0.85, wspace=0.3)

ax = fig.add_subplot(1, 1, 1)
ax.set_xlabel("No.of Cores")
ax.set_ylabel("Count #")
w_q = df['n_cores'].value_counts()
w_q = (list(w_q.index), list(w_q.values))
ax.tick_params(axis='both', which='major', labelsize=8.5)
bar = ax.bar(w_q[0], w_q[1], color='steelblue',
        edgecolor='black', linewidth=1)
```



Next, we plot a pie chart on `clock_speed` feature, which shows that `low` clock speed appears the most, which suggests that majority of the phones operated on a processor which was between `0-1 Ghz`, followed by `1-2 Ghz` and lastly the high end phones with `2-3 Ghz` clock speed.

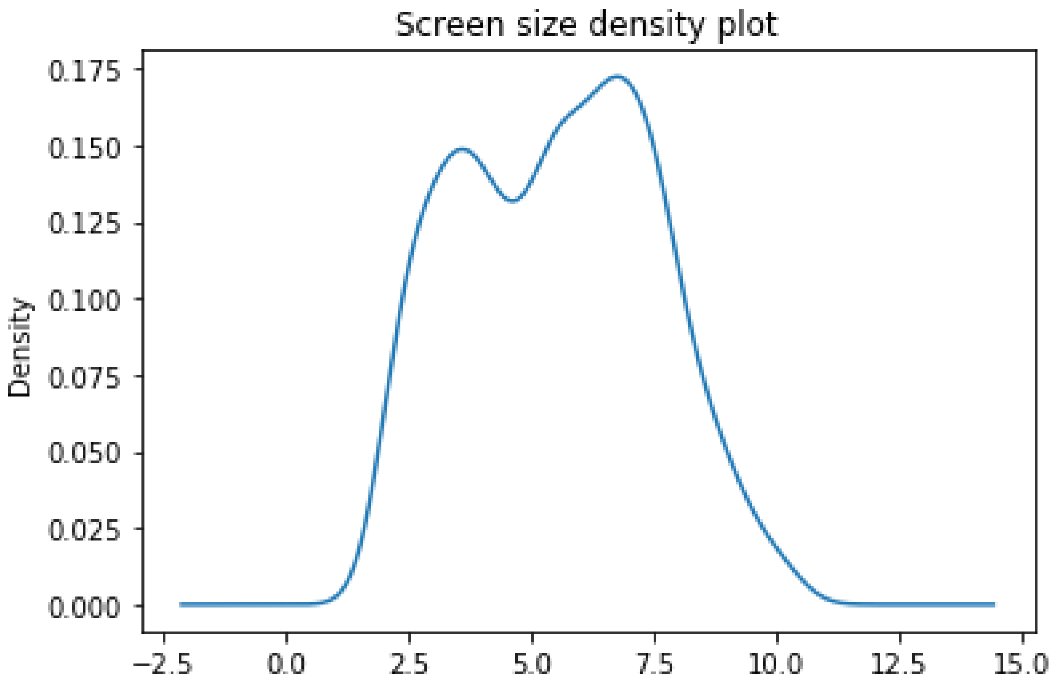
[33]:
```
labels = ['low', 'mid', 'high']
df['clock_speed'].value_counts().plot(kind='pie', autopct='%.2f')
plt.tight_layout()
plt.legend(labels)
plt.show()
```



Since `screen_size` is a continuos feature, we plot a density graph, to try to find which screen sizes were more prominent. We find that screen size across phones can be split into 2 distinct different categories, which aligns with today's trend of smaller 4.7 *inches* screens and larger 6+ *inches* screens. (Apple iPhone SE 2020 - 4.7 *inches*, iPhone 11 Pro Max - 6.5 *inches*)

[34]:
```
df['screen_size'].plot(kind='density', title="Screen size density plot")
```

[34]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f31dc186588>
```



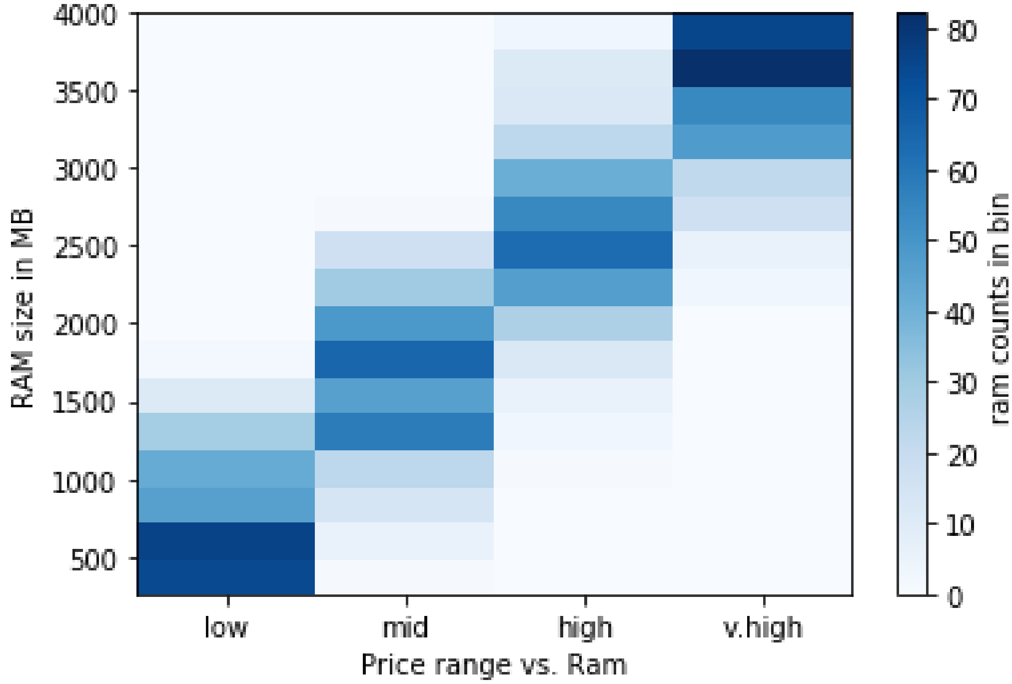
5.1 Bivariate visualizations

In this section we try to visualize and analyze two features at a time.

We start with plotting a 2d histogram for `ram` and `price_range`. We can clearly see the density of ram for each price range. Following are the average ram sizes for each price range: - `low` : 0 - 750 MB - `mid` : 1250 - 1800 MB - `high` : ~2500 MB - `v.high` : 3500 - 4000 MB

We can clearly see the pattern, as price range increases, average ram size also increases.

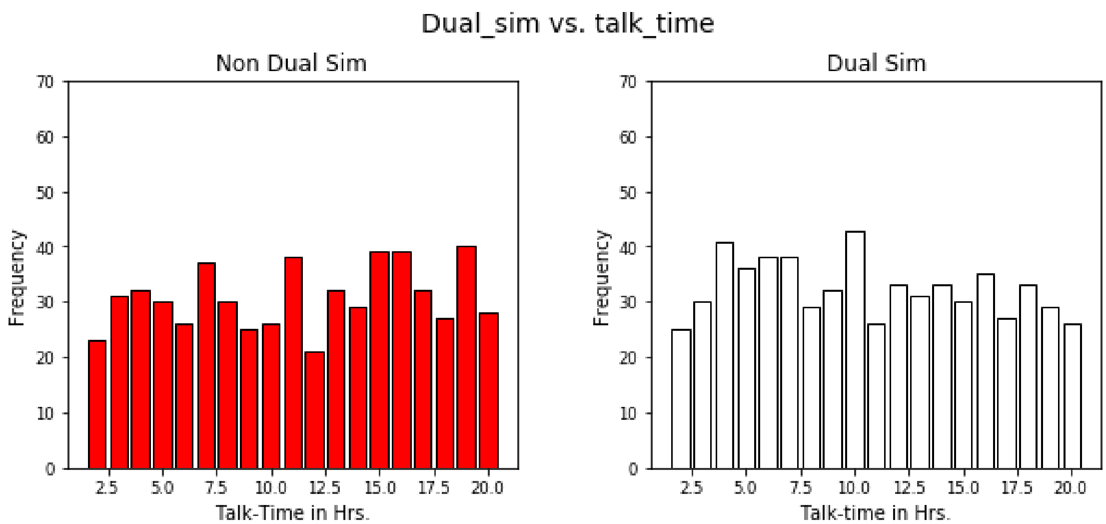
[35]:
```
labels = ['low', 'mid', 'high', 'v.high']
h, x, y, i = plt.hist2d(df['price_range'], df['ram'], bins=(4, 16), cmap='Blues')
bin_w = (max(x) - min(x)) / (len(x) - 1)
plt.xticks(np.arange(min(range(0,4))+bin_w/2, max(range(0, 4)), bin_w), labels)
plt.xlabel("Price range vs. Ram")
plt.ylabel("RAM size in MB")
cb = plt.colorbar(i)
cb.set_label('ram counts in bin')
plt.show()
```



Now we plot a bar graph between `dual_sim` and `talk_time` features. Even though there is no clear pattern visible, we can see that talk time for non dual sim phones falls around the higher end and vice versa. From this, we can assume that dual sim phones have less talk time on average as compared to non dual sim phones.

[36]:
```
# Using subplots or facets along with Bar Plots
fig = plt.figure(figsize = (10, 4))
title = fig.suptitle("Dual_sim vs. talk_time", fontsize=14)
fig.subplots_adjust(top=0.85, wspace=0.3)

# Non Dual Sim
ax1 = fig.add_subplot(1,2, 1)
ax1.set_title("Non Dual Sim")
ax1.set_xlabel("Talk-Time in Hrs.")
ax1.set_ylabel("Frequency")
rw_q = df[df['dual_sim'] == 0]['talk_time'].value_counts()
rw_q = (list(rw_q.index), list(rw_q.values))
ax1.set_ylim([0,70])
ax1.tick_params(axis='both', which='major', labelsize=8.5)
bar1 = ax1.bar(rw_q[0], rw_q[1], color='red',
               edgecolor='black', linewidth=1)

# Dual Sim
ax2 = fig.add_subplot(1,2, 2)
ax2.set_title("Dual Sim")
ax2.set_xlabel("Talk-time in Hrs.")
```

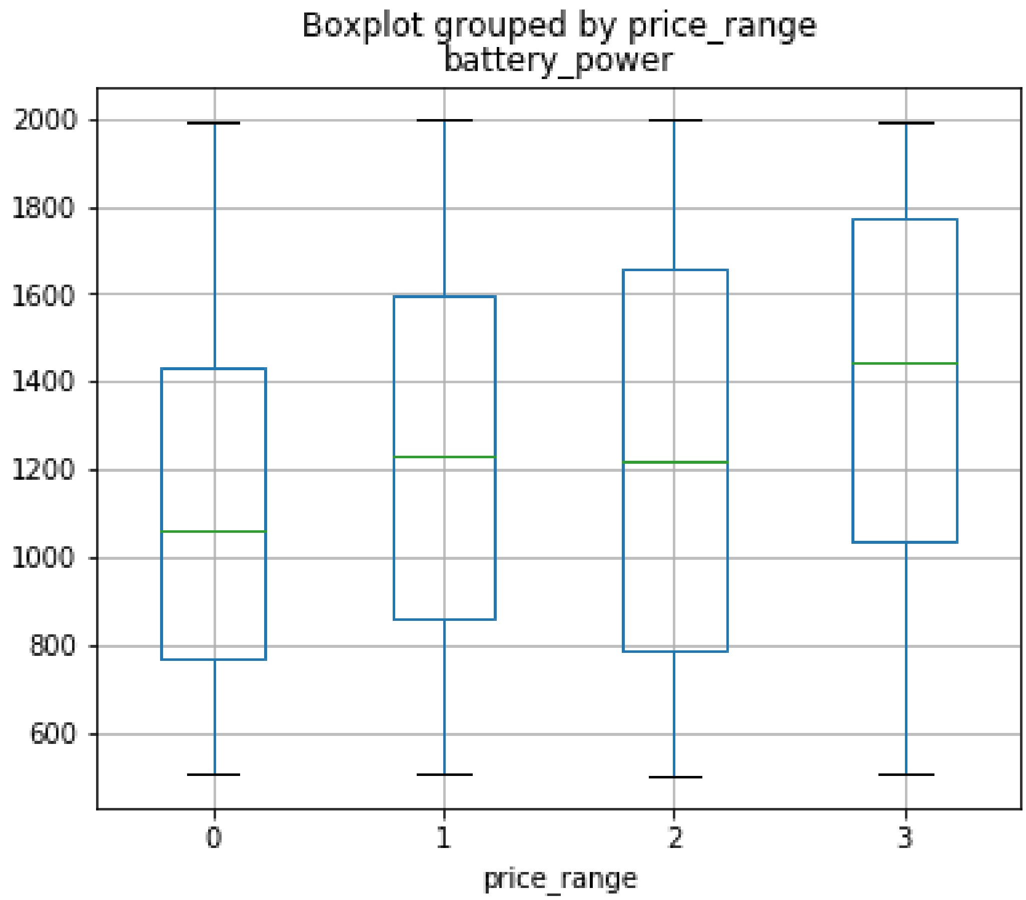
```
ax2.set_ylabel("Frequency")
ww_q = df[df['dual_sim'] == 1]['talk_time'].value_counts()
ww_q = (list(ww_q.index), list(ww_q.values))
ax2.set_ylim([0, 70])
ax2.tick_params(axis='both', which='major', labelsize=8.5)
bar2 = ax2.bar(ww_q[0], ww_q[1], color='white',
               edgecolor='black', linewidth=1)
```



Lastly, we plot a boxplot for `battery_power` vs `price_range`. Here we can see that as the price range increases, battery capacity also tends to increase. But, this is not true for `mid` and `high` range phones. High range phones have very little battery capacity increase as compared to `mid` range phones.

[37]:
```
df.boxplot(column='battery_power', by='price_range', figsize=(6,5))
```

[37]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f31dbd316d8>`



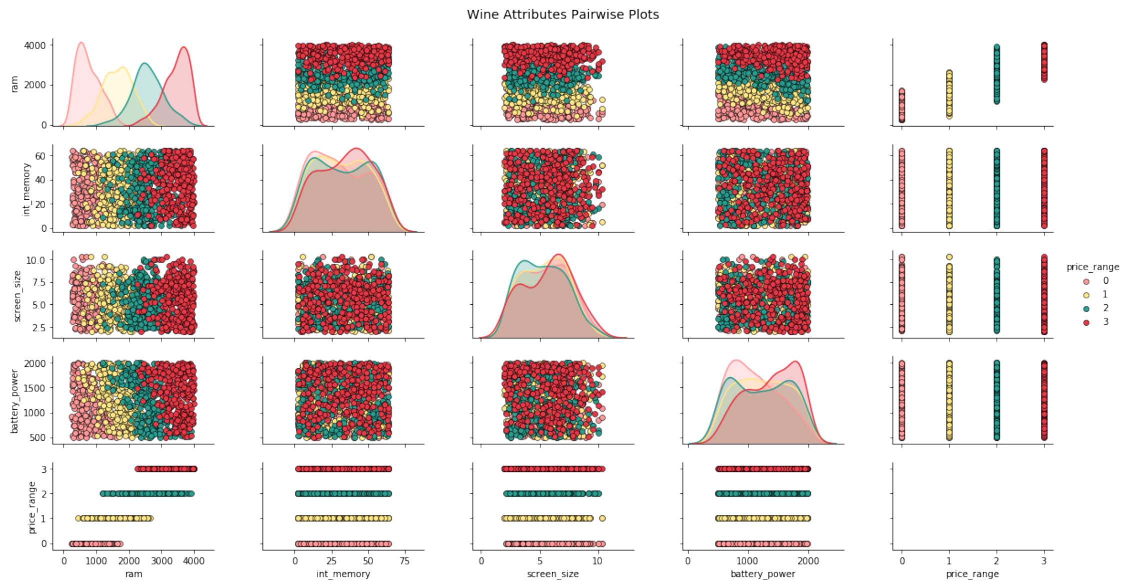
5.2 Multivariate visualizations

In this section we try to visualize and analyze three or more features at a time.

To start with, we plot 5 different features with each other in a pair plot, to find any dependence/pattern between features. Following observations can be concluded from the pair plot: - `ram` feature shows distinct seperation along target feature, meaning the average ram differs most between price range. - `low` and `v.high` prices phones tend to have lower or higher internal memory respectively. But `mid` and `high` priced phones can have a lot of options avaiable for internal memory sizes. - all phones tend to have bigger `screen_size`, except `high` priced phones. This may mean that there are people who tend to purchase high priced phones but are looking for smaller screen sizes. - Only `v.high` priced phones can offer higher battery capacity in general.

[38]:
```
# Scaling attribute values to avoid few outiers
cols = ['ram', 'int_memory', 'screen_size', 'battery_power','price_range']
pp = sns.pairplot(df[cols], hue='price_range', size=1.8, aspect=1.8,
                  palette={0: "#FF9999", 1: "#FFE888", 2:"#2A9D8F", 3:"#E63946"},
                  plot_kws=dict(edgecolor="black", linewidth=0.5))
fig = pp.fig
fig.subplots_adjust(top=0.93, wspace=0.3)
```

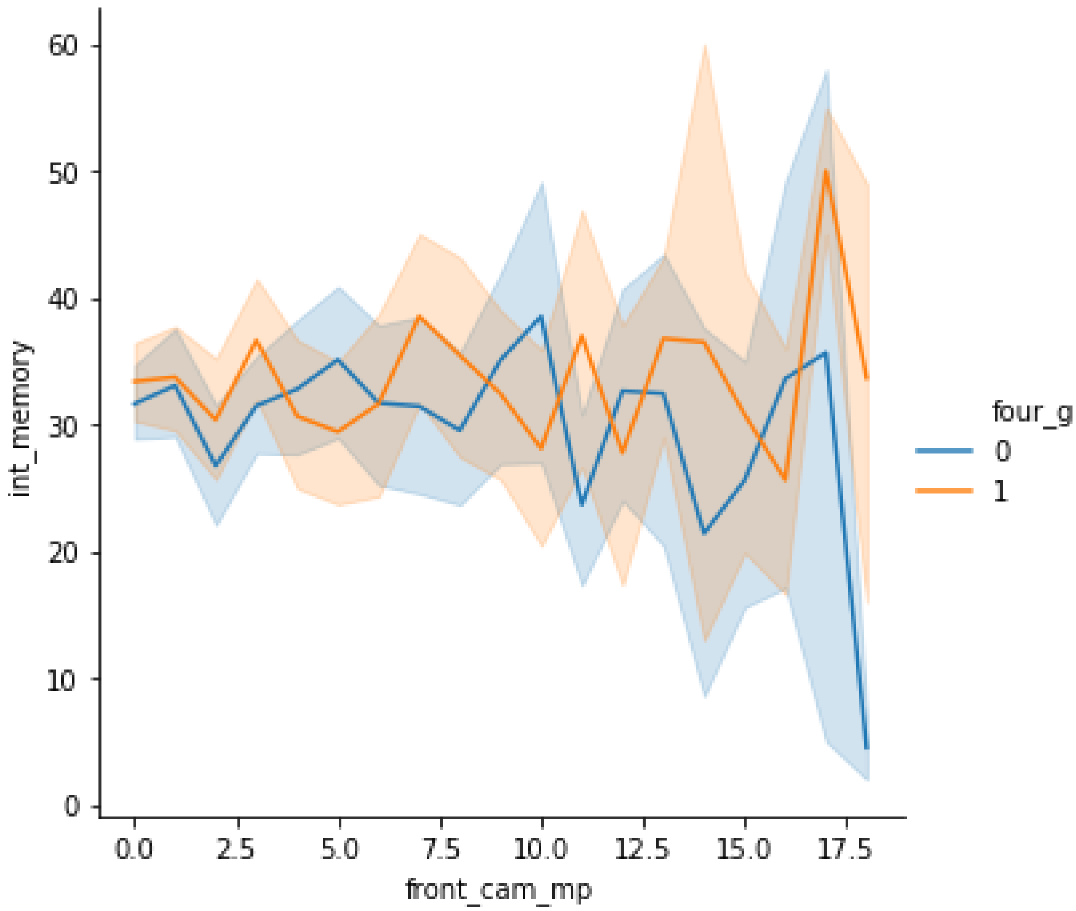
```
t = fig.suptitle('Wine Attributes Pairwise Plots', fontsize=14)
```



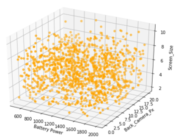
Secondly, we plot a relationship plot for `int_memory` vs. `four_g` vs. `front_cam_mp`. Here we can see that as front camera megapixels increases, internal memory of phones also tends to increase, presumably to cope with the multimedia possibilites whoch are opened with a good camera module.

[39]:
```
sns.relplot(y="int_memory", x="front_cam_mp", hue='four_g', kind="line", data=df)
```

[39]: <seaborn.axisgrid.FacetGrid at 0x7f31da8ef240>



[40]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt



fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(df['battery_power'], df['back_cam_mp'], df['screen_size'], c='orange')

ax.set_xlabel('Battery Power')
ax.set_ylabel('Back_Camera_Px')
ax.set_zlabel('Screen_Size')

plt.show()
```



6 Methodology

We are using classification task for this machine learning project. The 4 algorithms used to predit the models are: 1. K-Nearest Neigbors (commonly known as KNN) 2. Decision Tree 3. Random Forest 4. Support Vector Machine (also known as SVM)

Firstly,we have applied cross fold validation on the entire feature present in the dataset after pre-processing & found CV score as **'0.38'**. As the score seemed to be lower,we then decided to go for feature selection to see if any improvement in the accuracy. We applied `f-score` & `random forest importance` methods for feature selection and compared the accuracy of both. Finally we ended up with f-score as best features estimator and used it for further analysis.

We then applied the above mentioned algorithms on the best features given by f-score method and estimated the accuracy of all models. Also for each algorithm, we tuned the parameters and visualised to get the best accuracy score for corresponding model.

Lastly we evaluated the algorithms using the performance metrics and performance comparison using paired t-test

7 Feature selection

Feature selection is the process where you automatically select features which contribute most to your prediction variable. Sometimes having many features can decrease the accuracy of model.

1. Performance with full sets of features: We first accessed the performance using all the features of our data. We used `Stratified-K-fold` methods with `splits = 5` and `repetitions = 3` with scoring metric set to accuracy & lastly computed the result using `cross_val_score()`.

[41]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score,RepeatedStratifiedKFold

Data= df.drop(columns=['price_range'])
target = df[TARGET]
Data = preprocessing.MinMaxScaler().fit_transform(Data)

clf = KNeighborsClassifier(n_neighbors=1)
cv_method = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=786)
scoring_metric = 'accuracy'
cv_results_full = cross_val_score(estimator=clf, X=Data, y=target,
    cv=cv_method,scoring=scoring_metric)
```

```
cv_results_full.mean().round(2)
```

[41]: 0.38

With full set of features & 1 neigbor classifier, we achieved the accuracy score of **38%**.

2. Feature selection using f-score: F-score method selects the features based on relationship between descriptive feature and target feature using F-distribution. We now set number of features to 8. The `fs_indices_fscore` returns us top 8 features sorted highest to lowest.

[42]:
```
Data = df.drop(columns=['price_range'])
target = df[TARGET]
Data = preprocessing.MinMaxScaler().fit_transform(Data)
```

[43]:
```
from sklearn import feature_selection as fs
num_features = 8
fs_fit_fscore = fs.SelectKBest(fs.f_classif, k=num_features)
fs_fit_fscore.fit_transform(Data, target)
fs_indices_fscore = np.argsort(np.nan_to_num(fs_fit_fscore.scores_))[::-1][0:
 ↪num_features]
fs_indices_fscore
```

[43]:
```
array([ 9,  0,  6,  2, 14,  7, 10,  4])
```

[44]:
```
best_features_fscore = df.columns[fs_indices_fscore].values
best_features_fscore
```

[44]:
```
array(['ram', 'battery_power', 'int_memory', 'clock_speed', 'screen_size',
       'n_cores', 'talk_time', 'front_cam_mp'], dtype=object)
```

- We got `ram`, `battery_power`, `int_memory`, `clock_speed`, `screen_size`, `n_cores`, `talk_time` and `front_cam_mp` as best features based on F-score.

[45]:
```
feature_importances_fscore = fs_fit_fscore.scores_[fs_indices_fscore]
feature_importances_fscore
```

[45]:
```
array([2.16625306e+03, 2.01508823e+01, 3.42542997e+00, 3.07379301e+00,
       2.83311374e+00, 2.32208238e+00, 1.30146226e+00, 1.04149688e+00])
```

[46]:
```
import altair as alt

def plot_imp(best_features, scores, method_name, color):

    df = pd.DataFrame({'features': best_features,
                       'importances': scores})

    chart = alt.Chart(df,
                      width=500,
                      title=method_name + ' Feature Importances'
                     ).mark_bar(opacity=0.75,
                                color=color).encode(
```

```
        alt.X('features', title='Feature', sort=None, axis=alt.
↪AxisConfig(labelAngle=45)),
        alt.Y('importances', title='Importance')
    )

    return chart
```

- Plotting the best_features_fscores to visualised the feature importance.

[47]:
```
plot_imp(best_features_fscore, feature_importances_fscore, 'F-Score', 'red')
```

[47]:
```
<VegaLite 3 object>

If you see this message, it means the renderer has not been properly enabled
for the frontend that you are using. For more information, see
https://altair-viz.github.io/user_guide/troubleshooting.html
```

Accessing the performance of the selected features using cross validation.

[48]:
```
cv_results_fscore = cross_val_score(estimator=clf,
                             X=Data[:, fs_indices_fscore],
                             y=target,
                             cv=cv_method,
                             scoring=scoring_metric)
cv_results_fscore.mean().round(3)
```

[48]:
```
0.605
```

3. Feature selection using Random Forest Importance Random Forest importance (RFI) is widely used feature selector because of the accuracy, robustness and ease of use it gives. It tells us about how much accuracy is decreased when a variable is excluded and decrease in gini impurity when a variable is chosen to split node.

[49]:
```
Data= df.drop(columns=['price_range'])
target=df[TARGET]
Data=preprocessing.MinMaxScaler().fit_transform(Data)
```

[50]:
```
Data
```

[50]:
```
array([[0.72660428, 1.        , 0.        , ..., 1.        , 1.        ,
        0.77683957],
       [0.70320856, 0.        , 1.        , ..., 0.        , 0.        ,
        0.34499397],
       [0.76938503, 0.        , 0.        , ..., 1.        , 1.        ,
        0.67913148],
       ...,
       [0.40641711, 0.        , 0.5       , ..., 0.        , 1.        ,
        0.55367913],
       [0.43582888, 0.        , 0.5       , ..., 0.        , 1.        ,
```

```
 0.03498191],
[0.09291444, 1.        , 0.5       , ..., 1.        , 1.        ,
 0.16646562]])
```

[51]:
```
from sklearn.ensemble import RandomForestClassifier

model_rfi = RandomForestClassifier(n_estimators=100)
model_rfi.fit(Data, target)
fs_indices_rfi = np.argsort(model_rfi.feature_importances_)[::-1][0:num_features]
```

[52]:
```
best_features_rfi = df.columns[fs_indices_rfi].values
best_features_rfi
```

[52]:
```
array(['ram', 'battery_power', 'screen_size', 'int_memory', 'back_cam_mp',
       'talk_time', 'front_cam_mp', 'n_cores'], dtype=object)
```

- We got `ram`, `battery_power`, `screen_size`,`int_memory`, `talk_time`, `front_cam_mp`, `back_cam_mp` and `n_cores`,as best features based on random forest importance.

[53]:
```
feature_importances_rfi = model_rfi.feature_importances_[fs_indices_rfi]
feature_importances_rfi
```

[53]:
```
array([0.47525871, 0.10213975, 0.07127421, 0.06523768, 0.05342514,
       0.05248444, 0.04568845, 0.04036804])
```

- Plotting the `best_features_rfi` to visualise the feature importance

[54]:
```
plot_imp(best_features_rfi, feature_importances_rfi, 'Random Forest', 'green')
```

[54]:
```
<VegaLite 3 object>

If you see this message, it means the renderer has not been properly enabled
for the frontend that you are using. For more information, see
https://altair-viz.github.io/user_guide/troubleshooting.html
```

Accessing the performance of the selected features using cross validation.

[55]:
```
cv_results_rfi = cross_val_score(estimator=clf,
                             X=Data[:, fs_indices_rfi],
                             y=target,
                             cv=cv_method,
                             scoring=scoring_metric)
cv_results_rfi.mean().round(3)
```

[55]:
```
0.588
```

Finding the overall performance: We found that F-score feature selector gives us good accuracy score as compared to random forest importance.

Hence we choose `best_feature_f-score` for further fitting the model.

[56]:
```
print('Full Set of Features:', cv_results_full.mean().round(3))
print('F-Score:', cv_results_fscore.mean().round(3))
print('RFI:', cv_results_rfi.mean().round(3))
```

```
Full Set of Features: 0.383
F-Score: 0.605
RFI: 0.588
```

7.0.1 Splitting the data into training and test set

We have selected the sample(1200) of our entire data i.e(2000 rows) for model fitting and evaluation. We have split the data into `70 :30` ratio i.e 70% of our data to build a model and 30% data to test it to ensure that we measure the accuracy based on unseen data.

[57]:
```
Data = df[best_features_fscore].copy()
target = df[TARGET]
Data = preprocessing.MinMaxScaler().fit_transform(Data)
```

[58]:
```
from sklearn.model_selection import train_test_split

D_train, D_test, t_train, t_test = train_test_split(Data, target, test_size=0.3,␣
 ↪random_state=786)
```

8 Model fitting

8.1 1.K-Nearest Neighbor (KNN)

We fit a `KNeighborClassifier` with default parameter values as `n_neigbors = 5` and `P=2`. n_neigbors value is the number of neigbors to be used and `P=2` is the Euclidean distance metric. The score function returns the accuracy of classifier on the test data. Accuracy is ratio of total correctly predicted observations upon total number of observations. Computed accuracy found was 59.44%

[59]:
```
from sklearn.neighbors import KNeighborsClassifier
knn_classifier = KNeighborsClassifier(n_neighbors=5, p=2)
knn_classifier.fit(D_train, t_train)
knn_classifier.score(D_test, t_test)
```

[59]:
```
0.5944444444444444
```

8.2 Hyperparameter tuning using Grid Search

Grid-search is used to find the optimal hyperparameters of a model which results in the most *accurate* predictions.

Below we have defined a function for `grid search` to which we pass the classifier (KNN, DT, RF, and SVM) and training data. * We have defined different parameters for each algorithm in the `grid_params` method. * The function returns us best model parameters and model score based on the parameters given. * In addition we include repeated stratified cv method. * Also we tell sklean library which metric to optimize i.e. accuracy in our case.

[60]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

def grid_search(D_train, t_train, clf):

    if isinstance(clf, KNeighborsClassifier):
        grid_params = {
        'n_neighbors':[3, 5, 7, 9, 11, 13, 15],
        'p':[1, 2, 3]
        }
    elif isinstance(clf, DecisionTreeClassifier):
        grid_params = {
        'criterion':['gini','entropy'],
        'min_samples_split':[2, 3, 4],
        'max_depth':[1, 2, 3, 4, 5, 6, 7, 8]
        }
    elif isinstance(clf, RandomForestClassifier):
        grid_params = {
        'n_estimators':[110, 130, 150, 200],
        'criterion':['gini','entropy'],
        'min_samples_split':[2, 3, 4],
        'max_depth':[3, 4, 5]

        }
    elif isinstance(clf, SVC):
       grid_params = {
            'C':[1, 10, 50, 100],
            'gamma':[1, 0.1, 0.05, 0.001],
            'kernel':['rbf', 'poly', 'sigmoid']
        }
    else :
        raise ValueError("unkown classifier")

    gs = GridSearchCV(
        estimator = clf,
        param_grid = grid_params,
        verbose = 3,
        cv = cv_method,
        n_jobs = -1,
        refit = True
    )

    gs_results = gs.fit(D_train, t_train)
    p = gs_results.best_params_
    model = gs_results.best_estimator_
```

```
    return model, p, gs_results
```

- With `n_neigbors:[3, 5, 7, 9, 11, 13, 15]` and `P: [1, 2, 3]` the grid search function finds out the best parameter values and calculates the model score.

[61]:
```
knn_model, knn_best_estimate, knn_result = grid_search(D_train, t_train,␣
 ↪knn_classifier)
```

```
Fitting 15 folds for each of 21 candidates, totalling 315 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  92 tasks      | elapsed:    5.9s
[Parallel(n_jobs=-1)]: Done 315 out of 315 | elapsed:   20.9s finished
```

[62]:
```
knn_best_estimate
```

[62]:
```
{'n_neighbors': 15, 'p': 1}
```

[63]:
```
knn_model.score(D_test, t_test)
```

[63]:
```
0.6888888888888889
```

- KNN classifier with `n_neighbor = 15` and `p = 1` predicted the model mean score of 68.8%

[64]:
```
results_KNN = pd.DataFrame(knn_result.cv_results_['params'])
results_KNN['test_score'] = knn_result.cv_results_['mean_test_score']
results_KNN.head()
```

[64]:
```
   n_neighbors  p  test_score
0            3  1    0.585714
1            3  2    0.568651
2            3  3    0.552381
3            5  1    0.638095
4            5  2    0.600000
```

[65]:
```
results_KNN['metric'] = results_KNN['p'].replace([1, 2, 3], ["Manhattan",␣
 ↪"Euclidean", "Minkowski"])
results_KNN.head()
```

[65]:
```
   n_neighbors  p  test_score     metric
0            3  1    0.585714  Manhattan
1            3  2    0.568651  Euclidean
2            3  3    0.552381  Minkowski
3            5  1    0.638095  Manhattan
4            5  2    0.600000  Euclidean
```

8.2.1 Plotting the KNN Performance comparison.

We know visualise the hyper parameter tuning results from cross fold validation. We plot using altair module. The plot shows that at all values of `K` with Manhattan distance `p=1` outperforms others.

[66]:
```
import altair as alt

alt.Chart(results_KNN,
          title='KNN Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('n_neighbors', title='Number of Neighbors'),
    alt.Y('test_score', title='Mean CV Score', scale=alt.Scale(zero=False)),
    color='metric'
)
```

[66]:
```
<VegaLite 3 object>

If you see this message, it means the renderer has not been properly enabled
for the frontend that you are using. For more information, see
https://altair-viz.github.io/user_guide/troubleshooting.html
```

Advantages of KNN Classifier:

- The algorithm is simple and easy to implement.
- The algorithm is versatile and can be used for classification, regression and search.

Disadvantages:

- The algorithm gets slower as number of independent variables increases where predictions needs to be made rapidly.

Limitations:

- Need to have high computing resources to speedly handle the data.

8.3 2.Decision Tree Clasification

Decision trees are non-parametric supervised learning methods used for classification. The main aim of this is to define a model that gives value of target feature by learning decision rule inferred from data features. Fitting the decision tree classifier with default values and `random state = 786` which was selected at the very beginning. The score function returns the accuracy of classifier on the test data. Accuracy is ratio of total correctly predicted observations upon total number of observations.The accuracy measured was 76.38%.

[67]:
```
dt_classifier = DecisionTreeClassifier(random_state=786)
dt_classifier.fit(D_train, t_train)
dt_classifier.score(D_test, t_test)
```

[67]:
```
0.7638888888888888
```

- Out of Criterion = `gini, entropy`,`min_sample_split = [2, 3, 4]` & `max_depth = [1, 2, 3, 4, 5, 6, 7, 8]` the grid search function finds out the best parameter values and calculates the model score.

[68]:
```
dt_model, dt_best_estimate, dt_result = grid_search(D_train, t_train,␣
↪dt_classifier)
```

```
Fitting 15 folds for each of 48 candidates, totalling 720 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done 720 out of 720 | elapsed:    2.3s finished
```

[69]:
```
dt_best_estimate
```

[69]:
```
{'criterion': 'gini', 'max_depth': 4, 'min_samples_split': 2}
```

[70]:
```
dt_model.score(D_test, t_test)
```

[70]:
```
0.7666666666666667
```

With Criterion `gini` , `max_depth = 4` and `min_sample_splits` of 2 the model predicts the accuracy 76.6%

[71]:
```
results_DT = pd.DataFrame(dt_result.cv_results_['params'])
results_DT['test_score'] = dt_result.cv_results_['mean_test_score']
results_DT.head()
```

[71]:

	criterion	max_depth	min_samples_split	test_score
0	gini	1	2	0.483730
1	gini	1	3	0.483730
2	gini	1	4	0.483730
3	gini	2	2	0.779365
4	gini	2	3	0.779365

8.3.1 Plotting the DT Performance Comparison

Also from the plot we visualise the best hyperparamters as `gini` and `max_depth:4`

[72]:
```
alt.Chart(results_DT,
          title='DT Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('max_depth', title='Maximum Depth'),
    alt.Y('test_score', title='Mean CV Score', aggregate='average', scale=alt.
↪Scale(zero=False)),
    color='criterion'
)
```

[72]:
```
<VegaLite 3 object>

If you see this message, it means the renderer has not been properly enabled
for the frontend that you are using. For more information, see
https://altair-viz.github.io/user_guide/troubleshooting.html
```

Advantages of Decision tree classifier

- Inexpensive to construct.
- Easy to interpret for small size trees.
- Fast at classifying unknown records.

Disadvantages

- Decision tree models are often biased towards splits on features.
- Large trees can be difficult to interpret.
- Small change in training data can account for large change to decision logic.

8.4 3.Random Forest classifier

A random forest is a Meta estimator that fits number of decision tree classifier on various sub-samples and uses mean to advance the accuracy and avoid over-fitting. Fitting the random forest classifier with default estimator `n = 100` i.e. number of trees in the forest, criterion `gini` and `max_depth` 2.

The score function returns the accuracy of classifier on the test data. Accuracy is ratio of total correctly predicted observations upon total number of observations. The accuracy measured was *73.6%*.

[73]:
```
rf_classifier =␣
↪RandomForestClassifier(random_state=786,n_estimators=100,max_depth=2,criterion='gini')
rf_classifier.fit(D_train, t_train)
rf_classifier.score(D_test, t_test)
```

[73]:
```
0.7361111111111112
```

- Out of the given parameters given to grid search function `criterion = [gini, entropy]`, `n_estimators = [110, 130, 150, 200]`, `max_depth = [3, 4, 5]` and `min_sample_split = [2, 3, 4]` it calculates & returns best parameters with model score.

[74]:
```
rf_model, rf_best_estimate, rf_result = grid_search(D_train, t_train,␣
↪rf_classifier)
```

```
Fitting 15 folds for each of 72 candidates, totalling 1080 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  52 tasks      | elapsed:    5.1s
[Parallel(n_jobs=-1)]: Done 244 tasks      | elapsed:   26.0s
[Parallel(n_jobs=-1)]: Done 564 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 1012 tasks      | elapsed:  2.1min
[Parallel(n_jobs=-1)]: Done 1080 out of 1080 | elapsed:  2.3min finished
```

[75]:
```
rf_best_estimate
```

[75]:
```
{'criterion': 'gini',
 'max_depth': 5,
 'min_samples_split': 3,
```

```
 'n_estimators': 200}
```

[76]:
```
rf_model.score(D_test, t_test)
```

[76]:
```
0.7777777777777778
```

- The model predicts the accuracy score of 76.9%.*

[77]:
```
results_RF = pd.DataFrame(rf_result.cv_results_['params'])
results_RF['test_score'] = rf_result.cv_results_['mean_test_score']
results_RF.head()
```

[77]:

	criterion	max_depth	min_samples_split	n_estimators	test_score
0	gini	3	2	110	0.784921
1	gini	3	2	130	0.783730
2	gini	3	2	150	0.784921
3	gini	3	2	200	0.779365
4	gini	3	3	110	0.786111

8.4.1 Plotting the RF Performance Comparison

From the plot we visualise that at `max_depth = 4` , `gini` overpowers `entropy`.

[78]:
```
alt.Chart(results_RF,
          title='RF Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('max_depth', title='Maximum Depth'),
    alt.Y('test_score', title='Mean CV Score', aggregate='average', scale=alt.
 ↪Scale(zero=False)),
    color='criterion')
```

[78]:
```
<VegaLite 3 object>

If you see this message, it means the renderer has not been properly enabled
for the frontend that you are using. For more information, see
https://altair-viz.github.io/user_guide/troubleshooting.html
```

Advantages of Random forest classifier

- No need of any feature selection
- Easier to make parallel models
- If larger parts of features are lost , accuracy can still be maintained.

Disadvantages

- Fits for some noisy data
- Time complexity- much harder and time consuming to construct.

Limitations

- Heavy computation resources.

8.5 4.Support Vector Machine classifier

SVM is linear model for classification problem. The idea of SVM is simple. The algorithm creates a line or hyperplane which separates the data into classes. We fit the model with default kernel as rbf and regularisation value=1.0 parameters . The score function returns the accuracy of classifier on the test data. Accuracy is ratio of total correctly predicted observations upon total number of observations. The accuracy measured was 78.6%.

[79]:
```
svm_classifier = SVC()
svm_classifier.fit(D_train, t_train)
svm_classifier.score(D_test, t_test)
```

[79]:
```
0.7861111111111111
```

- The parameters passed to grid search function were gamma `values=[0.1, 0.05, 0.001, 1]` as the value must be between 0.1 to 1. `Kernels=[rbf, poly, sigmoid]` with `C=[1, 10, 50, 100]`.

[80]:
```
svm_model, svm_best_estimate, svm_result = grid_search(D_train, t_train,␣
↪svm_classifier)
```

```
Fitting 15 folds for each of 48 candidates, totalling 720 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 tasks      | elapsed:    2.6s
[Parallel(n_jobs=-1)]: Done 484 tasks      | elapsed:   13.5s
[Parallel(n_jobs=-1)]: Done 720 out of 720 | elapsed:   20.7s finished
```

[81]:
```
svm_best_estimate
```

[81]:
```
{'C': 50, 'gamma': 0.05, 'kernel': 'sigmoid'}
```

[82]:
```
svm_model.score(D_train, t_train)
```

[82]:
```
0.8345238095238096
```

The model predicts the accuracy score of 83.4% with best parameters .

[83]:
```
results_SVM = pd.DataFrame(svm_result.cv_results_['params'])
results_SVM['test_score'] = svm_result.cv_results_['mean_test_score']
results_SVM.head()
```

[83]:

	C	gamma	kernel	test_score
0	1	1.0	rbf	0.795238
1	1	1.0	poly	0.784524
2	1	1.0	sigmoid	0.172619
3	1	0.1	rbf	0.790873
4	1	0.1	poly	0.286905

8.5.1 Plotting the SVM Performance Comparison

From the plot we visualise that at max_depth =4 , gini overpowers entropy .

[84]:
```
alt.Chart(results_SVM,
          title='SVM Performance Comparison'
         ).mark_line(point=True).encode(
    alt.X('C', title='Regularisation Parameter'),
    alt.Y('test_score', title='Mean CV Score', aggregate='average', scale=alt.
 ↪Scale(zero=False)),
    color='kernel'
)
```

[84]:
```
<VegaLite 3 object>

If you see this message, it means the renderer has not been properly enabled
for the frontend that you are using. For more information, see
https://altair-viz.github.io/user_guide/troubleshooting.html
```

Advantages of Support Vector Machine

- Work well when there is clean margin of separation.
- Memory efficient #### Disadvantages
- Not suitable for larger data sets
- SVM does not perform well when data set has more noise or target class is overlapping.

8.6 Performance comparison

After testing the classifier by considering the train data and using it in cross validation way, we know perform the paired t-test in order to understand if the difference between performance is statistically significant for any 2 classifiers. Firstly we calculate the cross_val_score and then compare it with all models as: * KNN-DT * KNN-RF * KNN-SVM * DT-RF * DT-SVM * RF_SVM

From scipy library we import the stats module to run the t-test .

[85]:
```
from sklearn.model_selection import cross_val_score, StratifiedKFold

cv_method_ttest = StratifiedKFold(n_splits=10, random_state=786)
cv_results_KNN = cross_val_score(estimator=knn_model, X=Data, y=target,␣
 ↪cv=cv_method_ttest, n_jobs=-1, scoring='accuracy')
```

[86]:
```
cv_results_KNN.mean()
```

[86]:
```
0.7000298631849434
```

[87]:
```
cv_results_RF = cross_val_score(estimator=rf_model, X=Data, y=target,␣
 ↪cv=cv_method_ttest, n_jobs=-1, scoring='accuracy')
cv_results_RF.mean()
```

[87]:
```
0.7949359909252958
```

[88]:
```
cv_results_DT = cross_val_score(estimator=dt_model, X=Data, y=target,␣
 ↪cv=cv_method_ttest, n_jobs=-1, scoring='accuracy')
cv_results_DT.mean()
```

[88]: 0.7915683380790333

[89]:
```
cv_results_SVM = cross_val_score(estimator=svm_model, X=Data, y=target,␣
 ↪cv=cv_method_ttest, n_jobs=-1, scoring='accuracy')
cv_results_SVM.mean()
```

[89]: 0.823305264254462

[90]:
```
from scipy import stats

print(stats.ttest_rel(cv_results_KNN, cv_results_DT))
print(stats.ttest_rel(cv_results_KNN, cv_results_RF))
print(stats.ttest_rel(cv_results_KNN, cv_results_SVM))

print(stats.ttest_rel(cv_results_DT, cv_results_RF))
print(stats.ttest_rel(cv_results_DT, cv_results_SVM))

print(stats.ttest_rel(cv_results_RF, cv_results_SVM))
```

```
Ttest_relResult(statistic=-5.175690628053786, pvalue=0.0005827033357749185)
Ttest_relResult(statistic=-4.179672453115473, pvalue=0.002377244384685168)
Ttest_relResult(statistic=-6.222381065086182, pvalue=0.00015465915456455202)
Ttest_relResult(statistic=-0.3264812706568976, pvalue=0.7515244765497536)
Ttest_relResult(statistic=-3.0075809721373896, pvalue=0.014773665419268186)
Ttest_relResult(statistic=-4.532049993566178, pvalue=0.001422013597798 2204)
```

The Pair KNN-SVM gives statistically significant value of 0.0002 which is less than 0.05.

9 Model evaluation

Model evaluation is one of the important step required to determine the best model, how well the model will perform. The target variable for our dataset was multinomial. That is target feature is categorical with 4 different level {0, 1, 2, 3}. It refers to different price range ={'low', 'mid', 'high', 'v high'}. Hence we cannot use binary metric such as roc_auc curve to evaluate multinomial classifier. Below are the evaluation metrics used to find the accuracy, classification report and average model accuracy for each model.

[91]:
```
from sklearn import metrics
def print_model_stats(model, D_test, t_test):
    pred = model.predict(D_test)
    print("=========={model_name} Model Statistics=============".
 ↪format(model_name=model.__class__.__name__))
    print("Accuracy score:", metrics.accuracy_score(t_test, pred))
    print("Confusion Matrix:\n", metrics.confusion_matrix(t_test, pred))
    print("Classification report:\n", metrics.classification_report(t_test,␣
 ↪pred))
```

```
    print("Average model accuracy:", metrics.balanced_accuracy_score(t_test,␣
 ↪pred))
```

[92]:
```
print_model_stats(knn_model, D_test, t_test)
```

```
==========KNeighborsClassifier Model Statistics=============
Accuracy score: 0.6888888888888889
Confusion Matrix:
 [[71 12  0  0]
 [18 57 16  0]
 [ 1 29 51  8]
 [ 0  2 26 69]]
Classification report:
               precision    recall  f1-score   support

           0       0.79      0.86      0.82        83
           1       0.57      0.63      0.60        91
           2       0.55      0.57      0.56        89
           3       0.90      0.71      0.79        97

   micro avg       0.69      0.69      0.69       360
   macro avg       0.70      0.69      0.69       360
weighted avg       0.70      0.69      0.69       360

Average model accuracy: 0.6915423067928375
```

[93]:
```
print_model_stats(dt_model, D_test, t_test)
```

```
==========DecisionTreeClassifier Model Statistics=============
Accuracy score: 0.7666666666666667
Confusion Matrix:
 [[75  8  0  0]
 [17 53 21  0]
 [ 1  9 64 15]
 [ 0  0 13 84]]
Classification report:
               precision    recall  f1-score   support

           0       0.81      0.90      0.85        83
           1       0.76      0.58      0.66        91
           2       0.65      0.72      0.68        89
           3       0.85      0.87      0.86        97

   micro avg       0.77      0.77      0.77       360
   macro avg       0.77      0.77      0.76       360
weighted avg       0.77      0.77      0.76       360

Average model accuracy: 0.7677781363219282
```

[94]:
```
print_model_stats(rf_model, D_test, t_test)
```

```
==========RandomForestClassifier Model Statistics=============
Accuracy score: 0.7777777777777778
Confusion Matrix:
 [[75  8  0  0]
 [10 65 16  0]
 [ 0 19 62  8]
 [ 0  0 19 78]]
Classification report:
               precision    recall  f1-score   support

           0       0.88      0.90      0.89        83
           1       0.71      0.71      0.71        91
           2       0.64      0.70      0.67        89
           3       0.91      0.80      0.85        97

   micro avg       0.78      0.78      0.78       360
   macro avg       0.78      0.78      0.78       360
weighted avg       0.78      0.78      0.78       360

Average model accuracy: 0.779663274235098
```

[95]:
```
print_model_stats(svm_model, D_test, t_test)
```

```
==========SVC Model Statistics=============
Accuracy score: 0.8305555555555556
Confusion Matrix:
 [[78  5  0  0]
 [ 6 71 14  0]
 [ 0 15 65  9]
 [ 0  0 12 85]]
Classification report:
               precision    recall  f1-score   support

           0       0.93      0.94      0.93        83
           1       0.78      0.78      0.78        91
           2       0.71      0.73      0.72        89
           3       0.90      0.88      0.89        97

   micro avg       0.83      0.83      0.83       360
   macro avg       0.83      0.83      0.83       360
weighted avg       0.83      0.83      0.83       360

Average model accuracy: 0.8316511387024647
```

Hence we conclude that SVM gives us the best model accuracy and should be used for this predicting target feature.

10 Summary and Conclusion

After cleaning and visualizing the dataset, we were able to find clear pattern for features like `ram` which was proportional to the `price_range`, whereas some features had a very little information gain like `bluetooth` and `wifi`. We also noticed a pattern in `screen-size` for phones, had 2 distinct sizes which were manufactured the most, which aligned with the popular screen sizes provided by big brands(like Apple's iPhone).

The case study was to predict the cell phone price based on the descriptive features. We have successfully built a model based on the parameters given by grid search. That is we have fine-tuned the parameters and the best ones were applied to the model to train the data. The model was then tested and accuracy was computed for each algorithms. Out of 4, SVM gave us best accuracy with 84%.

Also we performed statistically significant ttest to determine if any difference between performance of any two classifier and we got KNN-SVM results as significant. Last but not the least, we used method evaluation techniques to verify the accuracy for multinomial classifier and it gave the same results.

There were also some limitations. The f-score method does not reveal information among the features but still we have used due to greater score than random forest importance.

Also we used few cases for feature selection and parameter tuning .we could have explored more taken more parameters and more feature selection methods. This might had helped us giving a better model.

11 References

- Sharma, A. (2018). Mobile Price Classification [online]. Retrieved from https://www.kaggle.com/iabhishekofficial/mobile-price-classification
- Vural, A. (2019). Feature Ranking [online]. Retrieved from http://www.featureranking.com
- Pupale, R. (2018). Support Vector Machines(SVM) — An Overview [online]. Retrieved from https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989
- https://www.apple.com/iphone-se/specs/
- https://www.apple.com/iphone-11-pro/specs/