Case Study : Mobile Price Range Prediction

This case study will help us to understand the stages in the data science project lifecycle with mobile prices data set to predict the price range for a unknown mobile. We will focus on the following stages namely -

- Business Understanding
- Data Acquisition
- Data Preparation
- Feature Engineering
- Feature Subset Selection
- Model Training
- Model Evaluation

1. Business Understanding

"TeraPhone" is multi-national brand in mobile manufacturing space, planning to enter the Indian market. They are planning to launch a searies of mobile handsets for Indian markets. They are doing a market survay for identifying the range of the products those are catering to the mobile handset needs of Indian folks. While doing so they have gathered a lot of information about the mobiles which are easily available in the markets. As they gathered data about 2000 mobile instruments present in the market, they are failing to identify the significant factors of mobile device which are making impact on the minds of the customers.

To fill this gap in the analysis, they have hired you so that you can help them to identify the various features of the mobile phones which are having quite a lot impact on the prices of the handsets. Using the knowledge of the various feature selection methods, you are going to list down the three significant factors that "TeraPhone" must take into consideration while determining the prices for theie new range of mobile devices.

As we are talking about the prediciton of the price range, this turns to be a classification problem as the price ranges can be seen as the discrete, finite set of values.

Classification can be of two types:

- Binary Classification : Predicts either of the two given classes. For example: identifying loan will be approved or not, student will take admission or not, customer will buy or not
- Multiclass Classification : Classify the data into more than two discrete classes. For example: identifying what customer is going to buy whether book, electronic item or appearals, classifying the customers into high , middle or low income ranges etc.

In the quick conversation with the "TeraPhone" marketing team, its revealed out that the price ranges to be considered are "Low", "Medium", "High" and "Very High". Looking at these class labels, this turns down to be multiclass classification problem. But actually the mobile prices are given, hence some preprocessing is required to convert them into the above mentioned categories.

In the conversation, following factors are listed out for which data is available -

- id : Unique Identifier for mobile device
- battery_power : Total energy a battery can store in one time measured in mAh
- blue : Has bluetooth or not
- clock_speed : speed at which microprocessor executes instructions
- dual_sim : Has dual sim support or not
- fc : Front Camera mega pixels
- four_g : Has 4G or not
- int_memory : Internal Memory in Gigabytes
- m_dep : Mobile Depth in cm
- mobile_wt : Weight of mobile phone
- n_cores : Number of cores of processor
- pc : Primary Camera mega pixels
- px_height : Pixel Resolution Height
- px_width : Pixel Resolution Width
- 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
- three_g : Has 3G or not
- touch_screen : Has touch screen or not
- wifi : Has wifi or not
- price : Actual market price of the device

Dataset deails can be found here

2. Data Acquisition

It's time to get access to the actual data and have initial look at the structure of the dataset.

2.1 Package Imports

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

2.2 Reading data from Mobiles Datasets

In [2]:

```
train_data = pd.read_csv("mobile_train_data.csv")
print("Data Imported!")
```

Data Imported!

Lets retain the original dataset as it is and work on the copy of it. Also have a quick look at the attributes of the data.

In [3]:

```
data = train_data
```

2.3 Confirm the imports

In [4]:

```
data.head()
```

Out[4]:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt	r
0	842	0	2.2	0	1	0	7	0.6	188	
1	1021	1	0.5	1	0	1	53	0.7	136	
2	563	1	0.5	1	2	1	41	0.9	145	
3	615	1	2.5	0	0	0	10	0.8	131	
4	1821	1	1.2	0	13	1	44	0.6	141	

5 rows × 21 columns

In [5]:

```
data.shape
```

Out[5]:

(2000, 21)

2000 mobile devices data is captured along with the 21 interesting characteristics!

Lets have a quick look at the columns and their respective data types.

In [6]:

```
data.columns
```

Out[6]:

```
Index(['battery_power', 'blue', 'clock_speed', 'dual_sim', 'fc', 'fo
ur_g',
       'int_memory', 'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_heig
ht',
       'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'three_g',
       'touch_screen', 'wifi', 'price'],
      dtype='object')
```

In [7]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   battery_power  2000 non-null   int64  
 1   blue           2000 non-null   int64  
 2   clock_speed    2000 non-null   float64
 3   dual_sim       2000 non-null   int64  
 4   fc             2000 non-null   int64  
 5   four_g         2000 non-null   int64  
 6   int_memory     2000 non-null   int64  
 7   m_dep          2000 non-null   float64
 8   mobile_wt      2000 non-null   int64  
 9   n_cores        2000 non-null   int64  
 10  pc             2000 non-null   int64  
 11  px_height      2000 non-null   int64  
 12  px_width       2000 non-null   int64  
 13  ram            2000 non-null   int64  
 14  sc_h           2000 non-null   int64  
 15  sc_w           2000 non-null   int64  
 16  talk_time      2000 non-null   int64  
 17  three_g        2000 non-null   int64  
 18  touch_screen   2000 non-null   int64  
 19  wifi           2000 non-null   int64  
 20  price          2000 non-null   int64  
dtypes: float64(2), int64(19)
memory usage: 328.2 KB
```

All the columns are numeric in nature

3. Data Preparation

3.1 Checking for unique data values

Lets define a fuction that will give us a report about the unique values of data for each attribute.

In [8]:

```
def show_unique_values(data_frame):
    print("Unique value for dataset attributes :\n")
    for column in data_frame.columns:
        print(column, " " ,data_frame[column].unique(), "\n")
```

In [9]:

```
show_unique_values(data)
```

```
Unique value for dataset attributes :

battery_power   [ 842 1021  563 ... 1139 1467  858]

blue   [0 1]

clock_speed   [2.2 0.5 2.5 1.2 1.7 0.6 2.9 2.8 2.1 1.  0.9 1.1 2.6
1.4 1.6 2.7 1.3 2.3
 2.  1.8 3.  1.5 1.9 2.4 0.8 0.7]

dual_sim   [0 1]

fc   [ 1  0  2 13  3  4  5  7 11 12 16  6 15  8  9 10 18 17 14 19]

four_g   [0 1]

int_memory   [ 7 53 41 10 44 22 24  9 33 17 52 46 13 23 49 19 39 47
38  8 57 51 21  5
 60 61  6 11 50 34 20 27 42 40 64 14 63 43 16 48 12 55 36 30 45 29 5
8 25
  3 54 15 37 31 32  4 18  2 56 26 35 59 28 62]

m_dep   [0.6 0.7 0.9 0.8 0.1 0.5 1.  0.3 0.4 0.2]

mobile_wt   [188 136 145 131 141 164 139 187 174  93 182 177 159 198
185 196 121 101
  81 156 199 114 111 132 143  96 200  88 150 107 100 157 160 119  87
152
 166 110 118 162 127 109 102 104 148 180 128 134 144 168 155 165  80
138
 142  90 197 172 116  85 163 178 171 103  83 140 194 146 192 106 135
153
  89  82 130 189 181  99 184 195 108 133 179 147 137 190 176  84  97
124
 183 113  92  95 151 117  94 173 105 115  91 112 123 129 154 191 175
86
  98 125 126 158 170 161 193 169 120 149 186 122 167]

n_cores   [2 3 5 6 1 8 4 7]

pc   [ 2  6  9 14  7 10  0 15  1 18 17 11 16  4 20 13  3 19  8  5 1
2]

px_height   [  20  905 1263 ...  528  915  483]

px_width   [ 756 1988 1716 ...  743 1890 1632]

ram   [2549 2631 2603 ... 2032 3057 3919]

sc_h   [ 9 17 11 16  8 13 19  5 14 18  7 10 12  6 15]

sc_w   [ 7  3  2  8  1 10  9  0 15 13  5 11  4 12  6 17 14 16 18]

talk_time   [19  7  9 11 15 10 18  5 20 12 13  2  4  3 16  6 14 17
8]

three_g   [0 1]

touch_screen   [0 1]

wifi   [1 0]
```

```
price    [11805 40303  7135 ...  5795 30699 31952]
```

As all the columns are numeric in nature, so the values are continuous.

3.2 Missing Values imputation (Data Cleansing)

Lets see how the missing values can be replaced in the dataset. First check whereall the missing values are present.

Take a closer look at the actual missing value count. 'False' means cell has a value whereas 'True" means cell is missing value. Output the count for different attributes of dataframe.

In [10]:

```
def show_missing_values(data):
    missing_data = data.isnull()
    for column in missing_data.columns.values.tolist():
        print(column)
        print (missing_data[column].value_counts())
        print("")
```

In [11]:

```
show_missing_values(data)
```

```
battery_power
False    2000
Name: battery_power, dtype: int64

blue
False    2000
Name: blue, dtype: int64

clock_speed
False    2000
Name: clock_speed, dtype: int64

dual_sim
False    2000
Name: dual_sim, dtype: int64

fc
False    2000
Name: fc, dtype: int64

four_g
False    2000
Name: four_g, dtype: int64

int_memory
False    2000
Name: int_memory, dtype: int64

m_dep
False    2000
Name: m_dep, dtype: int64

mobile_wt
False    2000
Name: mobile_wt, dtype: int64

n_cores
False    2000
Name: n_cores, dtype: int64

pc
False    2000
Name: pc, dtype: int64

px_height
False    2000
Name: px_height, dtype: int64

px_width
False    2000
Name: px_width, dtype: int64

ram
False    2000
Name: ram, dtype: int64

sc_h
False    2000
Name: sc_h, dtype: int64

sc_w
```

```
False    2000
Name: sc_w, dtype: int64

talk_time
False    2000
Name: talk_time, dtype: int64

three_g
False    2000
Name: three_g, dtype: int64

touch_screen
False    2000
Name: touch_screen, dtype: int64

wifi
False    2000
Name: wifi, dtype: int64

price
False    2000
Name: price, dtype: int64
```

Lets cross verify the report.

In [12]:

```
data.isnull().sum()
```

Out[12]:

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

Surprisingly none of the values is missing. So no need to bother about it!

3.3 Data Discretization (Target only)

In [13]:

```
import seaborn as sns
sns.set_style('whitegrid')
```

As discussed earlier, the target needs to be only one of the values i.e. 'Low', 'Medium', 'High' and 'Very High'. But the dataset has actual price ranges present in it. So lets go ahead and apply this tranformation using the binning technique.

In [14]:

```
print("max", max(data["price"]))
print("min", min(data["price"]))
```

```
max 49999
min 3038
```

The price range is between 3038 to 49999.

In [15]:

```
fig, ax = plt.subplots()
train_data["price"].hist(color='#A9C5D3', edgecolor='black',
                         grid=False, bins=4)
ax.set_title('Mobile Price Histogram', fontsize=12)
ax.set_xlabel('Price', fontsize=12)
ax.set_ylabel('Frequency', fontsize=12)
```

Out[15]:

```
Text(0, 0.5, 'Frequency')
```

There seems to be four bins present in the dataset. Lets try to create 4 bins and label them.

In [16]:

```
group_names = ['Low', 'Medium', 'High', 'Very_High']
data['price-binned'] = pd.cut(data['price'], 4, labels=group_names)
data[['price','price-binned']].tail(10)
```

Out[16]:

	price	price-binned
1990	28817	High
1991	38301	Very_High
1992	25036	Medium
1993	18956	Medium
1994	9162	Low
1995	8053	Low
1996	47441	Very_High
1997	5795	Low
1998	30699	High
1999	31952	High

Lets confirm the categories present in the target variable.

In [17]:

```
data["price-binned"].unique()
```

Out[17]:

```
[Low, Very_High, Medium, High]
Categories (4, object): [Low < Medium < High < Very_High]
```

In [18]:

```
data.columns
```

Out[18]:

```
Index(['battery_power', 'blue', 'clock_speed', 'dual_sim', 'fc', 'fo
ur_g',
       'int_memory', 'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_heig
ht',
       'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'three_g',
       'touch_screen', 'wifi', 'price', 'price-binned'],
      dtype='object')
```

3.4 Column Reduction (Target only)

As we have converted the price into the categories, so lets get rid of it from the normalized dataset.

In [19]:

```
data = data.drop(['price'], axis=1)
```

In [20]:

```
data.columns
```

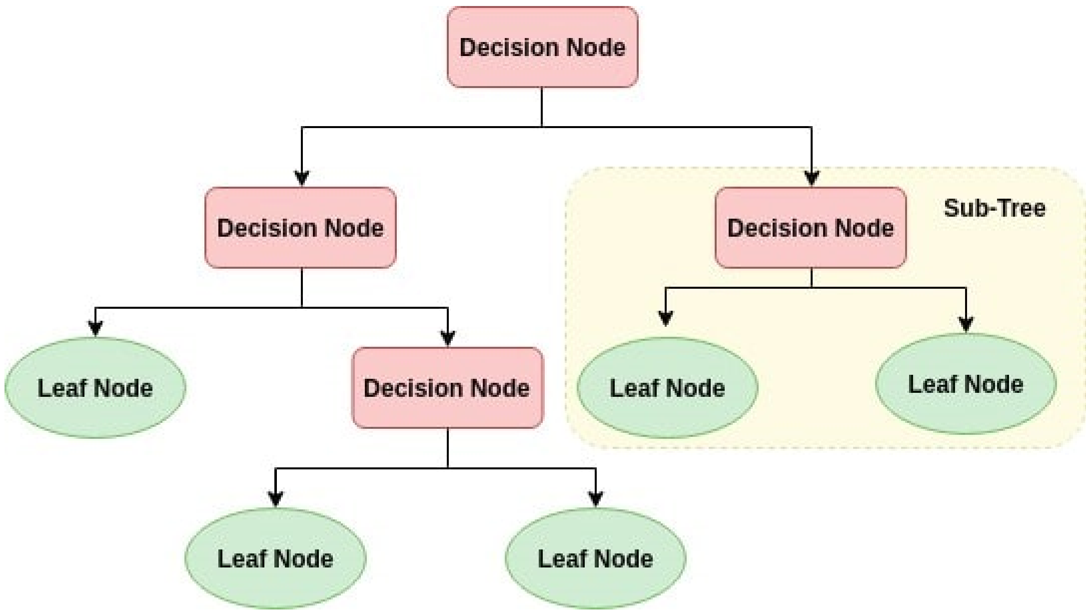
Out[20]:

```
Index(['battery_power', 'blue', 'clock_speed', 'dual_sim', 'fc', 'fo
ur_g',
       'int_memory', 'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_heig
ht',
       'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'three_g',
       'touch_screen', 'wifi', 'price-binned'],
      dtype='object')
```

Primer on Decision Tree Classification

Classification is a two-step process, learning step and prediction step. In the learning step, the model is developed based on given training data. In the prediction step, the model is used to predict the response for given data. Decision Tree is one of the easiest and popular classification algorithms to understand and interpret. It can be utilized for both classification and regression kind of problem.

A decision tree is a flowchart-like tree structure where an internal node represents feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome. The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in recursively manner call recursive partitioning. This flowchart-like structure helps you in decision making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.

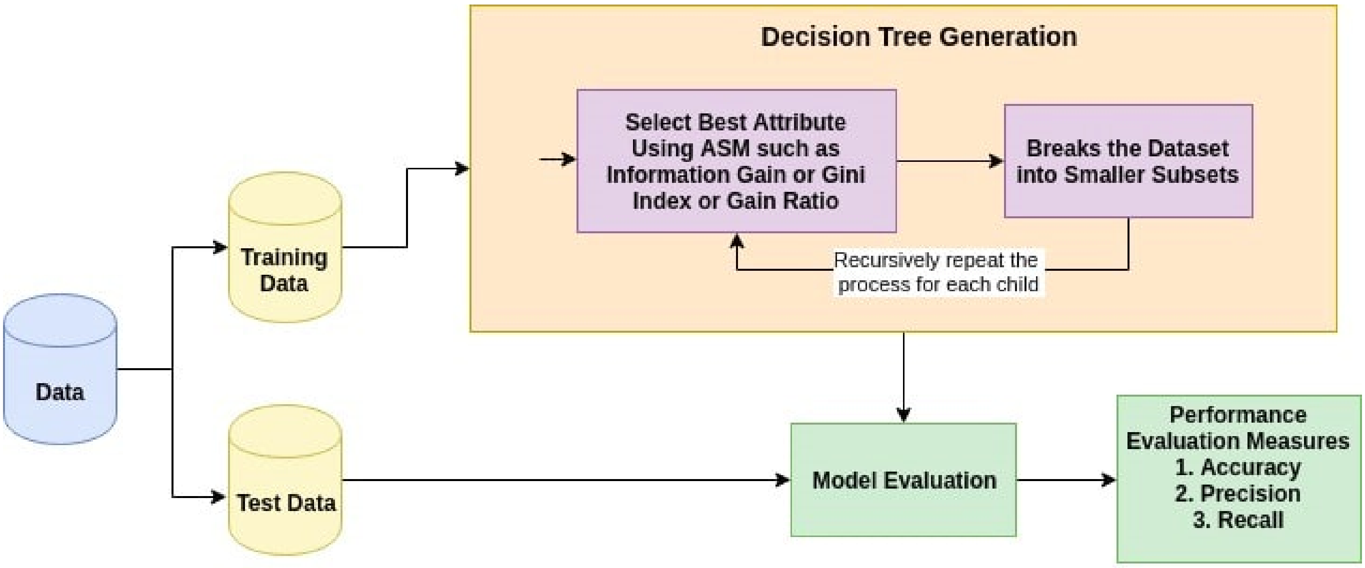


Decision Tree is a white box type of ML algorithm. It shares internal decision-making logic, which is not available in the black box type of algorithms such as Neural Network. Its training time is faster compared to the neural network algorithm. The time complexity of decision trees is a function of the number of records and number of attributes in the given data. Decision trees can handle high dimensional data with good accuracy.

The basic idea behind any decision tree algorithm is as follows:

1) Select the best attribute using Attribute Selection Measures(ASM) to split the records.
2) Make that attribute a decision node and breaks the dataset into smaller subsets.
3) Starts tree building by repeating this process recursively for each child until one of the condition will match:

- **All the tuples belong to the same attribute value.**
- **There are no more remaining attributes.**
- **There are no more instances.**



Attribute selection measure is a heuristic for selecting the splitting criterion that partition data into the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature(or attribute) by explaining the given dataset. Best score attribute will be selected as a splitting attribute (Source). In the case of a continuous-valued attribute, split points for branches also need to define. Most popular selection measures are

- **Information Gain**
- **Gain Ratio**
- **Gini Index.**

We will use following decistion tree for understanding feature selection process in more detail.

In [21]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

In [22]:

```
def prepare_decision_tree(data, show_matrix=False, show_accuracy=True, show_repo
rt=False, show_visual=False):
    # Split the data into independent and target attributes
    col_length = len(data.columns)
    X = data.iloc[:,0:col_length - 1]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    #Split the data into training and testing set
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =0.3)

    #Construct decision tree
    dt = DecisionTreeClassifier(random_state=100)
    dt.fit(X_train, y_train)

    #Use the decision tree for prediction on test data
    y_pred = dt.predict(X_test)

    #Prepare the confusion matrix
    actuals = np.array(y_test)
    predictions = np.array(y_pred)

    if show_matrix:
        print("Confusion Matrix : ")
        print(confusion_matrix(actuals, predictions), "\n")

    #Compute accuracy
    if show_accuracy:
        print ("Accuracy : ", accuracy_score(y_test,y_pred)*100, "\n")

    #Generate classification report
    if show_report:
        print("Classification Report : \n", classification_report(y_test, y_pred
), "\n")

    #Show the important features visually
    if show_visual:
        importances=pd.Series(dt.feature_importances_, index=X.columns).sort_val
ues()
        importances.plot(kind='barh', figsize=(12,8))

    return dt
```

In [23]:

```
help(DecisionTreeClassifier)
```

```
Help on class DecisionTreeClassifier in module sklearn.tree._classe
s:

class DecisionTreeClassifier(sklearn.base.ClassifierMixin, BaseDecis
ionTree)
 |  DecisionTreeClassifier(*, criterion='gini', splitter='best', max
_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fra
ction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes
=None, min_impurity_decrease=0.0, min_impurity_split=None, class_wei
ght=None, presort='deprecated', ccp_alpha=0.0)
 |
 |  A decision tree classifier.
 |
 |  Read more in the :ref:`User Guide <tree>`.
 |
 |  Parameters
 |  ----------
 |  criterion : {"gini", "entropy"}, default="gini"
 |      The function to measure the quality of a split. Supported cr
iteria are
 |      "gini" for the Gini impurity and "entropy" for the informati
on gain.
 |
 |  splitter : {"best", "random"}, default="best"
 |      The strategy used to choose the split at each node. Supporte
d
 |      strategies are "best" to choose the best split and "random"
to choose
 |      the best random split.
 |
 |  max_depth : int, default=None
 |      The maximum depth of the tree. If None, then nodes are expan
ded until
 |      all leaves are pure or until all leaves contain less than
 |      min_samples_split samples.
 |
 |  min_samples_split : int or float, default=2
 |      The minimum number of samples required to split an internal
node:
 |
 |      - If int, then consider `min_samples_split` as the minimum n
umber.
 |      - If float, then `min_samples_split` is a fraction and
 |        `ceil(min_samples_split * n_samples)` are the minimum
 |        number of samples for each split.
 |
 |      .. versionchanged:: 0.18
 |         Added float values for fractions.
 |
 |  min_samples_leaf : int or float, default=1
 |      The minimum number of samples required to be at a leaf node.
 |      A split point at any depth will only be considered if it lea
ves at
 |      least ``min_samples_leaf`` training samples in each of the l
eft and
 |      right branches.  This may have the effect of smoothing the m
odel,
 |      especially in regression.
 |
 |      - If int, then consider `min_samples_leaf` as the minimum nu
mber.
```

```
 |      - If float, then `min_samples_leaf` is a fraction and
 |        `ceil(min_samples_leaf * n_samples)` are the minimum
 |        number of samples for each node.
 |
 |      .. versionchanged:: 0.18
 |         Added float values for fractions.
 |
 |  min_weight_fraction_leaf : float, default=0.0
 |      The minimum weighted fraction of the sum total of weights (o
f all
 |      the input samples) required to be at a leaf node. Samples ha
ve
 |      equal weight when sample_weight is not provided.
 |
 |  max_features : int, float or {"auto", "sqrt", "log2"}, default=N
one
 |      The number of features to consider when looking for the best
split:
 |
 |          - If int, then consider `max_features` features at each
split.
 |          - If float, then `max_features` is a fraction and
 |            `int(max_features * n_features)` features are consider
ed at each
 |            split.
 |          - If "auto", then `max_features=sqrt(n_features)`.
 |          - If "sqrt", then `max_features=sqrt(n_features)`.
 |          - If "log2", then `max_features=log2(n_features)`.
 |          - If None, then `max_features=n_features`.
 |
 |      Note: the search for a split does not stop until at least on
e
 |      valid partition of the node samples is found, even if it req
uires to
 |      effectively inspect more than ``max_features`` features.
 |
 |  random_state : int, RandomState instance, default=None
 |      Controls the randomness of the estimator. The features are a
lways
 |      randomly permuted at each split, even if ``splitter`` is set
to
 |      ``"best"``. When ``max_features < n_features``, the algorith
m will
 |      select ``max_features`` at random at each split before findi
ng the best
 |      split among them. But the best found split may vary across d
ifferent
 |      runs, even if ``max_features=n_features``. That is the case,
if the
 |      improvement of the criterion is identical for several splits
and one
 |      split has to be selected at random. To obtain a deterministi
c behaviour
 |      during fitting, ``random_state`` has to be fixed to an integ
er.
 |      See :term:`Glossary <random_state>` for details.
 |
 |  max_leaf_nodes : int, default=None
 |      Grow a tree with ``max_leaf_nodes`` in best-first fashion.
 |      Best nodes are defined as relative reduction in impurity.
 |      If None then unlimited number of leaf nodes.
```

```
 |
 |  min_impurity_decrease : float, default=0.0
 |      A node will be split if this split induces a decrease of the
impurity
 |      greater than or equal to this value.
 |
 |      The weighted impurity decrease equation is the following::
 |
 |          N_t / N * (impurity - N_t_R / N_t * right_impurity
 |                              - N_t_L / N_t * left_impurity)
 |
 |      where ``N`` is the total number of samples, ``N_t`` is the n
umber of
 |      samples at the current node, ``N_t_L`` is the number of samp
les in the
 |      left child, and ``N_t_R`` is the number of samples in the ri
ght child.
 |
 |      ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the wei
ghted sum,
 |      if ``sample_weight`` is passed.
 |
 |      .. versionadded:: 0.19
 |
 |  min_impurity_split : float, default=0
 |      Threshold for early stopping in tree growth. A node will spl
it
 |      if its impurity is above the threshold, otherwise it is a le
af.
 |
 |      .. deprecated:: 0.19
 |         ``min_impurity_split`` has been deprecated in favor of
 |         ``min_impurity_decrease`` in 0.19. The default value of
 |         ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23
and it
 |         will be removed in 0.25. Use ``min_impurity_decrease`` in
stead.
 |
 |  class_weight : dict, list of dict or "balanced", default=None
 |      Weights associated with classes in the form ``{class_label:
weight}``.
 |      If None, all classes are supposed to have weight one. For
 |      multi-output problems, a list of dicts can be provided in th
e same
 |      order as the columns of y.
 |
 |      Note that for multioutput (including multilabel) weights sho
uld be
 |      defined for each class of every column in its own dict. For
example,
 |      for four-class multilabel classification weights should be
 |      [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] ins
tead of
 |      [{1:1}, {2:5}, {3:1}, {4:1}].
 |
 |      The "balanced" mode uses the values of y to automatically ad
just
 |      weights inversely proportional to class frequencies in the i
nput data
 |      as ``n_samples / (n_classes * np.bincount(y))``
 |
```

```
 |      For multi-output, the weights of each column of y will be mu
ltiplied.
 |
 |      Note that these weights will be multiplied with sample_weigh
t (passed
 |      through the fit method) if sample_weight is specified.
 |
 |  presort : deprecated, default='deprecated'
 |      This parameter is deprecated and will be removed in v0.24.
 |
 |      .. deprecated:: 0.22
 |
 |  ccp_alpha : non-negative float, default=0.0
 |      Complexity parameter used for Minimal Cost-Complexity Prunin
g. The
 |      subtree with the largest cost complexity that is smaller tha
n
 |      ``ccp_alpha`` will be chosen. By default, no pruning is perf
ormed. See
 |      :ref:`minimal_cost_complexity_pruning` for details.
 |
 |      .. versionadded:: 0.22
 |
 |  Attributes
 |  ----------
 |  classes_ : ndarray of shape (n_classes,) or list of ndarray
 |      The classes labels (single output problem),
 |      or a list of arrays of class labels (multi-output problem).
 |
 |  feature_importances_ : ndarray of shape (n_features,)
 |      The impurity-based feature importances.
 |      The higher, the more important the feature.
 |      The importance of a feature is computed as the (normalized)
 |      total reduction of the criterion brought by that feature.  I
t is also
 |      known as the Gini importance [4]_.
 |
 |      Warning: impurity-based feature importances can be misleadin
g for
 |      high cardinality features (many unique values). See
 |      :func:`sklearn.inspection.permutation_importance` as an alte
rnative.
 |
 |  max_features_ : int
 |      The inferred value of max_features.
 |
 |  n_classes_ : int or list of int
 |      The number of classes (for single output problems),
 |      or a list containing the number of classes for each
 |      output (for multi-output problems).
 |
 |  n_features_ : int
 |      The number of features when ``fit`` is performed.
 |
 |  n_outputs_ : int
 |      The number of outputs when ``fit`` is performed.
 |
 |  tree_ : Tree
 |      The underlying Tree object. Please refer to
 |      ``help(sklearn.tree._tree.Tree)`` for attributes of Tree obj
ect and
```

```
 |      :ref:`sphx_glr_auto_examples_tree_plot_unveil_tree_structur
e.py`
 |      for basic usage of these attributes.
 |  
 |  See Also
 |  --------
 |  DecisionTreeRegressor : A decision tree regressor.
 |  
 |  Notes
 |  -----
 |  The default values for the parameters controlling the size of th
e trees
 |  (e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully g
rown and
 |  unpruned trees which can potentially be very large on some data
sets. To
 |  reduce memory consumption, the complexity and size of the trees
should be
 |  controlled by setting those parameter values.
 |  
 |  References
 |  ----------
 |  
 |  .. [1] https://en.wikipedia.org/wiki/Decision_tree_learning
 |  
 |  .. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classi
fication
 |         and Regression Trees", Wadsworth, Belmont, CA, 1984.
 |  
 |  .. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of St
atistical
 |         Learning", Springer, 2009.
 |  
 |  .. [4] L. Breiman, and A. Cutler, "Random Forests",
 |         https://www.stat.berkeley.edu/~breiman/RandomForests/cc_h
ome.htm
 |  
 |  Examples
 |  --------
 |  >>> from sklearn.datasets import load_iris
 |  >>> from sklearn.model_selection import cross_val_score
 |  >>> from sklearn.tree import DecisionTreeClassifier
 |  >>> clf = DecisionTreeClassifier(random_state=0)
 |  >>> iris = load_iris()
 |  >>> cross_val_score(clf, iris.data, iris.target, cv=10)
 |  ...                             # doctest: +SKIP
 |  ...
 |  array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
 |          0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
 |  
 |  Method resolution order:
 |      DecisionTreeClassifier
 |      sklearn.base.ClassifierMixin
 |      BaseDecisionTree
 |      sklearn.base.MultiOutputMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, *, criterion='gini', splitter='best', max_depth=N
```

```
one, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_le
af=0.0, max_features=None, random_state=None, max_leaf_nodes=None, m
in_impurity_decrease=0.0, min_impurity_split=None, class_weight=Non
e, presort='deprecated', ccp_alpha=0.0)
 |      Initialize self.  See help(type(self)) for accurate signatur
e.
 |
 |  fit(self, X, y, sample_weight=None, check_input=True, X_idx_sort
ed=None)
 |      Build a decision tree classifier from the training set (X,
y).
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The training input samples. Internally, it will be conve
rted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csc_matrix``.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_output
s)
 |          The target values (class labels) as integers or strings.
 |
 |      sample_weight : array-like of shape (n_samples,), default=No
ne
 |          Sample weights. If None, then samples are equally weight
ed. Splits
 |          that would create child nodes with net zero or negative
weight are
 |          ignored while searching for a split in each node. Splits
are also
 |          ignored if they would result in any single class carryin
g a
 |          negative weight in either child node.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      X_idx_sorted : array-like of shape (n_samples, n_features),
default=None
 |          The indexes of the sorted training input samples. If man
y tree
 |          are grown on the same dataset, this allows the ordering
to be
 |          cached between trees. If None, the data will be sorted h
ere.
 |          Don't use this parameter unless you know what to do.
 |
 |      Returns
 |      -------
 |      self : DecisionTreeClassifier
 |          Fitted estimator.
 |
 |  predict_log_proba(self, X)
 |      Predict class log-probabilities of the input samples X.
 |
 |      Parameters
 |      ----------
```

```
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      Returns
 |      -------
 |      proba : ndarray of shape (n_samples, n_classes) or list of n
_outputs             such arrays if n_outputs > 1
 |          The class log-probabilities of the input samples. The or
der of the
 |          classes corresponds to that in the attribute :term:`clas
ses_`.
 |
 |  predict_proba(self, X, check_input=True)
 |      Predict class probabilities of the input samples X.
 |
 |      The predicted class probability is the fraction of samples o
f the same
 |      class in a leaf.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      Returns
 |      -------
 |      proba : ndarray of shape (n_samples, n_classes) or list of n
_outputs             such arrays if n_outputs > 1
 |          The class probabilities of the input samples. The order
of the
 |          classes corresponds to that in the attribute :term:`clas
ses_`.
 |
 |  ----------------------------------------------------------------
------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------
------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample th
at
 |      each label set be correctly predicted.
 |
```

```
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_output
s)
 |          True labels for X.
 |
 |      sample_weight : array-like of shape (n_samples,), default=No
ne
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of self.predict(X) wrt. y.
 |
 |  ----------------------------------------------------------------
------
 |  Data descriptors inherited from sklearn.base.ClassifierMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------
------
 |  Methods inherited from BaseDecisionTree:
 |
 |  apply(self, X, check_input=True)
 |      Return the index of the leaf that each sample is predicted a
s.
 |
 |      .. versionadded:: 0.17
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      Returns
 |      -------
 |      X_leaves : array-like of shape (n_samples,)
 |          For each datapoint x in X, return the index of the leaf
x
 |          ends up in. Leaves are numbered within
 |          ``[0; self.tree_.node_count)``, possibly with gaps in th
e
 |          numbering.
 |
 |  cost_complexity_pruning_path(self, X, y, sample_weight=None)
```

```
 |      Compute the pruning path during Minimal Cost-Complexity Prun
ing.
 |
 |      See :ref:`minimal_cost_complexity_pruning` for details on th
e pruning
 |      process.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The training input samples. Internally, it will be conve
rted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csc_matrix``.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_output
s)
 |          The target values (class labels) as integers or strings.
 |
 |      sample_weight : array-like of shape (n_samples,), default=No
ne
 |          Sample weights. If None, then samples are equally weight
ed. Splits
 |          that would create child nodes with net zero or negative
weight are
 |          ignored while searching for a split in each node. Splits
are also
 |          ignored if they would result in any single class carryin
g a
 |          negative weight in either child node.
 |
 |      Returns
 |      -------
 |      ccp_path : :class:`~sklearn.utils.Bunch`
 |          Dictionary-like object, with the following attributes.
 |
 |          ccp_alphas : ndarray
 |              Effective alphas of subtree during pruning.
 |
 |          impurities : ndarray
 |              Sum of the impurities of the subtree leaves for the
 |              corresponding alpha value in ``ccp_alphas``.
 |
 |  decision_path(self, X, check_input=True)
 |      Return the decision path in the tree.
 |
 |      .. versionadded:: 0.18
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
```

```
 |      Returns
 |      -------
 |      indicator : sparse matrix of shape (n_samples, n_nodes)
 |          Return a node indicator CSR matrix where non zero elemen
ts
 |          indicates that the samples goes through the nodes.
 |
 |  get_depth(self)
 |      Return the depth of the decision tree.
 |
 |      The depth of a tree is the maximum distance between the root
 |      and any leaf.
 |
 |      Returns
 |      -------
 |      self.tree_.max_depth : int
 |          The maximum depth of the tree.
 |
 |  get_n_leaves(self)
 |      Return the number of leaves of the decision tree.
 |
 |      Returns
 |      -------
 |      self.tree_.n_leaves : int
 |          Number of leaves.
 |
 |  predict(self, X, check_input=True)
 |      Predict class or regression value for X.
 |
 |      For a classification model, the predicted class for each sam
ple in X is
 |      returned. For a regression model, the predicted value based
on X is
 |      returned.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_featu
res)
 |          The input samples. Internally, it will be converted to
 |          ``dtype=np.float32`` and if a sparse matrix is provided
 |          to a sparse ``csr_matrix``.
 |
 |      check_input : bool, default=True
 |          Allow to bypass several input checking.
 |          Don't use this parameter unless you know what you do.
 |
 |      Returns
 |      -------
 |      y : array-like of shape (n_samples,) or (n_samples, n_output
s)
 |          The predicted classes, or the predict values.
 |
 |  ----------------------------------------------------------------
------
 |  Readonly properties inherited from BaseDecisionTree:
 |
 |  feature_importances_
 |      Return the feature importances.
 |
 |      The importance of a feature is computed as the (normalized)
```

```
total
 |      reduction of the criterion brought by that feature.
 |      It is also known as the Gini importance.
 |
 |      Warning: impurity-based feature importances can be misleadin
g for
 |      high cardinality features (many unique values). See
 |      :func:`sklearn.inspection.permutation_importance` as an alte
rnative.
 |
 |      Returns
 |      -------
 |      feature_importances_ : ndarray of shape (n_features,)
 |          Normalized total reduction of criteria by feature
 |          (Gini importance).
 |
 |  ----------------------------------------------------------------------
------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator a
nd
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested o
bjects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update
each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
```

```
|           Estimator instance.
```

In [24]:

```
prepare_decision_tree(data)
```

```
Accuracy :  27.0
```

Out[24]:

```
DecisionTreeClassifier(random_state=100)
```

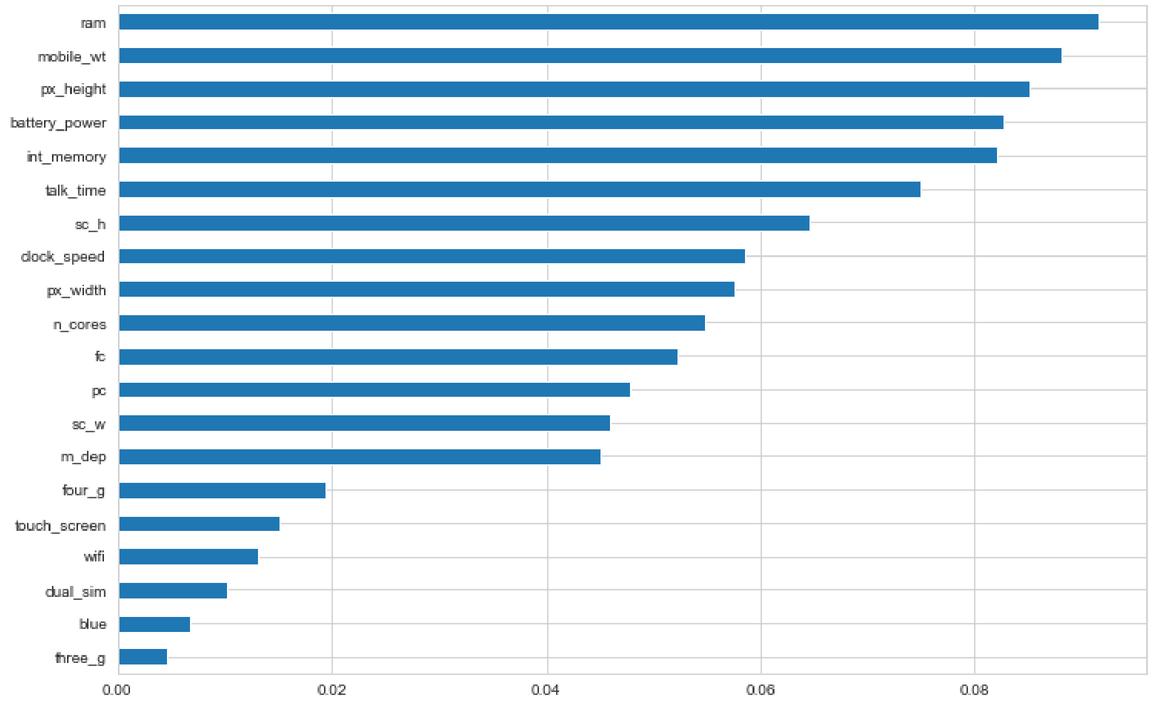
In [25]:

```
prepare_decision_tree(data, True, True, True, True)
```

```
Confusion Matrix :
[[35 43 43 32]
 [42 30 37 37]
 [45 32 38 26]
 [31 50 44 35]]

Accuracy :  23.0

Classification Report :
              precision    recall  f1-score   support

        High       0.23      0.23      0.23       153
         Low       0.19      0.21      0.20       146
      Medium       0.23      0.27      0.25       141
   Very_High       0.27      0.22      0.24       160

    accuracy                           0.23       600
   macro avg       0.23      0.23      0.23       600
weighted avg       0.23      0.23      0.23       600
```

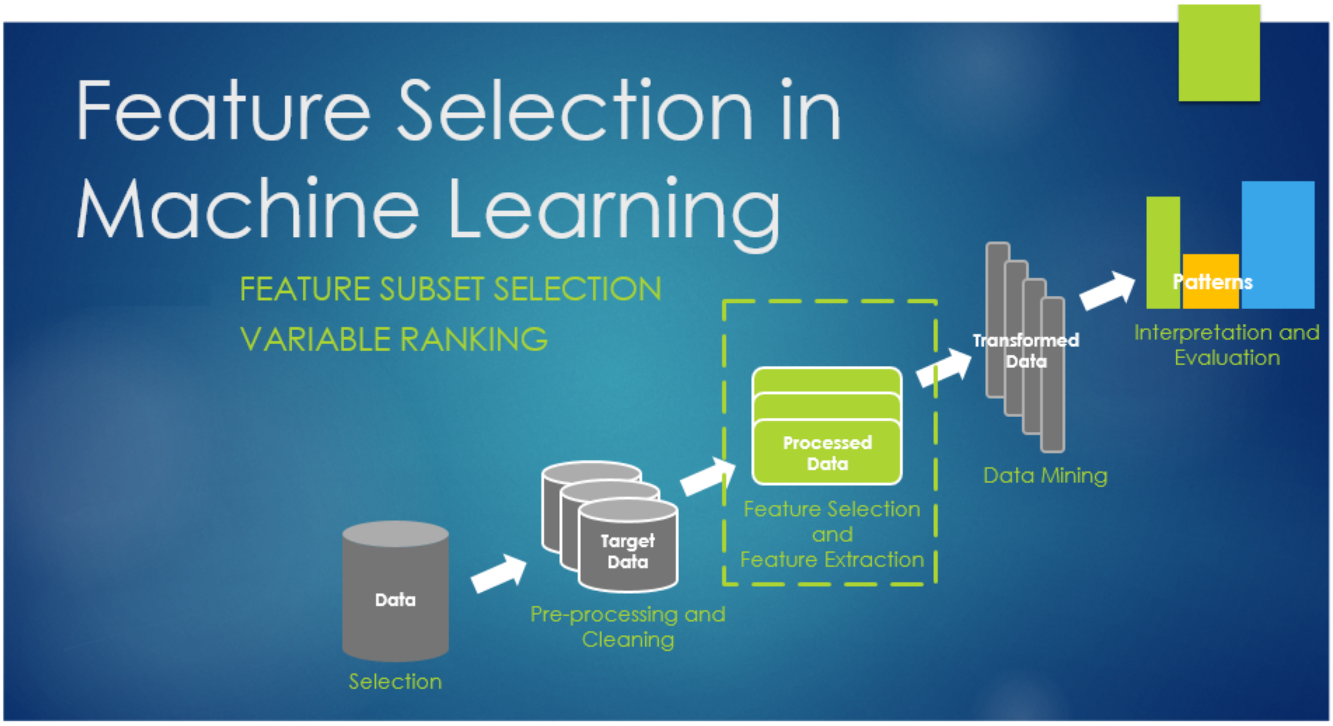
Out[25]:

```
DecisionTreeClassifier(random_state=100)
```



More details can be found here.

4. Feature Subset Selection



Adapted from this article.

4.1 Feature Selection methods

The accuracy of machine learning models depends a lot on the features which goes into building those models. Otherwise its just garbage in , garbage out. Feature selection plays such a vital role in creating an effective predictive model. It is even more important when the number of features are very large. Not every feature will be playing the significant role in the prediction, so you don't need to bother about each and every attribute present at your disposal for creating an algorithm. You can assist your algorithm by feeding in only those features that are really important. You not only reduce the training time and the evaluation time as well.

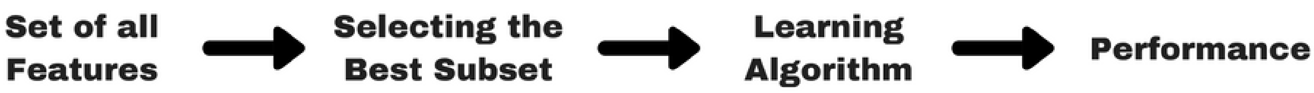
Top reasons to use feature selection are:

- **enables the machine learning algorithm to train faster**
- **reduces the complexity of a model and makes it easier to interpret**
- **improves the accuracy of a model if the right subset is chosen**
- **reduces overfitting**

There are following three methods those are used for the feature selection :

- **Filter methods**
- **Wrapper methods**
- **Embedded methods**

4.2 Filter Methods



These are generally used as a preprocessing step. The selection of features is not dependent of any machine learning algorithms. A lot of data exploration is done while using this method. Features are selected on the basis of their scores in various statistical tests for their correlation with the outcome variable.

Following table provides guidance on the type of method suitable for the type of attribute.

Feature\Response	Continuous	Categorical
Continuous	Pearson’s Correlation	LDA
Categorical	Anova	Chi-Square

- **Pearson’s Correlation: It is used as a measure for quantifying linear dependence between two continuous variables X and Y. Its value varies from -1 to +1.**
- **LDA: Linear discriminant analysis is used to find a linear combination of features that characterizes or separates two or more classes (or levels) of a categorical variable.**
- **ANOVA: ANOVA stands for Analysis of variance. It is similar to LDA except for the fact that it is operated using one or more categorical independent features and one continuous dependent feature. It provides a statistical test of whether the means of several groups are equal or not.**
- **Chi-Square: It is a is a statistical test applied to the groups of categorical features to evaluate the likelihood of correlation or association between them using their frequency distribution.**

Filter methods do not remove multicollinearity, must need to deal with multicollinearity of features as well before training models for data.

4.2.1. Univariate Filters

Univariate filters evaluate each feature independently with respect to the target variable.

- **Mutual Information (Information Gain)**
- **Gini index**
- **Gain Ratio**
- **Chi-Squared test**
- **Fisher Score**

Lets explore what different options are available in sklearn for the same.

In [26]:

```
from sklearn.feature_selection import SelectKBest
help(SelectKBest)
```

```
Help on class SelectKBest in module sklearn.feature_selection._univa
riate_selection:

class SelectKBest(_BaseFilter)
 |  SelectKBest(score_func=<function f_classif at 0x7fe296909b80>,
*, k=10)
 |
 |  Select features according to the k highest scores.
 |
 |  Read more in the :ref:`User Guide <univariate_feature_selection>
`.
 |
 |  Parameters
 |  ----------
 |  score_func : callable
 |      Function taking two arrays X and y, and returning a pair of
arrays
 |      (scores, pvalues) or a single array with scores.
 |      Default is f_classif (see below "See also"). The default fun
ction only
 |      works with classification tasks.
 |
 |      .. versionadded:: 0.18
 |
 |  k : int or "all", optional, default=10
 |      Number of top features to select.
 |      The "all" option bypasses selection, for use in a parameter
search.
 |
 |  Attributes
 |  ----------
 |  scores_ : array-like of shape (n_features,)
 |      Scores of features.
 |
 |  pvalues_ : array-like of shape (n_features,)
 |      p-values of feature scores, None if `score_func` returned on
ly scores.
 |
 |  Examples
 |  --------
 |  >>> from sklearn.datasets import load_digits
 |  >>> from sklearn.feature_selection import SelectKBest, chi2
 |  >>> X, y = load_digits(return_X_y=True)
 |  >>> X.shape
 |  (1797, 64)
 |  >>> X_new = SelectKBest(chi2, k=20).fit_transform(X, y)
 |  >>> X_new.shape
 |  (1797, 20)
 |
 |  Notes
 |  -----
 |  Ties between features with equal scores will be broken in an uns
pecified
 |  way.
 |
 |  See also
 |  --------
 |  f_classif: ANOVA F-value between label/feature for classificatio
n tasks.
 |  mutual_info_classif: Mutual information for a discrete target.
 |  chi2: Chi-squared stats of non-negative features for classificat
```

```
ion tasks.
 |  f_regression: F-value between label/feature for regression task
s.
 |  mutual_info_regression: Mutual information for a continuous targ
et.
 |  SelectPercentile: Select features based on percentile of the hig
hest scores.
 |  SelectFpr: Select features based on a false positive rate test.
 |  SelectFdr: Select features based on an estimated false discovery
rate.
 |  SelectFwe: Select features based on family-wise error rate.
 |  GenericUnivariateSelect: Univariate feature selector with config
urable mode.
 |  
 |  Method resolution order:
 |      SelectKBest
 |      _BaseFilter
 |      sklearn.feature_selection._base.SelectorMixin
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, score_func=<function f_classif at 0x7fe296909b80
>, *, k=10)
 |      Initialize self.  See help(type(self)) for accurate signatur
e.
 |  
 |  ----------------------------------------------------------------
------
 |  Data and other attributes defined here:
 |  
 |  __abstractmethods__ = frozenset()
 |  
 |  ----------------------------------------------------------------
------
 |  Methods inherited from _BaseFilter:
 |  
 |  fit(self, X, y)
 |      Run score function on (X, y) and get the appropriate feature
s.
 |  
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          The training input samples.
 |  
 |      y : array-like of shape (n_samples,)
 |          The target values (class labels in classification, real
numbers in
 |          regression).
 |  
 |      Returns
 |      -------
 |      self : object
 |  
 |  ----------------------------------------------------------------
------
 |  Methods inherited from sklearn.feature_selection._base.SelectorM
ixin:
```

```
 |
 |  get_support(self, indices=False)
 |      Get a mask, or integer index, of the features selected
 |
 |      Parameters
 |      ----------
 |      indices : boolean (default False)
 |          If True, the return value will be an array of integers,
rather
 |          than a boolean mask.
 |
 |      Returns
 |      -------
 |      support : array
 |          An index that selects the retained features from a featu
re vector.
 |          If `indices` is False, this is a boolean array of shape
 |          [# input features], in which an element is True iff its
 |          corresponding feature is selected for retention. If `ind
ices` is
 |          True, this is an integer array of shape [# output featur
es] whose
 |          values are indices into the input feature vector.
 |
 |  inverse_transform(self, X)
 |      Reverse the transformation operation
 |
 |      Parameters
 |      ----------
 |      X : array of shape [n_samples, n_selected_features]
 |          The input samples.
 |
 |      Returns
 |      -------
 |      X_r : array of shape [n_samples, n_original_features]
 |          `X` with columns of zeros inserted where features would
have
 |          been removed by :meth:`transform`.
 |
 |  transform(self, X)
 |      Reduce X to the selected features.
 |
 |      Parameters
 |      ----------
 |      X : array of shape [n_samples, n_features]
 |          The input samples.
 |
 |      Returns
 |      -------
 |      X_r : array of shape [n_samples, n_selected_features]
 |          The input samples with only the selected features.
 |
 |  ------------------------------------------------------------------
------
 |  Methods inherited from sklearn.base.TransformerMixin:
 |
 |  fit_transform(self, X, y=None, **fit_params)
 |      Fit to data, then transform it.
 |
 |      Fits transformer to X and y with optional parameters fit_par
ams
```

```
 |      and returns a transformed version of X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix, dataframe} of shape
(n_samples, n_features)
 |
 |      y : ndarray of shape (n_samples,), default=None
 |          Target values.
 |
 |      **fit_params : dict
 |          Additional fit parameters.
 |
 |      Returns
 |      -------
 |      X_new : ndarray array of shape (n_samples, n_features_new)
 |          Transformed array.
 |
 |  ----------------------------------------------------------------
------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------
------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator a
nd
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested o
bjects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update
```

```
each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
```

The important score functions supported are :

- **f_classif: ANOVA F-value between label/feature for classification tasks.**
- **mutual_info_classif: Mutual information for a discrete target.**
- **chi2: Chi-squared stats of non-negative features for classification tasks.**
- **f_regression: F-value between label/feature for regression tasks.**
- **mutual_info_regression: Mutual information for a continuous target.**
- **SelectPercentile: Select features based on percentile of the highest scores.**
- **SelectFpr: Select features based on a false positive rate test.**
- **SelectFdr: Select features based on an estimated false discovery rate.**
- **SelectFwe: Select features based on family-wise error rate.**
- **GenericUnivariateSelect: Univariate feature selector with configurable mode.**

In [27]:

```
from sklearn.feature_selection import f_classif
from sklearn.feature_selection import chi2
from sklearn.feature_selection import mutual_info_classif
```

In [28]:

```
def show_top_univariate_filters(data, score_func, top_k):
    X = data.iloc[:,0:20]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    if score_func == "chi2":
        func = chi2
    elif score_func == "f_classif":
        func = f_classif
    elif score_func == "mutual_info_classif":
        func = mutual_info_classif

    #apply SelectKBest class to extract top k best features
    bestfeatures = SelectKBest(score_func=func, k=top_k)
    fit = bestfeatures.fit(X,y)

    dfscores = pd.DataFrame(fit.scores_)
    dfcolumns = pd.DataFrame(X.columns)

    #concat two dataframes for better visualization
    featureScores = pd.concat([dfcolumns,dfscores],axis=1)
    featureScores.columns = ['Specs','Score']  #naming the dataframe columns
    print(featureScores.nlargest(top_k,'Score'))  #print 10 best features
```

In [29]:

```
show_top_univariate_filters(data, 'chi2', 5)
```

```
            Specs        Score
11      px_height  2033.383006
0   battery_power   918.479571
13            ram   500.179498
12       px_width   203.596995
4              fc    52.696243
```

In [30]:

```
show_top_univariate_filters(data, 'f_classif', 5)
```

```
            Specs     Score
4              fc  4.034505
11      px_height  2.224286
0   battery_power  1.966637
18   touch_screen  1.621116
10             pc  1.331369
```

The most significant attributes seems to be "px_height", "battery_power" and "fc".

4.2.2 Correlation Matrix with Heatmap

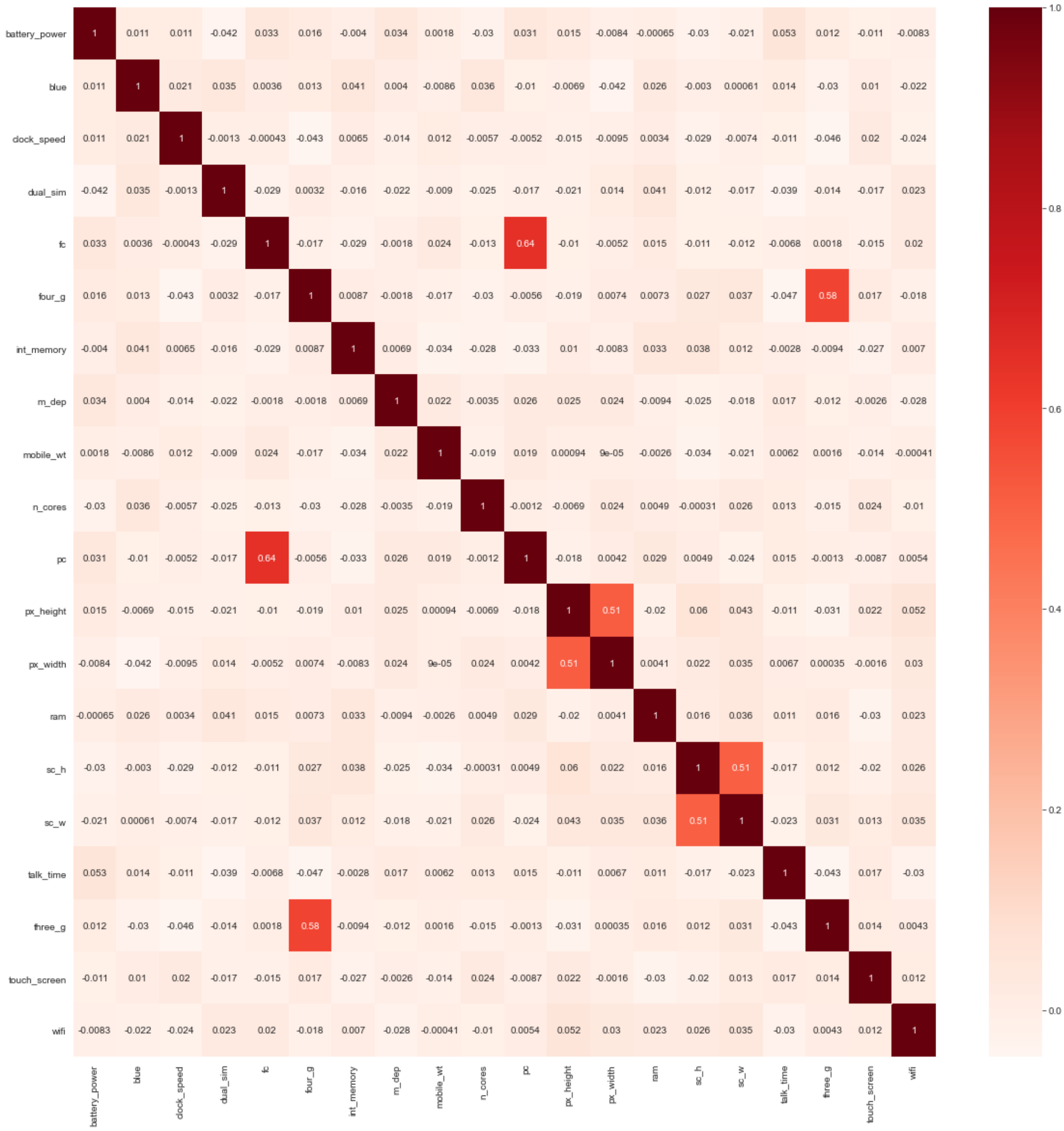
As the name suggest, in this method, you filter and take only the subset of the relevant features. The model is built after selecting the features. The filtering here is done using correlation matrix and it is most commonly done using Pearson correlation. Here we will first plot the Pearson correlation heatmap and see the correlation of independent variables with the output variable. The correlation coefficient has values between -1 to 1

- **A value closer to 0 implies weaker correlation (exact 0 implying no correlation)**
- **A value closer to 1 implies stronger positive correlation**
- **A value closer to -1 implies stronger negative correlation**

The relationship between the independent attributes also can help to identify the redundant attributes which further can be removed to limit the feature space. Lets have a look at this technique.

In [31]:

```
X = data.iloc[:,0:20]  #independent columns
y = data.iloc[:,-1]    #target column i.e price range

#get correlations of each features in dataset
corrmat = data.corr()
top_corr_features = corrmat.index

#plot heat map
plt.figure(figsize=(20,20))
sns.heatmap(data[top_corr_features].corr(), annot=True, cmap=plt.cm.Reds)
plt.show()
```


battery_power
blue
clock_speed
dual_sim
fc
four_g
int_memory
m_dep
mobile_wt
n_cores
pc
px_height
px_width
ram
sc_h
sc_w
talk_time
three_g
touch_screen
wifi
1.0
0.8
0.6
0.4
0.2
0.0

Few observations :

- **"pc" and "fc" are correlated , hence one of them can be ignored while model building**
- **"three_g" and "four_g" are correlated , hence one of them can be ignored while model building**
- **"px_height" and "px_width" are correlated , hence one of them can be ignored while model building**
- **"sc_w" and "sc_h" are correlated , hence one of them can be ignored while model building**

4.2.3 Using Feature Importance

As we are trying out classification problem, the classification implementations provides a built-in feature ranking mechanism, lets try that out with one of the decision tree classfier.

In [32]:

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier
```

In [33]:

```
import matplotlib.pyplot as plt
```

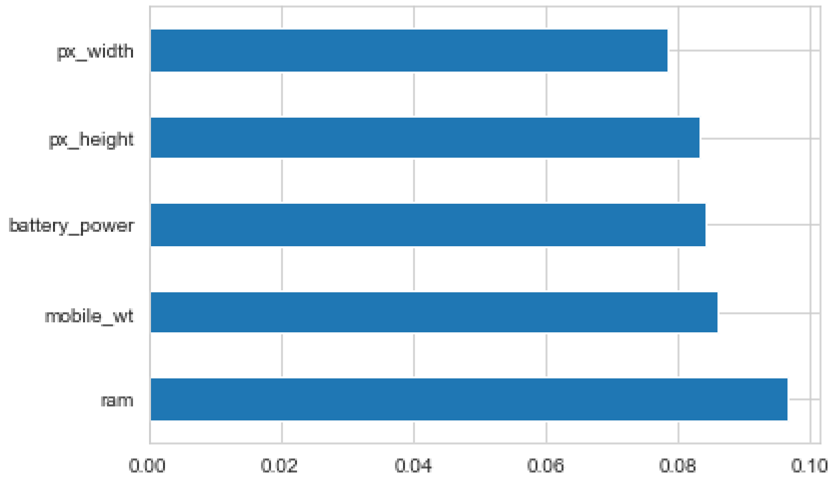
In [34]:

```
def show_top_decition_classifier_feature(data, classifier, top_k):
    #Prepare the independent and dependent attributes sets
    X = data.iloc[:,0:20]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    if classifier == "ExtraTreesClassifier":
        classifier = ExtraTreesClassifier
    elif classifier == "DecisionTreeClassifier":
        classifier = DecisionTreeClassifier

    model = classifier()
    model.fit(X,y)

    #use inbuilt class feature_importances of tree based classifiers
    print(model.feature_importances_)

    #plot graph of feature importances for better visualization
    feat_importances = pd.Series(model.feature_importances_, index=X.columns)
    feat_importances.nlargest(top_k).plot(kind='barh')
    plt.show()
```

In [35]:

```
show_top_decition_classifier_feature(data, "DecisionTreeClassifier", 5)
```

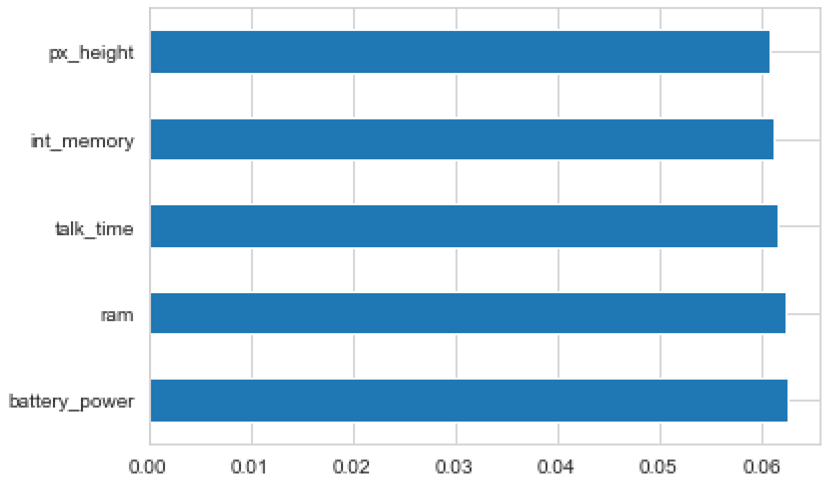
```
[0.08431254 0.01704859 0.06184454 0.01433624 0.0430978  0.01096343
 0.0777733  0.03669787 0.08615177 0.04722278 0.05460696 0.08344895
 0.07842286 0.09653316 0.05056465 0.05532983 0.06229554 0.01366141
 0.01478627 0.01090149]
```



In [36]:

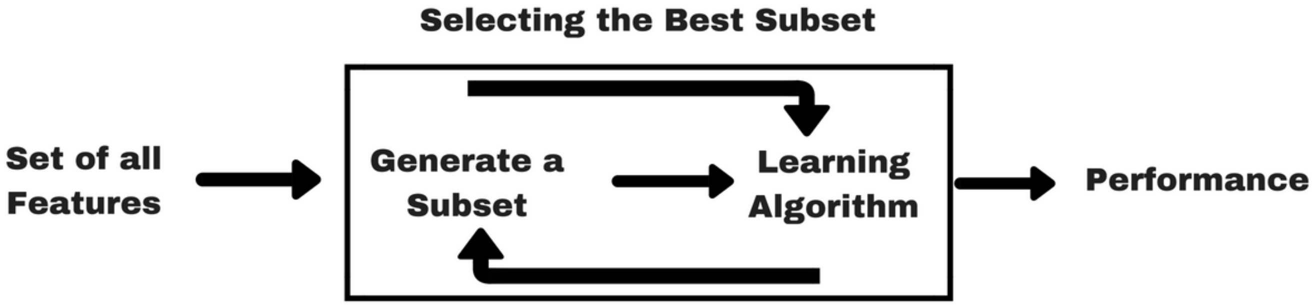
```
show_top_decition_classifier_feature(data, "ExtraTreesClassifier", 5)
```

```
[0.06249627 0.02902677 0.05938163 0.02903454 0.05620286 0.02601207
 0.06110183 0.05882095 0.06079747 0.05745755 0.06024863 0.06080147
 0.06079167 0.06238295 0.06001598 0.05936613 0.06148097 0.01961981
 0.0245604  0.03040004]
```



The most significant attributes seems to be "battery_power", "ram", "mobile_wt" and "px_height"

4.3 Wrapper Methods



In wrapper methods, a subset of features is used to train a model. Based on the inferences drawnfrom the previous model, needs to decide whether to add or remove features from feature subset. The problem is essentially reduced to a search problem. These methods are usually computationally very expensive.

Some common examples of wrapper methods are backward feature elimination, forward feature selection, recursive feature elimination, etc.

- **Backward Elimination: The backward elimination starts with all the features and removes the least significant feature at each iteration which improves the performance of the model. This is repeated until no improvement is observed on removal of features.**
- **Forward Selection: Forward selection is an iterative method which starts with having no feature in the model. In each iteration, a new feature is added to see if it improves the model. Its repeated till an addition of a new variable does not improve the performance of the model.**
- **Recursive Feature elimination: It is a greedy optimization algorithm which aims to find the best performing feature subset. It repeatedly creates models and keeps aside the best or the worst performing feature at each iteration. It constructs the next model with the left features until all the features are exhausted. It then ranks the features based on the order of their elimination.**

4.3.1. Backward Elimination Method

The backward elimination starts with all the features and removes the least significant feature at each iteration which improves the performance of the model. This is repeated until no improvement is observed on removal of features.

Lets write a function that will help us to try out Backward Feature Elimination, It will accept a dataset and list of features that needs to be dropped in an iteration.

In [37]:

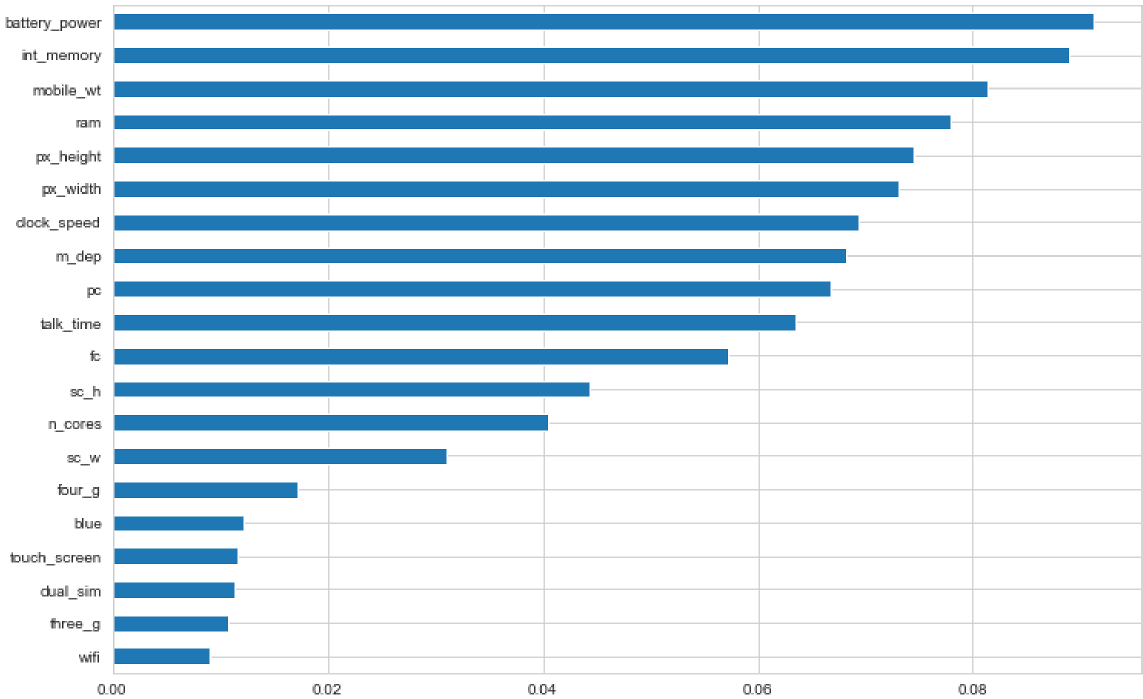
```
def predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_v
isual):
    #Prepare the dataset by removing the features mentioned
    for feature in features_to_be_removed:
        data = data.drop(feature, axis=1)

    #Call Decision tree function to get the accuracy results
    prepare_decision_tree(data, show_visual = show_visual)
```

Lets see the accuracy score with all features present i.e. feature removal list is empty.

In [38]:

```
features_to_be_removed = []
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

```
Accuracy :  24.5
```

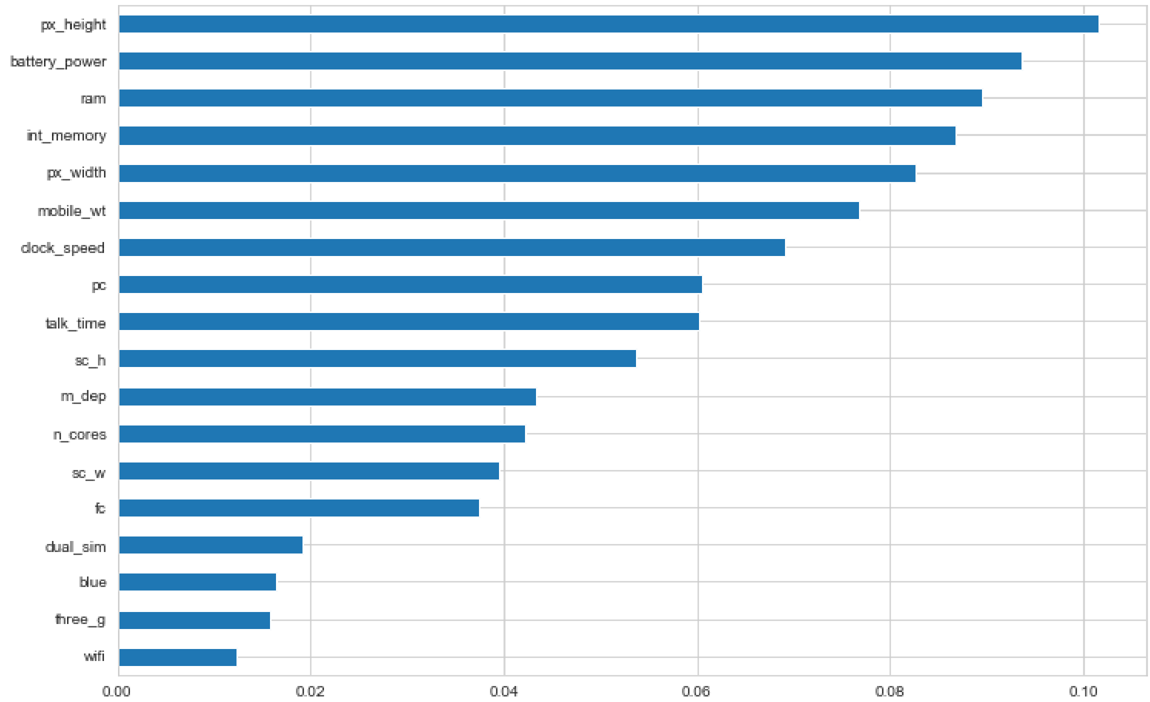


"blue" and "three_g" seems to have little impact on the model performance. Lets try to remove them.

In [39]:

```
features_to_be_removed = ["touch_screen", "four_g"]
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

```
Accuracy :  25.0
```

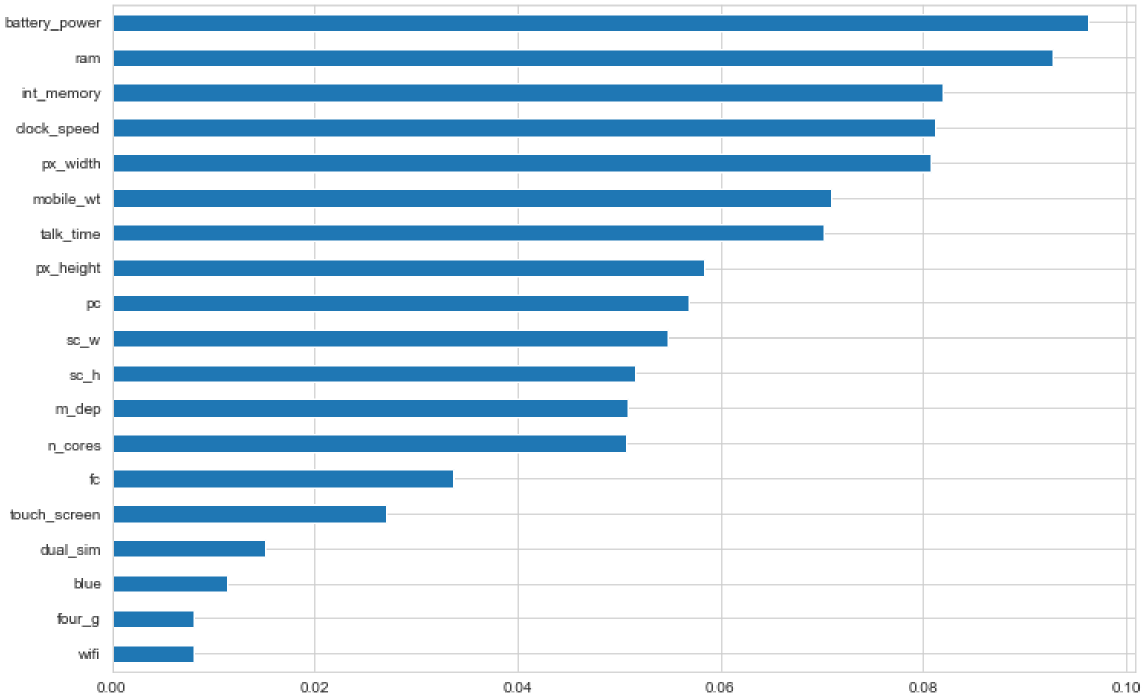


Accuracy decreased so lets try them one by one.

In [40]:

```
features_to_be_removed = ["three_g"]
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

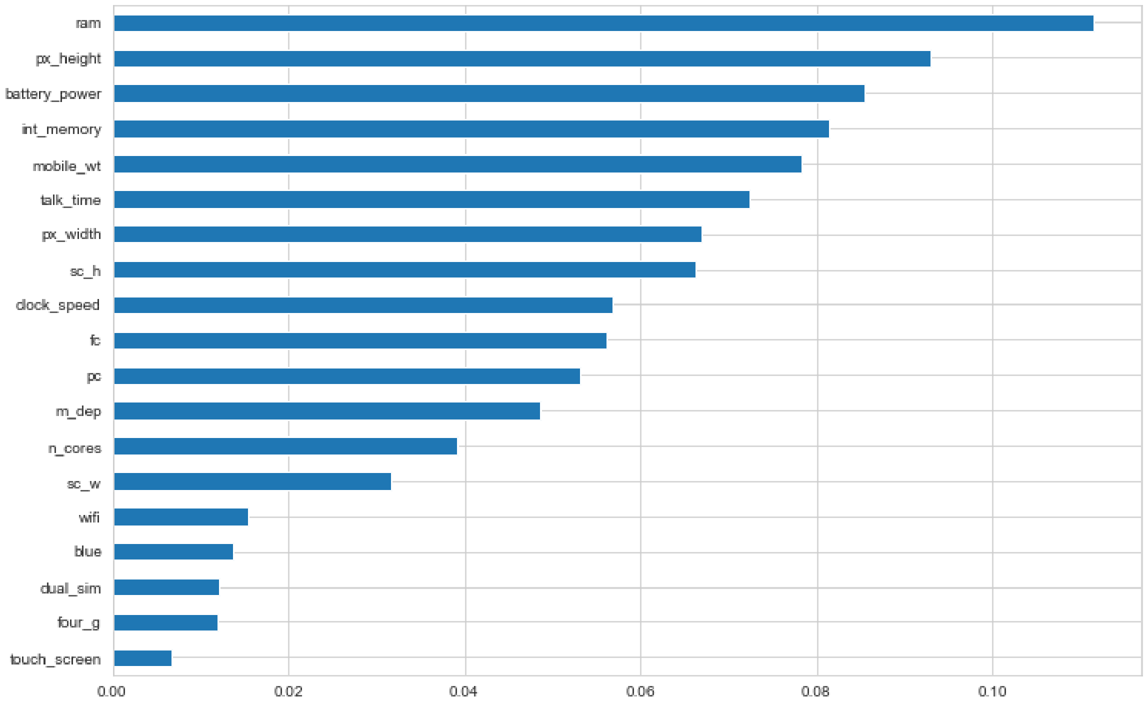
```
Accuracy :  28.000000000000004
```



In [41]:

```
features_to_be_removed = ["three_g"]
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

```
Accuracy :  22.833333333333332
```



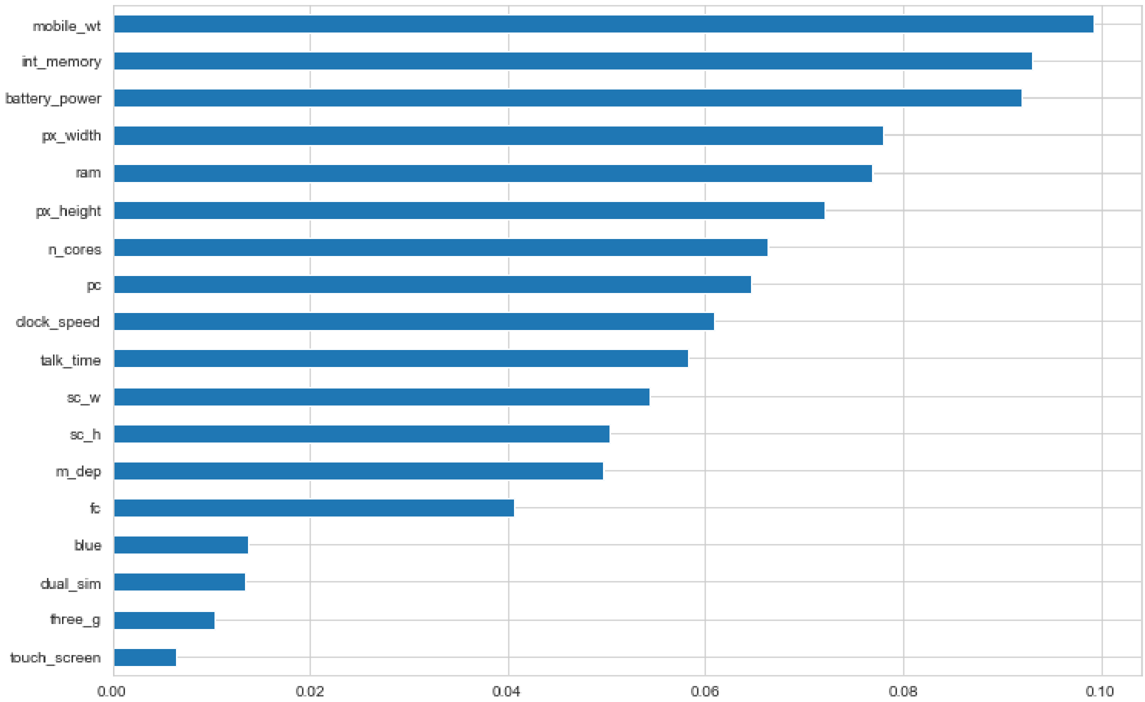
When "dual_sim" is removed, the accuracy is improved but when "four_g" is removed, then accuray decreased, which means "four_g" cant be ignored.

When "dual_sim" is removed, the next least significant attribute seems to be "wifi", lets try removing it.

In [42]:

```
features_to_be_removed = ["four_g", "wifi"]
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

```
Accuracy :  28.000000000000004
```

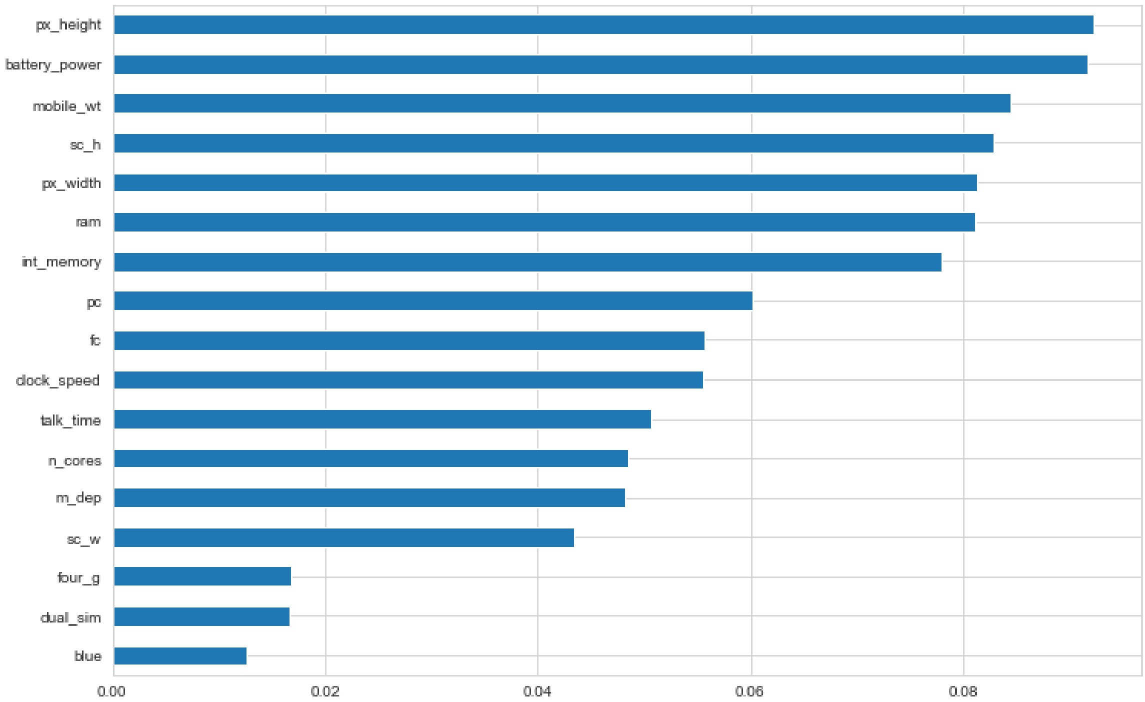


We can try out the other combinations of the attributes and see its effect on the accuracy.

In [43]:

```
features_to_be_removed = ['three_g', 'wifi', 'touch_screen']
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

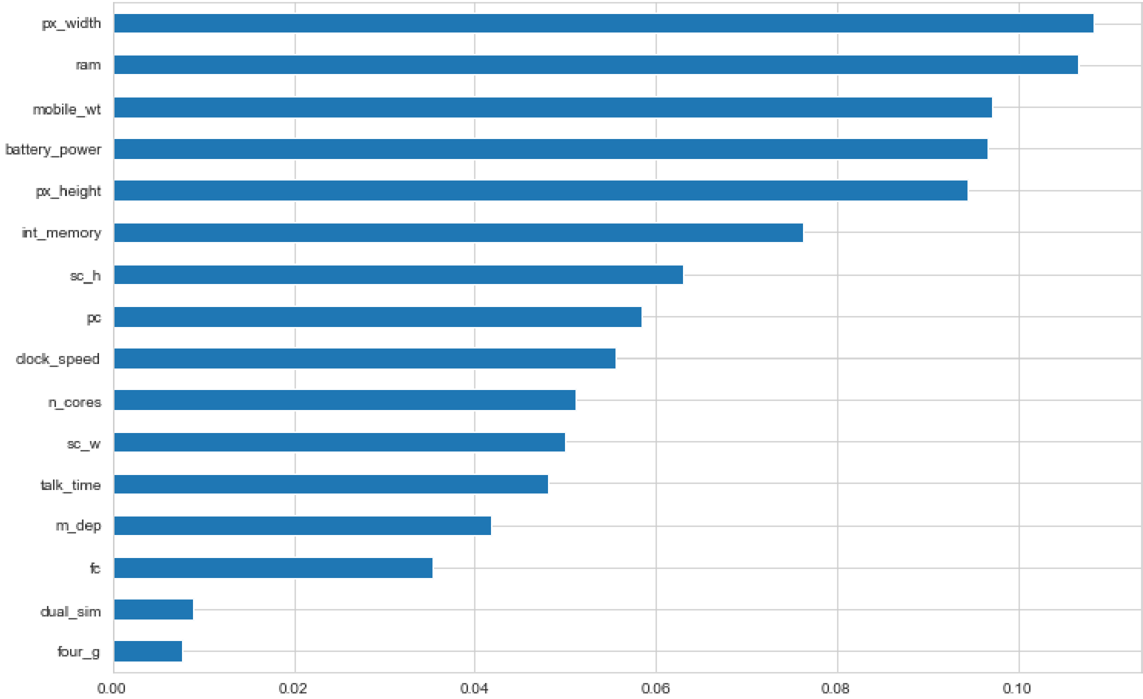
```
Accuracy :  25.5
```



In [44]:

```
features_to_be_removed = ['three_g', 'wifi', 'touch_screen', 'blue']
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

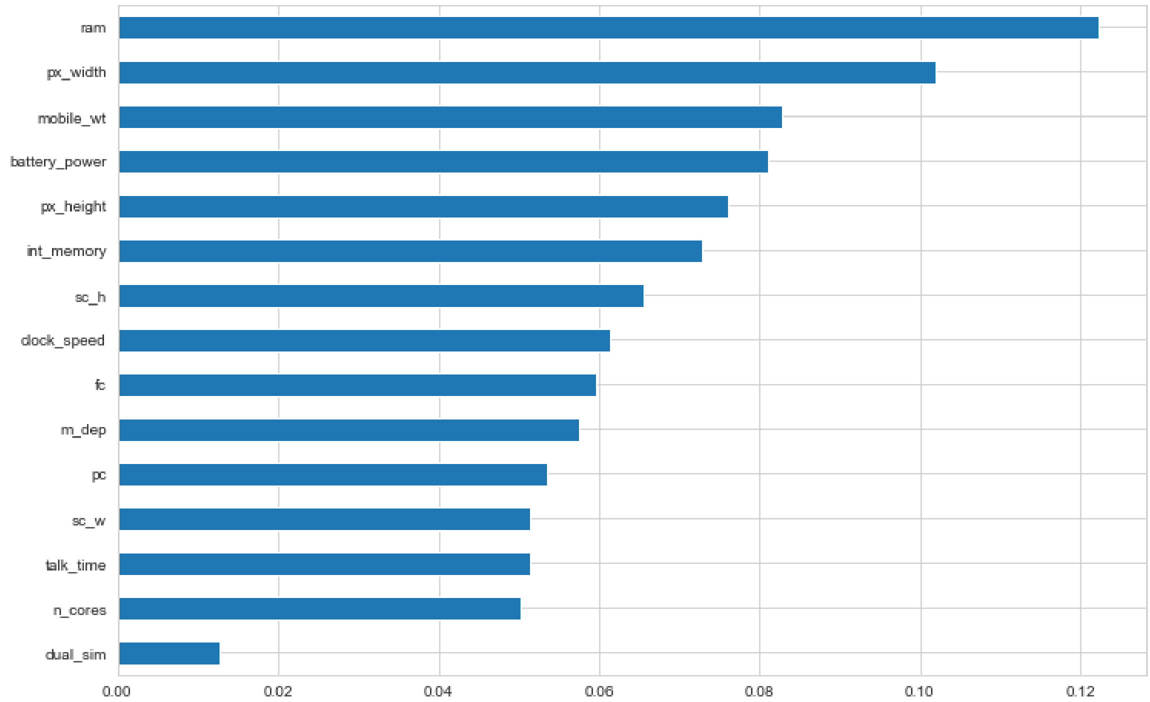
```
Accuracy :  27.166666666666668
```



In [45]:

```
features_to_be_removed = ['three_g', 'wifi', 'touch_screen', 'blue', 'four_g']
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

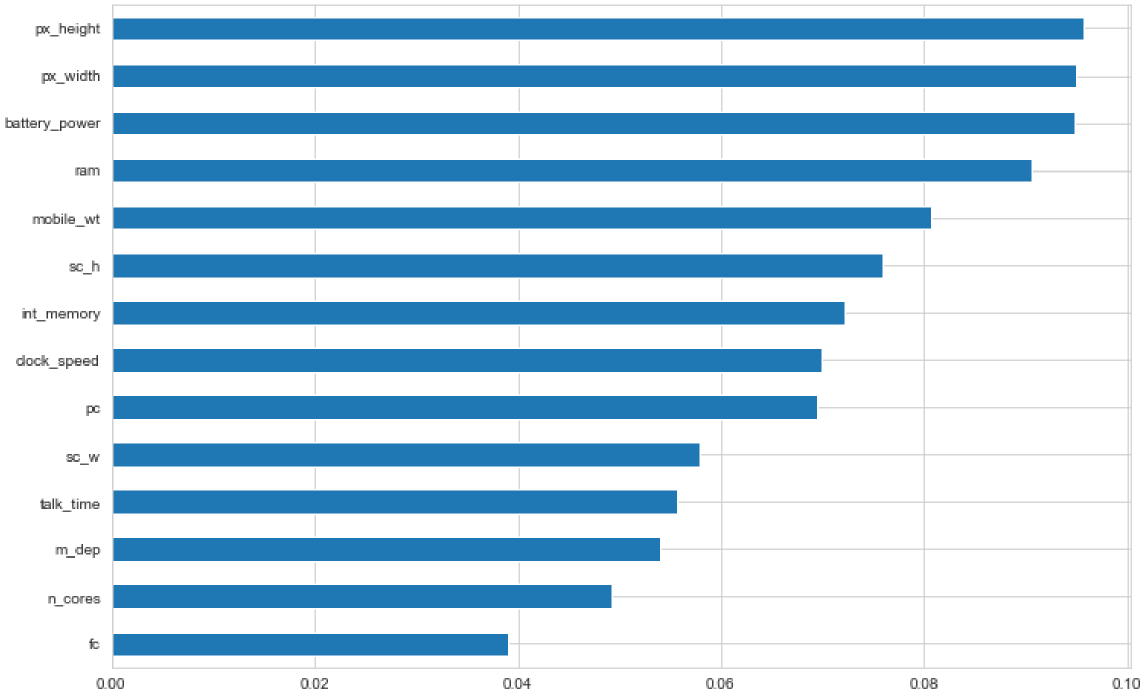
```
Accuracy :  22.666666666666664
```



In [46]:

```
features_to_be_removed = ['three_g', 'wifi', 'touch_screen', 'blue', 'four_g',
'dual_sim']
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

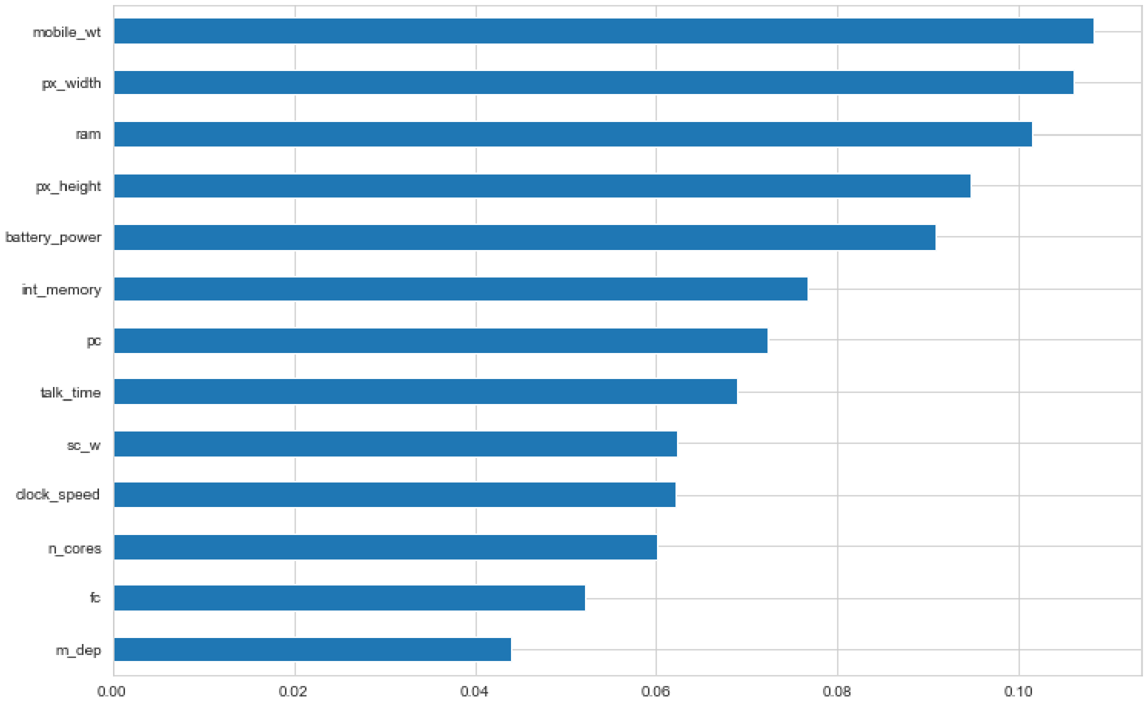
```
Accuracy :  25.666666666666664
```



In [47]:

```
features_to_be_removed = ['three_g', 'wifi', 'touch_screen', 'blue', 'four_g',
'dual_sim', 'sc_h']
predict_accuracy_by_feature_elimination(data, features_to_be_removed, show_visua
l=True)
```

```
Accuracy :  25.666666666666664
```



4.3.2. Backward Elimination Method using "mlxtend"

But next question in your mind, what if the attribute space is too wide then this iterative approach will become cumborsome to follow. Is there anything else that can simplify the feature selection process. Fortunately, we have a library that can be used for this purpose, named "mlxtend".

In [48]:

```
#execute only first time
!pip install mlxtend
```

```
zsh:1: command not found: pip
```

First we need to obtain an instance of Decision tree on which feature selection approaches can be tried out.

In [49]:

```
dt = prepare_decision_tree(data, show_visual = False)
```

```
Accuracy :  26.166666666666664
```

Lets use the function from mlxtend to obtain the best features list.

In [50]:

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

In [51]:

```
def get_top_k_features_by_mlxtend(data, dt, top_k, forward=True, cv_cnt=0, show_
results=True):
    #Preprare the independant and target attributes
    col_length = len(data.columns)
    X = data.iloc[:,0:col_length-1]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    #Prepare a model using the specified feature selection method
    sfs_model = SFS(dt,
                   k_features=top_k,
                   forward=forward,
                   floating=False,
                   verbose=2,
                   scoring='accuracy',
                   cv=cv_cnt)

    #Lets fit the model and identify the features
    sfs_model = sfs_model.fit(X, y)

    #Show outcomes
    #print("Subsets : \n", sfs_model.subsets_ , "\n")
    if show_results:
        print("Score : " , sfs_model.k_score_, "\n")
        print("Top" , top_k , " Feature Names : " , sfs_model.k_feature_names_,
"\n")

    return sfs_model
```

In [52]:

```
get_top_k_features_by_mlxtend(data, dt, 3, forward=False)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.4s finishe
d

[2020-12-15 10:21:46] Features: 19/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  19 out of  19 | elapsed:    0.4s finishe
d

[2020-12-15 10:21:46] Features: 18/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:    0.4s finishe
d

[2020-12-15 10:21:47] Features: 17/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  17 out of  17 | elapsed:    0.3s finishe
d

[2020-12-15 10:21:47] Features: 16/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  16 out of  16 | elapsed:    0.5s finishe
d

[2020-12-15 10:21:48] Features: 15/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  15 out of  15 | elapsed:    0.5s finishe
d

[2020-12-15 10:21:48] Features: 14/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  14 out of  14 | elapsed:    0.3s finishe
d

[2020-12-15 10:21:48] Features: 13/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  13 out of  13 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:49] Features: 12/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  12 out of  12 | elapsed:    0.2s finishe
```

```
d

[2020-12-15 10:21:49] Features: 11/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  11 out of  11 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:49] Features: 10/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  10 out of  10 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:49] Features: 9/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   9 out of   9 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:49] Features: 8/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   8 out of   8 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:49] Features: 7/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   7 out of   7 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:49] Features: 6/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   6 out of   6 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:49] Features: 5/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.


Score :  0.999

Top 3  Feature Names :  ('battery_power', 'clock_speed', 'int_memor
y')
```

```
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    0.0s finishe
d

[2020-12-15 10:21:49] Features: 4/3 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.0s finishe
d

[2020-12-15 10:21:49] Features: 3/3 -- score: 0.999
```

Out[52]:

```
SequentialFeatureSelector(cv=0,
                          estimator=DecisionTreeClassifier(random_st
ate=100),
                          forward=False, k_features=3, scoring='accu
racy',
                          verbose=2)
```

In [53]:

```
get_top_k_features_by_mlxtend(data, dt, 5, forward=False)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.4s finishe
d

[2020-12-15 10:21:50] Features: 19/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  19 out of  19 | elapsed:    0.4s finishe
d

[2020-12-15 10:21:50] Features: 18/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:    0.4s finishe
d

[2020-12-15 10:21:51] Features: 17/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  17 out of  17 | elapsed:    0.5s finishe
d

[2020-12-15 10:21:51] Features: 16/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  16 out of  16 | elapsed:    0.3s finishe
d

[2020-12-15 10:21:51] Features: 15/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  15 out of  15 | elapsed:    0.3s finishe
d

[2020-12-15 10:21:52] Features: 14/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  14 out of  14 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:52] Features: 13/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  13 out of  13 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:52] Features: 12/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  12 out of  12 | elapsed:    0.2s finishe
```

```
d

[2020-12-15 10:21:52] Features: 11/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  11 out of  11 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:52] Features: 10/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  10 out of  10 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:53] Features: 9/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s

Score :  1.0

Top 5  Feature Names :  ('battery_power', 'blue', 'clock_speed', 'f
c', 'int_memory')


[Parallel(n_jobs=1)]: Done   9 out of   9 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:53] Features: 8/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   8 out of   8 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:53] Features: 7/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   7 out of   7 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:53] Features: 6/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done   6 out of   6 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:53] Features: 5/5 -- score: 1.0
```

Out[53]:

```
SequentialFeatureSelector(cv=0,
                          estimator=DecisionTreeClassifier(random_st
ate=100),
                          forward=False, k_features=5, scoring='accu
racy',
                          verbose=2)
```

More details on "mlxtend" can be found here.

4.3.3 Forward Feature Selection using "mlxtend"

Forward selection is an iterative method which starts with having no feature in the model. In each iteration, a new feature is added to see if it improves the model. Its repeated till an addition of a new variable does not improve the performance of the model.

Lets use the same function which we have defined earlier for feature selection using mlxtend to obtain the best features list but with "forward selection" technique.

In [54]:

```
get_top_k_features_by_mlxtend(data, dt, 5, forward=True)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.1s finishe
d

[2020-12-15 10:21:53] Features: 1/5 -- score: 0.847[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  19 out of  19 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:53] Features: 2/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:53] Features: 3/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s
[Parallel(n_jobs=1)]: Done  17 out of  17 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:54] Features: 4/5 -- score: 1.0[Parallel(n_jobs=
1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaini
ng:    0.0s

Score :  1.0

Top 5  Feature Names :  ('battery_power', 'blue', 'clock_speed', 'du
al_sim', 'ram')


[Parallel(n_jobs=1)]: Done  16 out of  16 | elapsed:    0.2s finishe
d

[2020-12-15 10:21:54] Features: 5/5 -- score: 1.0
```

Out[54]:

```
SequentialFeatureSelector(cv=0,
                          estimator=DecisionTreeClassifier(random_st
ate=100),
                          k_features=5, scoring='accuracy', verbose=
2)
```

4.3.4. RFE

The Recursive Feature Elimination (or RFE) works by recursively removing attributes and building a model on those attributes that remain.It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

You can learn more about the RFE class in the scikit-learn documentation.

The example below uses RFE with the decission tree algorithm to select the top k features. The choice of algorithm does not matter too much as long as it is skillful and consistent.

In [55]:

```
from sklearn.feature_selection import RFE
```

In [56]:

```
def get_top_k_features_by_rfe(data, dt, top_k, show_results=True):
    #Preprare the independant and target attributes
    col_length = len(data.columns)
    X = data.iloc[:,0:col_length-1]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    #Initializing RFE model
    rfe = RFE(dt, top_k)

    #Transforming data using RFE
    X_rfe = rfe.fit_transform(X,y)

    #Fitting the data to model
    model = dt.fit(X_rfe,y)

    #Prepare top k feature list
    indx= 0
    feature_list = []
    for col in X.columns:
        if rfe.ranking_[indx] == 1:
            feature_list.append(col)
        indx = indx + 1

    if show_results:
        print("Num Features: %d\n" % rfe.n_features_)
        print("Selected Features :" , feature_list)
        #print("Feature Ranking: %s" % rfe.ranking_)

    return feature_list
```

In [57]:

```
get_top_k_features_by_rfe(data, dt, 5, show_results=True)
```

```
/Users/nsumita/opt/anaconda3/lib/python3.8/site-packages/sklearn/uti
ls/validation.py:68: FutureWarning: Pass n_features_to_select=5 as k
eyword args. From version 0.25 passing these as positional arguments
will result in an error
  warnings.warn("Pass {} as keyword args. From version 0.25 "

Num Features: 5

Selected Features : ['battery_power', 'mobile_wt', 'px_height', 'px_
width', 'ram']
```

Out[57]:

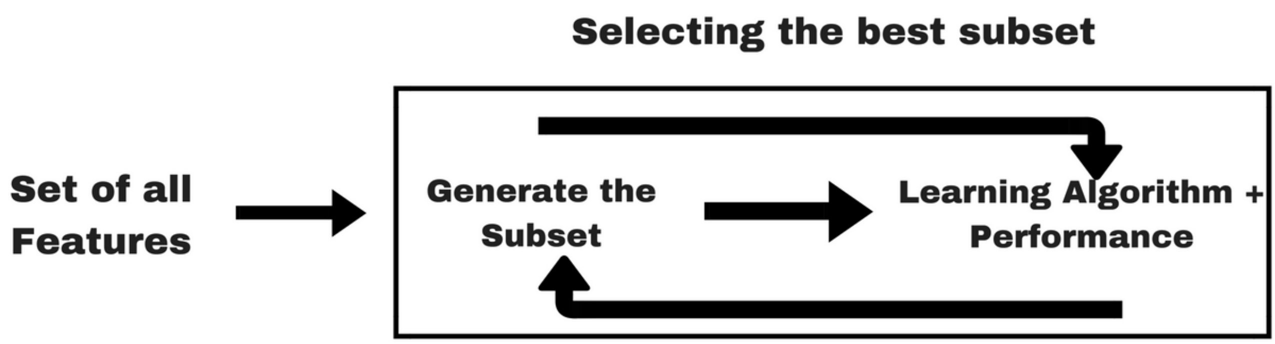
```
['battery_power', 'mobile_wt', 'px_height', 'px_width', 'ram']
```

In [58]:

```
feature_list= get_top_k_features_by_rfe(data, dt, 7, show_results=True)
```

```
/Users/nsumita/opt/anaconda3/lib/python3.8/site-packages/sklearn/uti
ls/validation.py:68: FutureWarning: Pass n_features_to_select=7 as k
eyword args. From version 0.25 passing these as positional arguments
will result in an error
  warnings.warn("Pass {} as keyword args. From version 0.25 "

Num Features: 7

Selected Features : ['battery_power', 'int_memory', 'mobile_wt', 'p
c', 'px_height', 'px_width', 'ram']
```

4.4 Embedded techniques



Embedded methods learn which features best contribute to the accuracy of the model while the model is being created. The most common type of embedded feature selection methods are regularization methods.

Regularization methods are also called penalization methods that introduce additional constraints into the optimization of a predictive algorithm (such as a regression algorithm) that bias the model toward lower complexity (fewer coefficients).

In the classificaiton problems, another type of technique called "ensembling" is used which helps to improve the accuracy of prediction by using more than one models. These are not really embedded techniques but can be correlated with them as they also help to improve the prediction accuracy by affecting the performane of sequence/collection of models.

The goal of ensemble methods is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

Two families of ensemble methods are usually distinguished:

- **In averaging methods, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.**
  Examples: Bagging methods, Forests of randomized trees, ...
- **By contrast, in boosting methods, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.**
  Examples: AdaBoost, Gradient Tree Boosting, ...

More details can be obtained here.

4.4.1 Bagging

The sklearn.ensemble module includes two averaging algorithms based on randomized decision trees: the RandomForest algorithm and the Extra-Trees method, specifically designed for trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

Lets try to build a bagging classifier using the decision tree that we have obtained earlier.

In [59]:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
```

In [60]:

```
def get_bagging_classifier(data):
    # Split the data into independent and target attributes
    col_length = len(data.columns)
    X = data.iloc[:,0:col_length - 1]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    #Split the data into training and testing set
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =0.3)

    forests = RandomForestClassifier(n_estimators=100, random_state=100)
    forests.fit(X_train, y_train)
    print(forests.score(X_test, y_test))

    return forests
```

In [61]:

```
get_bagging_classifier(data)
```

```
0.235
```

Out[61]:

```
RandomForestClassifier(random_state=100)
```

4.4.2 Boosting

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction.

In [62]:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import AdaBoostClassifier
```

In [63]:

```
def get_boosting_classifier(data):
    # Split the data into independent and target attributes
    col_length = len(data.columns)
    X = data.iloc[:,0:col_length - 1]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    #Split the data into training and testing set
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =0.3)
    clf = AdaBoostClassifier(n_estimators=100)
    clf.fit(X_train, y_train)
    print("Score : " , clf.score(X_test, y_test))

    print("Feature Importance : \n", clf.feature_importances_)

    importances=pd.Series(clf.feature_importances_, index=X_train.columns).sort_
values()
    importances.plot(kind='barh', figsize=(12,8))

    scores = cross_val_score(clf, X_train, y_train, cv=5)
    print("Score after cross validation : ", scores.mean())

    return clf
```

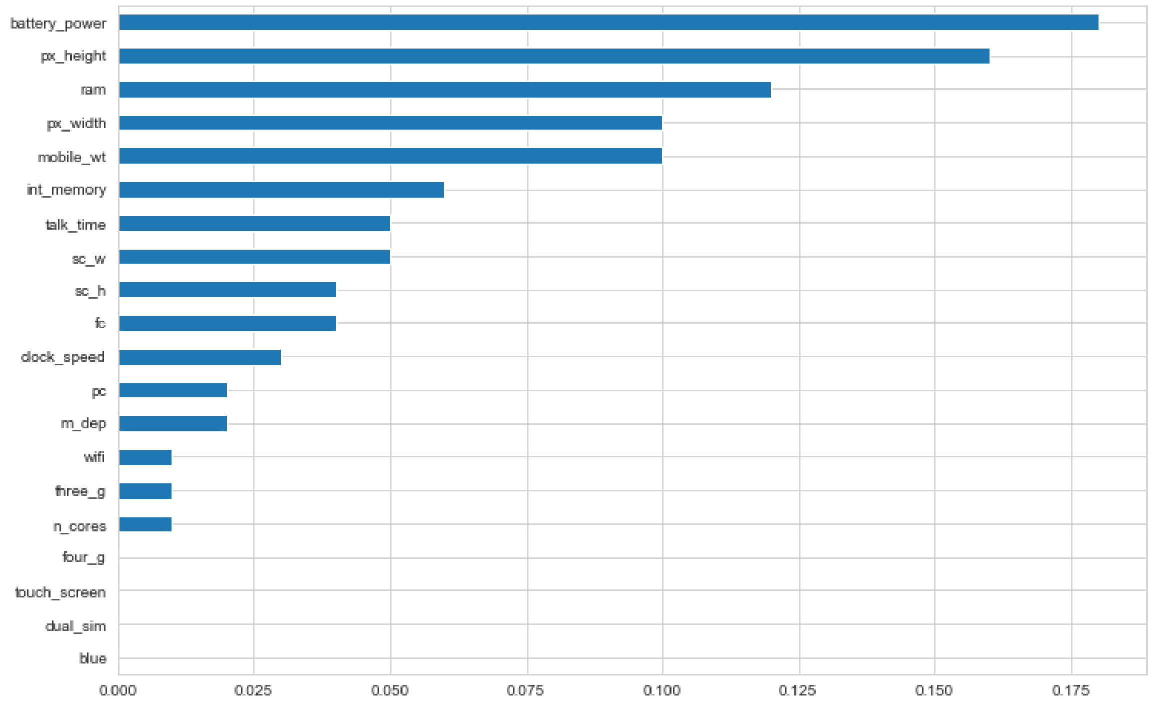
In [64]:

```
get_boosting_classifier(data)
```

```
Score :  0.275
Feature Importance :
 [0.18 0.   0.03 0.   0.04 0.   0.06 0.02 0.1  0.01 0.02 0.16 0.1
0.12
 0.04 0.05 0.05 0.01 0.   0.01]
Score after cross validation :  0.24928571428571428
```

Out[64]:

```
AdaBoostClassifier(n_estimators=100)
```



Lets try the model building with another advanced algorithm i.e. GradientBoostingClassifier.

In [65]:

```
from sklearn.ensemble import GradientBoostingClassifier
```

In [66]:

```
def get_gradient_boosting_classifier(data):

    # Split the data into independent and target attributes
    col_length = len(data.columns)
    X = data.iloc[:,0:col_length - 1]  #independent columns
    y = data.iloc[:,-1]    #target column i.e price range

    #Split the data into training and testing set
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size =0.3)
    clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,max_dep
th=1, random_state=0)
    clf.fit(X_train, y_train)
    print(clf.score(X_test, y_test))

    print("Score : " , clf.feature_importances_)

    importances=pd.Series(clf.feature_importances_, index=X_train.columns).sort_
values()
    importances.plot(kind='barh', figsize=(12,8))

    return clf
```

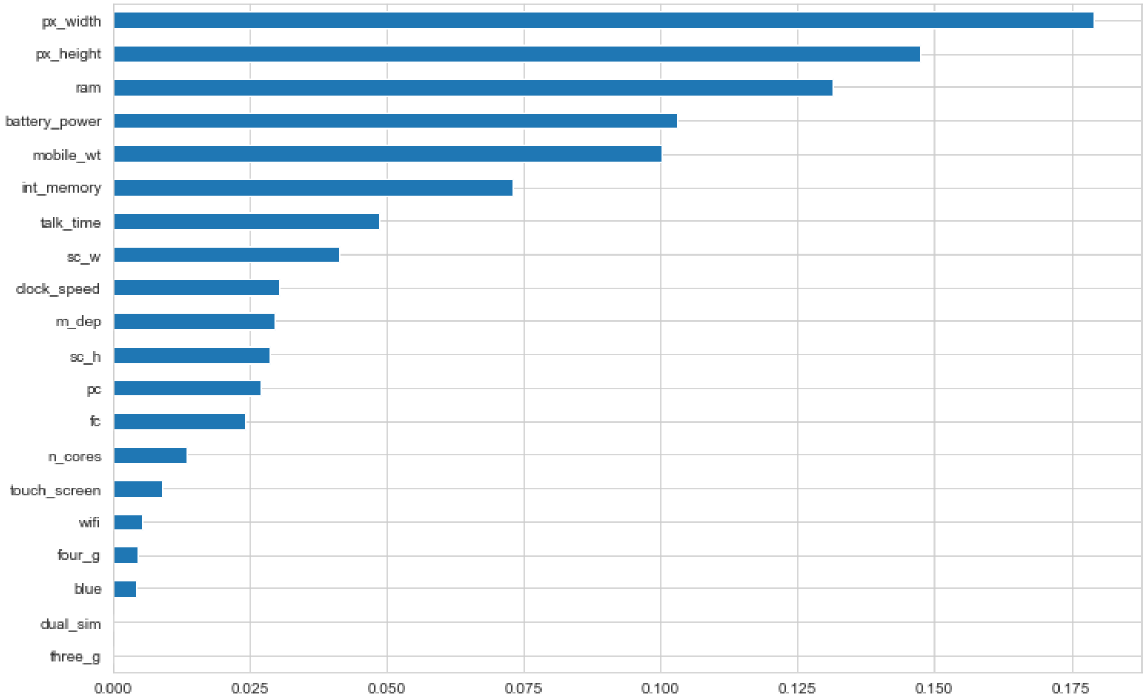
In [67]:

```
get_gradient_boosting_classifier(data)
```

```
0.23833333333333334
Score :  [0.10295247 0.00431918 0.03021268 0.         0.02423577 0.0
0451126
 0.07292279 0.02937785 0.1001392  0.01343848 0.02712459 0.14734599
 0.17892804 0.13144907 0.02867111 0.04131097 0.04857754 0.
 0.00908705 0.00539598]
```

Out[67]:

```
GradientBoostingClassifier(learning_rate=1.0, max_depth=1, random_st
ate=0)
```



4.5 Difference between Filter and Wrapper methods

The main differences between the filter and wrapper methods for feature selection are:

- **Filter methods measure the relevance of features by their correlation with dependent variable while wrapper methods measure the usefulness of a subset of feature by actually training a model on it.**
- **Filter methods are much faster compared to wrapper methods as they do not involve training the models. On the other hand, wrapper methods are computationally very expensive as well.**
- **Filter methods use statistical methods for evaluation of a subset of features while wrapper methods use cross validation.**
- **Filter methods might fail to find the best subset of features in many occasions but wrapper methods can always provide the best subset of features.**
- **Using the subset of features from the wrapper methods make the model more prone to overfitting as compared to using subset of features from the filter methods.**

5. What are the three best features?

Type your answer here :

```
- 1)
- 2)
- 3)
```