1. Create a jupyter notebook to perform all your work in.
2. Pick any business-related data set from Kaggle. Describe in 1 paragraph the dataset and the "business use" you hope to use it for.
3. Using pandas and numpy, wrangle the data to make it manageable.
4. Using k-means and Hierarchical clustering, run clustering on the data set. Describe the results, meaning, and what you learned from the data.
5. Use 5-fold cross-validation and perform classification on the data set using DecisionTrees and Random Forest. Describe the results, what you learned about the data, and how well your classifications performed in testing.
6. Repeat problem 5 but this time perform regression on your data using SVM and XgBoost. Again describe your results, what you learned about the data, and how accurate regression was.
7. Using the python visualization library of your choice, create relevant visualizations for the above tasks and to otherwise help interpret your data. Provide some functional descriptions of your visualization and what can be deduced from them.
8. Submit all of the above in one jupyter notebook along with your dataset

In [1]:

```
# import necessary libraries
import numpy as np
import pandas as pd
import scipy as sp
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from collections import Counter
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_blobs
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.metrics import roc_auc_score,precision_recall_curve,roc_curve
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import r2_score
```

In [2]:

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, c
all drive.mount("/content/drive", force_remount=True).
```

In [3]:

```
df = pd.read_csv("/content/drive/MyDrive/train (1).csv")
```

In [4]:

```
df
```

Out[4]:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_
0	842	0	2.2	0	1	0	7	0.6	1
1	1021	1	0.5	1	0	1	53	0.7	1
2	563	1	0.5	1	2	1	41	0.9	1
3	615	1	2.5	0	0	0	10	0.8	1
4	1821	1	1.2	0	13	1	44	0.6	1
...	...	...	...	...	...	...	...	...	
1995	794	1	0.5	1	0	1	2	0.8	1
1996	1965	1	2.6	1	0	0	39	0.2	1
1997	1911	0	0.9	1	1	1	36	0.7	1
1998	1512	0	0.9	0	4	1	46	0.1	1
1999	510	1	2.0	1	5	1	45	0.9	1

2000 rows × 21 columns

Data Description

The dataset is from Kaggle, which is to use the features of a smart phone to predict its price level. Dataset has 2000 samples and 21 featurs. The target variable is categorical label, from 1 to 4 indicating from cheap to expensive. As for the features, it has both continuous variables including battery power, ram, weight, and height, etc, and categorical variables for example touch screen, wifi, 3g, 4g, and dual sim, etc.

Business Use

Training this machine learning model to predict price range of the smartphone could help people better evaluate the value of smartphone, only if they know the features of smartphone.

Data Preprocessing

In [5]:

```
def describe(df):
    print(f"Dataset Shape: {df.shape}")
    summary = pd.DataFrame(df.dtypes,columns=['dtypes'])
    summary = summary.reset_index()
    summary['Name'] = summary['index']
    summary = summary[['Name','dtypes']]
    summary['Missing'] = df.isnull().sum().values
    summary['Uniques'] = df.nunique().values
    return summary
describe(df)
```

```
Dataset Shape: (2000, 21)
```

Out[5]:

	Name	dtypes	Missing	Uniques
0	battery_power	int64	0	1094
1	blue	int64	0	2
2	clock_speed	float64	0	26
3	dual_sim	int64	0	2
4	fc	int64	0	20
5	four_g	int64	0	2
6	int_memory	int64	0	63
7	m_dep	float64	0	10
8	mobile_wt	int64	0	121
9	n_cores	int64	0	8
10	pc	int64	0	21
11	px_height	int64	0	1137
12	px_width	int64	0	1109
13	ram	int64	0	1562
14	sc_h	int64	0	15
15	sc_w	int64	0	19
16	talk_time	int64	0	19
17	three_g	int64	0	2
18	touch_screen	int64	0	2
19	wifi	int64	0	2
20	price_range	int64	0	4

In [6]:

```
X_cols = [i for i in df.columns if i not in ['price_range']]

X = df[X_cols]
y = df["price_range"]
```

In [7]:

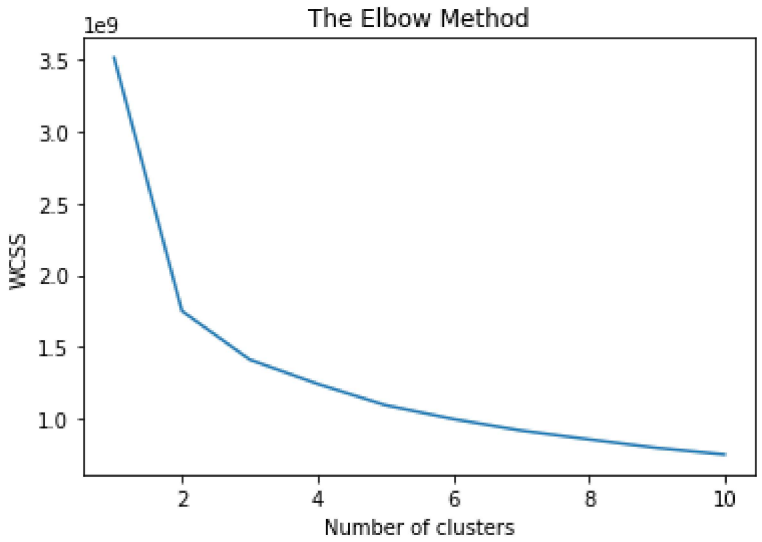
```
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.3, random_state=20
22)
```

Unsupervised Learning

K-Means

In [8]:

```
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i,
        init = 'k-means++',
        random_state = 0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



In [9]:

```
kmeans = KMeans(n_clusters=3).fit(X)
```

In [10]:

```
Counter(kmeans.labels_)
Counter({2: 50, 0: 50, 3: 50, 1: 50})
```

Out[10]:

```
Counter({0: 50, 1: 50, 2: 50, 3: 50})
```

In [11]:

```
kmeans_labels = pd.DataFrame(kmeans.labels_, columns=["cluster"])
```

In [12]:

```
X_kmeans = pd.concat([X, kmeans_labels], axis=1)
```

In [13]:

```
pd.pivot_table(X_kmeans,index=['cluster'])
```

Out[13]:

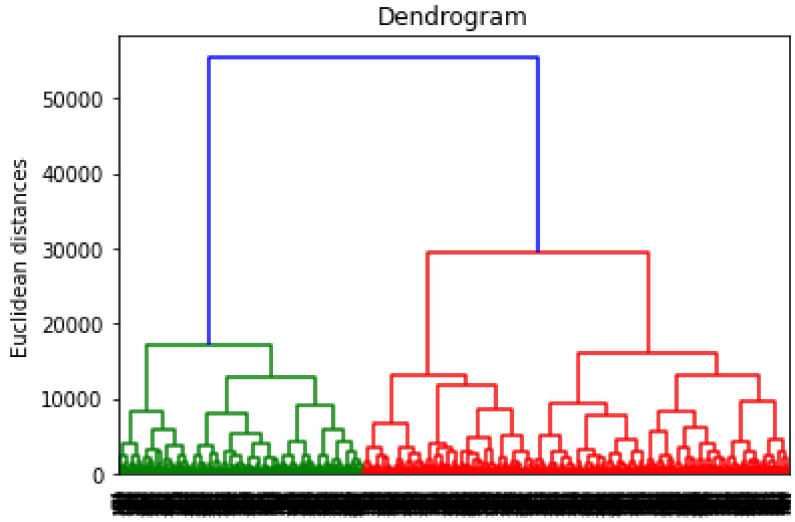
cluster	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	
0	1250.734752	0.473759	1.535461	0.489362	4.188652	0.514894	31.588652	0
1	1245.815359	0.509804	1.511438	0.545752	4.397059	0.529412	33.022876	0
2	1219.370425	0.503660	1.518302	0.497804	4.355783	0.521230	31.644217	0

The k menas clustering algorithm split the dataset to three group. From the pivot table group by cluster, we can see that the major different between different group is battery power, int memory, dul_sum, and px_width. The above results suggest that the major differences between smartphone depend on these hardwares.

Hierarchical Clustering

In [14]:

```
dendrogram = sch.dendrogram(sch.linkage(X, method = 'ward'))
plt.title('Dendrogram')
plt.ylabel('Euclidean distances')
plt.show()
```



In [15]:

```
hc = AgglomerativeClustering(n_clusters = 4, affinity = 'euclidean', linkage = 'ward')
hc_labels = pd.DataFrame(hc.fit_predict(X), columns=["cluster"])
```

In [16]:

```
X_hc = pd.concat([X, hc_labels], axis=1)
pd.pivot_table(X_hc,index=['cluster'])
```

Out[16]:

cluster	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	
0	1199.400000	0.492105	1.540395	0.486842	4.352632	0.514474	32.232895	0
1	1344.449416	0.488327	1.546693	0.480545	4.266537	0.523346	31.573930	0
2	1216.435897	0.514793	1.505917	0.556213	4.416174	0.536489	32.623274	0
3	1176.771689	0.474886	1.439726	0.547945	4.013699	0.506849	31.173516	0

The hierarchy clustering split the dataset to four clusters. Similar to the K-means cluster, it separate the groups mainly according to the mobile hardware and phone appearance, for example, ram, px_height, px_width, and mobile weight.

Supervised Learning

Decision tree

In [17]:

```
# define models and parameters
model = DecisionTreeClassifier()
max_features = ['auto', 'sqrt', 'log2']
ccp_alpha = [0.1, .01, .001]
max_depth = [5, 6, 7, 8, 9]
criterion = ['gini', 'entropy']
# define grid search
grid = dict(max_features=max_features, ccp_alpha=ccp_alpha, max_depth=max_depth, criter
ion=criterion)
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring=
'accuracy',error_score=0)
grid_result = grid_search.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.617381 using {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max_dept
h': 8, 'max_features': 'log2'}
0.321190 (0.107835) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 5, 'max_features': 'auto'}
0.313810 (0.119208) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 5, 'max_features': 'sqrt'}
0.337143 (0.115166) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 5, 'max_features': 'log2'}
0.321190 (0.107835) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 6, 'max_features': 'auto'}
0.336905 (0.114833) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 6, 'max_features': 'sqrt'}
0.272143 (0.060920) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 6, 'max_features': 'log2'}
0.288333 (0.083038) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 7, 'max_features': 'auto'}
0.304048 (0.097397) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 7, 'max_features': 'sqrt'}
0.324286 (0.140820) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 7, 'max_features': 'log2'}
0.330000 (0.127387) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 8, 'max_features': 'auto'}
0.271905 (0.060985) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 8, 'max_features': 'sqrt'}
0.313810 (0.119969) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 8, 'max_features': 'log2'}
0.320952 (0.107980) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 9, 'max_features': 'auto'}
0.272143 (0.060920) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 9, 'max_features': 'sqrt'}
0.320476 (0.131588) with: {'ccp_alpha': 0.1, 'criterion': 'gini', 'max_dep
th': 9, 'max_features': 'log2'}
0.380952 (0.123502) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 5, 'max_features': 'auto'}
0.371667 (0.117387) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 5, 'max_features': 'sqrt'}
0.377381 (0.145894) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 5, 'max_features': 'log2'}
0.450476 (0.127475) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 6, 'max_features': 'auto'}
0.367857 (0.138971) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 6, 'max_features': 'sqrt'}
0.422619 (0.148769) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 6, 'max_features': 'log2'}
0.396429 (0.144555) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 7, 'max_features': 'auto'}
0.389524 (0.130061) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 7, 'max_features': 'sqrt'}
0.439762 (0.088216) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 7, 'max_features': 'log2'}
0.350952 (0.119220) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 8, 'max_features': 'auto'}
0.446667 (0.150082) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 8, 'max_features': 'sqrt'}
0.373810 (0.120251) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 8, 'max_features': 'log2'}
0.433810 (0.165420) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 9, 'max_features': 'auto'}
0.373810 (0.112006) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
depth': 9, 'max_features': 'sqrt'}
0.401429 (0.127361) with: {'ccp_alpha': 0.1, 'criterion': 'entropy', 'max_
```

```
depth': 9, 'max_features': 'log2'}
0.456429 (0.115957) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 5, 'max_features': 'auto'}
0.505952 (0.118871) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 5, 'max_features': 'sqrt'}
0.564286 (0.093723) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 5, 'max_features': 'log2'}
0.515952 (0.122439) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 6, 'max_features': 'auto'}
0.494524 (0.135129) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 6, 'max_features': 'sqrt'}
0.529524 (0.135466) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 6, 'max_features': 'log2'}
0.473810 (0.119150) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 7, 'max_features': 'auto'}
0.500952 (0.131090) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 7, 'max_features': 'sqrt'}
0.493333 (0.116049) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 7, 'max_features': 'log2'}
0.514286 (0.078083) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 8, 'max_features': 'auto'}
0.459048 (0.132627) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 8, 'max_features': 'sqrt'}
0.543095 (0.098418) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 8, 'max_features': 'log2'}
0.535476 (0.117155) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 9, 'max_features': 'auto'}
0.511190 (0.116641) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 9, 'max_features': 'sqrt'}
0.503333 (0.120108) with: {'ccp_alpha': 0.01, 'criterion': 'gini', 'max_de
pth': 9, 'max_features': 'log2'}
0.543333 (0.107229) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 5, 'max_features': 'auto'}
0.546429 (0.104361) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 5, 'max_features': 'sqrt'}
0.518333 (0.119905) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 5, 'max_features': 'log2'}
0.561667 (0.124741) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 6, 'max_features': 'auto'}
0.580714 (0.102791) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 6, 'max_features': 'sqrt'}
0.576905 (0.107601) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 6, 'max_features': 'log2'}
0.551429 (0.143195) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 7, 'max_features': 'auto'}
0.601190 (0.107481) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 7, 'max_features': 'sqrt'}
0.501429 (0.104871) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 7, 'max_features': 'log2'}
0.567143 (0.104444) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 8, 'max_features': 'auto'}
0.555952 (0.087255) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 8, 'max_features': 'sqrt'}
0.617381 (0.111406) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 8, 'max_features': 'log2'}
0.538810 (0.079869) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 9, 'max_features': 'auto'}
0.593095 (0.084218) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 9, 'max_features': 'sqrt'}
0.570000 (0.106249) with: {'ccp_alpha': 0.01, 'criterion': 'entropy', 'max
_depth': 9, 'max_features': 'log2'}
```

```
0.534286 (0.107656) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 5, 'max_features': 'auto'}
0.521190 (0.116922) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 5, 'max_features': 'sqrt'}
0.461667 (0.084005) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 5, 'max_features': 'log2'}
0.521190 (0.100817) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 6, 'max_features': 'auto'}
0.565000 (0.111202) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 6, 'max_features': 'sqrt'}
0.531667 (0.118069) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 6, 'max_features': 'log2'}
0.523095 (0.078581) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 7, 'max_features': 'auto'}
0.577857 (0.102330) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 7, 'max_features': 'sqrt'}
0.585476 (0.119334) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 7, 'max_features': 'log2'}
0.587381 (0.087578) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 8, 'max_features': 'auto'}
0.557857 (0.094493) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 8, 'max_features': 'sqrt'}
0.565714 (0.106807) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 8, 'max_features': 'log2'}
0.597619 (0.080079) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 9, 'max_features': 'auto'}
0.557857 (0.091679) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 9, 'max_features': 'sqrt'}
0.561905 (0.101642) with: {'ccp_alpha': 0.001, 'criterion': 'gini', 'max_d
epth': 9, 'max_features': 'log2'}
0.457143 (0.122461) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 5, 'max_features': 'auto'}
0.505714 (0.127601) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 5, 'max_features': 'sqrt'}
0.542619 (0.139340) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 5, 'max_features': 'log2'}
0.575952 (0.119056) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 6, 'max_features': 'auto'}
0.486667 (0.127572) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 6, 'max_features': 'sqrt'}
0.517619 (0.078760) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 6, 'max_features': 'log2'}
0.560476 (0.097620) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 7, 'max_features': 'auto'}
0.485000 (0.136210) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 7, 'max_features': 'sqrt'}
0.575952 (0.115892) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 7, 'max_features': 'log2'}
0.591667 (0.107734) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 8, 'max_features': 'auto'}
0.580952 (0.067444) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 8, 'max_features': 'sqrt'}
0.563095 (0.089030) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 8, 'max_features': 'log2'}
0.519048 (0.099392) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 9, 'max_features': 'auto'}
0.573571 (0.063885) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 9, 'max_features': 'sqrt'}
0.608095 (0.081561) with: {'ccp_alpha': 0.001, 'criterion': 'entropy', 'ma
x_depth': 9, 'max_features': 'log2'}
```

In [18]:

```
clf_tree = DecisionTreeClassifier(ccp_alpha=0.001, criterion="gini", max_depth=7, max_f
eatures="log2").fit(X_train, y_train)
y_pred_tree = clf_tree.predict(X_test)
cm_tree = confusion_matrix(y_test, y_pred_tree)
```

In [19]:

```
cm_tree
```

Out[19]:

```
array([[107,  22,   5,   8],
       [ 53,  64,  20,   9],
       [ 50,  28,  62,  14],
       [ 21,   8,  26, 103]])
```

In [20]:

```
print(classification_report(y_test, y_pred_tree))
```

```
              precision    recall  f1-score   support

           0       0.46      0.75      0.57       142
           1       0.52      0.44      0.48       146
           2       0.55      0.40      0.46       154
           3       0.77      0.65      0.71       158

    accuracy                           0.56       600
   macro avg       0.58      0.56      0.56       600
weighted avg       0.58      0.56      0.56       600
```

The confusion matrix of the optimal decision tree model suggests that the model performs not good, with an overall accuracy of 56%, and average recall and percision around 56% and 58%. It means that the model makes wrong predictions on about 50% of samples.

We can know from the model that decision tree is prone to overfitting, because the accuracy in test dataset is lower than accuracy in training dataset.

Random Forest

In [21]:

```
# define models and parameters
model = RandomForestClassifier()
n_estimators = [10, 100, 1000]
max_features = ['sqrt', 'log2']
# define grid search
grid = dict(n_estimators=n_estimators,max_features=max_features)
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring=
'accuracy',error_score=0)
grid_result = grid_search.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.875714 using {'max_features': 'log2', 'n_estimators': 1000}
0.779762 (0.026716) with: {'max_features': 'sqrt', 'n_estimators': 10}
0.866667 (0.014677) with: {'max_features': 'sqrt', 'n_estimators': 100}
0.869762 (0.016333) with: {'max_features': 'sqrt', 'n_estimators': 1000}
0.777143 (0.028446) with: {'max_features': 'log2', 'n_estimators': 10}
0.867857 (0.019605) with: {'max_features': 'log2', 'n_estimators': 100}
0.875714 (0.017311) with: {'max_features': 'log2', 'n_estimators': 1000}
```

In [22]:

```
clf_rf = RandomForestClassifier(max_features="sqrt", n_estimators=1000).fit(X_train, y_
train)
y_pred_rf = clf_rf.predict(X_test)
cm_rf = confusion_matrix(y_test, y_pred_rf)
```

In [23]:

```
cm_rf
```

Out[23]:

```
array([[136,   6,   0,   0],
       [ 15, 121,  10,   0],
       [  0,  13, 133,   8],
       [  0,   0,  10, 148]])
```

In [ ]:

```
print(classification_report(y_test, y_pred_rf))
```

```
              precision    recall  f1-score   support

           0       0.90      0.96      0.93       142
           1       0.86      0.83      0.85       146
           2       0.87      0.84      0.86       154
           3       0.94      0.94      0.94       158

    accuracy                           0.89       600
   macro avg       0.89      0.89      0.89       600
weighted avg       0.89      0.89      0.89       600
```

Random Forest model performs excellent in this dataset. It has an overall accuracy of 89%, and average precision and recall of 89% as well. The precision and recall in each label classes are high.

In terms of data preprocessing, random forest has great advantages of handling binary features, categorical features, and numerical features. Thus there is very little pre-processing that needs to be done because the data does not need to be rescaled or transformed. Other advantages of random forest are it can handle high dimensional data since we are working with subsets of data, and it is robust to outliers and nonlinear features

SVM

In [24]:

```
# define model and parameters
model = SVC()
kernel = ['poly', 'rbf', 'sigmoid']
C = [50, 10, 1.0, 0.1, 0.01]
gamma = ['scale']
# define grid search
grid = dict(kernel=kernel,C=C,gamma=gamma)
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring=
'accuracy',error_score=0)
grid_result = grid_search.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.958333 using {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}
0.958333 (0.009554) with: {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}
0.955952 (0.008618) with: {'C': 50, 'gamma': 'scale', 'kernel': 'rbf'}
0.175476 (0.017490) with: {'C': 50, 'gamma': 'scale', 'kernel': 'sigmoid'}
0.955000 (0.013222) with: {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}
0.951905 (0.008637) with: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
0.172619 (0.015577) with: {'C': 10, 'gamma': 'scale', 'kernel': 'sigmoid'}
0.946429 (0.012910) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'poly'}
0.940476 (0.012688) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'rbf'}
0.187381 (0.017146) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'sigmoi
d'}
0.917857 (0.016853) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'poly'}
0.888333 (0.012941) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'rbf'}
0.454048 (0.026975) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'sigmoi
d'}
0.699524 (0.021660) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'poly'}
0.293571 (0.012108) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'rbf'}
0.255714 (0.001750) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'sigmoi
d'}
```

In [26]:

```
svm = SVC(C=50, gamma="scale", kernel="poly").fit(X_train, y_train)
```

In [27]:

```
y_pred_svm = svm.predict(X_test)
```

In [34]:

```
cm_svm = confusion_matrix(y_test, y_pred_svm)
cm_svm
```

Out[34]:

```
array([[140,   2,   0,   0],
       [  2, 141,   3,   0],
       [  0,   1, 153,   0],
       [  0,   0,   3, 155]])
```

In [32]:

```
print(classification_report(y_test, y_pred_svm))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       142
           1       0.98      0.97      0.97       146
           2       0.96      0.99      0.98       154
           3       1.00      0.98      0.99       158

    accuracy                           0.98       600
   macro avg       0.98      0.98      0.98       600
weighted avg       0.98      0.98      0.98       600
```

In [29]:

```
r2_score(y_test, y_pred_svm)
```

Out[29]:

```
0.98530773614475
```

SVM's optimal model uses C of 50, gamma of scale and poly kernel. SVM performs better than decision tree and random forest. It is worthy to be mentioned that SVM performs better in testing set than in the training set which indicates that SVM has great generalization ability

XgBoost

In [25]:

```
# define models and parameters
model = XGBClassifier()
n_estimators = [10, 100, 1000]
learning_rate = [0.001, 0.01, 0.1]
subsample = [0.5, 0.7, 1.0]
max_depth = [3, 7, 9]
# define grid search
grid = dict(learning_rate=learning_rate, n_estimators=n_estimators, subsample=subsample
, max_depth=max_depth)
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring=
'accuracy',error_score=0)
grid_result = grid_search.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.911429 using {'learning_rate': 0.1, 'max_depth': 3, 'n_estimator
s': 1000, 'subsample': 0.5}
0.781667 (0.017146) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 10, 'subsample': 0.5}
0.786190 (0.013227) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 10, 'subsample': 0.7}
0.774762 (0.023163) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 10, 'subsample': 1.0}
0.786190 (0.017587) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 100, 'subsample': 0.5}
0.787381 (0.012769) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 100, 'subsample': 0.7}
0.778095 (0.019556) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 100, 'subsample': 1.0}
0.803333 (0.015346) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 1000, 'subsample': 0.5}
0.800000 (0.015703) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 1000, 'subsample': 0.7}
0.795000 (0.021301) with: {'learning_rate': 0.001, 'max_depth': 3, 'n_esti
mators': 1000, 'subsample': 1.0}
0.847143 (0.019957) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 10, 'subsample': 0.5}
0.851429 (0.015854) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 10, 'subsample': 0.7}
0.840000 (0.018313) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 10, 'subsample': 1.0}
0.852619 (0.015456) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 100, 'subsample': 0.5}
0.858333 (0.013278) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 100, 'subsample': 0.7}
0.843095 (0.014432) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 100, 'subsample': 1.0}
0.873571 (0.015474) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 1000, 'subsample': 0.5}
0.875952 (0.017225) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 1000, 'subsample': 0.7}
0.858333 (0.013469) with: {'learning_rate': 0.001, 'max_depth': 7, 'n_esti
mators': 1000, 'subsample': 1.0}
0.852619 (0.019098) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 10, 'subsample': 0.5}
0.855952 (0.015740) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 10, 'subsample': 0.7}
0.839048 (0.020054) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 10, 'subsample': 1.0}
0.856905 (0.014013) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 100, 'subsample': 0.5}
0.863571 (0.013814) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 100, 'subsample': 0.7}
0.845476 (0.017664) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 100, 'subsample': 1.0}
0.878571 (0.015865) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 1000, 'subsample': 0.5}
0.879286 (0.017360) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 1000, 'subsample': 0.7}
0.859762 (0.014074) with: {'learning_rate': 0.001, 'max_depth': 9, 'n_esti
mators': 1000, 'subsample': 1.0}
0.782619 (0.017490) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 10, 'subsample': 0.5}
0.786429 (0.015386) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 10, 'subsample': 0.7}
0.777857 (0.019308) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
```

```
ators': 10, 'subsample': 1.0}
0.803095 (0.014689) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 100, 'subsample': 0.5}
0.801667 (0.014975) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 100, 'subsample': 0.7}
0.795000 (0.021301) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 100, 'subsample': 1.0}
0.907381 (0.014723) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 1000, 'subsample': 0.5}
0.908810 (0.014631) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 1000, 'subsample': 0.7}
0.894762 (0.014218) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estim
ators': 1000, 'subsample': 1.0}
0.851667 (0.021344) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 10, 'subsample': 0.5}
0.853810 (0.018648) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 10, 'subsample': 0.7}
0.841429 (0.013414) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 10, 'subsample': 1.0}
0.872143 (0.017603) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 100, 'subsample': 0.5}
0.876429 (0.017143) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 100, 'subsample': 0.7}
0.859762 (0.014549) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 100, 'subsample': 1.0}
0.910952 (0.013071) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 1000, 'subsample': 0.5}
0.904286 (0.012791) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 1000, 'subsample': 0.7}
0.891905 (0.014373) with: {'learning_rate': 0.01, 'max_depth': 7, 'n_estim
ators': 1000, 'subsample': 1.0}
0.855714 (0.017291) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 10, 'subsample': 0.5}
0.858571 (0.017487) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 10, 'subsample': 0.7}
0.844048 (0.017625) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 10, 'subsample': 1.0}
0.876190 (0.018519) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 100, 'subsample': 0.5}
0.879524 (0.014194) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 100, 'subsample': 0.7}
0.859524 (0.013469) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 100, 'subsample': 1.0}
0.910714 (0.012976) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 1000, 'subsample': 0.5}
0.902381 (0.011857) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 1000, 'subsample': 0.7}
0.889524 (0.012266) with: {'learning_rate': 0.01, 'max_depth': 9, 'n_estim
ators': 1000, 'subsample': 1.0}
0.801190 (0.015954) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 10, 'subsample': 0.5}
0.797381 (0.012472) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 10, 'subsample': 0.7}
0.793333 (0.019986) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 10, 'subsample': 1.0}
0.905476 (0.013916) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 100, 'subsample': 0.5}
0.907619 (0.013544) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 100, 'subsample': 0.7}
0.896429 (0.014226) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 100, 'subsample': 1.0}
```

```
0.911429 (0.012386) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 1000, 'subsample': 0.5}
0.910238 (0.013731) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 1000, 'subsample': 0.7}
0.899524 (0.015750) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estima
tors': 1000, 'subsample': 1.0}
0.863810 (0.016745) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 10, 'subsample': 0.5}
0.870476 (0.014313) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 10, 'subsample': 0.7}
0.859524 (0.015246) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 10, 'subsample': 1.0}
0.906429 (0.014534) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 100, 'subsample': 0.5}
0.905000 (0.012764) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 100, 'subsample': 0.7}
0.894286 (0.015908) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 100, 'subsample': 1.0}
0.907619 (0.013291) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 1000, 'subsample': 0.5}
0.909048 (0.011800) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 1000, 'subsample': 0.7}
0.899524 (0.012294) with: {'learning_rate': 0.1, 'max_depth': 7, 'n_estima
tors': 1000, 'subsample': 1.0}
0.868095 (0.015234) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 10, 'subsample': 0.5}
0.871667 (0.014953) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 10, 'subsample': 0.7}
0.860000 (0.017013) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 10, 'subsample': 1.0}
0.903095 (0.014455) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 100, 'subsample': 0.5}
0.904048 (0.011434) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 100, 'subsample': 0.7}
0.890000 (0.013502) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 100, 'subsample': 1.0}
0.905476 (0.013669) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 1000, 'subsample': 0.5}
0.908095 (0.015009) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 1000, 'subsample': 0.7}
0.897381 (0.015783) with: {'learning_rate': 0.1, 'max_depth': 9, 'n_estima
tors': 1000, 'subsample': 1.0}
```

In [33]:

```
xgb = XGBClassifier(learning_rate=0.1, max_depth=3, n_estimators=1000, subsample=0.5).f
it(X_train, y_train)
```

In [41]:

```
y_pred_xgb = xgb.predict(X_test)
cm_xgb = confusion_matrix(y_test, y_pred_xgb)
cm_xgb
```

Out[41]:

```
array([[137,   5,   0,   0],
       [  8, 130,   8,   0],
       [  0,   9, 138,   7],
       [  0,   0,  10, 148]])
```

In [36]:

```
print(classification_report(y_test, y_pred_xgb))
```

```
              precision    recall  f1-score   support

           0       0.94      0.96      0.95       142
           1       0.90      0.89      0.90       146
           2       0.88      0.90      0.89       154
           3       0.95      0.94      0.95       158

    accuracy                           0.92       600
   macro avg       0.92      0.92      0.92       600
weighted avg       0.92      0.92      0.92       600
```

In [38]:

```
r2_score(y_test, y_pred_xgb)
```

Out[38]:

```
0.937223963527568
```

Using grid search method, I find the optimal model with learning rate of 0.1, maximum depth equals to 3, n estimeators equals to 1000 and subsample equals to 0.5. XGBoost also performs good in this problem according to the confusion matrix. And XGBoost has really fast training speed.

Data Visualization

EDA

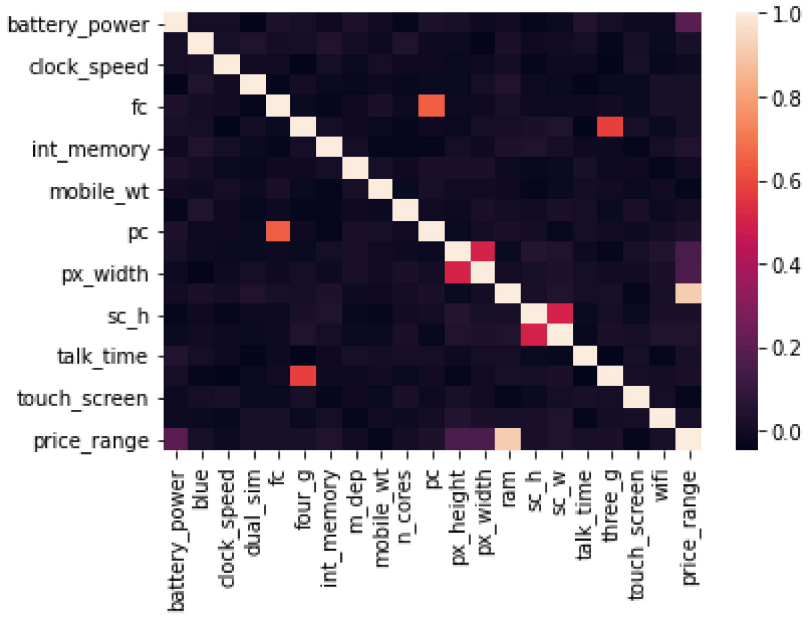
By plotting the correlation plot, we can do feature selection for the modeling that drop the high-correlated features to avoid multi-conlinearity. Fortunately, there is no highly correlated features.

In [48]:

```
sns.heatmap(df.corr())
```

Out[48]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb4cabba210>
```



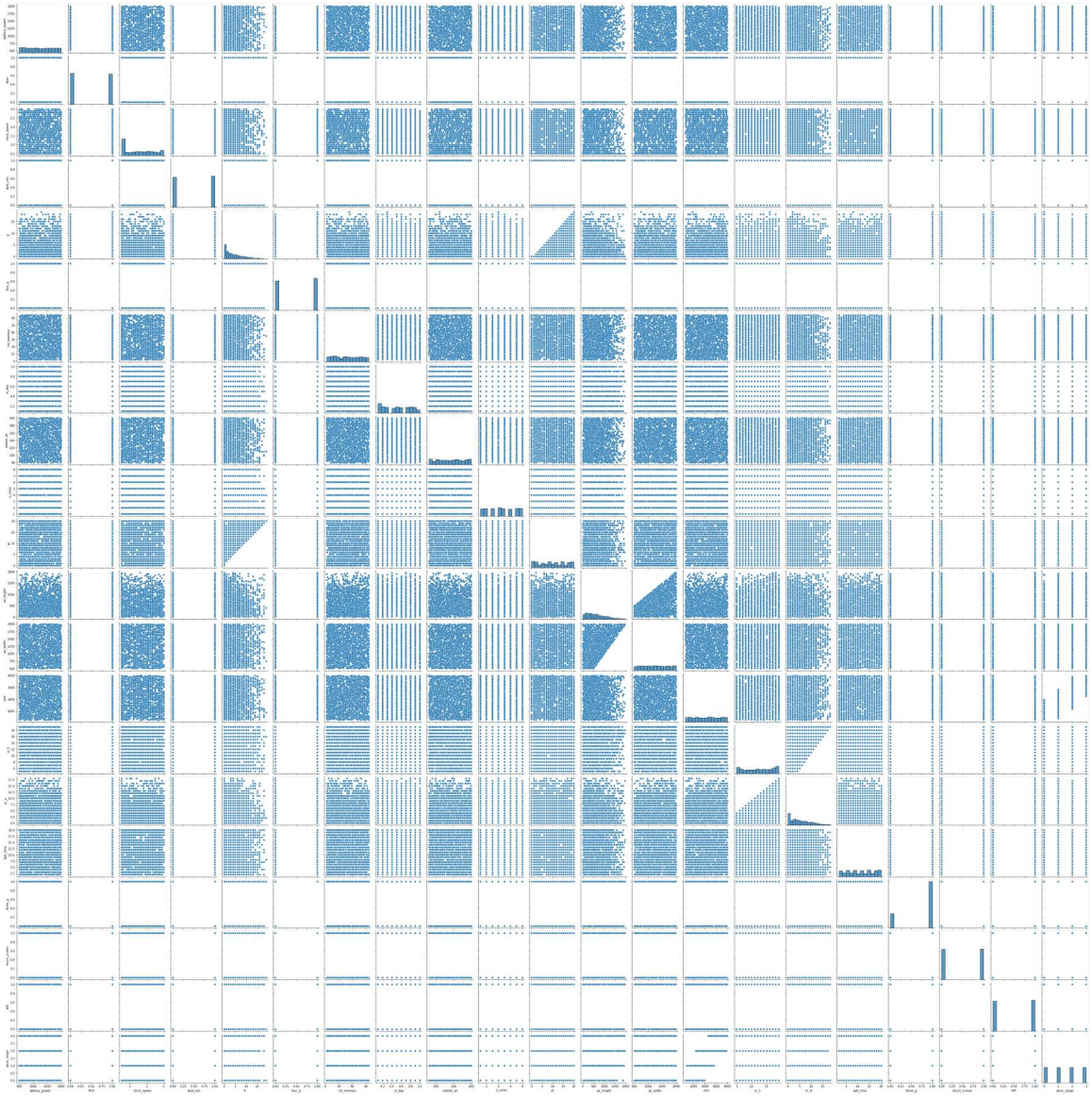
By plotting the distribution of each feature and the joint distributions of different features, we can quickly know the distribution of the features. Further, we could look at the distribution by target to find deeper relationship between dependent and independent variables. In our dataset, majority of the features are categorical features, and the target varible is also categorical. Therefore, the relationship between features and label is not so clear in the plot.

In [49]:

```
sns.pairplot(df)
```

Out[49]:

```
<seaborn.axisgrid.PairGrid at 0x7fb4cbcade10>
```



Confusion Matrix

Using visualized confusion matrix, we can analyse the performance of models more intuitively and it is much easier for other people to understand the results.

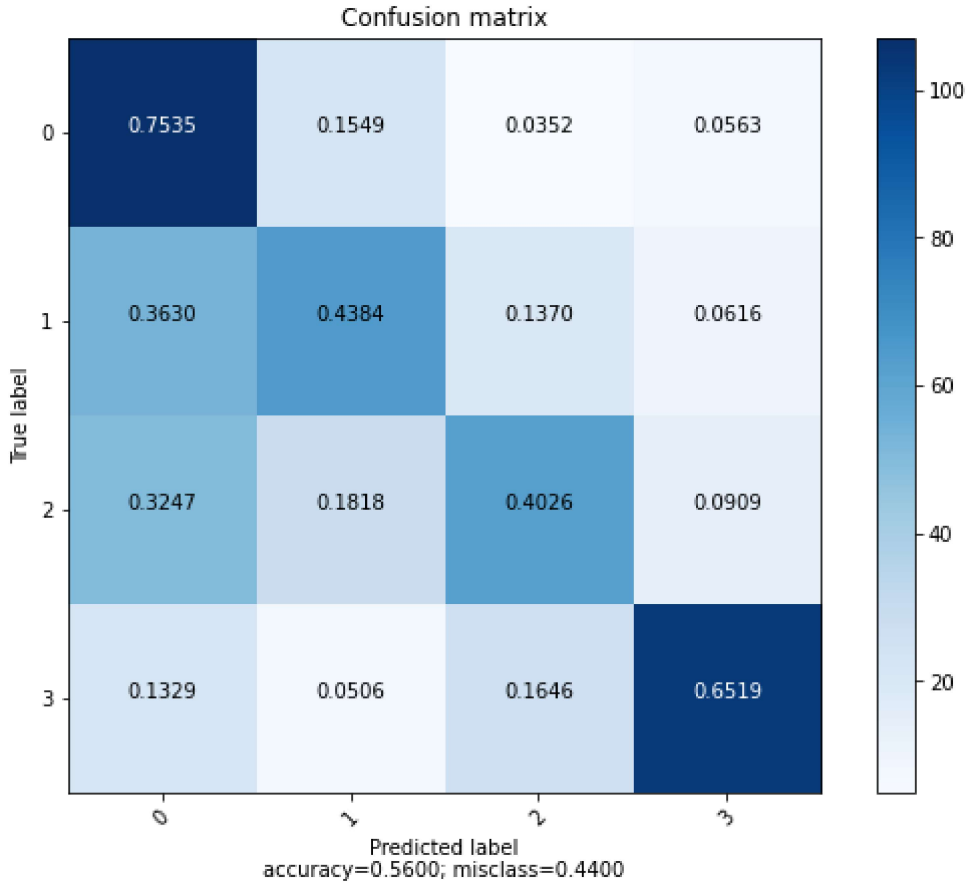
In [39]:

```
def plot_confusion_matrix(cm,
                          target_names,
                          title='Confusion matrix',
                          cmap=None,
                          normalize=True):
    """
    given a sklearn confusion matrix (cm), make a nice plot

    Arguments
    ---------
    cm:           confusion matrix from sklearn.metrics.confusion_matrix

    target_names: given classification classes such as [0, 1, 2]
                  the class names, for example: ['high', 'medium', 'low']

    title:        the text to display at the top of the matrix

    cmap:         the gradient of the values displayed from matplotlib.pyplot.cm
                  see http://matplotlib.org/examples/color/colormaps_reference.html
                  plt.get_cmap('jet') or plt.cm.Blues

    normalize:    If False, plot the raw numbers
                  If True, plot the proportions

    Usage
    -----
    plot_confusion_matrix(cm           = cm,                  # confusion matrix create
d by
                                                              # sklearn.metrics.confusi
on_matrix
                          normalize    = True,                # show proportions
                          target_names = y_labels_vals,       # list of names of the cl
asses
                          title        = best_estimator_name) # title of graph

    Citiation
    ---------
    http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.
html

    """
    import matplotlib.pyplot as plt
    import numpy as np
    import itertools

    accuracy = np.trace(cm) / float(np.sum(cm))
    misclass = 1 - accuracy

    if cmap is None:
        cmap = plt.get_cmap('Blues')

    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()

    if target_names is not None:
        tick_marks = np.arange(len(target_names))
        plt.xticks(tick_marks, target_names, rotation=45)
```

```
        plt.yticks(tick_marks, target_names)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]


    thresh = cm.max() / 1.5 if normalize else cm.max() / 2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        if normalize:
            plt.text(j, i, "{:0.4f}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
        else:
            plt.text(j, i, "{:,}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")


    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label\naccuracy={:0.4f}; misclass={:0.4f}'.format(accuracy, m
isclass))
    plt.show()
```

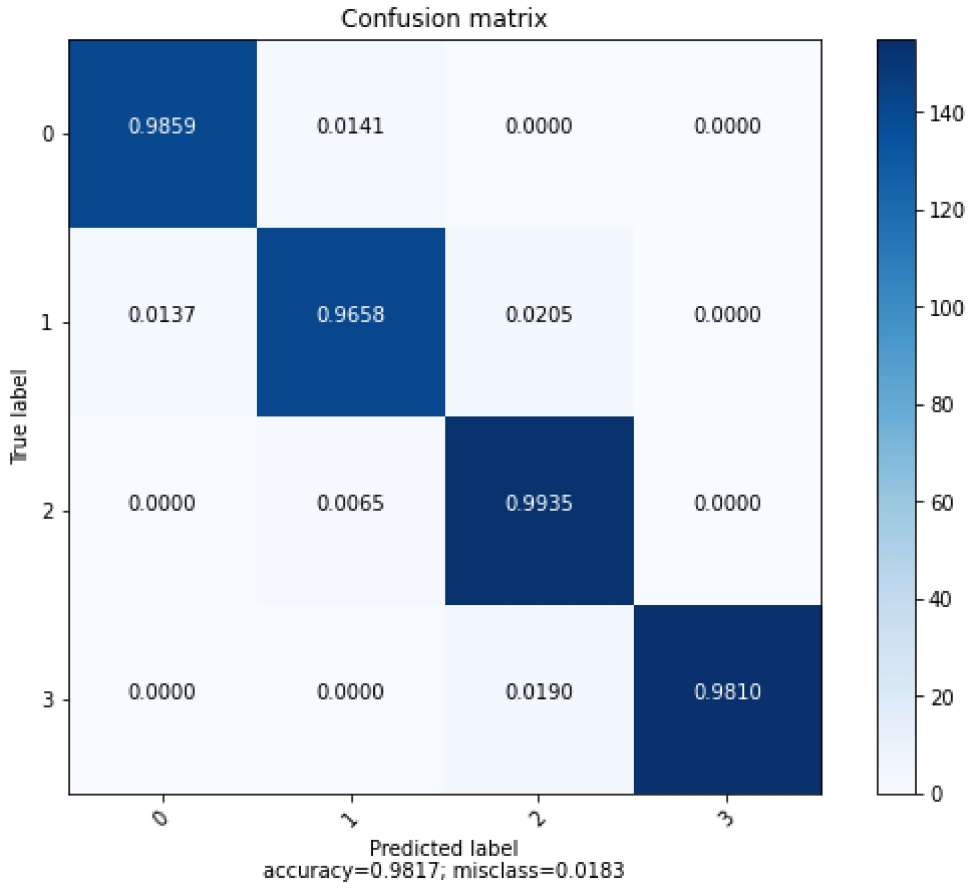
In [47]:

```
plot_confusion_matrix(cm_tree, target_names=[0,1,2,3], title='Confusion matrix', cmap=N
one, normalize=True)
```



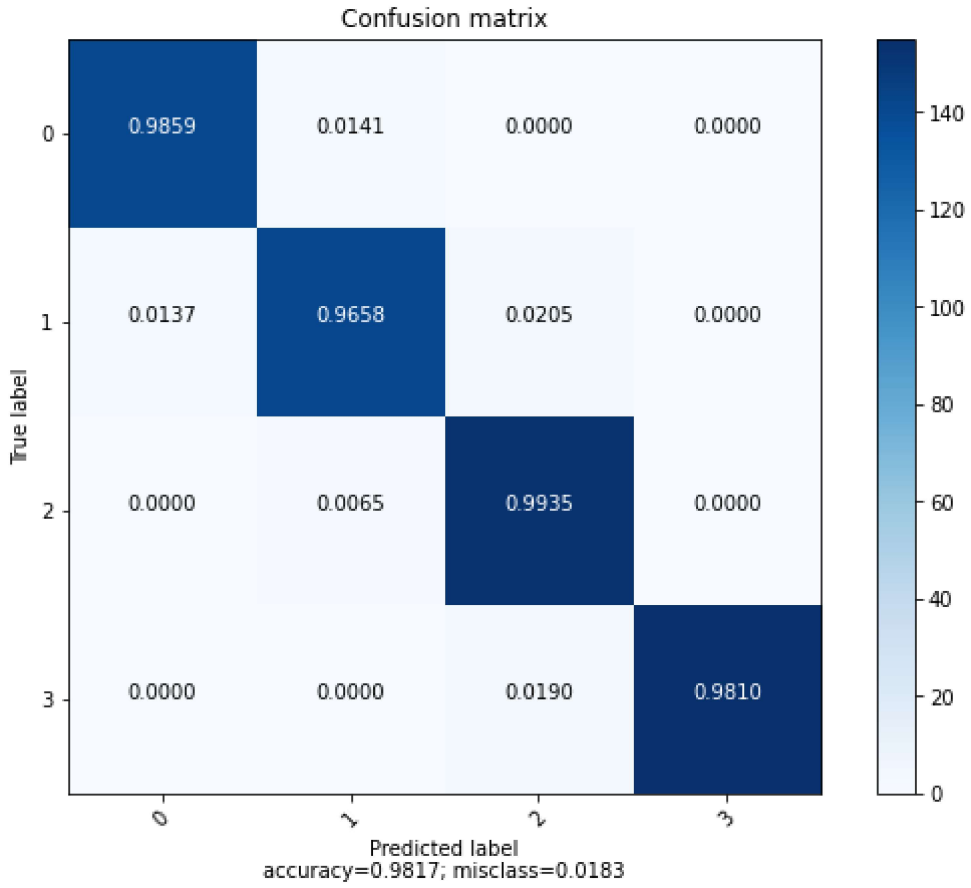
In [46]:

```
plot_confusion_matrix(cm_rf, target_names=[0,1,2,3], title='Confusion matrix', cmap=None, normalize=True)
```



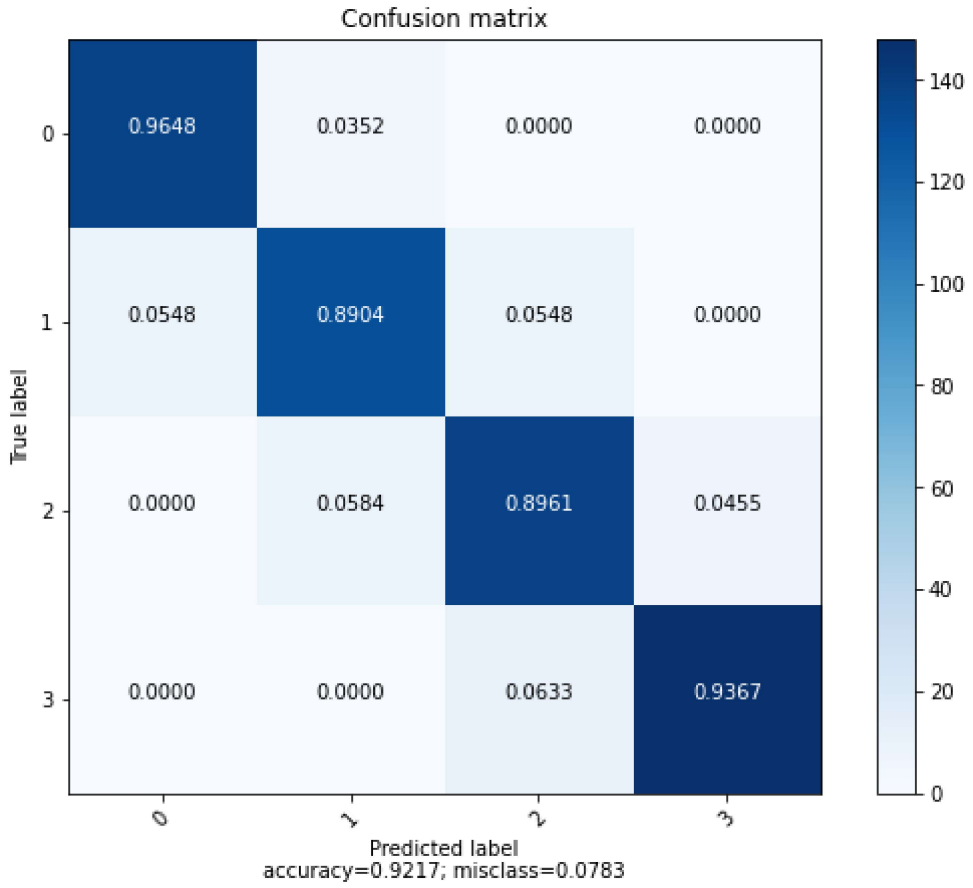
In [45]:

```
plot_confusion_matrix(cm_svm, target_names=[0,1,2,3], title='Confusion matrix', cmap=None, normalize=True)
```



In [44]:

```
plot_confusion_matrix(cm_xgb, target_names=[0,1,2,3], title='Confusion matrix', cmap=None, normalize=True)
```



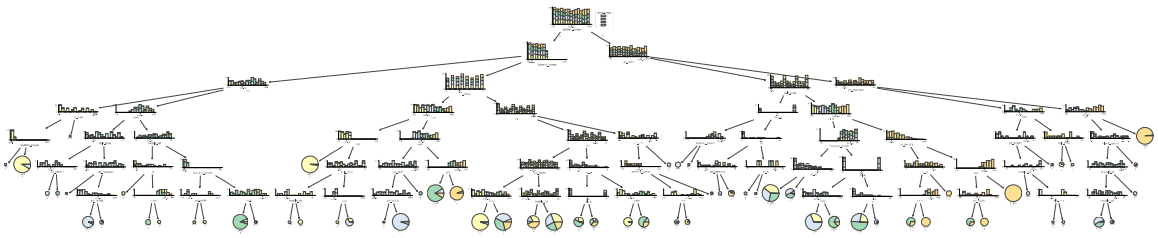
Decision Tree Plot

In [56]:

```
from dtreeviz.trees import dtreeviz # remember to load the package

viz = dtreeviz(clf_tree, X, y,
               target_name="price_range",
               feature_names=X.columns,
               class_names=[0,1,2,3])

viz
```

```
findfont: Font family ['Arial'] not found. Falling back to DejaVu Sans.
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X
does not have valid feature names, but DecisionTreeClassifier was fitted w
ith feature names
  "X does not have valid feature names, but"
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:3208: Vis
ibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different le
ngths or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray.
  return asarray(a).size
/usr/local/lib/python3.7/dist-packages/matplotlib/cbook/__init__.py:1376:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequence
s (which is a list-or-tuple of lists-or-tuples-or ndarrays with different
lengths or shapes) is deprecated. If you meant to do this, you must specif
y 'dtype=object' when creating the ndarray.
  X = np.atleast_1d(X.T if isinstance(X, np.ndarray) else np.asarray(X))
findfont: Font family ['Arial'] not found. Falling back to DejaVu Sans.
findfont: Font family ['Arial'] not found. Falling back to DejaVu Sans.
findfont: Font family ['Arial'] not found. Falling back to DejaVu Sans.
```

Out[56]:

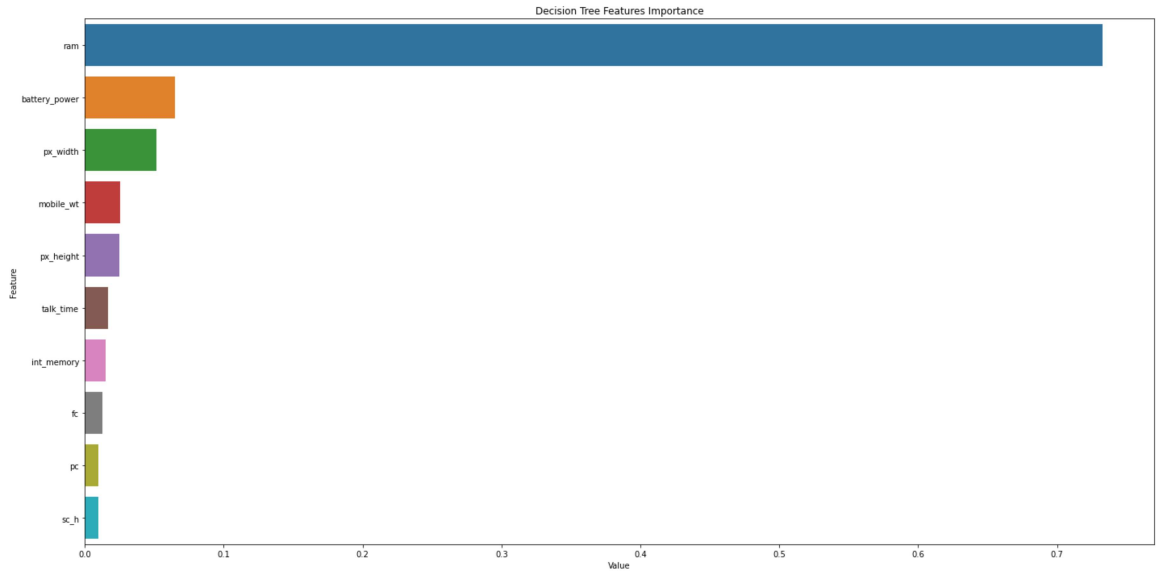


By visualizing the decision tree model, we could know that how decision tree split the sample to each class step by step. In our model, the most important features are battery power and px width, then is ram, int memory and other hardwares feature, which is consistent with the clustering results.

Importance Plot

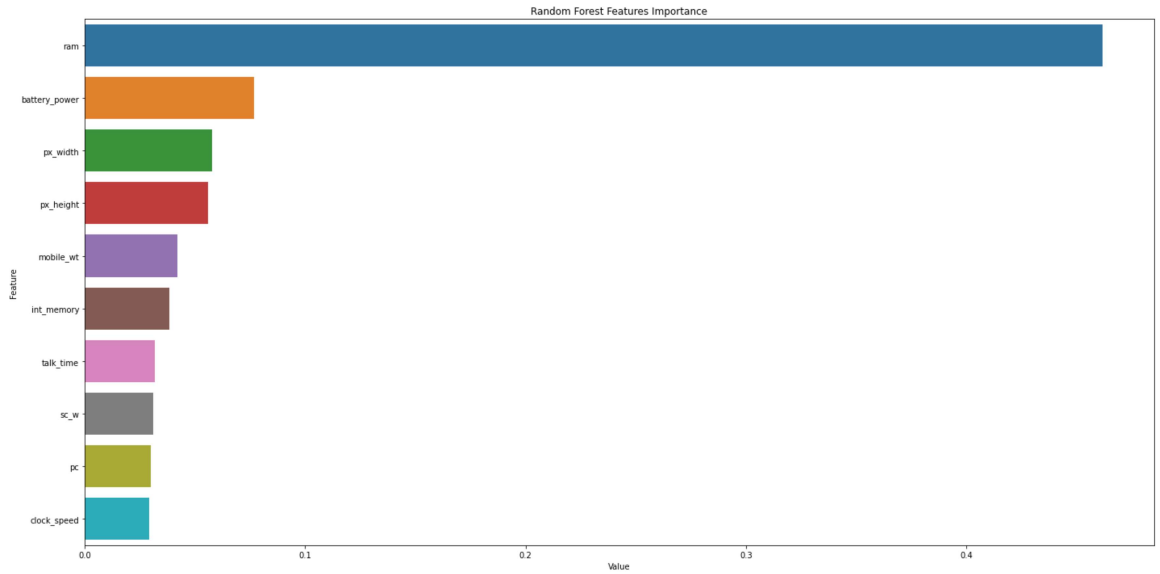
In [65]:

```
feature_imp = pd.DataFrame(sorted(zip(clf_tree.feature_importances_,X_train.columns)),
columns=['Value','Feature']).sort_values(by="Value", ascending=False).iloc[:10,:]

plt.figure(figsize=(20, 10))
sns.barplot(x="Value", y="Feature", data=feature_imp.sort_values(by="Value", ascending=
False))
plt.title('Decision Tree Features Importance')
plt.tight_layout()
plt.show()
```



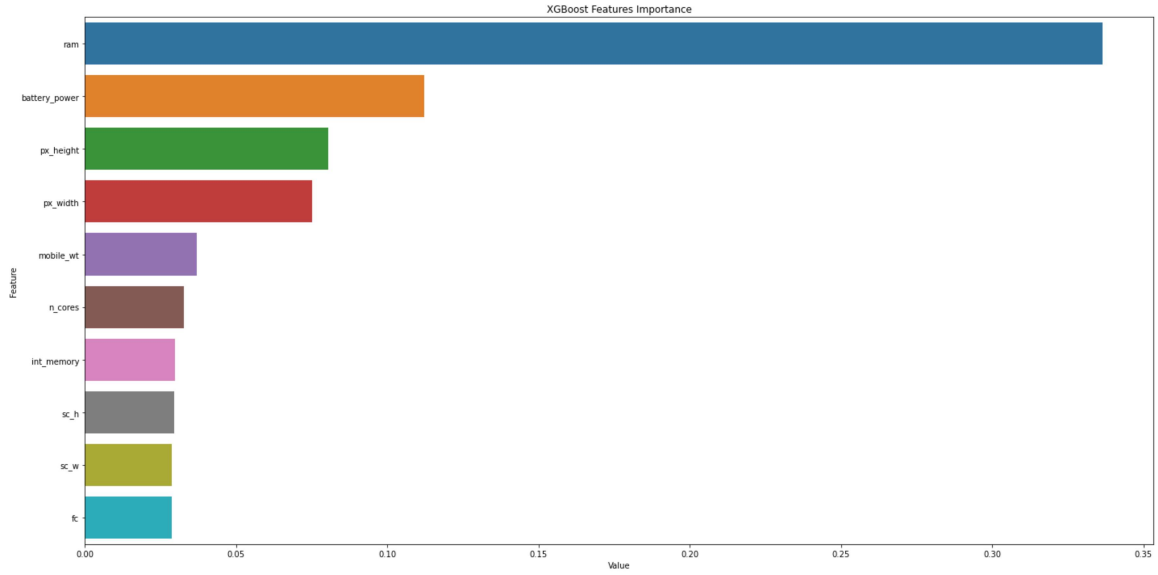
In [68]:

```
feature_imp = pd.DataFrame(sorted(zip(clf_rf.feature_importances_,X_train.columns)), co
lumns=['Value','Feature']).sort_values(by="Value", ascending=False).iloc[:10,:]

plt.figure(figsize=(20, 10))
sns.barplot(x="Value", y="Feature", data=feature_imp.sort_values(by="Value", ascending=
False))
plt.title('Random Forest Features Importance')
plt.tight_layout()
plt.show()
```



In [69]:

```
feature_imp = pd.DataFrame(sorted(zip(xgb.feature_importances_,X_train.columns)), colum
ns=['Value','Feature']).sort_values(by="Value", ascending=False).iloc[:10,:]

plt.figure(figsize=(20, 10))
sns.barplot(x="Value", y="Feature", data=feature_imp.sort_values(by="Value", ascending=
False))
plt.title('XGBoost Features Importance')
plt.tight_layout()
plt.show()
```



The importance plot could tell us the which indicators have the largest influence on the price range. According to the importance plot of random forest, decision tree and xgboost, we can know that ram is the dominat factor of moblie phone price. Then is the battery power.