Decision Tree Q2 (Fraud check)

Use decision trees to prepare a model on fraud data treating those who have taxable_income <= 30000 as "Risky" and others are "Good"

Data Description :

Undergrad : person is under graduated or not
Marital.Status : marital status of a person
Taxable.Income : Taxable income is the amount of how much tax an individual owes to the government
Work Experience : Work experience of an individual person
Urban : Whether that person belongs to urban area or not

1. Import Libs

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from io import StringIO
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
from sklearn import metrics
from sklearn import externals
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

2. Import Data

In [2]:

```
fraud_check = pd.read_csv('Fraud_check.csv')
fraud_check
```

Out[2]:

	Undergrad	Marital.Status	Taxable.Income	City.Population	Work.Experience	Urban
0	NO	Single	68833	50047	10	YES
1	YES	Divorced	33700	134075	18	YES
2	NO	Married	36925	160205	30	YES
3	YES	Single	50190	193264	15	YES
4	NO	Married	81002	27533	28	NO
...	...	...	...	...	...	...
595	YES	Divorced	76340	39492	7	YES
596	YES	Divorced	69967	55369	2	YES
597	NO	Divorced	47334	154058	0	YES
598	YES	Married	98592	180083	17	NO
599	NO	Divorced	96519	158137	16	NO

600 rows × 6 columns

3. EDA

In [3]:

```
fraud_check.describe()
```

Out[3]:

	Taxable.Income	City.Population	Work.Experience
count	600.000000	600.000000	600.000000
mean	55208.375000	108747.368333	15.558333
std	26204.827597	49850.075134	8.842147
min	10003.000000	25779.000000	0.000000
25%	32871.500000	66966.750000	8.000000
50%	55074.500000	106493.500000	15.000000
75%	78611.750000	150114.250000	24.000000
max	99619.000000	199778.000000	30.000000

In [4]:

```
fraud_check.isna().sum()
```

Out[4]:

```
Undergrad          0
Marital.Status     0
Taxable.Income     0
City.Population    0
Work.Experience    0
Urban              0
dtype: int64
```

In [5]:

```
fraud_check.dtypes
```

Out[5]:

```
Undergrad          object
Marital.Status     object
Taxable.Income      int64
City.Population     int64
Work.Experience     int64
Urban              object
dtype: object
```

checking outliers

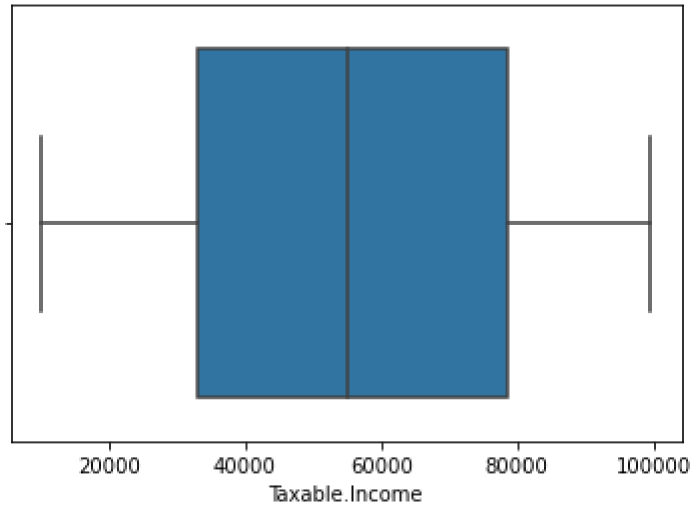
In [6]:

```
sns.boxplot(fraud_check['Taxable.Income'])
```

```
C:\Users\shubham\anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futu
reWarning: Pass the following variable as a keyword arg: x. From version 0.1
2, the only valid positional argument will be `data`, and passing other argu
ments without an explicit keyword will result in an error or misinterpretati
on.
  warnings.warn(
```

Out[6]:

```
<AxesSubplot:xlabel='Taxable.Income'>
```

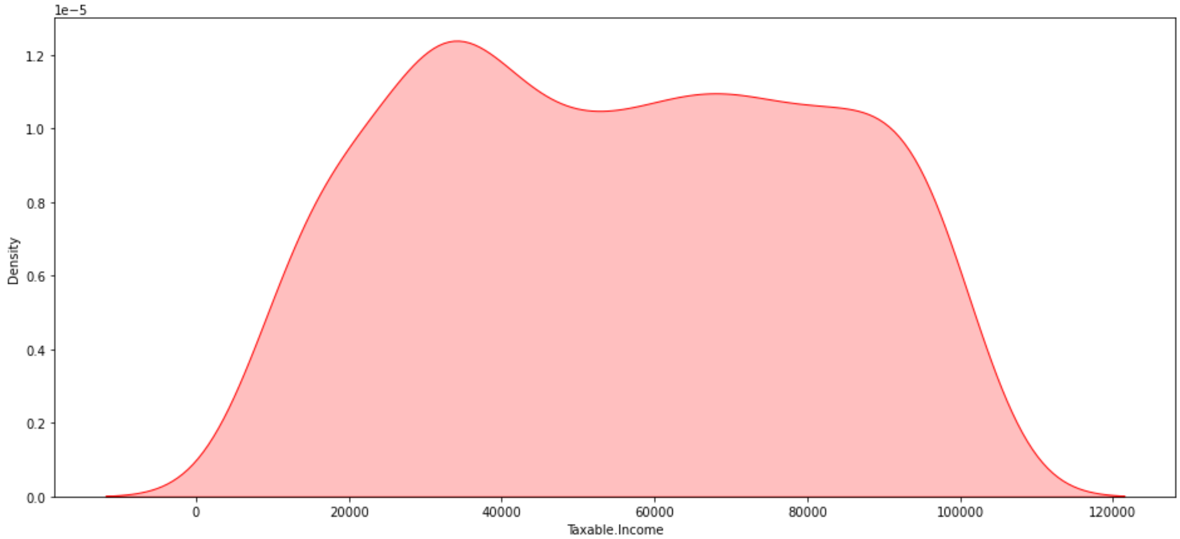


There are no outliers in the data

In [7]:

```
plt.figure(figsize=(16,7))
print("Skewness =",fraud_check['Taxable.Income'].skew())
print("Kurtosis =",fraud_check['Taxable.Income'].kurtosis())
sns.kdeplot(fraud_check['Taxable.Income'],shade=True,color='r')
plt.show()
```

```
Skewness = 0.030014788906377175
Kurtosis = -1.1997824607083138
```



Sales Data is skewed to the right and Data has negative kurtosis

In [8]:

```
obj_colum = fraud_check.select_dtypes(include='object')
obj_colum
```

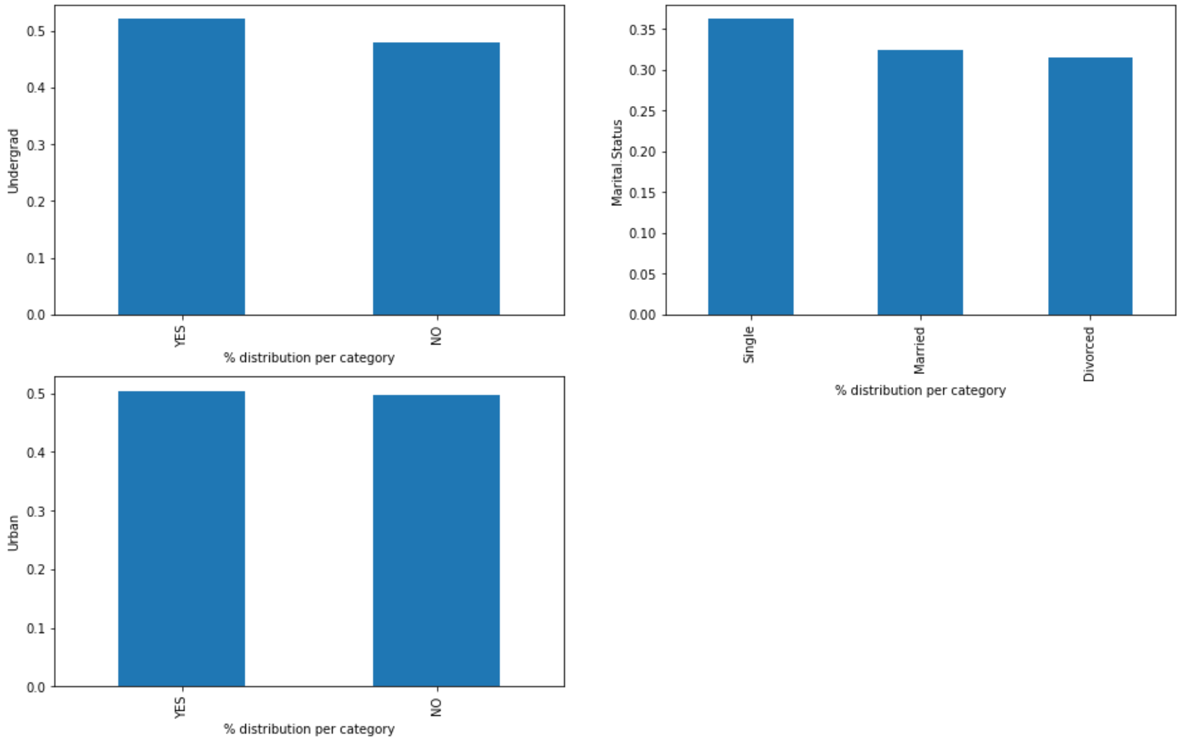
Out[8]:

	Undergrad	Marital.Status	Urban
0	NO	Single	YES
1	YES	Divorced	YES
2	NO	Married	YES
3	YES	Single	YES
4	NO	Married	NO
...	...	...	...
595	YES	Divorced	YES
596	YES	Divorced	YES
597	NO	Divorced	YES
598	YES	Married	NO
599	NO	Divorced	NO

600 rows × 3 columns

In [9]:

```
plt.figure(figsize=(16,10))
for i,col in enumerate(obj_colum,1):
    plt.subplot(2,2,i)
    fraud_check[col].value_counts(normalize=True).plot.bar()
    plt.ylabel(col)
    plt.xlabel('% distribution per category')
```



In [10]:

```
num_columns = fraud_check.select_dtypes(include=['float64','int64'])
num_columns
```

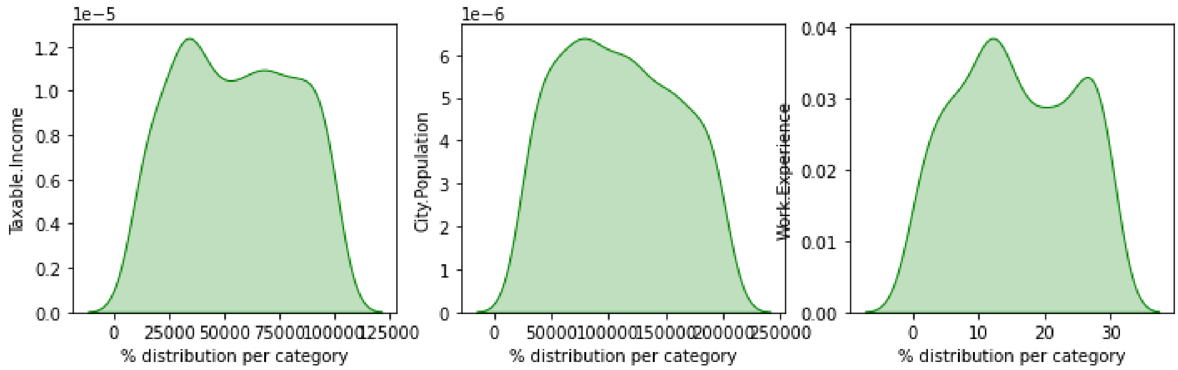
Out[10]:

	Taxable.Income	City.Population	Work.Experience
0	68833	50047	10
1	33700	134075	18
2	36925	160205	30
3	50190	193264	15
4	81002	27533	28
...	...	...	...
595	76340	39492	7
596	69967	55369	2
597	47334	154058	0
598	98592	180083	17
599	96519	158137	16

600 rows × 3 columns

In [11]:

```
plt.figure(figsize=(16,30))
for i,col in enumerate(num_columns,1):
    plt.subplot(8,4,i)
    sns.kdeplot(fraud_check[col],color='g',shade=True)
    plt.ylabel(col)
    plt.xlabel('% distribution per category')
```



In [12]:

```
pd.DataFrame(data=[num_columns.skew(),num_columns.kurtosis()],index=['skewness','kurtosis']
```

Out[12]:

	Taxable.Income	City.Population	Work.Experience
skewness	0.030015	0.125009	0.018529
kurtosis	-1.199782	-1.120154	-1.167524

In [13]:

```
df = pd.get_dummies(fraud_check, columns = ['Undergrad','Marital.Status','Urban'])
```

In [14]:

```
df
```

Out[14]:

	Taxable.Income	City.Population	Work.Experience	Undergrad_NO	Undergrad_YES	Marital.S
0	68833	50047	10	1	0	
1	33700	134075	18	0	1	
2	36925	160205	30	1	0	
3	50190	193264	15	0	1	
4	81002	27533	28	1	0	
...	...	...	...	...	...	
595	76340	39492	7	0	1	
596	69967	55369	2	0	1	
597	47334	154058	0	1	0	
598	98592	180083	17	0	1	
599	96519	158137	16	1	0	

600 rows × 10 columns

In [15]:

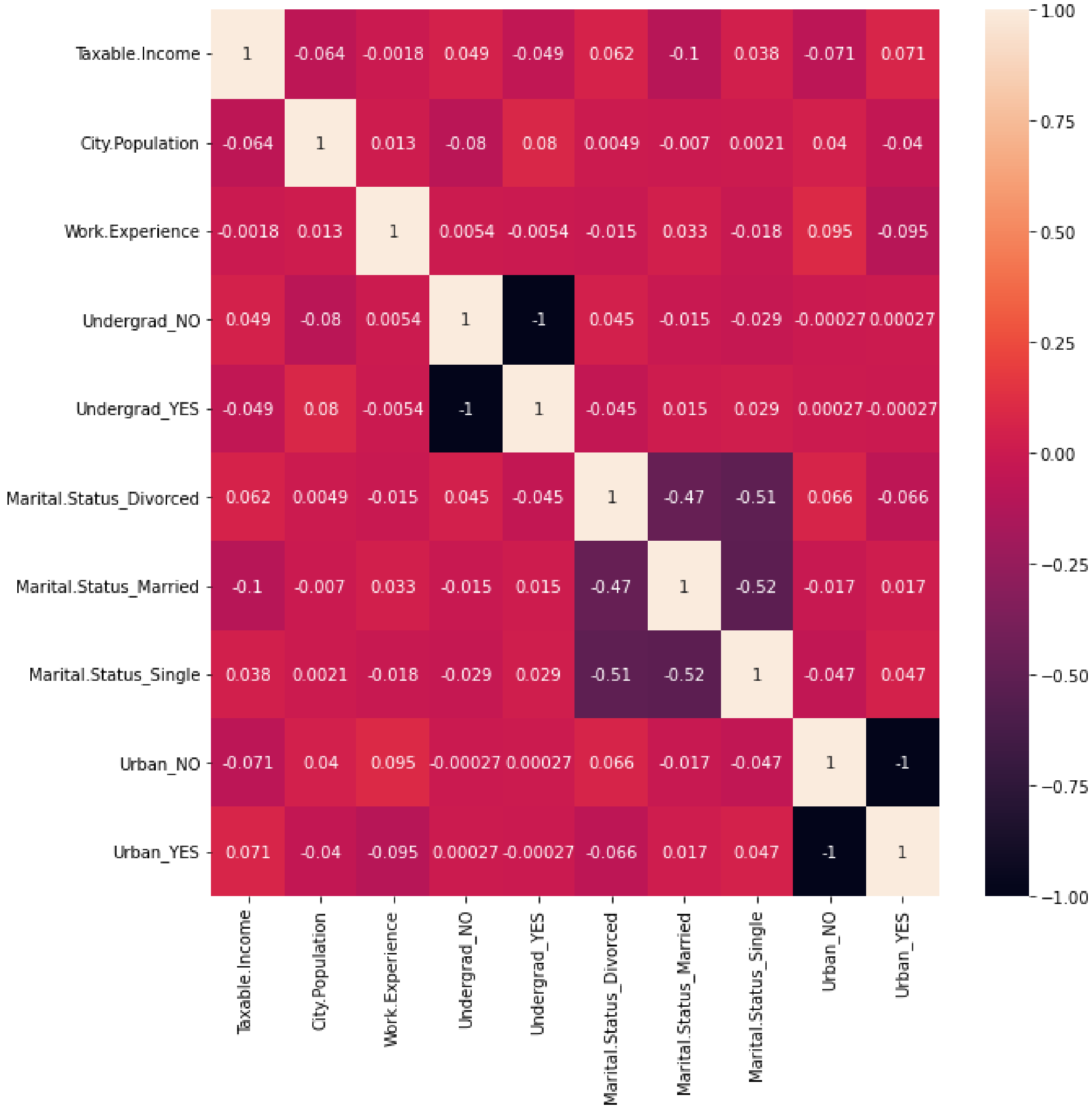
```
corr = df.corr()
```

In [16]:

```
plt.figure(figsize=(10,10))
sns.heatmap(corr,annot=True)
```

Out[16]:

<AxesSubplot:>



4. Model Building

Since the target variable is continious, we create a class of taxable_income <= 30000 as "Risky" and others are "Good"

In [17]:

```
df["Taxable.Income"]
```

Out[17]:

```
0      68833
1      33700
2      36925
3      50190
4      81002
       ...
595    76340
596    69967
597    47334
598    98592
599    96519
Name: Taxable.Income, Length: 600, dtype: int64
```

for <= 30000 = "Risky" and > 30000 = "Good"

Use cut when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable.

In [18]:

```
df['Taxable.Income'] = pd.cut(df["Taxable.Income"],bins=[0,30000,100000],labels=['Riskey','
```

Droping the Sales column

In [19]:

```
df.head(20)
```

Out[19]:

	Taxable.Income	City.Population	Work.Experience	Undergrad_NO	Undergrad_YES	Marital.St
0	Good	50047	10	1	0	
1	Good	134075	18	0	1	
2	Good	160205	30	1	0	
3	Good	193264	15	0	1	
4	Good	27533	28	1	0	
5	Good	116382	0	1	0	
6	Good	80890	8	1	0	
7	Good	131253	3	0	1	
8	Good	102481	12	1	0	
9	Good	155482	4	0	1	
10	Riskey	102602	19	1	0	
11	Good	94875	6	1	0	
12	Riskey	148033	14	1	0	
13	Good	86649	16	1	0	
14	Good	57529	13	1	0	
15	Good	107764	29	1	0	
16	Riskey	34551	29	0	1	
17	Good	57194	25	0	1	
18	Good	59269	6	0	1	
19	Riskey	126953	30	1	0	

In [20]:

```
x = df.iloc[:,1:10]
x
```

Out[20]:

	City.Population	Work.Experience	Undergrad_NO	Undergrad_YES	Marital.Status_Divorced
0	50047	10	1	0	0
1	134075	18	0	1	1
2	160205	30	1	0	0
3	193264	15	0	1	0
4	27533	28	1	0	0
...	...	...	...	...	...
595	39492	7	0	1	1
596	55369	2	0	1	1
597	154058	0	1	0	1
598	180083	17	0	1	0
599	158137	16	1	0	1

600 rows × 9 columns

In [21]:

```
y =df['Taxable.Income']

df['Taxable.Income'].value_counts()
```

Out[21]:

```
Good      476
Riskey    124
Name: Taxable.Income, dtype: int64
```

5. Model Traning

Decision Tree - Model using Entropy Criteria and Gini Criteria

Splitting data into training and testing data set

In [22]:

```
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.2, stratify = y)
```

In [23]:

```
y_train.value_counts()
```

Out[23]:

```
Good      381
Riskey     99
Name: Taxable.Income, dtype: int64
```

Building Decision Tree Classifier using Entropy Criteria

In [24]:

```
model = DecisionTreeClassifier(criterion = 'entropy',max_depth=5)
model.fit(x_train,y_train)
```

Out[24]:

```
DecisionTreeClassifier(criterion='entropy', max_depth=5)
```

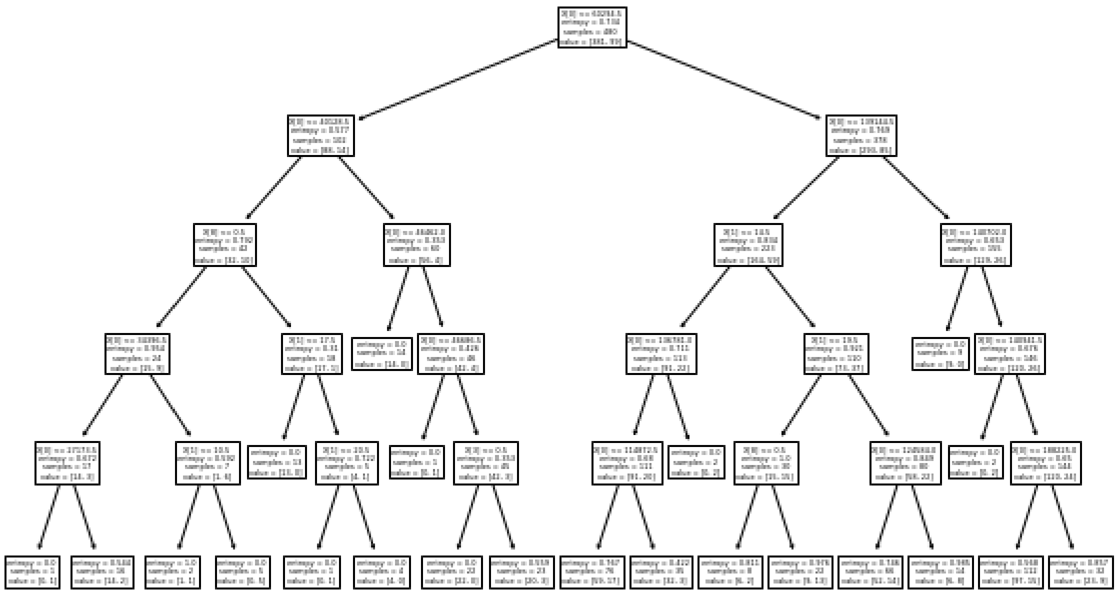
In [25]:

```
#PLot the decision tree
plt.figure(figsize=(10,6))
tree.plot_tree(model)
```

Out[25]:

```
[Text(0.53125, 0.9166666666666666, 'X[0] <= 60294.5\nentropy = 0.734\nsample
s = 480\nvalue = [381, 99]'),
 Text(0.2890625, 0.75, 'X[0] <= 40128.5\nentropy = 0.577\nsamples = 102\nval
ue = [88, 14]'),
 Text(0.203125, 0.5833333333333334, 'X[8] <= 0.5\nentropy = 0.792\nsamples =
42\nvalue = [32, 10]'),
 Text(0.125, 0.4166666666666667, 'X[0] <= 34396.5\nentropy = 0.954\nsamples
= 24\nvalue = [15, 9]'),
 Text(0.0625, 0.25, 'X[0] <= 27173.5\nentropy = 0.672\nsamples = 17\nvalue =
[14, 3]'),
 Text(0.03125, 0.08333333333333333, 'entropy = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.09375, 0.08333333333333333, 'entropy = 0.544\nsamples = 16\nvalue =
[14, 2]'),
 Text(0.1875, 0.25, 'X[1] <= 10.5\nentropy = 0.592\nsamples = 7\nvalue = [1,
6]'),
 Text(0.15625, 0.08333333333333333, 'entropy = 1.0\nsamples = 2\nvalue = [1,
1]'),
 Text(0.21875, 0.08333333333333333, 'entropy = 0.0\nsamples = 5\nvalue = [0,
5]'),
 Text(0.28125, 0.4166666666666667, 'X[1] <= 17.5\nentropy = 0.31\nsamples =
18\nvalue = [17, 1]'),
 Text(0.25, 0.25, 'entropy = 0.0\nsamples = 13\nvalue = [13, 0]'),
 Text(0.3125, 0.25, 'X[1] <= 20.5\nentropy = 0.722\nsamples = 5\nvalue = [4,
1]'),
 Text(0.28125, 0.08333333333333333, 'entropy = 0.0\nsamples = 1\nvalue = [0,
1]'),
 Text(0.34375, 0.08333333333333333, 'entropy = 0.0\nsamples = 4\nvalue = [4,
0]'),
 Text(0.375, 0.5833333333333334, 'X[0] <= 46462.0\nentropy = 0.353\nsamples
= 60\nvalue = [56, 4]'),
 Text(0.34375, 0.4166666666666667, 'entropy = 0.0\nsamples = 14\nvalue = [1
4, 0]'),
 Text(0.40625, 0.4166666666666667, 'X[0] <= 46686.5\nentropy = 0.426\nsample
s = 46\nvalue = [42, 4]'),
 Text(0.375, 0.25, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.4375, 0.25, 'X[3] <= 0.5\nentropy = 0.353\nsamples = 45\nvalue = [4
2, 3]'),
 Text(0.40625, 0.08333333333333333, 'entropy = 0.0\nsamples = 22\nvalue = [2
2, 0]'),
 Text(0.46875, 0.08333333333333333, 'entropy = 0.559\nsamples = 23\nvalue =
[20, 3]'),
 Text(0.7734375, 0.75, 'X[0] <= 139144.5\nentropy = 0.769\nsamples = 378\nva
lue = [293, 85]'),
 Text(0.671875, 0.5833333333333334, 'X[1] <= 14.5\nentropy = 0.834\nsamples
= 223\nvalue = [164, 59]'),
 Text(0.59375, 0.4166666666666667, 'X[0] <= 136781.0\nentropy = 0.711\nsampl
es = 113\nvalue = [91, 22]'),
 Text(0.5625, 0.25, 'X[0] <= 114872.5\nentropy = 0.68\nsamples = 111\nvalue
= [91, 20]'),
 Text(0.53125, 0.08333333333333333, 'entropy = 0.767\nsamples = 76\nvalue =
[59, 17]'),
 Text(0.59375, 0.08333333333333333, 'entropy = 0.422\nsamples = 35\nvalue =
```

```
[32, 3]'),
 Text(0.625, 0.25, 'entropy = 0.0\nsamples = 2\nvalue = [0, 2]'),
 Text(0.75, 0.4166666666666667, 'X[1] <= 19.5\nentropy = 0.921\nsamples = 11
0\nvalue = [73, 37]'),
 Text(0.6875, 0.25, 'X[8] <= 0.5\nentropy = 1.0\nsamples = 30\nvalue = [15,
15]'),
 Text(0.65625, 0.08333333333333333, 'entropy = 0.811\nsamples = 8\nvalue =
[6, 2]'),
 Text(0.71875, 0.08333333333333333, 'entropy = 0.976\nsamples = 22\nvalue =
[9, 13]'),
 Text(0.8125, 0.25, 'X[0] <= 124584.0\nentropy = 0.849\nsamples = 80\nvalue
= [58, 22]'),
 Text(0.78125, 0.08333333333333333, 'entropy = 0.746\nsamples = 66\nvalue =
[52, 14]'),
 Text(0.84375, 0.08333333333333333, 'entropy = 0.985\nsamples = 14\nvalue =
[6, 8]'),
 Text(0.875, 0.5833333333333334, 'X[0] <= 140702.0\nentropy = 0.653\nsamples
= 155\nvalue = [129, 26]'),
 Text(0.84375, 0.4166666666666667, 'entropy = 0.0\nsamples = 9\nvalue = [9,
0]'),
 Text(0.90625, 0.4166666666666667, 'X[0] <= 140941.5\nentropy = 0.676\nsampl
es = 146\nvalue = [120, 26]'),
 Text(0.875, 0.25, 'entropy = 0.0\nsamples = 2\nvalue = [0, 2]'),
 Text(0.9375, 0.25, 'X[0] <= 188225.0\nentropy = 0.65\nsamples = 144\nvalue
= [120, 24]'),
 Text(0.90625, 0.08333333333333333, 'entropy = 0.568\nsamples = 112\nvalue =
[97, 15]'),
 Text(0.96875, 0.08333333333333333, 'entropy = 0.857\nsamples = 32\nvalue =
[23, 9]')]
```



In [26]:

```
pred_train = model.predict(x_train)
```

accuracy check

In [27]:

```
accuracy_score(y_train,pred_train)
```

Out[27]:

```
0.83125
```

In [28]:

```
confusion_matrix(y_train,pred_train)
```

Out[28]:

```
array([[366,  15],
       [ 66,  33]], dtype=int64)
```

In [29]:

```
pred_test = model.predict(x_test)
```

accuracy check

In [30]:

```
accuracy_score(y_test,pred_test)
```

Out[30]:

```
0.7333333333333333
```

In [31]:

```
confusion_matrix(y_test,pred_test)
```

Out[31]:

```
array([[84, 11],
       [21,  4]], dtype=int64)
```

In [32]:

```
df_Entropy=pd.DataFrame({'Actual':y_test, 'Predicted':pred_test})
```

In [33]:

```
df_Entropy
```

Out[33]:

	Actual	Predicted
133	Good	Good
74	Good	Good
268	Good	Good
72	Good	Good
377	Good	Good
...	...	...
415	Good	Riskey
113	Good	Good
319	Good	Good
510	Good	Good
170	Good	Good

120 rows × 2 columns

In [34]:

```
model.feature_importances_
```

Out[34]:

```
array([0.6583675 , 0.20794652, 0.        , 0.04134832, 0.        ,
       0.        , 0.        , 0.        , 0.09233766])
```

In [35]:

```
feature_importance = pd.DataFrame({'feature': list(x_train.columns),
                   'importance': model.feature_importances_}).\
                    sort_values('importance', ascending = False)
```

In [36]:

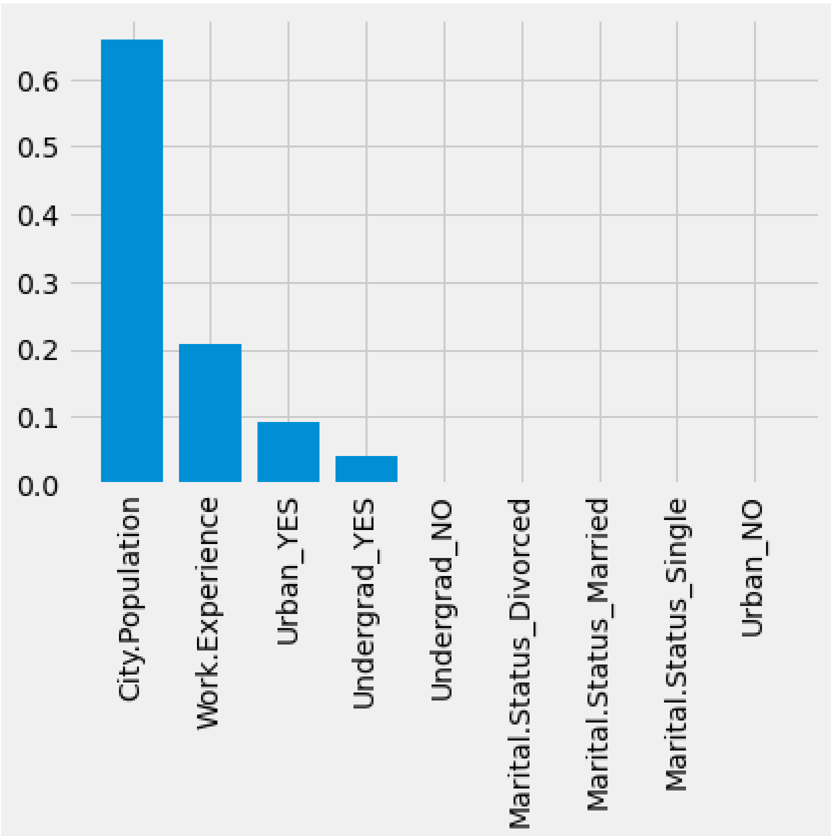
```
feature_importance
```

Out[36]:

	feature	importance
0	City.Population	0.658368
1	Work.Experience	0.207947
8	Urban_YES	0.092338
3	Undergrad_YES	0.041348
2	Undergrad_NO	0.000000
4	Marital.Status_Divorced	0.000000
5	Marital.Status_Married	0.000000
6	Marital.Status_Single	0.000000
7	Urban_NO	0.000000

In [37]:

```
plt.style.use('fivethirtyeight')
plt.bar(feature_importance['feature'],feature_importance['importance'], orientation = 'vert
plt.xticks(rotation = 90)
plt.show()
```



As seen in the above chart, City.Population is most important feature

END