Diamond Price Prediction Model

March 9, 2022

[1]:
```
import sklearn

import os
import pandas as pd
import numpy as np

import shap
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
import warnings
warnings.filterwarnings("ignore")

%matplotlib inline
```

[ ]:
```

```

[2]:
```
#Load the Data
diamonds = pd.read_csv("C:/Users/Owner/Downloads/wholesale_diamonds.csv")
```

[3]:
```
#a short preview of our dataset setructure
diamonds.head()
```

[3]:
```
   index  carat        cut color clarity  depth  table  cost (dollars)  \
0      0   0.23      Ideal     E     SI2   61.5   55.0             326
1      1   0.23       Good     E     VS1   56.9   65.0             327
2      2   0.29    Premium     I     VS2   62.4   58.0             334
3      3   0.31       Good     J     SI2   63.3   58.0             335
4      4   0.24  Very Good     J    VVS2   62.8   57.0             336

   length (mm)  width (mm)  height (mm)  year
0         3.95        3.98         2.43  2010
1         4.05        4.07         2.31  2010
2         4.20        4.23         2.63  2010
3         4.34        4.35         2.75  2010
4         3.94        3.96         2.48  2010
```

[4]:
```
#a little more preview of the dataset
diamonds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 407280 entries, 0 to 407279
Data columns (total 12 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   index           407280 non-null  int64
 1   carat           405232 non-null  float64
 2   cut             407280 non-null  object
 3   color           407280 non-null  object
 4   clarity         407280 non-null  object
 5   depth           407280 non-null  float64
 6   table           407280 non-null  float64
 7   cost (dollars)  407280 non-null  int64
 8   length (mm)     407280 non-null  float64
 9   width (mm)      407280 non-null  float64
 10  height (mm)     407280 non-null  float64
 11  year            407280 non-null  int64
dtypes: float64(6), int64(3), object(3)
memory usage: 37.3+ MB
```

Noticed that the index column is unnecessary and need to to drop

[5]:
```
#drop the index column
diamonds = diamonds.drop(columns={"index"})
```

[6]:
```
diamonds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 407280 entries, 0 to 407279
Data columns (total 11 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   carat           405232 non-null  float64
 1   cut             407280 non-null  object
 2   color           407280 non-null  object
 3   clarity         407280 non-null  object
 4   depth           407280 non-null  float64
 5   table           407280 non-null  float64
 6   cost (dollars)  407280 non-null  int64
 7   length (mm)     407280 non-null  float64
 8   width (mm)      407280 non-null  float64
 9   height (mm)     407280 non-null  float64
 10  year            407280 non-null  int64
dtypes: float64(6), int64(2), object(3)
```

```
memory usage: 34.2+ MB
```

[7]:
```python
#Renaming the columns name
#columns:Cost,Length,Width and height and summarize statistics
diamonds.rename(columns ={'cost (dollars)':'price', 'length (mm)':'x', 'width␣
↪(mm)':'y', 'height (mm)':'z'}, inplace = True)
diamonds.describe()
```

[7]:
```
               carat          depth          table          price  \
count  405232.000000  407280.000000  407280.000000  407280.000000
mean        0.797742      61.747793      57.457113    4372.968506
std         0.474774       1.434209       2.239837    4503.620949
min         0.200000      43.000000      43.000000    -998.000000
25%         0.400000      61.000000      56.000000    1043.000000
50%         0.700000      61.800000      57.000000    2655.000000
75%         1.040000      62.500000      59.000000    5960.000000
max         4.130000      79.000000      95.000000   26930.000000

                   x              y              z           year
count  407280.000000  407280.000000  407280.000000  407280.000000
mean        5.730165       5.732369       3.538519    2015.500000
std         1.122960       1.114266       0.712168       3.452057
min         0.000000       0.000000       0.000000    2010.000000
25%         4.710000       4.720000       2.910000    2012.750000
50%         5.690000       5.710000       3.520000    2015.500000
75%         6.530000       6.530000       4.030000    2018.250000
max        10.140000      10.100000      31.800000    2021.000000
```

[8]:
```python
# Price is int64, best if all numeric attributes have the same datatype,␣
↪especially as float64
diamonds["price"] = diamonds["price"].astype(float)
```

[9]:
```python
#preview the data set again
diamonds.head()
```

[9]:
```
   carat        cut color clarity  depth  table  price     x     y     z  year
0   0.23      Ideal     E     SI2   61.5   55.0  326.0  3.95  3.98  2.43  2010
1   0.23       Good     E     VS1   56.9   65.0  327.0  4.05  4.07  2.31  2010
2   0.29    Premium     I     VS2   62.4   58.0  334.0  4.20  4.23  2.63  2010
3   0.31       Good     J     SI2   63.3   58.0  335.0  4.34  4.35  2.75  2010
4   0.24  Very Good     J    VVS2   62.8   57.0  336.0  3.94  3.96  2.48  2010
```

0.0.1 Cleaning the Data

[10]:
```python
#check if we have a null value
diamonds.isnull().sum()
```

[10]:
```
carat       2048
cut            0
color          0
clarity        0
depth          0
table          0
price          0
x              0
y              0
z              0
year           0
dtype: int64
```

Noticed there are 2048 null values in Carat. They need to be dropped

[11]:
```
#Drop the null values
diamonds = diamonds.dropna(axis = 0, how='any')
```

[12]:
```
#check if the null values are removed or not
diamonds.isnull().sum()
```

[12]:
```
carat       0
cut         0
color       0
clarity     0
depth       0
table       0
price       0
x           0
y           0
z           0
year        0
dtype: int64
```

[13]:
```
diamonds.isnull().values.any()
```

[13]:
```
False
```

WE Noticed there are negative values in price. They need to dropped to 0

[14]:
```
#drop negaticve values to 0
diamonds.drop(diamonds[diamonds['price'] <= 0].index, inplace = True)
diamonds.describe()
```

[14]:
```
               carat          depth          table          price  \
count  403192.000000  403192.000000  403192.000000  403192.000000
mean        0.797700      61.747838      57.457640    4397.571519
std         0.474791       1.434470       2.240196    4501.692426
min         0.200000      43.000000      43.000000     304.000000
```

```
25%          0.400000      61.000000      56.000000     1053.000000
50%          0.700000      61.800000      57.000000     2677.500000
75%          1.040000      62.500000      59.000000     5983.000000
max          4.130000      79.000000      95.000000    26930.000000

                    x              y              z            year
count  403192.000000  403192.000000  403192.00000  403192.000000
mean        5.730006       5.732210       3.53842    2015.500243
std         1.123061       1.114364       0.71237       3.452026
min         0.000000       0.000000       0.00000    2010.000000
25%         4.710000       4.720000       2.91000    2013.000000
50%         5.690000       5.710000       3.52000    2016.000000
75%         6.530000       6.530000       4.03000    2019.000000
max        10.140000      10.100000      31.80000    2021.000000
```

[15]:
```
diamonds.shape
```

[15]:
```
(403192, 11)
```

[16]:
```
diamonds = diamonds.drop(columns={"year"})
```

[17]:
```
diamonds.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 403192 entries, 0 to 407279
Data columns (total 10 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   carat    403192 non-null  float64
 1   cut      403192 non-null  object
 2   color    403192 non-null  object
 3   clarity  403192 non-null  object
 4   depth    403192 non-null  float64
 5   table    403192 non-null  float64
 6   price    403192 non-null  float64
 7   x        403192 non-null  float64
 8   y        403192 non-null  float64
 9   z        403192 non-null  float64
dtypes: float64(7), object(3)
memory usage: 33.8+ MB
```

[18]:
```
diamonds.head()
```

[18]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326.0	3.95	3.98	2.43
1	0.23	Good	E	VS1	56.9	65.0	327.0	4.05	4.07	2.31
2	0.29	Premium	I	VS2	62.4	58.0	334.0	4.20	4.23	2.63
3	0.31	Good	J	SI2	63.3	58.0	335.0	4.34	4.35	2.75

```
4   0.24   Very Good     J     VVS2    62.8    57.0   336.0   3.94   3.96   2.48
```

[19]:
```
diamonds.shape
```

[19]:
```
(403192, 10)
```

0.0.2 Exploring and Visualization the data

It's easier to work a dataset when all its attributes are numerical. The cut, color and clarity attributes are non-numeric (They are objects). We still have to convert them to be numerical. Let's find out what categories exist for each of them.

[20]:
```
# The diamond cut categories
diamonds["cut"].value_counts()
```

[20]:
```
Ideal        161196
Premium      102668
Very Good     90813
Good          36701
Fair          11814
Name: cut, dtype: int64
```

[21]:
```
#The diamond color categories
diamonds["color"].value_counts()
```

[21]:
```
G    85245
E    72607
F    70154
H    63045
D    50803
I    40276
J    21062
Name: color, dtype: int64
```

[22]:
```
# The diamond clarity categories
diamonds["clarity"].value_counts()
```

[22]:
```
SI1     97900
VS2     91040
SI2     69326
VS1     60653
VVS2    37631
VVS1    28089
IF      13083
I1       5470
Name: clarity, dtype: int64
```

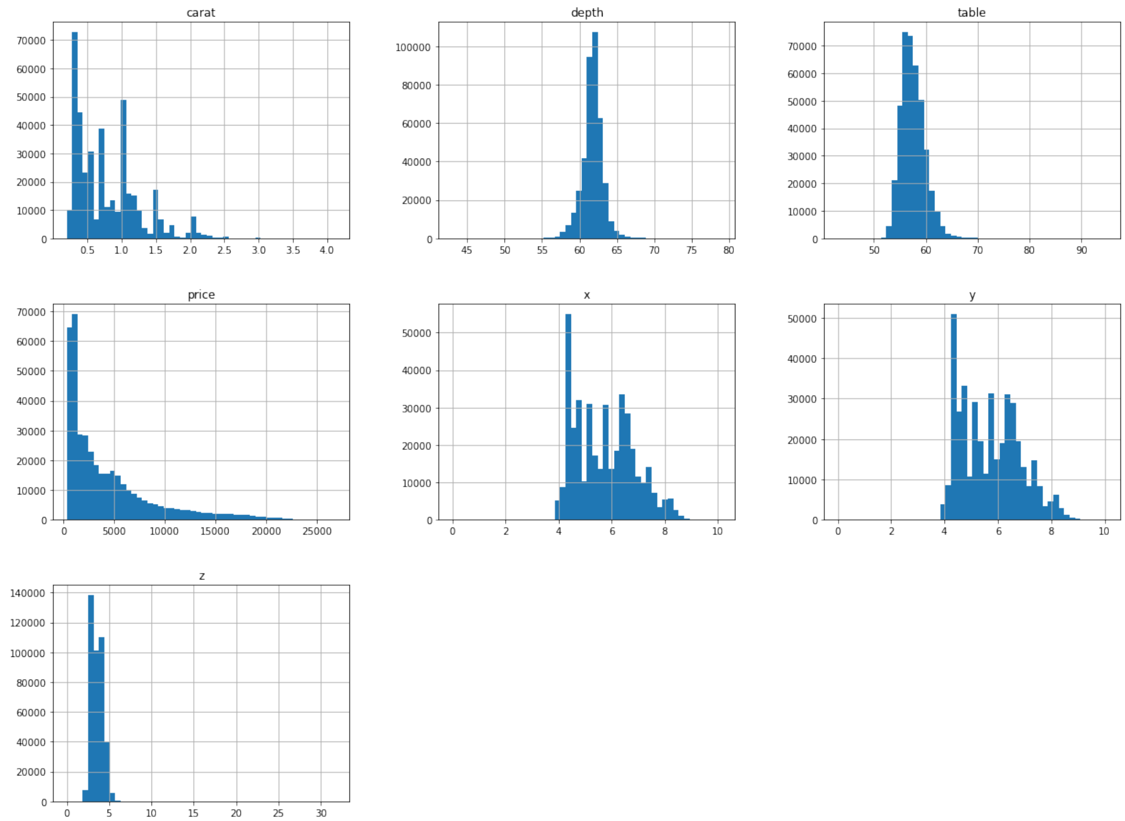
Let's take a preview of the summary of the numerical attributes and then an histogram on the dataset.

[23]:
```
# Summary of each numerical attribute
diamonds.describe()
```

[23]:
```
               carat          depth          table          price  \
count  403192.000000  403192.000000  403192.000000  403192.000000
mean        0.797700      61.747838      57.457640    4397.571519
std         0.474791       1.434470       2.240196    4501.692426
min         0.200000      43.000000      43.000000     304.000000
25%         0.400000      61.000000      56.000000    1053.000000
50%         0.700000      61.800000      57.000000    2677.500000
75%         1.040000      62.500000      59.000000    5983.000000
max         4.130000      79.000000      95.000000   26930.000000

                   x              y             z
count  403192.000000  403192.000000  403192.00000
mean        5.730006       5.732210       3.53842
std         1.123061       1.114364       0.71237
min         0.000000       0.000000       0.00000
25%         4.710000       4.720000       2.91000
50%         5.690000       5.710000       3.52000
75%         6.530000       6.530000       4.03000
max        10.140000      10.100000      31.80000
```

[24]:
```
diamonds.to_csv("C:/Users/Owner/Desktop/diamond_set/new_diamonds.csv")
```

[25]:
```
diamonds.hist(bins = 50, figsize = (20, 15))
plt.show()
```

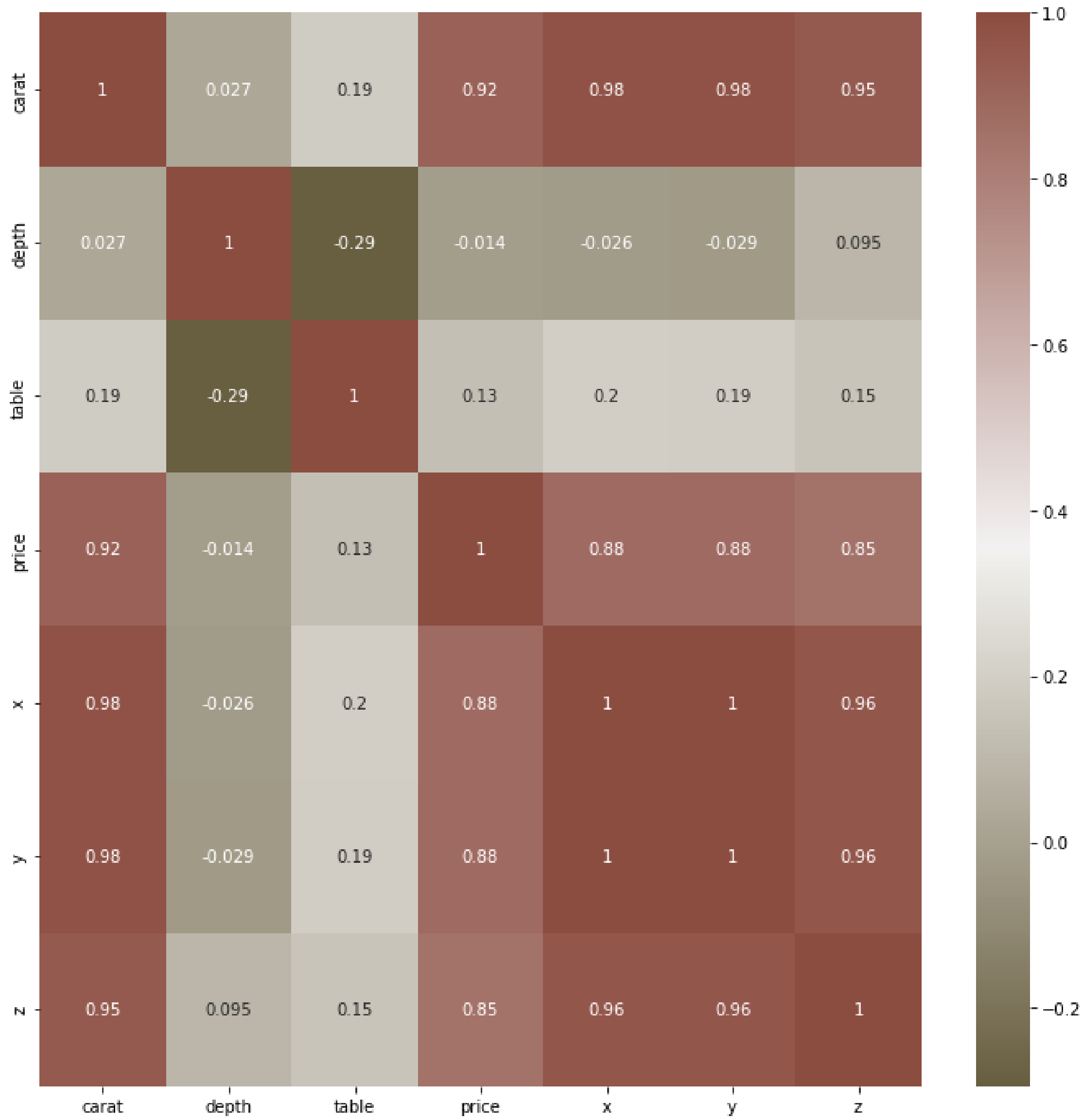


0.0.3 Understanding the correlation between variables

We have learned that to avoid a sampling bias, it is a good practice to perform stratified sampling. Stratified sampling is a technique that divides the dataset into homogeneous subgroups called strata. To use this, we need a right attribute of the dataset to predict the price. To get the right attribute, we need to pick an attribute most correlated to price. To select the correct attribute, we use the standard correlation coefficient

[26]:
```
# Create a correlation matrix between every pair of attributes
corr_matrix = diamonds.corr()

# Plot the correlation with seaborn
cmap = sns.diverging_palette(70,20,s=50, l=40, n=6,as_cmap=True)
corrmat= diamonds.corr()
f, ax = plt.subplots(figsize=(12,12))
sns.heatmap(corrmat,cmap=cmap,annot=True, )
```

[26]: <AxesSubplot:>



Carat has the strongest correlation with price with the number of 0.92 X,y,z also have strong correlations with price depth and table have the weakest correlation with price

[27]:
```
diamonds.corr()
```

[27]:

	carat	depth	table	price	x	y	z
carat	1.000000	0.027497	0.188253	0.915346	0.976104	0.975667	0.946206
depth	0.027497	1.000000	-0.294142	-0.013748	-0.025652	-0.028533	0.094620
table	0.188253	-0.294142	1.000000	0.134365	0.200761	0.194485	0.153990
price	0.915346	-0.013748	0.134365	1.000000	0.880170	0.882150	0.849546
x	0.976104	-0.025652	0.200761	0.880170	1.000000	0.998137	0.962398
y	0.975667	-0.028533	0.194485	0.882150	0.998137	1.000000	0.961687

```
z      0.946206  0.094620  0.153990  0.849546  0.962398  0.961687  1.000000
```

[28]:

```python
diamonds.describe()
```

[28]:

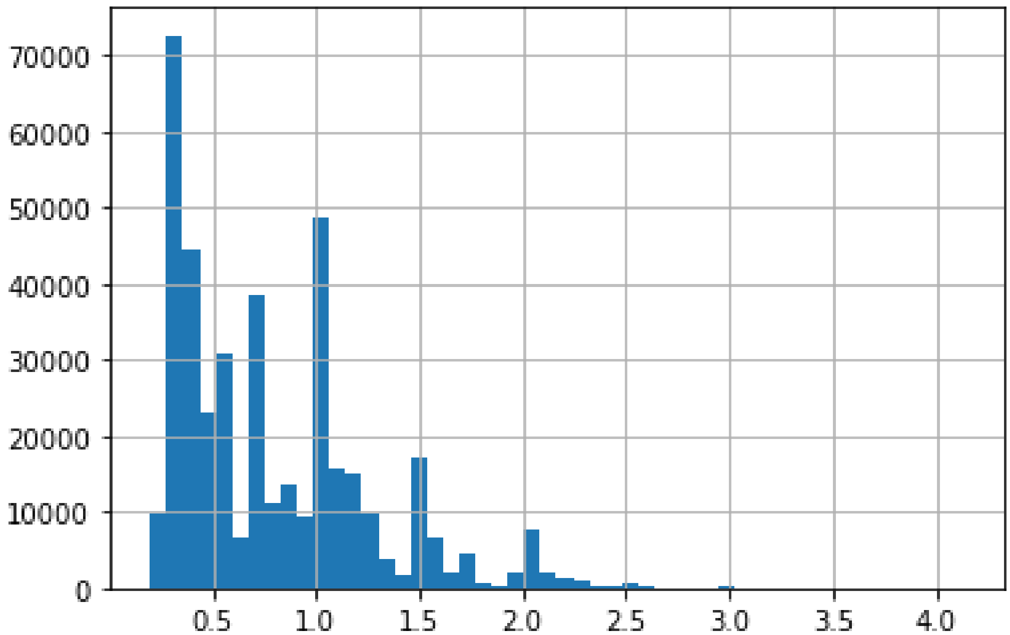
```
               carat          depth          table          price  \
count  403192.000000  403192.000000  403192.000000  403192.000000
mean        0.797700      61.747838      57.457640    4397.571519
std         0.474791       1.434470       2.240196    4501.692426
min         0.200000      43.000000      43.000000     304.000000
25%         0.400000      61.000000      56.000000    1053.000000
50%         0.700000      61.800000      57.000000    2677.500000
75%         1.040000      62.500000      59.000000    5983.000000
max         4.130000      79.000000      95.000000   26930.000000

                   x              y             z
count  403192.000000  403192.000000  403192.00000
mean        5.730006       5.732210       3.53842
std         1.123061       1.114364       0.71237
min         0.000000       0.000000       0.00000
25%         4.710000       4.720000       2.91000
50%         5.690000       5.710000       3.52000
75%         6.530000       6.530000       4.03000
max        10.140000      10.100000      31.80000
```

We see that carat correlates best with price. Its score is pretty high! Now we use this for our Stratified Sampling.

Let's take a closer look at the carat's histogram.

[29]:

```python
diamonds["carat"].hist(bins = 50)
plt.show()
```



[30]:
```
diamonds.describe()
```

[30]:
```
               carat          depth          table          price  \
count  403192.000000  403192.000000  403192.000000  403192.000000
mean        0.797700      61.747838      57.457640    4397.571519
std         0.474791       1.434470       2.240196    4501.692426
min         0.200000      43.000000      43.000000     304.000000
25%         0.400000      61.000000      56.000000    1053.000000
50%         0.700000      61.800000      57.000000    2677.500000
75%         1.040000      62.500000      59.000000    5983.000000
max         4.130000      79.000000      95.000000   26930.000000

                   x              y             z
count  403192.000000  403192.000000  403192.00000
mean        5.730006       5.732210       3.53842
std         1.123061       1.114364       0.71237
min         0.000000       0.000000       0.00000
25%         4.710000       4.720000       2.91000
50%         5.690000       5.710000       3.52000
75%         6.530000       6.530000       4.03000
max        10.140000      10.100000      31.80000
```

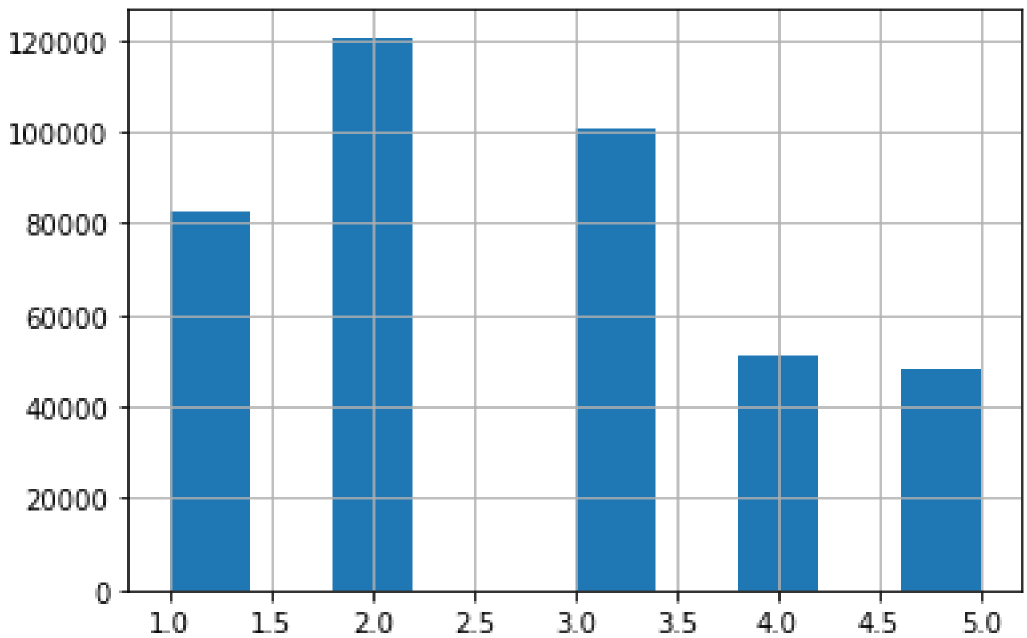
[31]:
```
# Divide the diamond carats by 0.4 to limit the number of carat categories
# Round up to have discrete categories
diamonds["carat_cat"] = np.ceil(diamonds["carat"] / 0.35)
```

```
# Merge categories > 5 in 5
diamonds["carat_cat"].where(diamonds["carat_cat"] < 5, 5.0, inplace = True)
```

[32]:
```
# Check the distribution of the diamonds in the categories
diamonds["carat_cat"].value_counts()
```

[32]:
```
2.0    120609
3.0    100673
1.0     82624
4.0     50871
5.0     48415
Name: carat_cat, dtype: int64
```

[33]:
```
diamonds["carat_cat"].hist()
plt.show()
```



[34]:
```
# Import the sklearn module
from sklearn.model_selection import StratifiedShuffleSplit

# Run the split. Creates on split and shares 20% of the dataset for the test set
split = StratifiedShuffleSplit(n_splits = 1, test_size = 0.2, random_state = 42)

# Separate the stratified train set and the test set
for x_train,x_test in split.split(diamonds, diamonds["carat_cat"]):
    strat_train_set = diamonds.iloc[x_train]
    strat_test_set = diamonds.iloc[x_test]
```

[35]:
```
for set in (strat_train_set, strat_test_set):
    set.drop(["carat_cat"], axis = 1, inplace = True)
```

[36]:
```
# Redefined diamonds dataset
diamonds = strat_train_set.copy()
diamonds.head()
```

[36]:

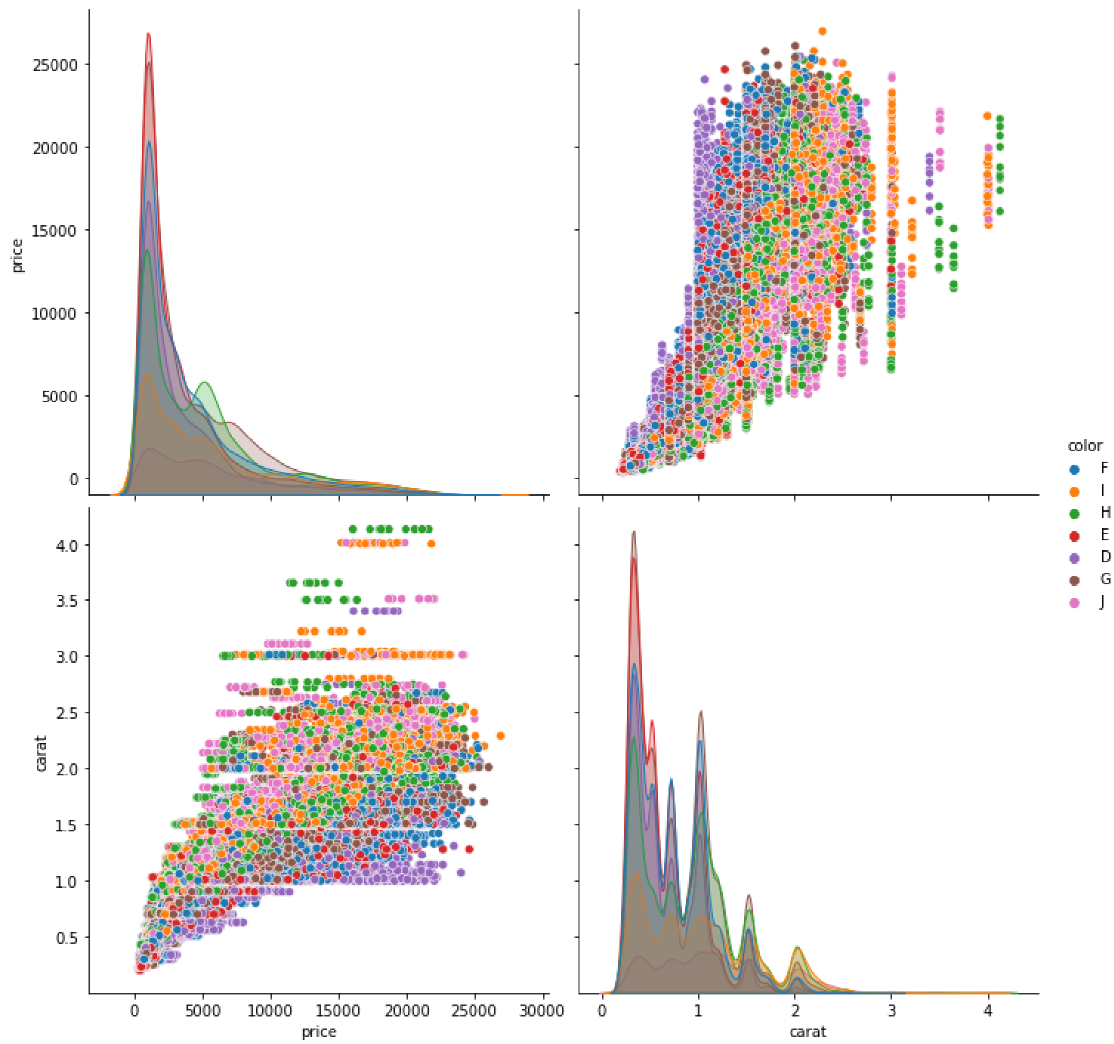
	carat	cut	color	clarity	depth	table	price	x	y	z
79064	1.09	Ideal	F	VS2	61.2	56.0	7616.0	6.68	6.65	4.08
80834	1.50	Very Good	I	VS2	60.1	60.0	9339.0	7.38	7.35	4.43
106815	1.19	Premium	H	SI2	61.9	58.0	4525.0	6.84	6.77	4.21
269312	0.75	Premium	I	SI1	62.0	60.0	2744.0	5.80	5.75	3.57
8251	1.21	Premium	H	VS2	60.8	62.0	5461.0	6.82	6.78	4.13

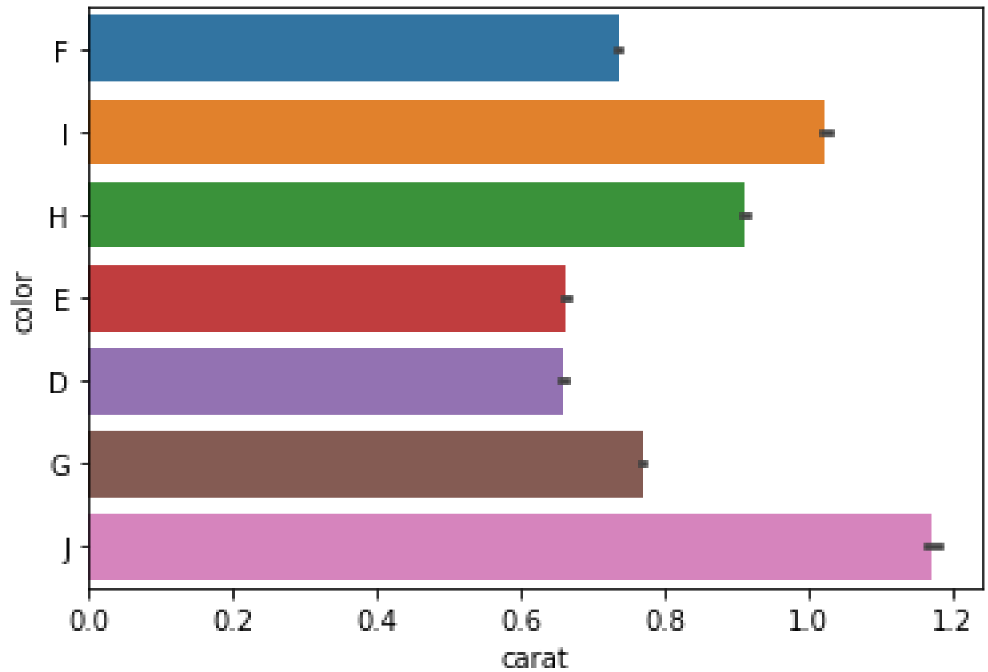
We can now perform a Stratified Sampling based on the carat categories : I will use Scikit-Learn's StratifiedShuffleSplit class.

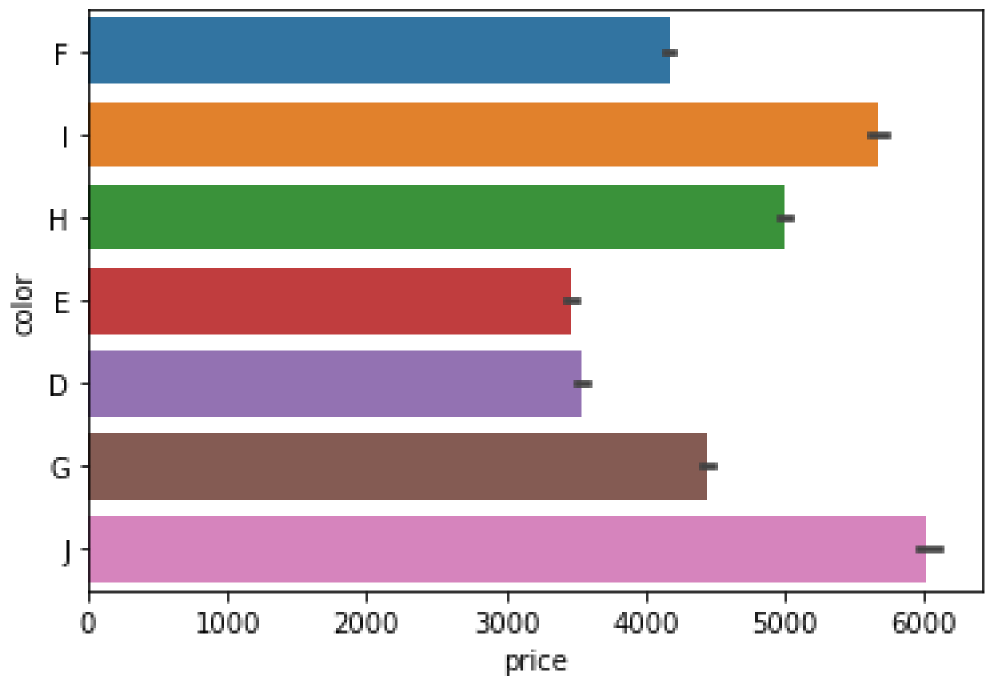
0.1 Data Visualization

0.1.1 Let's play arround with visualization of the data set

[37]:
```
sns.pairplot(diamonds[["price", "carat", "color"]], hue = "color", height = 5)
plt.show()
sns.barplot(x = "carat", y = "color", data = diamonds)
plt.show()
sns.barplot(x = "price", y = "color", data = diamonds)
plt.show()
```

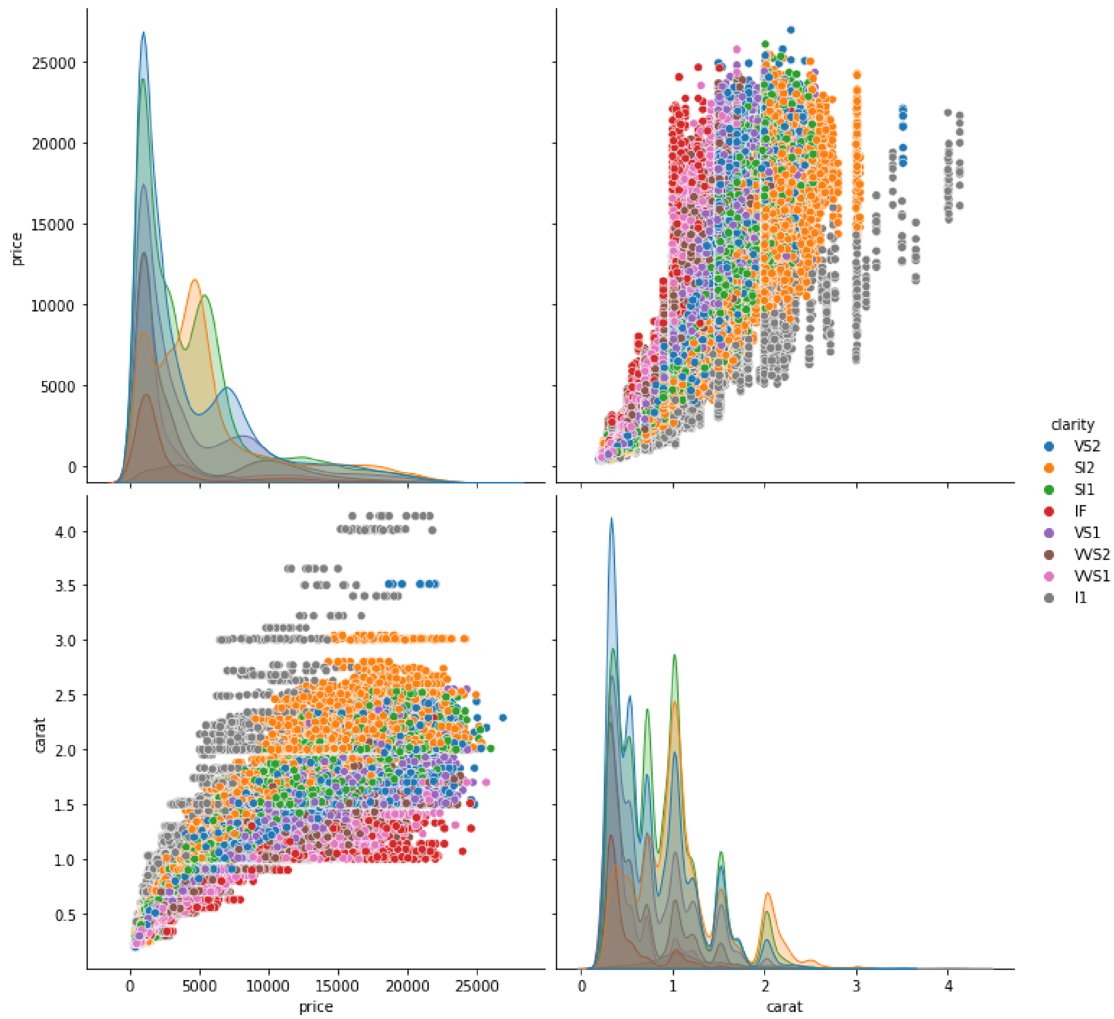

price
carat
color
F
I
H
E
D
G
J
0
5000
10000
15000
20000
25000
30000
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
1
2
3
4

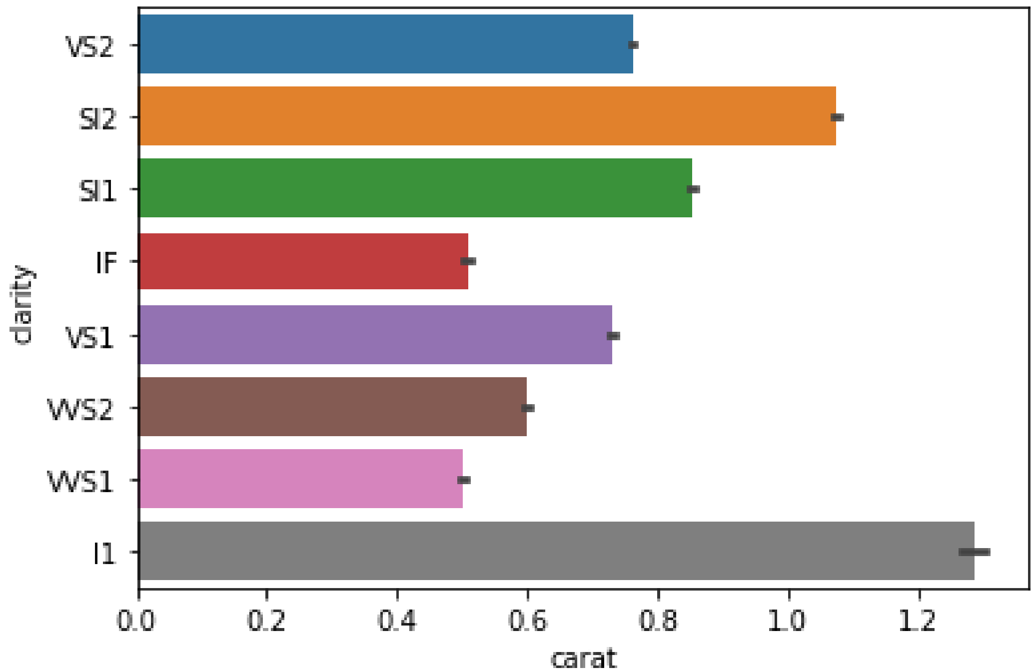


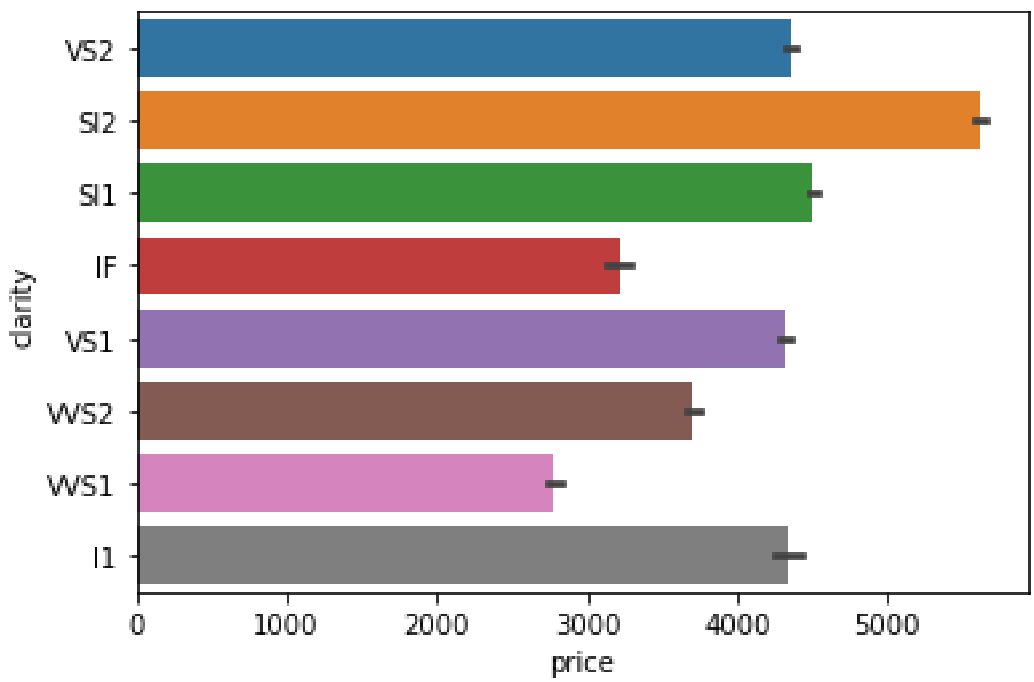


[38]:
```
sns.pairplot(diamonds[["price", "carat", "clarity"]], hue = "clarity", height =␣
↪5)
```

```
plt.show()
sns.barplot(x = "carat", y = "clarity", data = diamonds)
plt.show()
sns.barplot(x = "price", y = "clarity", data = diamonds)
plt.show()
```







[39]:
```
diamonds.describe()
```

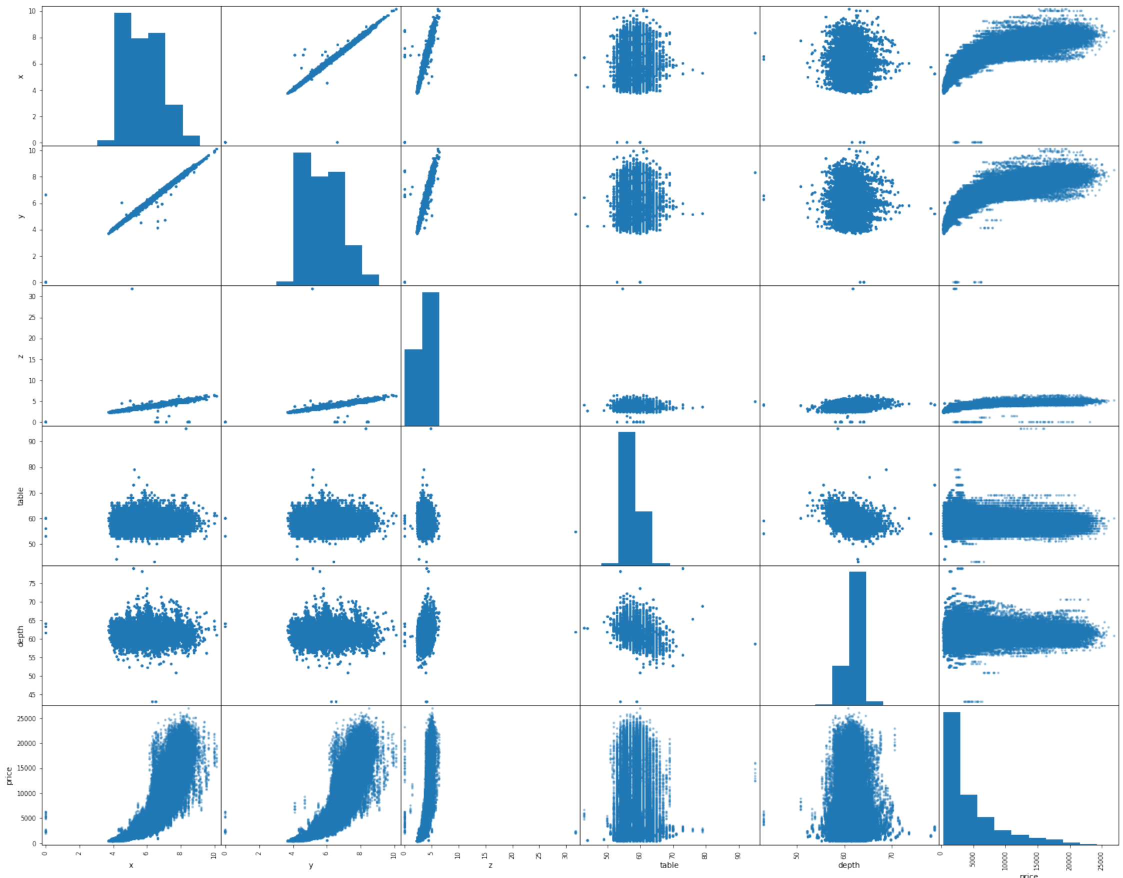
```
[39]:                carat          depth          table          price  \
count  322553.000000  322553.000000  322553.000000  322553.000000
mean        0.797825      61.748797      57.458247    4396.330634
std         0.474806       1.434381       2.239518    4496.460572
min         0.200000      43.000000      43.000000     304.000000
25%         0.400000      61.000000      56.000000    1054.000000
50%         0.700000      61.800000      57.000000    2680.000000
75%         1.040000      62.500000      59.000000    5984.000000
max         4.130000      79.000000      95.000000   26930.000000

                   x              y              z
count  322553.000000  322553.000000  322553.000000
mean        5.730354       5.732535       3.538676
std         1.122976       1.114397       0.713041
min         0.000000       0.000000       0.000000
25%         4.710000       4.720000       2.910000
50%         5.690000       5.710000       3.520000
75%         6.530000       6.530000       4.030000
max        10.140000      10.100000      31.800000
```

```
[40]: diamonds.head()
```

```
[40]:         carat        cut color clarity  depth  table   price     x     y     z
79064    1.09      Ideal     F     VS2   61.2   56.0  7616.0  6.68  6.65  4.08
80834    1.50  Very Good     I     VS2   60.1   60.0  9339.0  7.38  7.35  4.43
106815   1.19    Premium     H     SI2   61.9   58.0  4525.0  6.84  6.77  4.21
269312   0.75    Premium     I     SI1   62.0   60.0  2744.0  5.80  5.75  3.57
8251     1.21    Premium     H     VS2   60.8   62.0  5461.0  6.82  6.78  4.13
```

```
[41]: diamonds.to_csv("C:/Users/Owner/Desktop/diamond_set/new_diamond.csv")
```

```
[42]: from pandas.plotting import scatter_matrix

attributes = ["x", "y", "z", "table", "depth", "price"]
scatter_matrix(diamonds[attributes], figsize=(25, 20))
plt.show()
```



[43]:
```python
# Do not stratify the label
diamonds = strat_train_set.drop("price", axis = 1)

# Set a new dataset label variable
diamond_labels = strat_train_set["price"].copy()

# Drop all the category, so we could have only numeric
diamonds_num = diamonds.drop(["cut", "color", "clarity"], axis = 1)
diamonds_num.head()
```

[43]:

	carat	depth	table	x	y	z
79064	1.09	61.2	56.0	6.68	6.65	4.08
80834	1.50	60.1	60.0	7.38	7.35	4.43
106815	1.19	61.9	58.0	6.84	6.77	4.21
269312	0.75	62.0	60.0	5.80	5.75	3.57
8251	1.21	60.8	62.0	6.82	6.78	4.13

[44]:
```python
from sklearn.preprocessing import StandardScaler
```

```
# Perform the feature scaling on the numeric attributes of the dataset
num_scaler = StandardScaler()
diamonds_num_scaled = num_scaler.fit_transform(diamonds_num)

# Preview
pd.DataFrame(diamonds_num_scaled).head()
```

[44]:

	0	1	2	3	4	5
0	0.615358	-0.382603	-0.651144	0.845652	0.823285	0.759178
1	1.478870	-1.149485	1.134957	1.468997	1.451428	1.250035
2	0.825971	0.105414	0.241906	0.988131	0.930967	0.941496
3	-0.100725	0.175130	1.134957	0.062019	0.015672	0.043931
4	0.868094	-0.661469	2.028008	0.970321	0.939940	0.829301

0.1.2 Build prediction model based on multiple features

[45]:

```
from sklearn.preprocessing import StandardScaler

# Perform the feature scaling on the numeric attributes of the dataset
num_scaler = StandardScaler()
diamonds_num_scaled = num_scaler.fit_transform(diamonds_num)

# Preview
pd.DataFrame(diamonds_num_scaled).head()
```

[45]:

	0	1	2	3	4	5
0	0.615358	-0.382603	-0.651144	0.845652	0.823285	0.759178
1	1.478870	-1.149485	1.134957	1.468997	1.451428	1.250035
2	0.825971	0.105414	0.241906	0.988131	0.930967	0.941496
3	-0.100725	0.175130	1.134957	0.062019	0.015672	0.043931
4	0.868094	-0.661469	2.028008	0.970321	0.939940	0.829301

[46]:

```
#Handling Catagorical Variables
# We need only the category attributes to work with here
diamonds_cat = diamonds[["cut", "color", "clarity"]]
diamonds_cat.head()
```

[46]:

	cut	color	clarity
79064	Ideal	F	VS2
80834	Very Good	I	VS2
106815	Premium	H	SI2
269312	Premium	I	SI1
8251	Premium	H	VS2

[47]:

```
from sklearn.preprocessing import OneHotEncoder

# Perform the one-hot encoding on the category attributes of the dataset
```

```python
cat_encoder = OneHotEncoder()
diamonds_cat_encoded = cat_encoder.fit_transform(diamonds_cat)

# Convert the encoded categories to arrays and Preview
pd.DataFrame(diamonds_cat_encoded.toarray()).head()
```

[47]:
```
    0    1    2    3    4    5    6    7    8    9    10   11   12   13   14  \
0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  1.0
4  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0

    15   16   17   18   19
0  0.0  0.0  1.0  0.0  0.0
1  0.0  0.0  1.0  0.0  0.0
2  1.0  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0  0.0
4  0.0  0.0  1.0  0.0  0.0
```

[48]:
```python
from sklearn.compose import ColumnTransformer

num_attribs = list(diamonds_num)
cat_attribs = ["cut", "color", "clarity"]

# Pipeline to transform our dataset
pipeline = ColumnTransformer([
    ("num", StandardScaler(), num_attribs), # Perform feaured scaling on␣
 ↪numeric attributes
    ("cat", OneHotEncoder(), cat_attribs) # Perform One-Hot encoding on the␣
 ↪category attributes
])
```

[49]:
```python
# Transformed dataset to feed the ML Algorithm
diamonds_ready = pipeline.fit_transform(diamonds)

# Preview
pd.DataFrame(diamonds_ready).head()
```

[49]:
```
          0         1         2         3         4         5    6    7    8  \
0  0.615358 -0.382603 -0.651144  0.845652  0.823285  0.759178  0.0  0.0  1.0
1  1.478870 -1.149485  1.134957  1.468997  1.451428  1.250035  0.0  0.0  0.0
2  0.825971  0.105414  0.241906  0.988131  0.930967  0.941496  0.0  0.0  0.0
3 -0.100725  0.175130  1.134957  0.062019  0.015672  0.043931  0.0  0.0  0.0
4  0.868094 -0.661469  2.028008  0.970321  0.939940  0.829301  0.0  0.0  0.0

    9  …   16   17   18   19   20   21   22   23   24   25
```

```
0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
1  0.0  …  1.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
2  1.0  …  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
3  1.0  …  1.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
4  1.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0

[5 rows x 26 columns]
```

[50]:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from random import randint

# Our test set
# Remove target feature from test set
X_test = strat_test_set.drop("price", axis = 1)
# Have stand alone features
y_test = strat_test_set["price"].copy()


# Our models performance holder
models_rmse = [] # Holds Models originalMSE
cvs_rmse_mean = [] # Holds the Cross Validation RMSE Mean
tests_rmse = [] # Holds the tests RMSE
tests_accuracy = [] # Holds the tests accuracy
models = [] # Holds the models name


def display_model_performance(model_name, model, diamonds = diamonds_ready,␣
↪labels = diamond_labels,
                              models_rmse = models_rmse, cvs_rmse_mean =␣
↪cvs_rmse_mean, tests_rmse = tests_rmse,
                              tests_accuracy = tests_accuracy, pipeline =␣
↪pipeline, X_test = X_test,
                              y_test = y_test, cv = True):
     # Fit dataset in model
    model.fit(diamonds, labels)

    # Setup predictions
    predictions = model.predict(diamonds)

    # Get models performance
    model_mse = mean_squared_error(labels, predictions)
    model_rmse = np.sqrt(model_mse)

    # Cross validation
    cv_score = cross_val_score(model, diamonds, labels, scoring =␣
↪"neg_mean_squared_error", cv = 10)
```

```
cv_rmse = np.sqrt(-cv_score)
cv_rmse_mean = cv_rmse.mean()

print("RMSE: %.4f" %model_rmse)
models_rmse.append(model_rmse)

print("CV-RMSE: %.4f" %cv_rmse_mean)
cvs_rmse_mean.append(cv_rmse_mean)

print("--- Test Performance ---")

X_test_prepared = pipeline.transform(X_test)

 # Fit test dataset in model
model.fit(X_test_prepared, y_test)

# Setup test predictions
test_predictions = model.predict(X_test_prepared)

# Get models performance on test
test_model_mse = mean_squared_error(y_test, test_predictions)
test_model_rmse = np.sqrt(test_model_mse)
print("RMSE: %.4f" %test_model_rmse)
tests_rmse.append(test_model_rmse)

# Tests accuracy
test_accuracy = round(model.score(X_test_prepared, y_test) * 100, 2)
print("Accuracy:", str(test_accuracy)+"%")
tests_accuracy.append(test_accuracy)

# Check how well model works on Test set by comparing prices
start = randint(1, len(y_test))
some_data = X_test.iloc[start:start + 7]
some_labels = y_test.iloc[start:start + 7]
some_data_prepared = pipeline.transform(some_data)
print("Predictions:\t", model.predict(some_data_prepared))
print("Labels:\t\t", list(some_labels))

models.append(model_name)

# Preview plot
plt.scatter(diamond_labels, model.predict(diamonds_ready))
plt.xlabel("Actual")
plt.ylabel("Predicted")
x_lim = plt.xlim()
y_lim = plt.ylim()
```

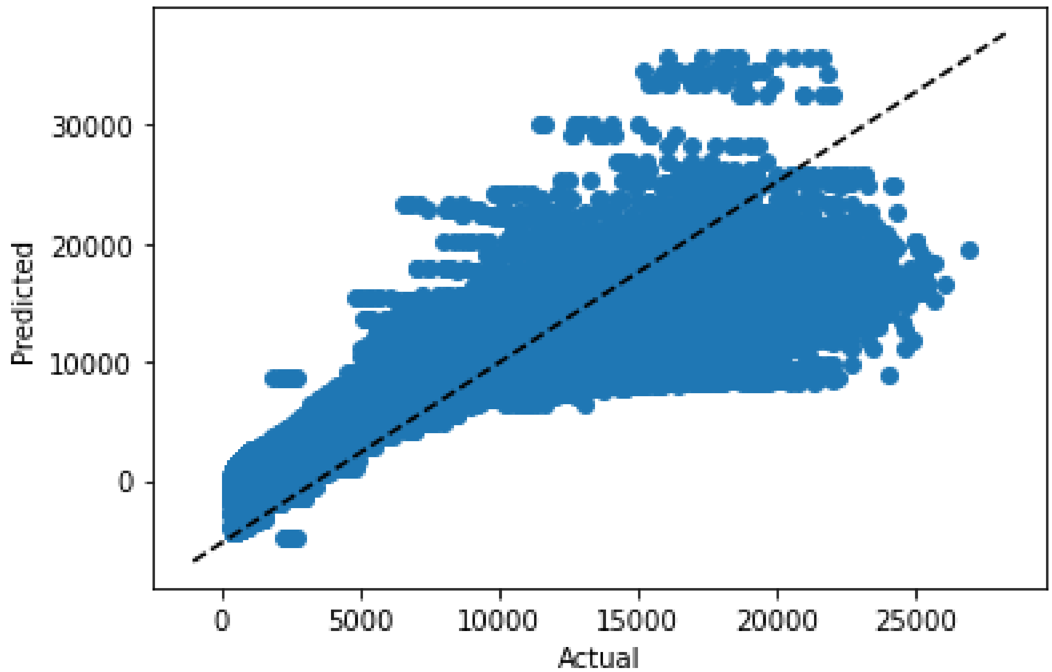
```
    plt.plot(x_lim, y_lim, "k--")
    plt.show()

    print("------- Test -------")
    plt.scatter(y_test, model.predict(X_test_prepared))
    plt.xlabel("Actual")
    plt.ylabel("Predicted")
    plt.plot(x_lim, y_lim, "k--")
    plt.show()
```

We can now start fitting models and get their performance error. We are using Root Mean Squared Error for our performance measure.

0.2 Linear Regression

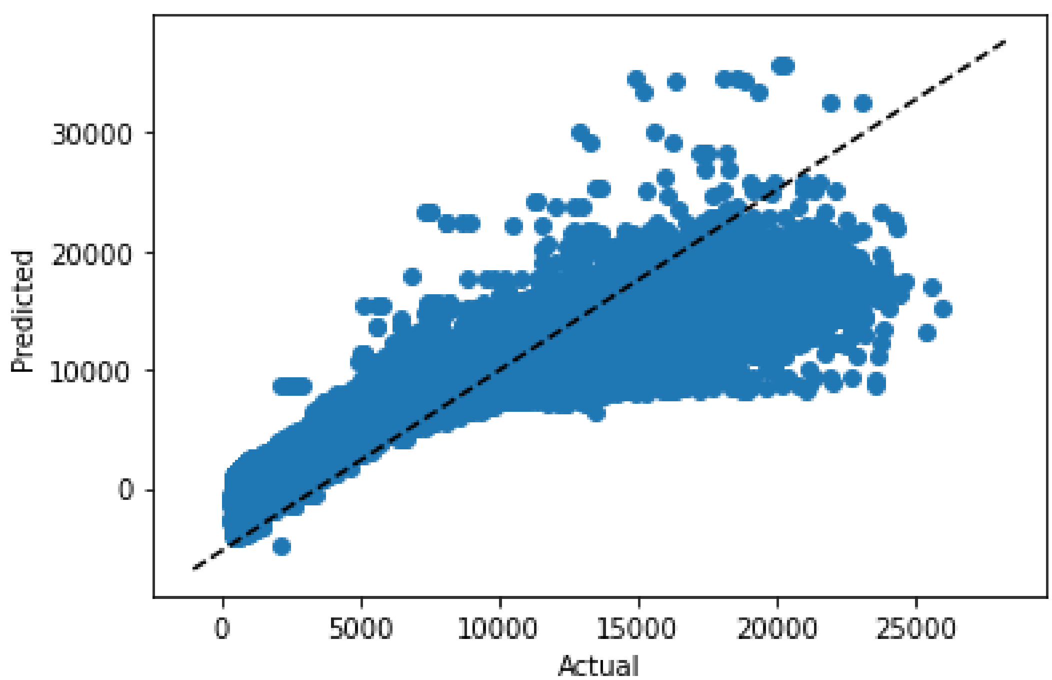
[51]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression(normalize = True)
display_model_performance("Linear Regression", lin_reg)
```

```
RMSE: 1372.0532
CV-RMSE: 1372.2307
--- Test Performance ---
RMSE: 1376.8318
Accuracy: 90.73%
Predictions:     [ 1036.3606775   2871.16259804   122.81520923 13419.49156411
  1017.92549942  3454.78791488  1347.93573761]
Labels:          [644.0, 3110.0, 722.0, 14433.0, 831.0, 2906.0, 843.0]
```



30000
20000
10000
0
Predicted
0
5000
10000
15000
20000
25000
Actual


```
------- Test -------
```

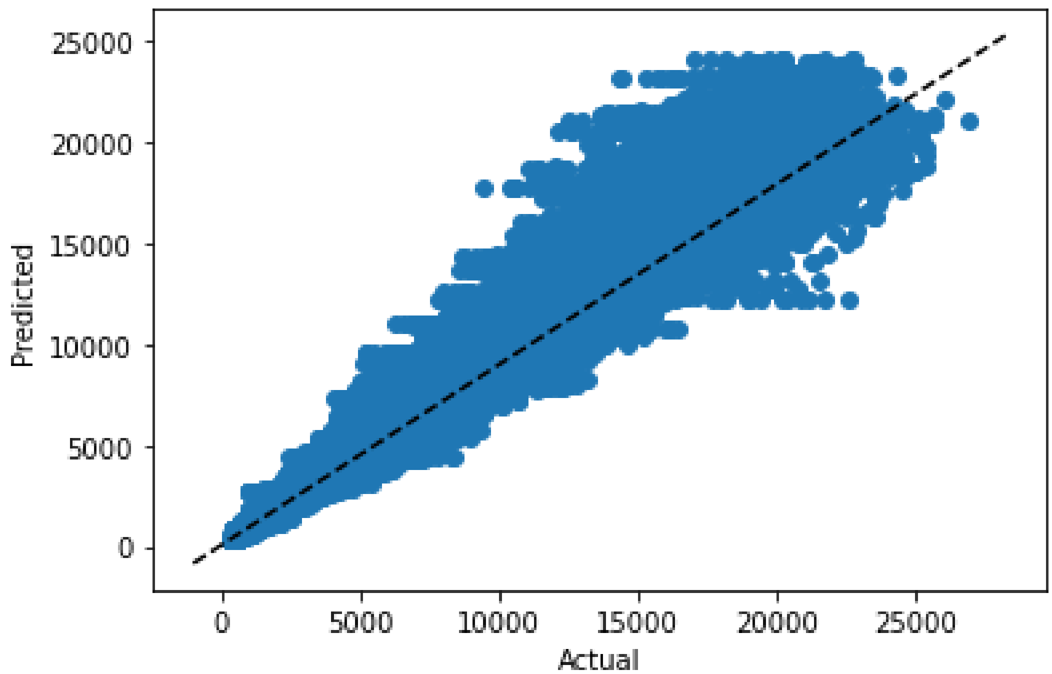


30000
20000
10000
0
Predicted
0
5000
10000
15000
20000
25000
Actual

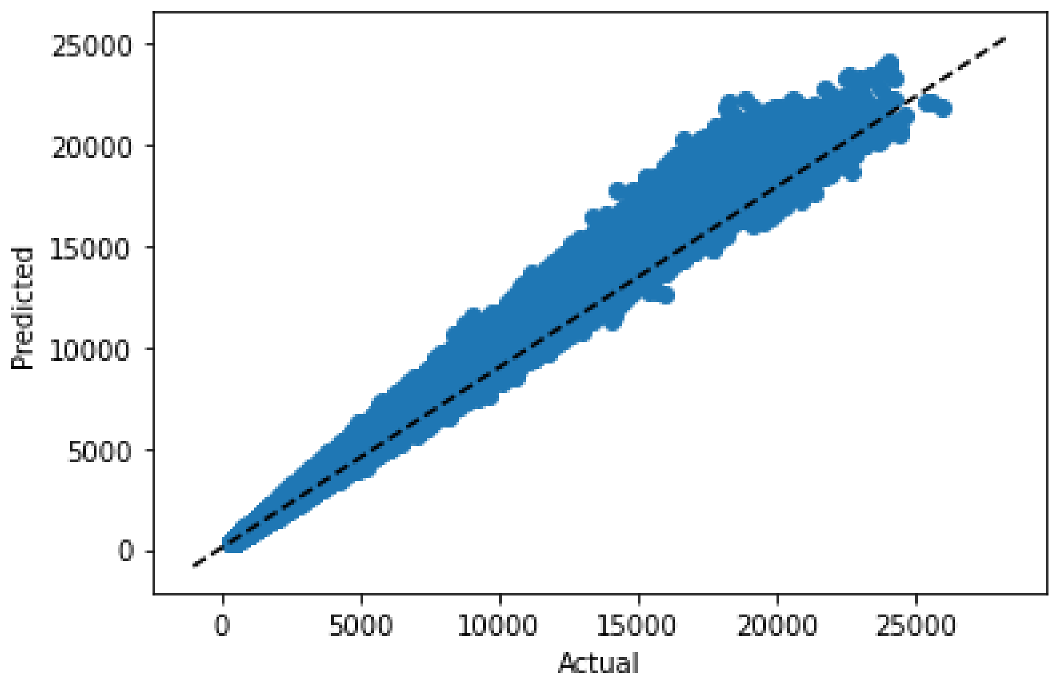
0.3 Decision Tree Regression

[52]:
```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state = 42)
display_model_performance("Decision Tree Regression", tree_reg)
```

```
RMSE: 510.7221
CV-RMSE: 573.9419
--- Test Performance ---
RMSE: 425.3440
Accuracy: 99.12%
Predictions:     [ 441.          2532.25        8786.6         3430.33333333 8069.
 2515.          6425.5       ]
Labels:          [441.0, 2557.0, 8296.0, 3304.0, 8069.0, 2491.0, 6293.0]
```



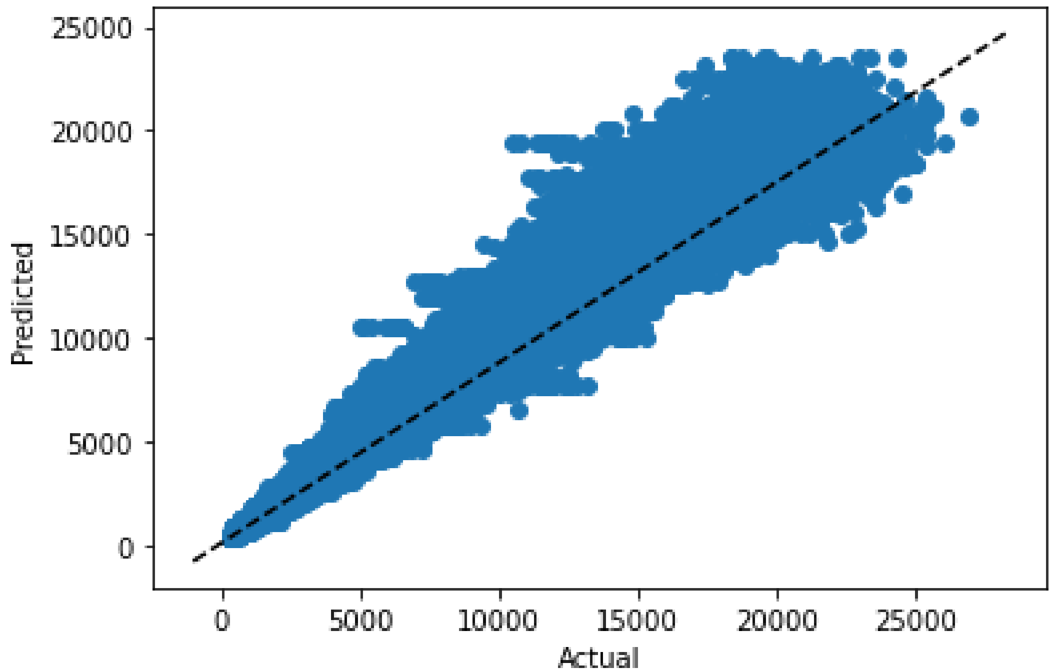

```
------- Test -------
```



0.3.1 Random Forest Regression

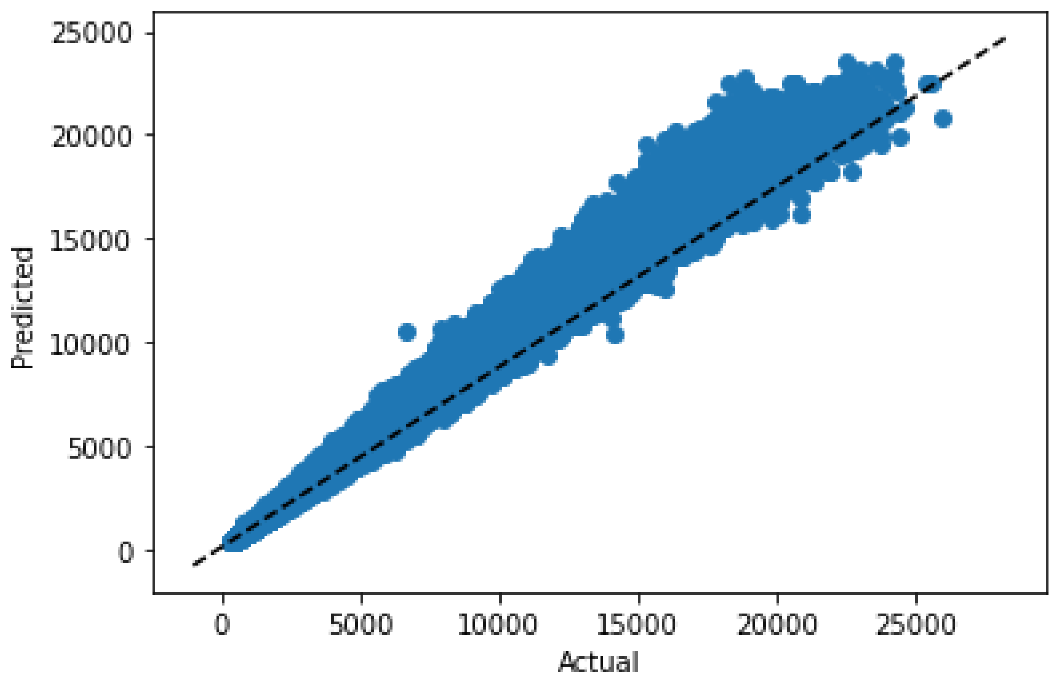
[53]:
```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators = 10, random_state = 42)
display_model_performance("Random Forest Regression", forest_reg)
```

```
RMSE: 513.7440
CV-RMSE: 577.4522
--- Test Performance ---
RMSE: 455.3055
Accuracy: 98.99%
Predictions:      [ 3790.48333333   919.65166667  5636.62357143 20116.74833333
  1000.49166667  6283.13333333  4879.45      ]
Labels:           [3909.0, 1021.0, 5792.0, 20072.0, 1007.0, 6452.0, 4482.0]
```



Predicted
Actual
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
25000


```
------- Test -------
```

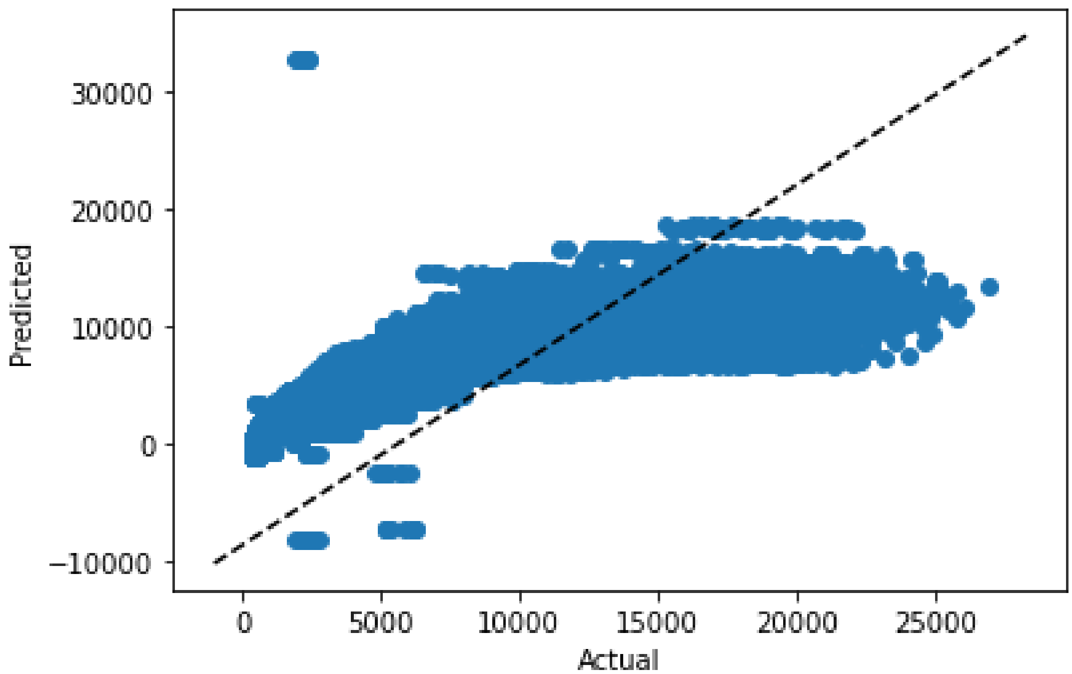


Predicted
Actual
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
25000

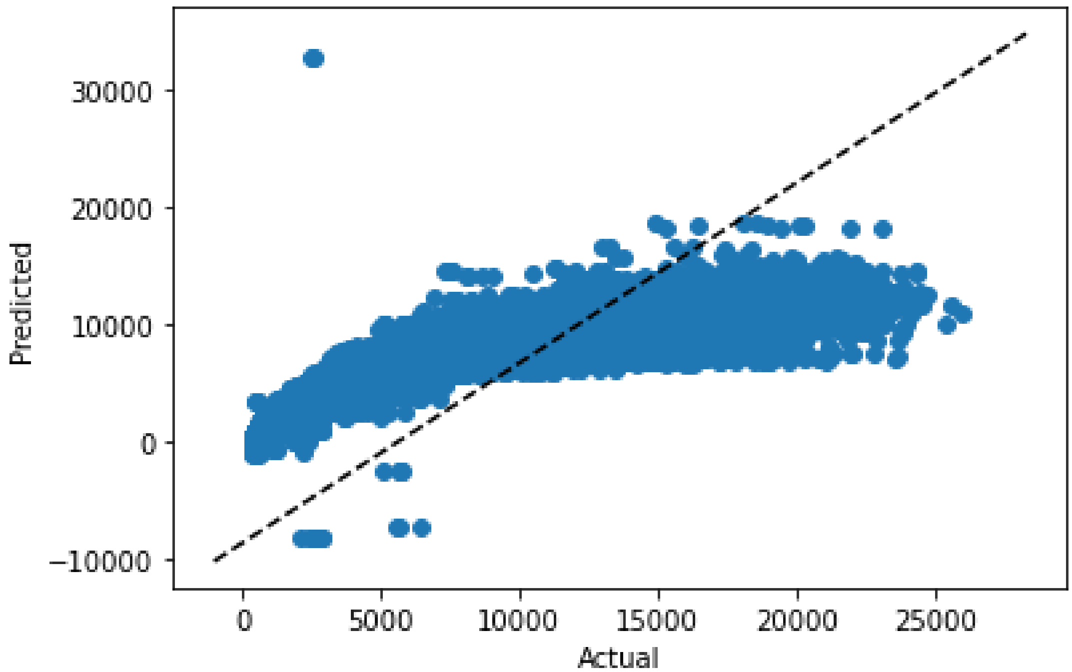
0.4 Ridge Regression

[54]:
```
from sklearn.linear_model import Ridge

ridge_reg = Ridge(normalize = True)
display_model_performance("Ridge Regression", ridge_reg)
```

```
RMSE: 2043.7825
CV-RMSE: 2043.9321
--- Test Performance ---
RMSE: 2059.8497
Accuracy: 79.26%
Predictions:      [ 1240.11985888  4294.6933905   6012.59313031 11976.15806539
 11343.10027794   526.42254813  2783.02404763]
Labels:           [1025.0, 2850.0, 4422.0, 21233.0, 17192.0, 680.0, 1412.0]
```



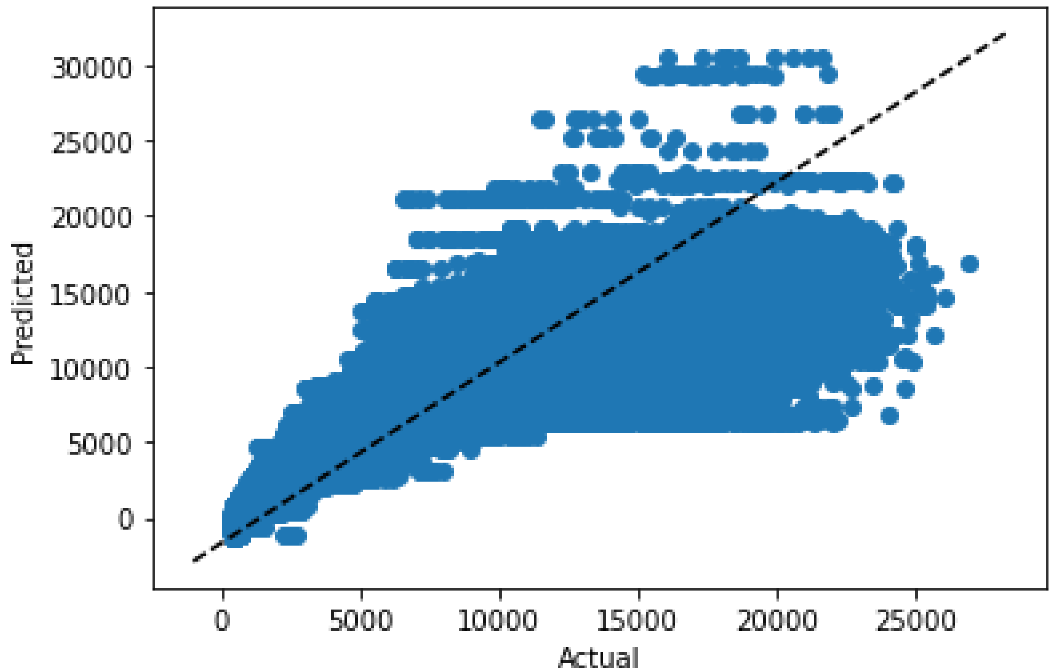

```
------- Test -------
```



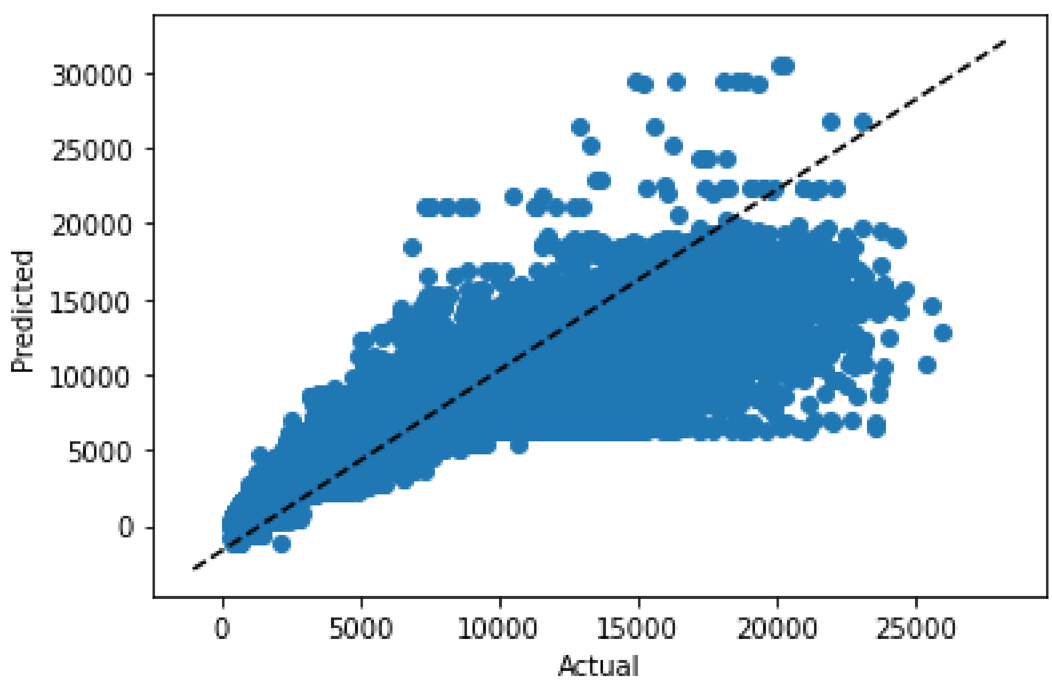
0.5 Lasso Regression

[55]:
```
from sklearn.linear_model import Lasso

lasso_reg = Lasso(normalize = True)
display_model_performance("Lasso Regression", lasso_reg)
```

```
RMSE: 1898.2986
CV-RMSE: 1889.7609
--- Test Performance ---
RMSE: 1732.4304
Accuracy: 85.33%
Predictions:      [ 815.15575328 2310.74823845 6080.5518979   981.33269608
2476.92518125
 6548.26027978 3338.63234175]
Labels:           [558.0, 2043.0, 4177.0, 996.0, 1835.0, 11860.0, 2959.0]
```


30000
25000
20000
15000
10000
5000
0
Predicted
0
5000
10000
15000
20000
25000
Actual

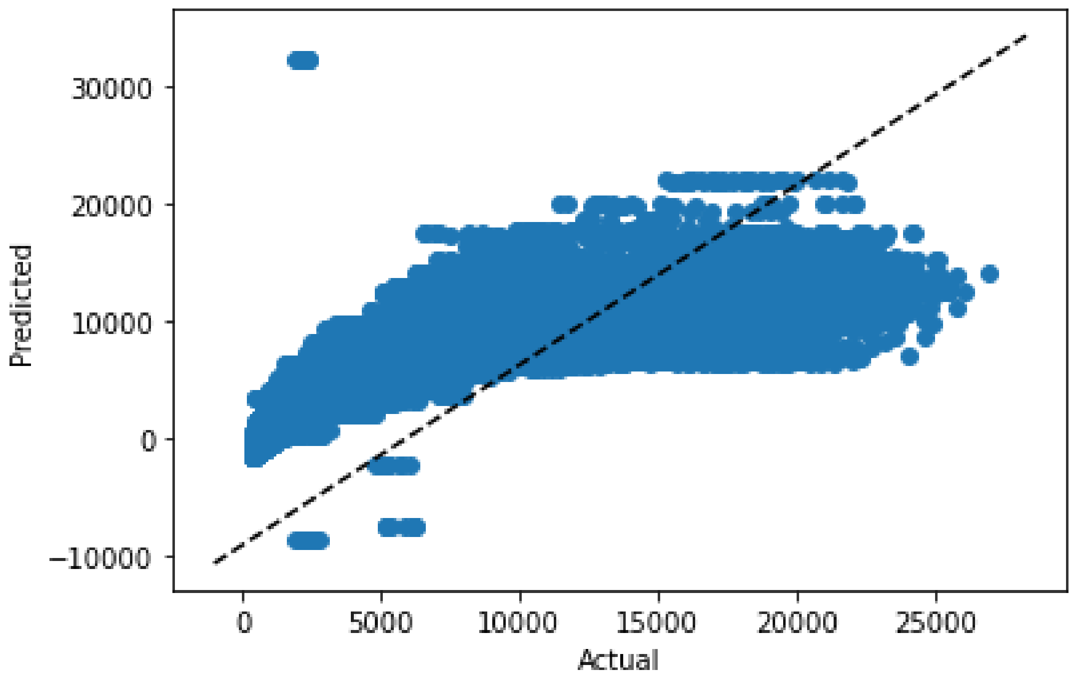
------- Test -------


30000
25000
20000
15000
10000
5000
0
Predicted
0
5000
10000
15000
20000
25000
Actual

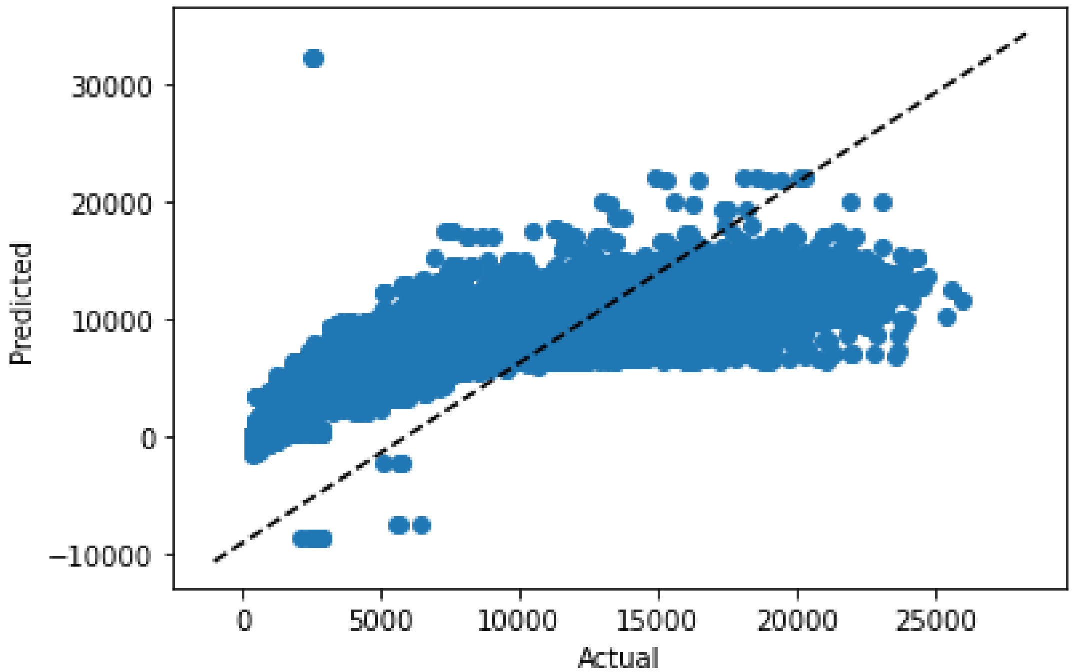
0.6 Elastic Net Regression

[56]:
```
from sklearn.linear_model import ElasticNet

net_reg = ElasticNet()
display_model_performance("Elastic Net Regression", net_reg)
```

```
RMSE: 1987.6651
CV-RMSE: 1987.7956
--- Test Performance ---
RMSE: 2004.1156
Accuracy: 80.36%
Predictions:      [ 5668.43803649  1050.13153585  3117.56088973   239.48574925
  6472.05781275 11379.39931957  6828.97197001]
Labels:           [4851.0, 1250.0, 2055.0, 837.0, 7021.0, 18377.0, 6147.0]
```




```
------- Test -------
```



0.7 Save the model

[57]:
```
#create Pickle file
import pickle
pickle_out = open("forest_reg.pkl","wb")
pickle.dump(forest_reg,pickle_out)
pickle_out.close()
```

0.8 Model Deployment of the prediction using Streamlit

[58]:
```
#import streamlit as st
```

[59]:
```
#loading the trained model
#pickle_in = open('model.pkl', 'rb')
#regressor = pickle.load(pickle_in)
```

[60]:
```
#diamonds.to_csv("model_prediction_dataset.csv")
```

[61]:
```
#diamonds.to_csv("C:/Users/Owner/Downloads/prediction_model.csv")
```

[ ]:
```
```