Diamond Price Estimation based on Features - ML Based Algorithms Application

Yinuo Song

1. Explore Dataset & Examine what Features affect the Price of Diamonds.

1.1) Importing Libraries

In [63]:
```
# Ignore warnings :
import warnings
warnings.filterwarnings('ignore')

# Handle table-like data and matrices :
import numpy as np
import pandas as pd
import math
```

In [64]:
```
# Modelling Algorithms :

# Classification
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier , GradientBoostingCl
assifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis , Q
uadraticDiscriminantAnalysis

# Regression
from sklearn.linear_model import LinearRegression,Ridge,Lasso,RidgeCV, E
lasticNet
from sklearn.ensemble import RandomForestRegressor,BaggingRegressor,Grad
ientBoostingRegressor,AdaBoostRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.neural_network import MLPRegressor
```

In [65]:
```
# Modelling Helpers :
from sklearn.preprocessing import Imputer , Normalizer , scale
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import RFECV
from sklearn.model_selection import GridSearchCV , KFold , cross_val_sco
re

#preprocessing :
from sklearn.preprocessing import MinMaxScaler , StandardScaler, Imputer
, LabelEncoder
```

In [66]:
```
#evaluation metrics :

# Regression
from sklearn.metrics import mean_squared_log_error,mean_squared_error, r
2_score,mean_absolute_error

# Classification
from sklearn.metrics import accuracy_score,precision_score,recall_score,
f1_score
```

In [67]:
```
# Visualisation
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import seaborn as sns
import missingno as msno

# Configure visualisations
%matplotlib inline
mpl.style.use( 'ggplot' )
plt.style.use('fivethirtyeight')
sns.set(context="notebook", palette="dark", style = 'whitegrid' , color_
codes=True)
params = {
    'axes.labelsize': "large",
    'xtick.labelsize': 'x-large',
    'legend.fontsize': 20,
    'figure.dpi': 150,
    'figure.figsize': [25, 7]
}
plt.rcParams.update(params)
```

In [68]:
```
# Center all plots
from IPython.core.display import HTML
HTML("""
<style>
.output_png {
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
</style>
""");
```

1.2) Extract Dataset

In [5]:
```
df = pd.read_csv('/Users/yinuo/Desktop/diamonds.csv')
diamonds = df.copy()
```

In [6]:
```
# How the data looks
df.head()
```

Out[6]:

	Unnamed: 0	carat	cut	color	clarity	depth	table	price	x	y	z
0	1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

1.3) Features

Qualitative Features (Categorical) : Cut, Color, Clarity.

Quantitative Features (Numerical) : Carat, Depth , Table , Price , X , Y, Z.

Price is the Target Variable.

1.4) Drop the 'Unnamed: 0' column as we already have Index

In [7]:
```
df.drop(['Unnamed: 0'] , axis=1 , inplace=True)
df.head()
```

Out[7]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

In [8]:
```
df.shape
```

Out[8]:
```
(53940, 10)
```

In [9]:
```
# So, We have 53,940 rows and 10 columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
carat      53940 non-null float64
cut        53940 non-null object
color      53940 non-null object
clarity    53940 non-null object
depth      53940 non-null float64
table      53940 non-null float64
price      53940 non-null int64
x          53940 non-null float64
y          53940 non-null float64
z          53940 non-null float64
dtypes: float64(6), int64(1), object(3)
memory usage: 4.1+ MB
```

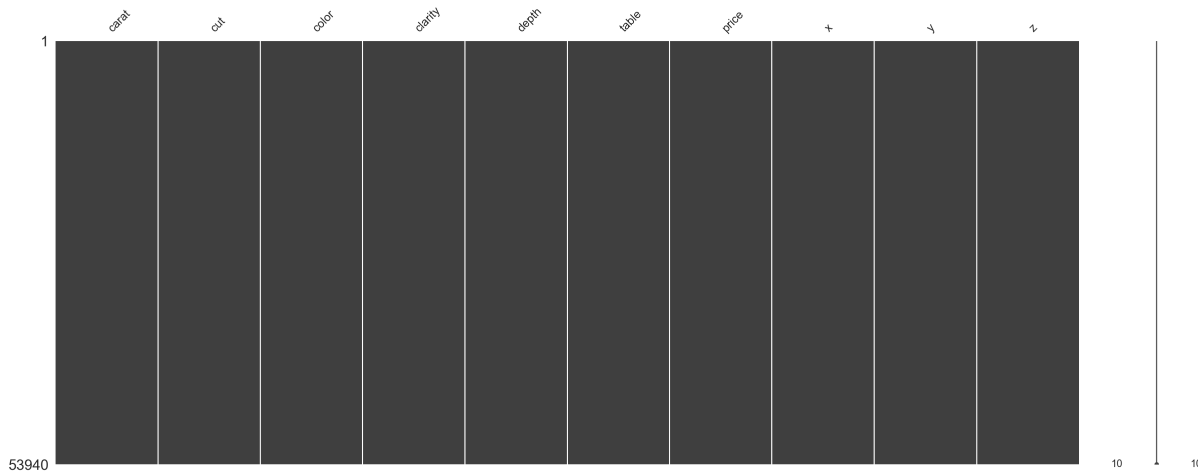
1.5) Examine NaN Values

In [10]:
```
# It seems there are no Null Values.
# Let's Confirm
df.isnull().sum()
```

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

In [11]:
```
msno.matrix(df) # just to visualize. no missing values.
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12cb02950>
```



In [12]:
```
df.describe()
```

Out[12]:

	carat	depth	table	price	x	y	
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	5394
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	3

In [13]:
```
df.loc[(df['x']==0) | (df['y']==0) | (df['z']==0)]
```

Out[13]:

	carat	cut	color	clarity	depth	table	price	x	y	z
2207	1.00	Premium	G	SI2	59.1	59.0	3142	6.55	6.48	0.0
2314	1.01	Premium	H	I1	58.1	59.0	3167	6.66	6.60	0.0
4791	1.10	Premium	G	SI2	63.0	59.0	3696	6.50	6.47	0.0
5471	1.01	Premium	F	SI2	59.2	58.0	3837	6.50	6.47	0.0
10167	1.50	Good	G	I1	64.0	61.0	4731	7.15	7.04	0.0
11182	1.07	Ideal	F	SI2	61.6	56.0	4954	0.00	6.62	0.0
11963	1.00	Very Good	H	VS2	63.3	53.0	5139	0.00	0.00	0.0
13601	1.15	Ideal	G	VS2	59.2	56.0	5564	6.88	6.83	0.0
15951	1.14	Fair	G	VS1	57.5	67.0	6381	0.00	0.00	0.0
24394	2.18	Premium	H	SI2	59.4	61.0	12631	8.49	8.45	0.0
24520	1.56	Ideal	G	VS2	62.2	54.0	12800	0.00	0.00	0.0
26123	2.25	Premium	I	SI1	61.3	58.0	15397	8.52	8.42	0.0
26243	1.20	Premium	D	VVS1	62.1	59.0	15686	0.00	0.00	0.0
27112	2.20	Premium	H	SI1	61.2	59.0	17265	8.42	8.37	0.0
27429	2.25	Premium	H	SI2	62.8	59.0	18034	0.00	0.00	0.0
27503	2.02	Premium	H	VS2	62.7	53.0	18207	8.02	7.95	0.0
27739	2.80	Good	G	SI2	63.8	58.0	18788	8.90	8.85	0.0
49556	0.71	Good	F	SI2	64.1	60.0	2130	0.00	0.00	0.0
49557	0.71	Good	F	SI2	64.1	60.0	2130	0.00	0.00	0.0
51506	1.12	Premium	G	I1	60.4	59.0	2383	6.71	6.67	0.0

In [14]:
```
len(df[(df['x']==0) | (df['y']==0) | (df['z']==0)])
```

Out[14]: 20

1.6) Dropping Rows with Dimensions 'Zero'

In [15]:
```
df = df[(df[['x','y','z']] != 0).all(axis=1)]
```

In [16]:
```
# Just to Confirm
df.loc[(df['x']==0) | (df['y']==0) | (df['z']==0)]
```

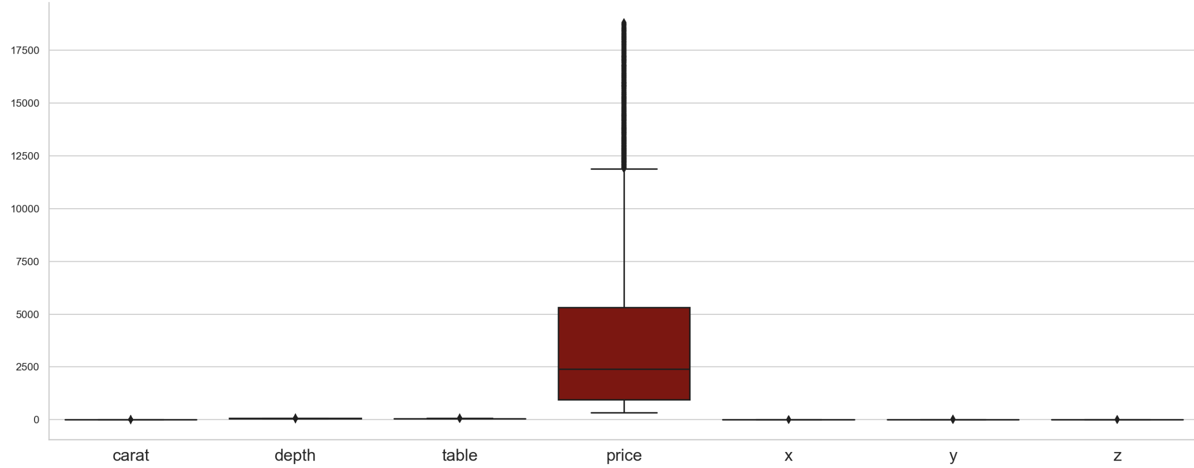
Out[16]:

carat	cut	color	clarity	depth	table	price	x	y	z

1.7) Scaling of all Features

In [17]:
```
sns.factorplot(data=df , kind='box' , size=7, aspect=2.5)
```

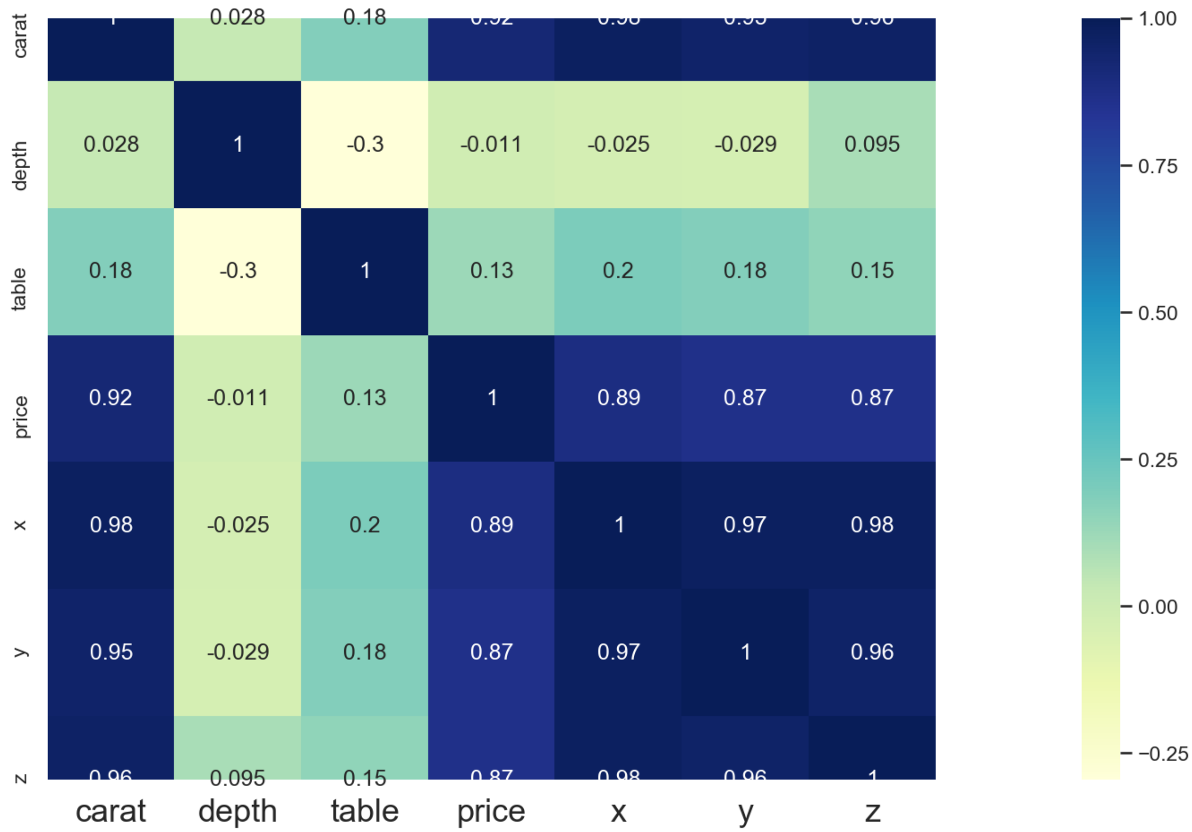
Out[17]: <seaborn.axisgrid.FacetGrid at 0x118e35910>



2. Correlation Between Features

In [21]:
```
# Correlation Map
corr = df.corr()
sns.heatmap(data=corr, square=True , annot=True, cbar=True, cmap="YlGnB
u")
```

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x12d6ac7d0>

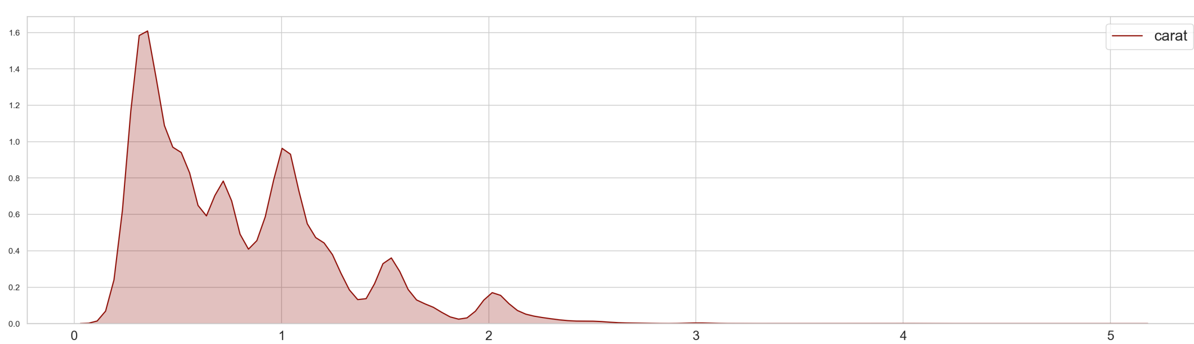


3. Visualization Of All Features

3.1) Carat

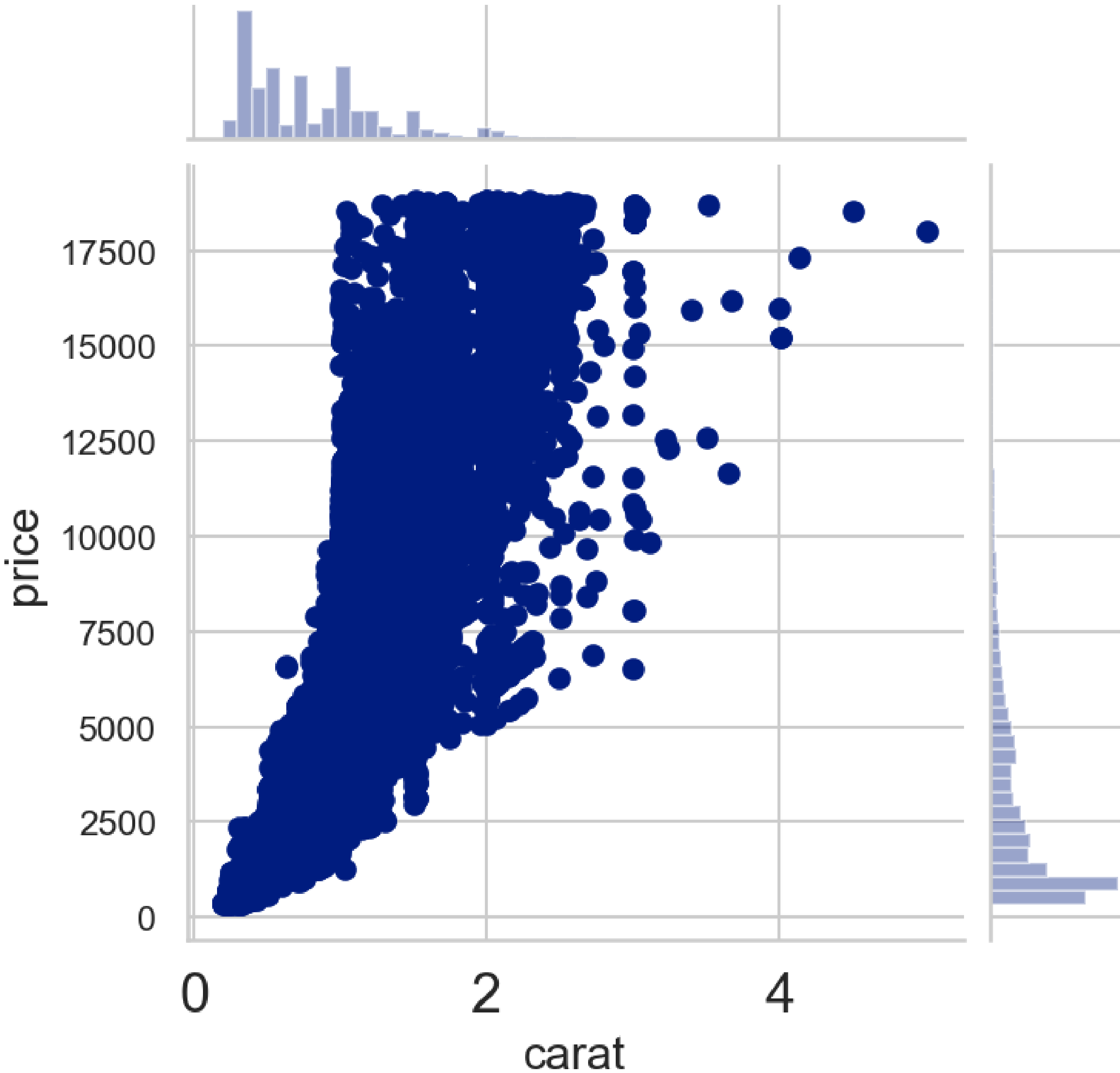
In [22]:
```
# Visualize via kde plots
sns.kdeplot(df['carat'], shade=True , color='r')
```

Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x12dc9cf10>



In [23]:
```
# Cara & price
sns.jointplot(x='carat' , y='price' , data=df , size=5)
```

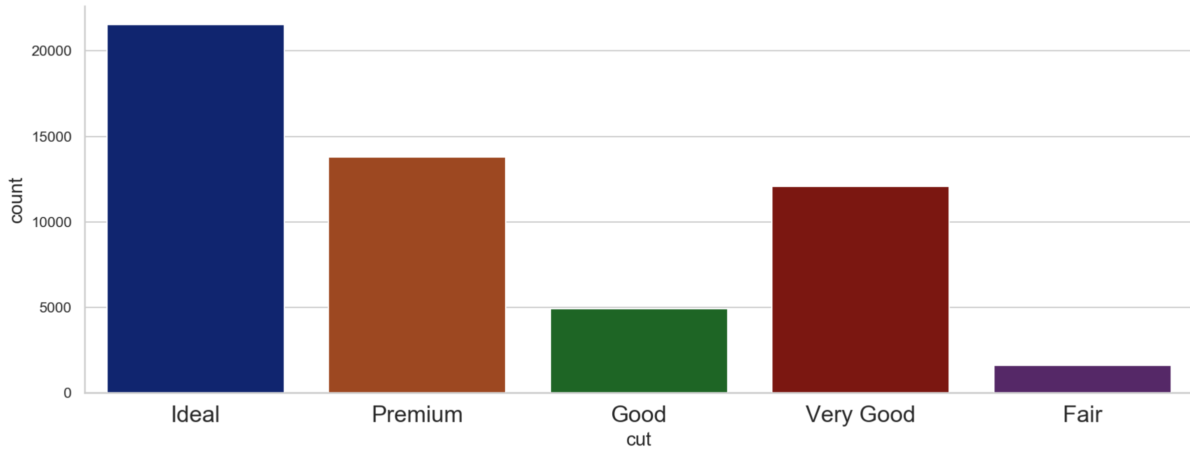
Out[23]: <seaborn.axisgrid.JointGrid at 0x12dce7050>



3.2) Cut

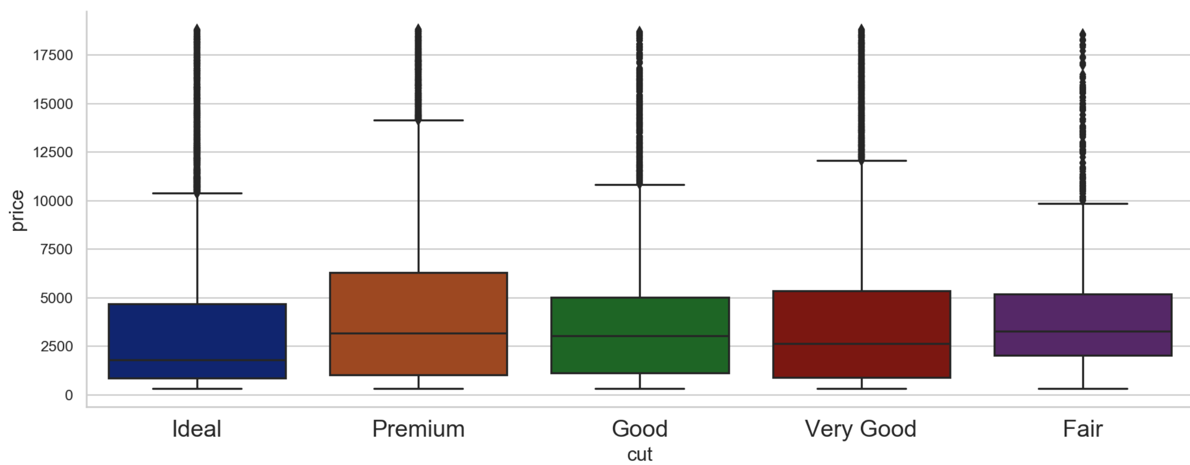
In [24]:
```
sns.factorplot(x='cut', data=df , kind='count',aspect=2.5 )
```

Out[24]: <seaborn.axisgrid.FacetGrid at 0x12da20e90>



In [25]:
```
# cut & price
sns.factorplot(x='cut', y='price', data=df, kind='box' ,aspect=2.5 )
```

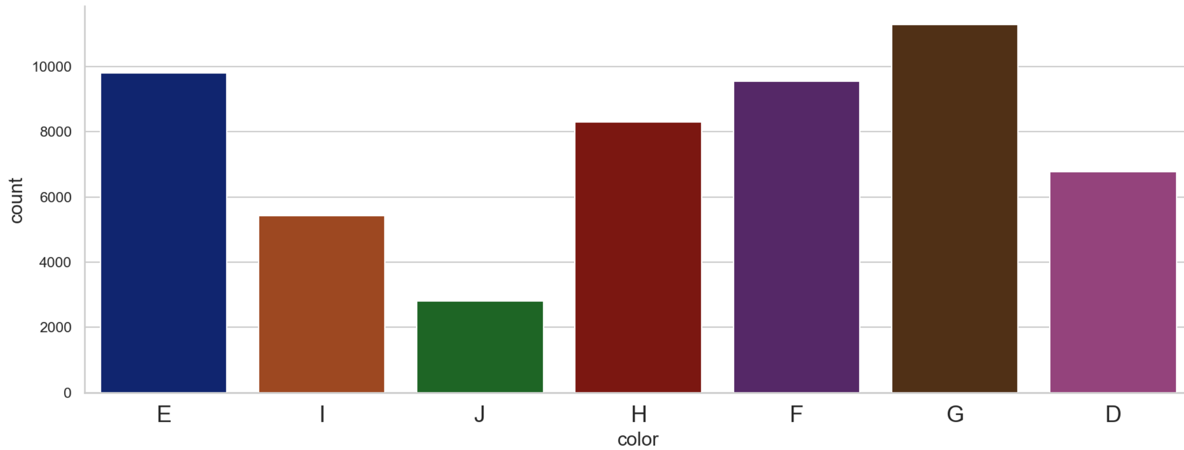
Out[25]: <seaborn.axisgrid.FacetGrid at 0x12d637990>



3.3) Color

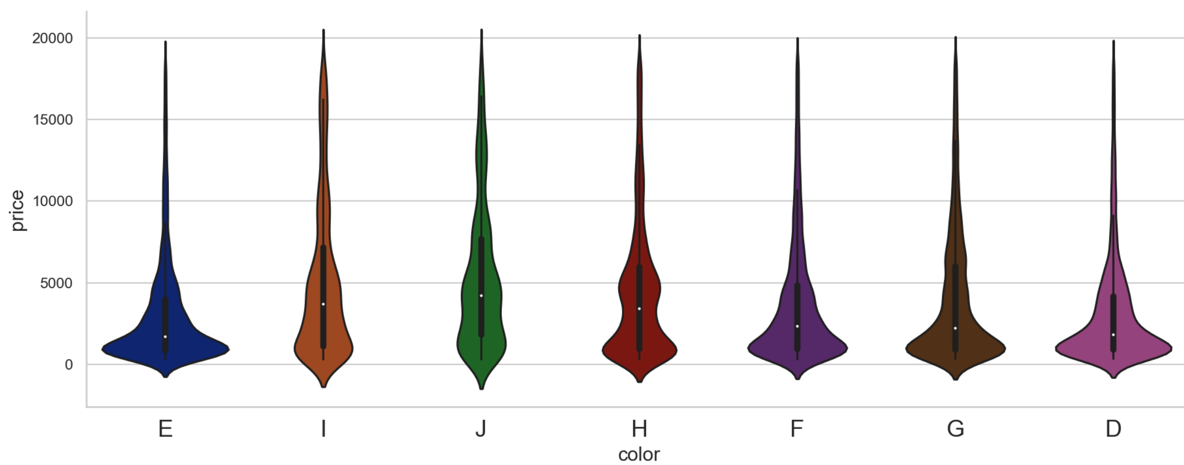
In [26]:
```
sns.factorplot(x='color', data=df , kind='count',aspect=2.5 )
```

Out[26]: <seaborn.axisgrid.FacetGrid at 0x12d54a590>



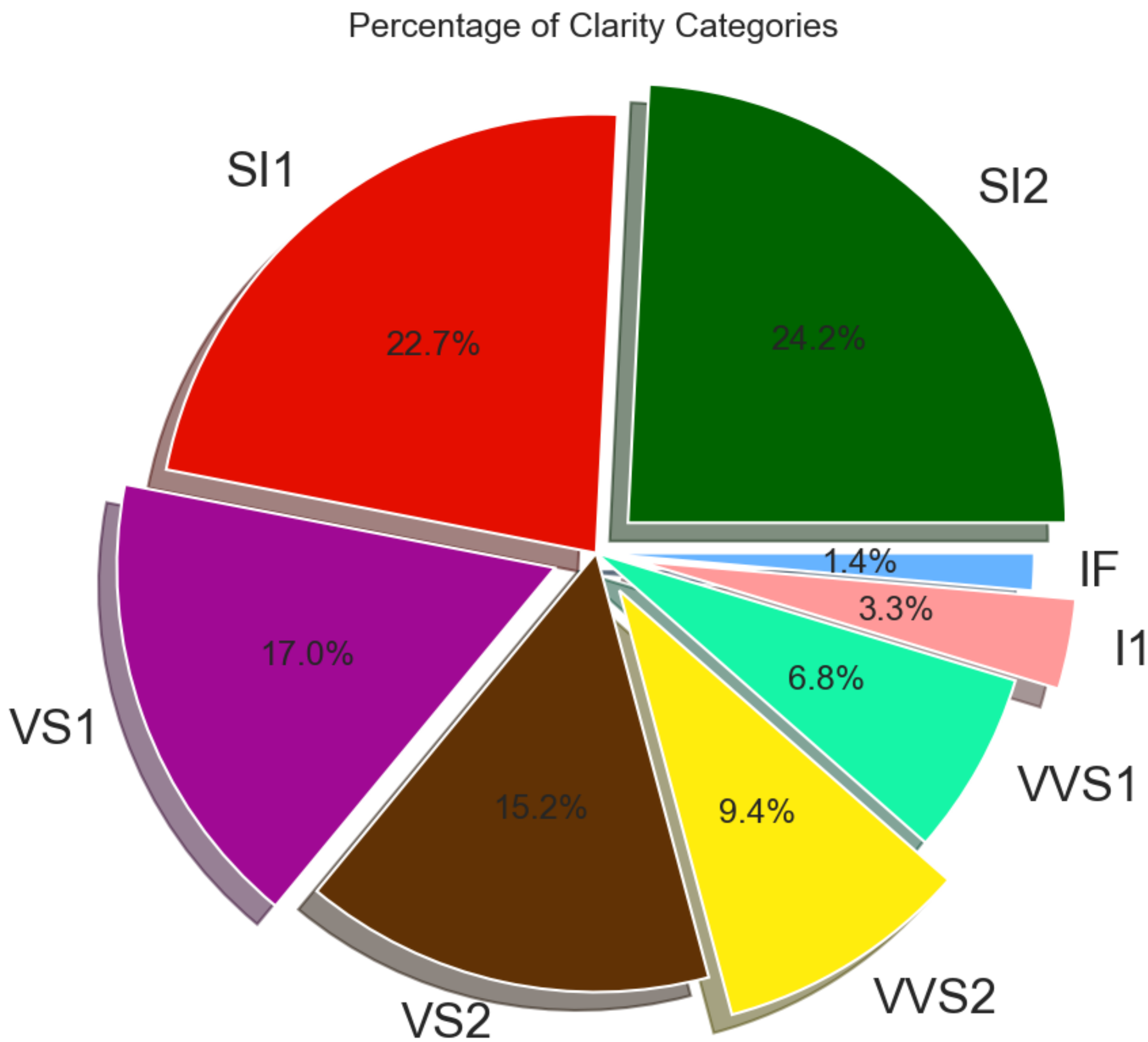
In [27]:
```
# color & price
sns.factorplot(x='color', y='price' , data=df , kind='violin', aspect=2.
5)
```

Out[27]: <seaborn.axisgrid.FacetGrid at 0x12dd29ed0>



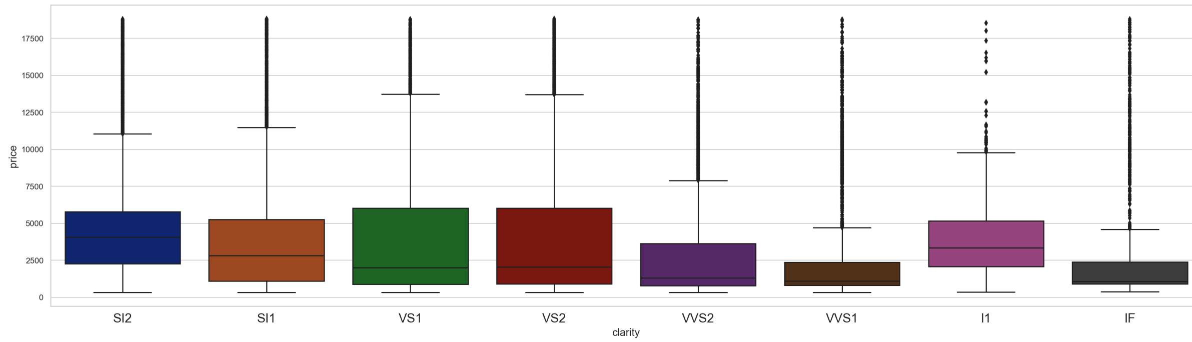
3.4) Clarity

In [29]:
```
labels = df.clarity.unique().tolist()
sizes = df.clarity.value_counts().tolist()
colors = ['#006400', '#E40E00', '#A00994', '#613205', '#FFED0D', '#16F5A
7','#ff9999','#66b3ff']
explode = (0.1, 0.0, 0.1, 0, 0.1, 0, 0.1,0)
plt.pie(sizes, explode=explode, labels=labels, colors=colors,autopct='%
1.1f%%', shadow=True, startangle=0)
plt.axis('equal')
plt.title("Percentage of Clarity Categories")
plt.plot()
fig=plt.gcf()
fig.set_size_inches(6,6)
plt.show()
```



In [30]:
```
sns.boxplot(x='clarity', y='price', data=df )
```

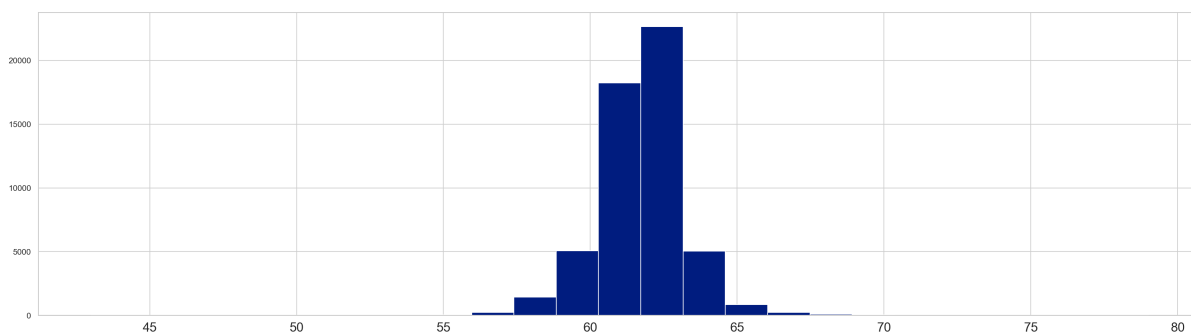
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x1365781d0>



3.5) Depth

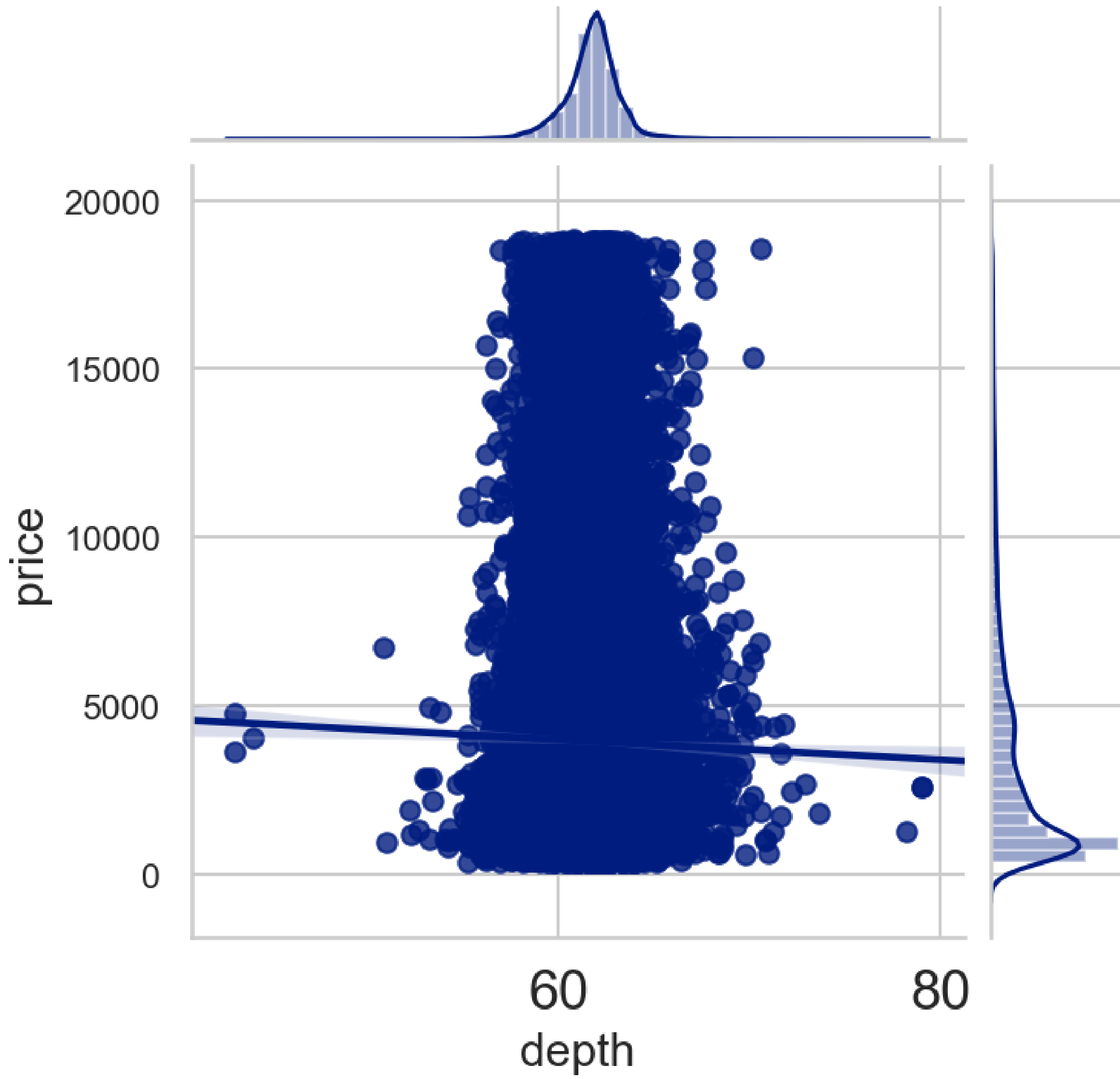
In [31]:
```
plt.hist('depth' , data=df , bins=25)
```

Out[31]:
```
(array([3.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00,
        2.0000e+00, 4.0000e+00, 1.1000e+01, 4.3000e+01, 2.1900e+02,
        1.4240e+03, 5.0730e+03, 1.8242e+04, 2.2649e+04, 5.0330e+03,
        8.5100e+02, 2.3400e+02, 8.7000e+01, 2.7000e+01, 1.1000e+01,
        3.0000e+00, 1.0000e+00, 0.0000e+00, 0.0000e+00, 3.0000e+00]),
 array([43.  , 44.44, 45.88, 47.32, 48.76, 50.2 , 51.64, 53.08, 54.52,
        55.96, 57.4 , 58.84, 60.28, 61.72, 63.16, 64.6 , 66.04, 67.48,
        68.92, 70.36, 71.8 , 73.24, 74.68, 76.12, 77.56, 79.  ]),
 <a list of 25 Patch objects>)
```



In [32]:
```
sns.jointplot(x='depth', y='price' , data=df , kind='regplot', size=5)
```

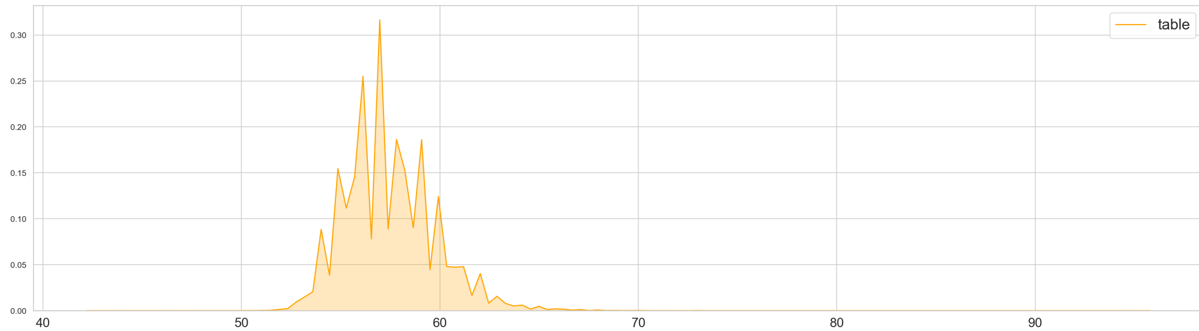
Out[32]:
```
<seaborn.axisgrid.JointGrid at 0x132b24390>
```



3.6) Table

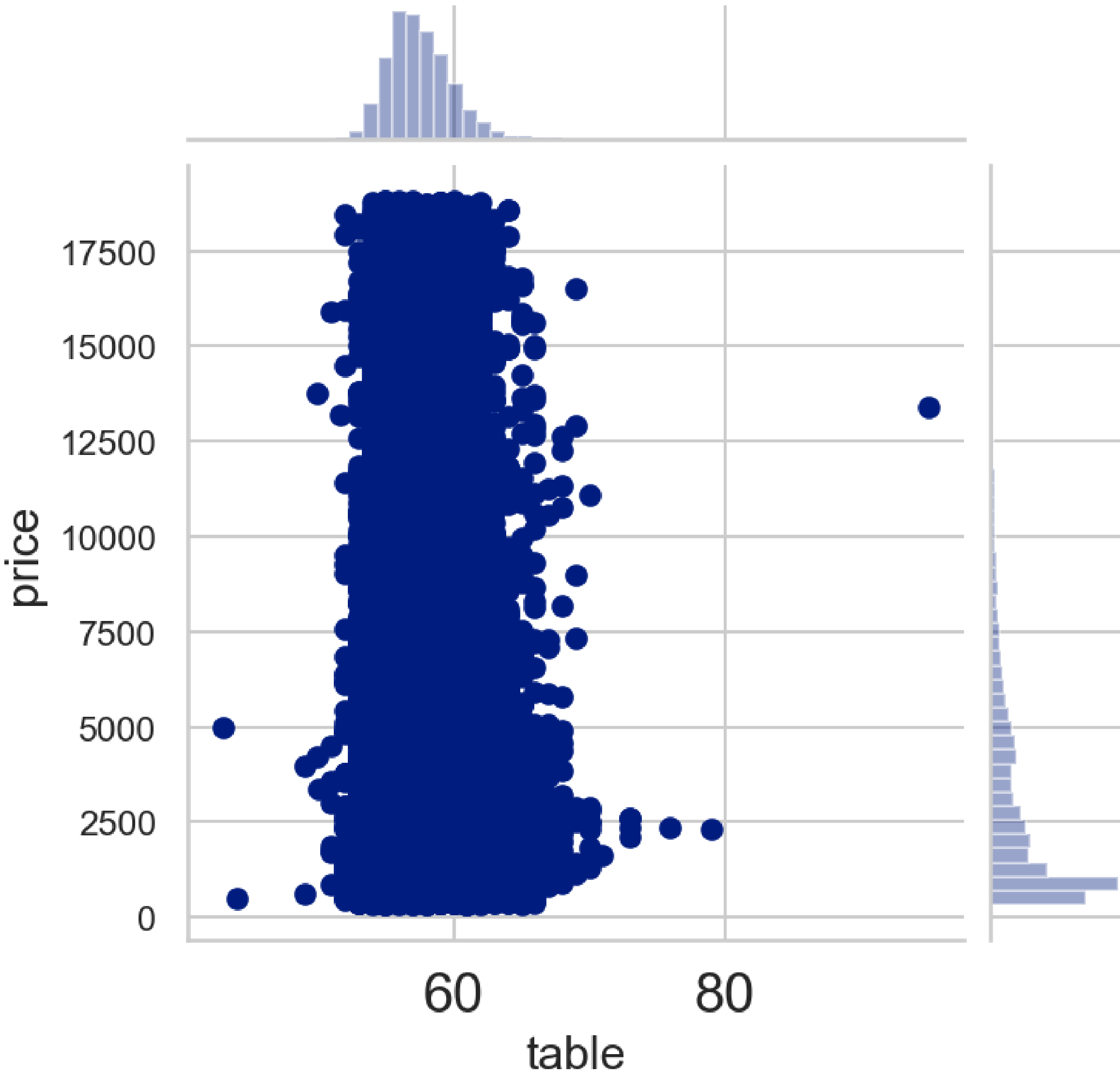
In [33]:
```
sns.kdeplot(df['table'] ,shade=True , color='orange')
```

Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x13466bad0>



In [34]:
```
sns.jointplot(x='table', y='price', data=df , size=5)
```

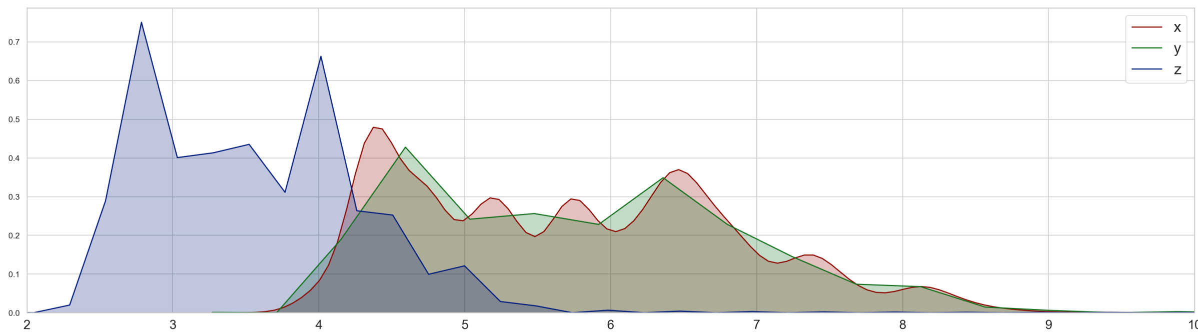
Out[34]: <seaborn.axisgrid.JointGrid at 0x135346710>



3.7) Dimensions

In [35]:
```
sns.kdeplot(df['x'] ,shade=True , color='r' )
sns.kdeplot(df['y'] , shade=True , color='g' )
sns.kdeplot(df['z'] , shade= True , color='b')
plt.xlim(2,10)
```

Out[35]: (2, 10)



4. Feature Engineering

4.1) Create New Feature 'Volume'

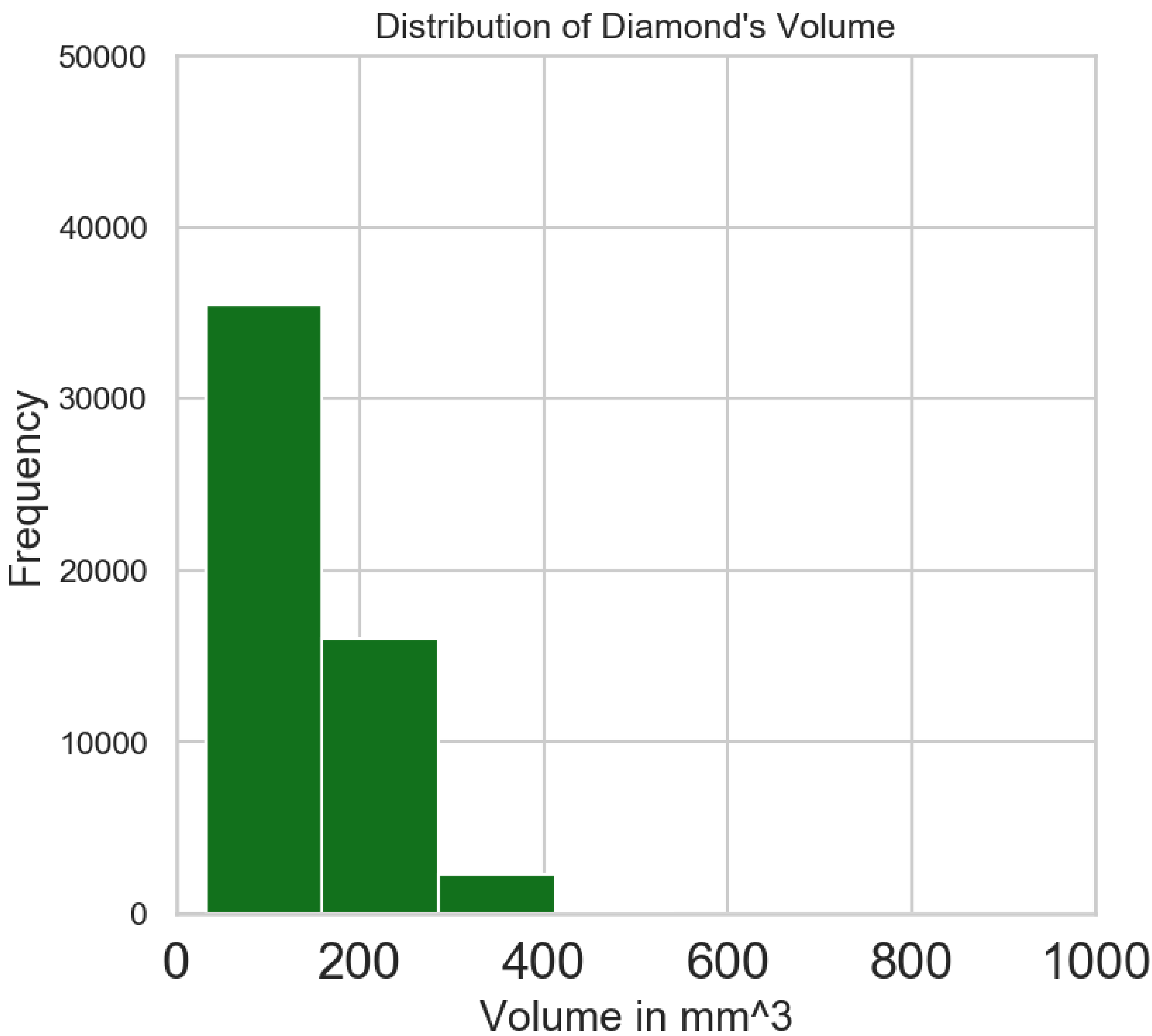
In [36]:
```
df['volume'] = df['x']*df['y']*df['z']
df.head()
```

Out[36]:

	carat	cut	color	clarity	depth	table	price	x	y	z	volume
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43	38.202030
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31	34.505856
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31	38.076885
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63	46.724580
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75	51.917250

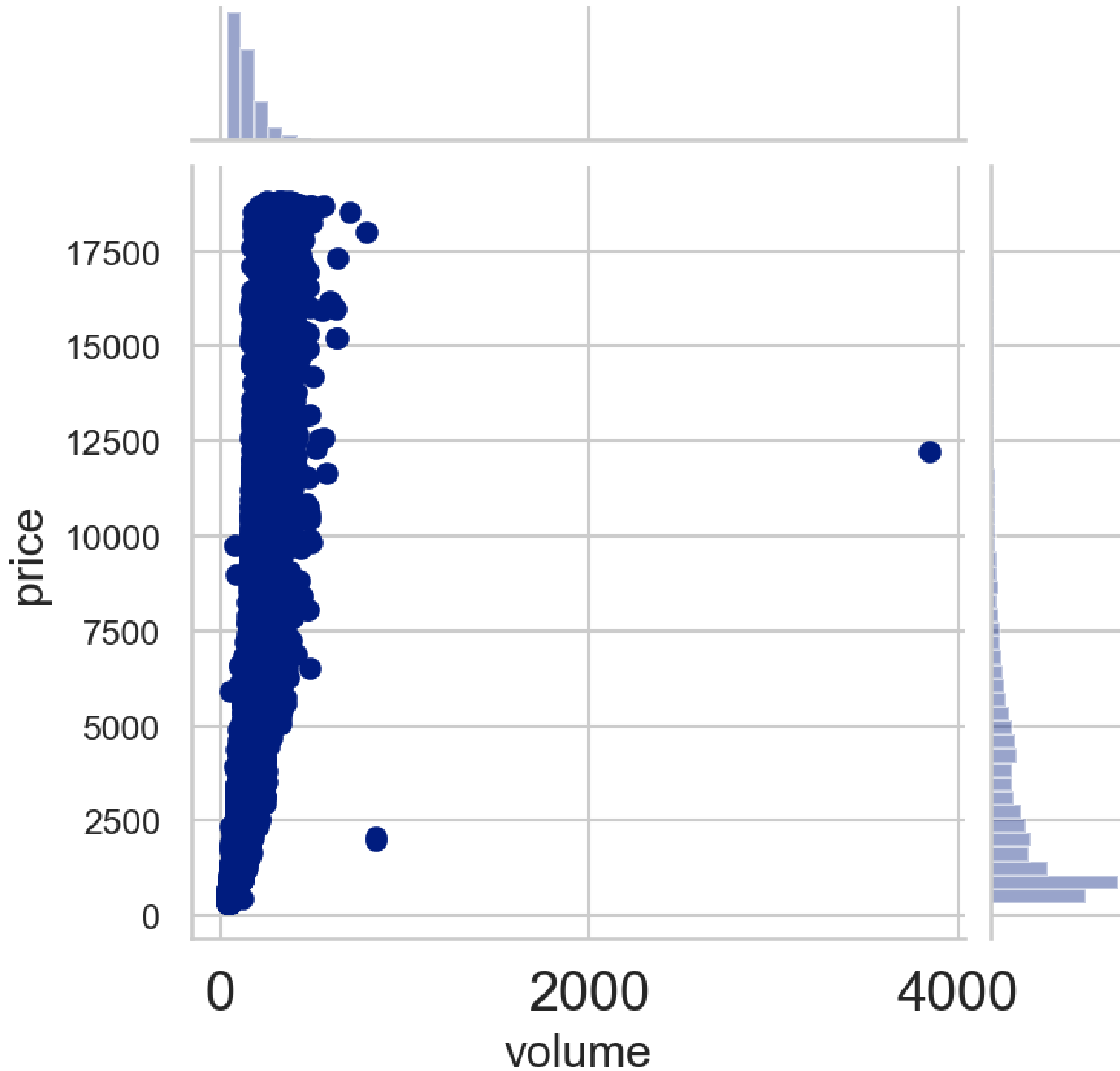
In [37]:
```
plt.figure(figsize=(5,5))
plt.hist( x=df['volume'] , bins=30 ,color='g')
plt.xlabel('Volume in mm^3')
plt.ylabel('Frequency')
plt.title('Distribution of Diamond\'s Volume')
plt.xlim(0,1000)
plt.ylim(0,50000)
```

Out[37]: (0, 50000)



In [38]:
```
sns.jointplot(x='volume', y='price' , data=df, size=5)
```

Out[38]: <seaborn.axisgrid.JointGrid at 0x138815310>



4.2) Drop X, Y, Z

In [39]:
```
df.drop(['x','y','z'], axis=1, inplace= True)
df.head()
```

Out[39]:

	carat	cut	color	clarity	depth	table	price	volume
0	0.23	Ideal	E	SI2	61.5	55.0	326	38.202030
1	0.21	Premium	E	SI1	59.8	61.0	326	34.505856
2	0.23	Good	E	VS1	56.9	65.0	327	38.076885
3	0.29	Premium	I	VS2	62.4	58.0	334	46.724580
4	0.31	Good	J	SI2	63.3	58.0	335	51.917250

5. Feature Encoding

In [45]:
```
label_cut = LabelEncoder()
label_color = LabelEncoder()
label_clarity = LabelEncoder()


df['cut'] = label_cut.fit_transform(df['cut'])
df['color'] = label_color.fit_transform(df['color'])
df['clarity'] = label_clarity.fit_transform(df['clarity'])
```

6. Feature Scaling

In [46]:
```
# Split the data into train and test.
X = df.drop(['price'], axis=1)
y = df['price']

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2, r
andom_state=66)
```

In [48]:
```
# Applying Feature Scaling ( StandardScaler )
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

7. Modelling Algorithms

In [49]:
```
# Collect all R2 Scores.
R2_Scores = []
models = ['Linear Regression' , 'Lasso Regression' , 'AdaBoost Regressio
n' , 'Ridge Regression' , 'GradientBoosting Regression',
          'RandomForest Regression' ,
         'KNeighbours Regression']
```

7.1) Linear Regression

In [50]:
```
clf_lr = LinearRegression()
clf_lr.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_lr, X = X_train, y = y_trai
n, cv = 5,verbose = 1)
y_pred = clf_lr.predict(X_test)
print('')
print('####### Linear Regression #######')
print('Score : %.4f' % clf_lr.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)

R2_Scores.append(r2)
```

```
####### Linear Regression #######
Score : 0.8814
[0.87116164 0.88350756 0.87757769 0.87635168 0.88384912]

MSE    : 1911398.80
MAE    : 926.72
RMSE   : 1382.53
R2     : 0.88

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    0.0s finished
```

7.2) Lasso Regression

In [51]:
```
clf_la = Lasso(normalize=True)
clf_la.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_la, X = X_train, y = y_trai
n, cv = 5,verbose = 1)
y_pred = clf_la.predict(X_test)
print('')
print('###### Lasso Regression ######')
print('Score : %.4f' % clf_la.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)

R2_Scores.append(r2)
```

```
###### Lasso Regression ######
Score : 0.8659
[0.84325995 0.86900907 0.86386374 0.86539938 0.86976969]

MSE    : 2162331.94
MAE    : 909.60
RMSE   : 1470.49
R2     : 0.87

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    0.1s finished
```

7.3) AdaBosst Regression

In [52]:
```
clf_ar = AdaBoostRegressor(n_estimators=1000)
clf_ar.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_ar, X = X_train, y = y_trai
n, cv = 5,verbose = 1)
y_pred = clf_ar.predict(X_test)
print('')
print('###### AdaBoost Regression ######')
print('Score : %.4f' % clf_ar.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)

R2_Scores.append(r2)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.

###### AdaBoost Regression ######
Score : 0.8548
[0.87489599 0.86588851 0.86935994 0.90093199 0.87837076]

MSE    : 2340156.64
MAE    : 1290.77
RMSE   : 1529.76
R2     : 0.85

[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    9.8s finished
```

7.4) Ridge Regression

In [53]:
```
clf_rr = Ridge(normalize=True)
clf_rr.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_rr, X = X_train, y = y_trai
n, cv = 5,verbose = 1)
y_pred = clf_rr.predict(X_test)
print('')
print('###### Ridge Regression ######')
print('Score : %.4f' % clf_rr.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)

R2_Scores.append(r2)
```

```
###### Ridge Regression ######
Score : 0.7537
[0.74232856 0.75599775 0.74753493 0.75626    0.74960313]

MSE    : 3970442.17
MAE    : 1346.18
RMSE   : 1992.60
R2     : 0.75

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    0.0s finished
```

7.5) GradientBoosting Regression

In [55]:
```
clf_gbr = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
max_depth=1, random_state=0, loss='ls',verbose = 1)
clf_gbr.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_gbr, X = X_train, y = y_tra
in, cv = 5,verbose = 1)
y_pred = clf_gbr.predict(X_test)
print('')
print('###### Gradient Boosting Regression #######')
print('Score : %.4f' % clf_gbr.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)

R2_Scores.append(r2)
```

```
      Iter       Train Loss   Remaining Time 
         1    14009477.5296            0.55s
         2    12437807.7359            0.50s
         3    11113339.5845            0.48s
         4     9945244.2308            0.47s
         5     8973416.9156            0.50s
         6     8109014.7842            0.50s
         7     7387120.0500            0.49s
         8     6753937.9878            0.49s
         9     6197182.6819            0.47s
        10     5724689.0901            0.46s
        20     3200362.4597            0.37s
        30     2393542.3170            0.31s
        40     2102586.3335            0.26s
        50     1923964.9187            0.21s
        60     1790574.6006            0.17s
        70     1688380.2826            0.13s
        80     1609829.0076            0.09s
        90     1548089.0039            0.04s
       100     1499127.4566            0.00s
      Iter       Train Loss   Remaining Time 
         1    13994442.1962            0.46s
         2    12429322.7982            0.47s
         3    11112606.0983            0.48s
         4     9944843.0686            0.49s
         5     8977395.9870            0.47s
         6     8111748.5741            0.45s
         7     7395490.7272            0.43s
         8     6765223.5285            0.42s
         9     6204866.4570            0.41s
        10     5734465.9748            0.40s
        20     3206145.1577            0.30s
        30     2394369.2846            0.25s
        40     2101114.6326            0.21s
        50     1921108.4005            0.18s
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
```

```
         60        1785959.4111            0.14s
         70        1683385.7302            0.11s
         80        1604163.5538            0.07s
         90        1542370.2912            0.04s
        100        1493476.7608            0.00s
      Iter       Train Loss      Remaining Time 
         1       14044115.9884            0.43s
         2       12472837.6750            0.41s
         3       11137657.6396            0.40s
         4        9974212.6419            0.38s
         5        8994369.5031            0.38s
         6        8133396.8459            0.37s
         7        7407925.9669            0.37s
         8        6764110.5537            0.37s
         9        6215416.1793            0.36s
        10        5736700.1166            0.35s
        20        3210108.0310            0.30s
        30        2402276.2056            0.26s
        40        2112221.2275            0.22s
        50        1934266.1687            0.18s
        60        1801087.0287            0.14s
        70        1699719.1554            0.11s
        80        1621327.8312            0.07s
        90        1559382.1164            0.04s
       100        1510393.9635            0.00s
      Iter       Train Loss      Remaining Time 
         1       14049930.2441            0.41s
         2       12464124.5936            0.45s
         3       11134339.1520            0.42s
         4        9963572.7604            0.41s
         5        8988544.3119            0.43s
         6        8123782.2835            0.43s
         7        7389901.0249            0.42s
         8        6746492.7030            0.41s
         9        6199732.4929            0.40s
        10        5719212.8946            0.40s
        20        3190875.3245            0.34s
        30        2381512.2819            0.29s
        40        2090340.2810            0.24s
        50        1911382.9450            0.19s
        60        1777779.4025            0.15s
        70        1675708.2272            0.11s
        80        1597212.1456            0.07s
        90        1535230.7915            0.04s
       100        1486232.9351            0.00s
      Iter       Train Loss      Remaining Time 
         1       13979667.1721            0.41s
         2       12410196.9258            0.39s
         3       11091464.1339            0.36s
         4        9924417.4531            0.35s
         5        8957051.8356            0.34s
         6        8090860.3178            0.33s
         7        7375141.7273            0.33s
         8        6738456.6139            0.33s
         9        6185985.1013            0.33s
        10        5710402.7142            0.32s
        20        3187460.0845            0.28s
```

```
          30     2381173.1454            0.24s
          40     2090773.7598            0.21s
          50     1911732.9770            0.18s
          60     1778590.7605            0.15s
          70     1677144.9024            0.11s
          80     1598482.5518            0.07s
          90     1537106.7445            0.04s
         100     1488486.3117            0.00s
        Iter       Train Loss   Remaining Time
           1    13978748.2331            0.45s
           2    12405054.9778            0.41s
           3    11080465.6241            0.39s
           4     9914747.0919            0.37s
           5     8945923.9930            0.37s
           6     8080995.1785            0.36s
           7     7359121.7076            0.35s
           8     6730987.4249            0.35s
           9     6173506.2064            0.35s
          10     5705021.9472            0.35s
          20     3193418.0981            0.29s
          30     2392723.0847            0.25s
          40     2103994.3744            0.21s
          50     1925922.2525            0.18s
          60     1792394.0684            0.14s
          70     1690611.3128            0.11s
          80     1611922.8661            0.07s
          90     1550358.7743            0.04s
         100     1501582.8989            0.00s

###### Gradient Boosting Regression #######
Score : 0.9058
[0.90486253 0.90678932 0.90033344 0.90344783 0.90514653]

MSE    : 1518030.06
MAE    : 720.72
RMSE   : 1232.08
R2     : 0.91

[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    1.9s finished
```

7.6) RandomForest Regression

In [56]:
```
clf_rf = RandomForestRegressor()
clf_rf.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_rf, X = X_train, y = y_trai
n, cv = 5,verbose = 1)
y_pred = clf_rf.predict(X_test)
print('')
print('###### Random Forest ######')
print('Score : %.4f' % clf_rf.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.

###### Random Forest ######
Score : 0.9802
[0.97715703 0.97904597 0.97989043 0.9758083  0.9795579 ]

MSE    : 319186.39
MAE    : 286.40
RMSE   : 564.97
R2     : 0.98

[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    4.6s finished
```

Tuning Parameters

In [57]:
```
no_of_test=[100]
params_dict={'n_estimators':no_of_test,'n_jobs':[-1],'max_features':["au
to",'sqrt','log2']}
clf_rf=GridSearchCV(estimator=RandomForestRegressor(),param_grid=params_
dict,scoring='r2')
clf_rf.fit(X_train,y_train)
print('Score : %.4f' % clf_rf.score(X_test, y_test))
pred=clf_rf.predict(X_test)
r2 = r2_score(y_test, pred)
print('R2     : %0.2f ' % r2)
R2_Scores.append(r2)
```

```
Score : 0.9820
R2     : 0.98
```

7.7) KNeighbours Regression

In [58]:
```
clf_knn = KNeighborsRegressor()
clf_knn.fit(X_train , y_train)
accuracies = cross_val_score(estimator = clf_knn, X = X_train, y = y_tra
in, cv = 5,verbose = 1)
y_pred = clf_knn.predict(X_test)
print('')
print('###### KNeighbours Regression ######')
print('Score : %.4f' % clf_knn.score(X_test, y_test))
print(accuracies)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred)**0.5
r2 = r2_score(y_test, y_pred)

print('')
print('MSE    : %0.2f ' % mse)
print('MAE    : %0.2f ' % mae)
print('RMSE   : %0.2f ' % rmse)
print('R2     : %0.2f ' % r2)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:    1.2s finished

###### KNeighbours Regression ######
Score : 0.9590
[0.95429058 0.95856983 0.95504994 0.94931403 0.95517559]

MSE    : 660416.40
MAE    : 424.98
RMSE   : 812.66
R2     : 0.96
```

Tuning Parameters

In [59]:
```
n_neighbors=[]
for i in range (0,50,5):
    if(i!=0):
        n_neighbors.append(i)
params_dict={'n_neighbors':n_neighbors,'n_jobs':[-1]}
clf_knn=GridSearchCV(estimator=KNeighborsRegressor(),param_grid=params_d
ict,scoring='r2')
clf_knn.fit(X_train,y_train)
print('Score : %.4f' % clf_knn.score(X_test, y_test))
pred=clf_knn.predict(X_test)
r2 = r2_score(y_test, pred)
print('R2     : %0.2f ' % r2)
R2_Scores.append(r2)
```

```
Score : 0.9590
R2     : 0.96
```

8. Visualizing R2-Score of Algorithms

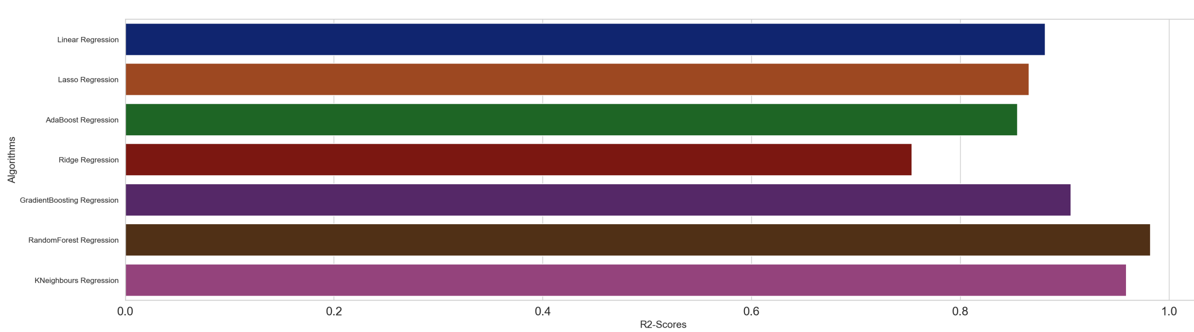
In [60]:
```
compare = pd.DataFrame({'Algorithms' : models , 'R2-Scores' : R2_Scores
})
compare.sort_values(by='R2-Scores' ,ascending=False)
```

Out[60]:

	Algorithms	R2-Scores
5	RandomForest Regression	0.982025
6	KNeighbours Regression	0.959033
4	GradientBoosting Regression	0.905833
0	Linear Regression	0.881432
1	Lasso Regression	0.865866
2	AdaBoost Regression	0.854835
3	Ridge Regression	0.753705

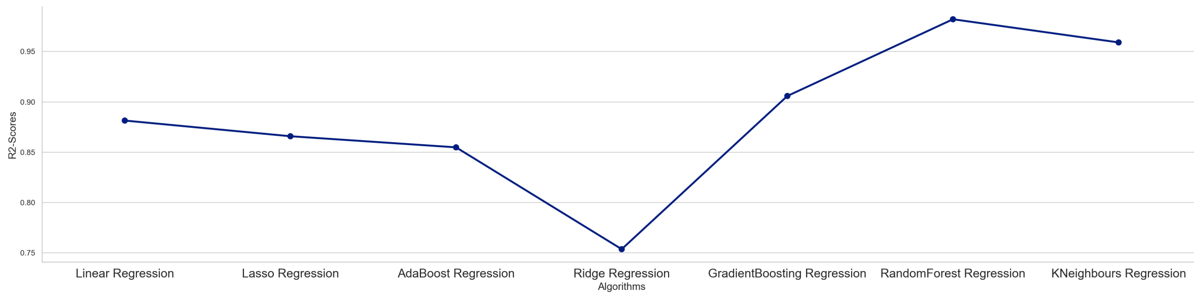
In [61]:
```
sns.barplot(x='R2-Scores' , y='Algorithms' , data=compare)
```

Out[61]: <matplotlib.axes._subplots.AxesSubplot at 0x13b94b190>



In [62]:
```
sns.factorplot(x='Algorithms', y='R2-Scores' , data=compare, size=6 , as
pect=4)
```

Out[62]: <seaborn.axisgrid.FacetGrid at 0x13ba8d750>



In [ ]: