Diamond, the most precious gem in the world

What makes a diamond more expensive than another?

Today we're gonna explore the diamond prices dataset to learn more about the factors that influence a diamond price

1. Data Preparation

In [1]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

#import os
#for dirname, _, filenames in os.walk('/kaggle/input'):
#    for filename in filenames:
#        print(os.path.join(dirname, filename))

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

In [2]:

```
df = pd.read_csv('/kaggle/input/diamonds-prices/Diamonds Prices2022.csv')
df.head()
```

Out[2]:

	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

In [3]:

```
df.drop('Unnamed: 0', axis=1, inplace=True)
```

In [4]:

```
df.cut.value_counts()
```

Out[4]:

```
Ideal        21551
Premium      13793
Very Good    12083
Good          4906
Fair          1610
Name: cut, dtype: int64
```

In [5]:

```
df.color.value_counts()
```

Out[5]:

```
G    11292
E     9799
F     9543
H     8304
D     6775
I     5422
J     2808
Name: color, dtype: int64
```

The color chart range from D to Z, D being colorless and Z being yellowish

D - F Colorless

GIA	AGS
Colorless	
D	0
E	0.5
F	1.0

G - J Near Colorless

Near Colorless	
G	1.5
H	2.0
I	2.5
J	3.0

K - M Faint Color

Faint Color	
K	3.5
L	4.0
M	4.5

N - Z Noticeable Color

Noticeable Color	
N	5.0
O	5.5
P	6.0
Q	6.5
R	7.0
S	7.5
T	8.0
U	8.5
V	9.0
W	9.5
X	10
Y	
Z	

In [6]:

```
color_map = {'D': 0, 'E': 0.5, 'F': 1, 'G': 1.5, 'H': 2, 'I': 2.5, 'J': 2}
```

In [7]:

```
df.clarity.value_counts()
```

Out[7]:

```
SI1     13067
VS2     12259
SI2      9194
VS1      8171
VVS2     5066
VVS1     3655
IF       1790
I1        741
Name: clarity, dtype: int64
```

The clarity of a diamond range from Flawless to I3.

In this dataset, it only goes from IF (internally flawless, to I1)

here we have 8 different values, we are going to replace them with numeral ranging from 1 to 8 (keeping 0 for a flawless diamond)

In [8]:

```
clarity_map = {'IF': 1, 'VVS1':2, 'VVS2': 3, 'VS1': 4, 'VS2': 5, 'SI1': 6, 'SI2'
: 7, 'I1': 8}
```

finally, depth and table are a percentage of the maximum diameter.

An excellent depth is around 60% of the diameter, and for table it between 53 and 60% of the diameter

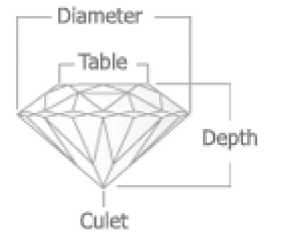


Table % = Table + Diameter
Depth % = Depth + Diameter

	Depth %	Table %
Excellent	59.0% - 61.0%	53% - 60%
Very Good	58.0% - 62.0%	61% - 62%
Good	56% - 64%	62% – 64%
Fair	64% - 70%	64% - 66%
Poor	over 70%	over 66% or under 53%

In [9]:

```
df['color'] = df['color'].map(color_map)
df['clarity'] = df['clarity'].map(clarity_map)
df.head(10)
```

Out[9]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	0.5	7	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	0.5	6	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	0.5	4	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	2.5	5	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	2.0	7	63.3	58.0	335	4.34	4.35	2.75
5	0.24	Very Good	2.0	3	62.8	57.0	336	3.94	3.96	2.48
6	0.24	Very Good	2.5	2	62.3	57.0	336	3.95	3.98	2.47
7	0.26	Very Good	2.0	6	61.9	55.0	337	4.07	4.11	2.53
8	0.22	Fair	0.5	5	65.1	61.0	337	3.87	3.78	2.49
9	0.23	Very Good	2.0	4	59.4	61.0	338	4.00	4.05	2.39

In [10]:

```
df.columns[:7]
```

Out[10]:

```
Index(['carat', 'cut', 'color', 'clarity', 'depth', 'table', 'pric
e'], dtype='object')
```

In [11]:

```
corr = df[['carat', 'cut', 'color', 'clarity', 'depth', 'table', 'price']].corr
()
corr
```

Out[11]:

	carat	color	clarity	depth	table	price
carat	1.000000	0.269395	0.352833	0.028234	0.181602	0.921591
color	0.269395	1.000000	-0.039515	0.045659	0.018440	0.166630
clarity	0.352833	-0.039515	1.000000	0.067355	0.160328	0.146791
depth	0.028234	0.045659	0.067355	1.000000	-0.295798	-0.010630
table	0.181602	0.018440	0.160328	-0.295798	1.000000	0.127118
price	0.921591	0.166630	0.146791	-0.010630	0.127118	1.000000

2. Data Visualization

In [12]:

```
sns.heatmap(corr, annot=True, linewidths=0.5)
plt.show()
```

Carat is the parameter with the greater influence on the price.

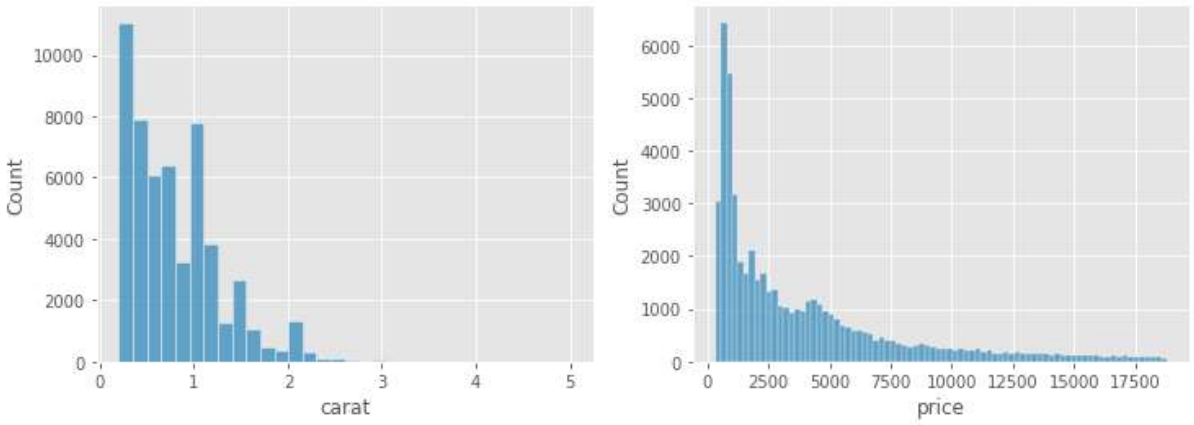
The correlation between the carat and the price is 0.92

In [13]:

```
plt.style.use('ggplot')
```

In [14]:

```
fig, ax = plt.subplots(1,2, figsize=(12,4))

sns.histplot(ax=ax[0], data=df.carat, bins=32)

sns.histplot(ax=ax[1], data=df.price)

plt.show()
```

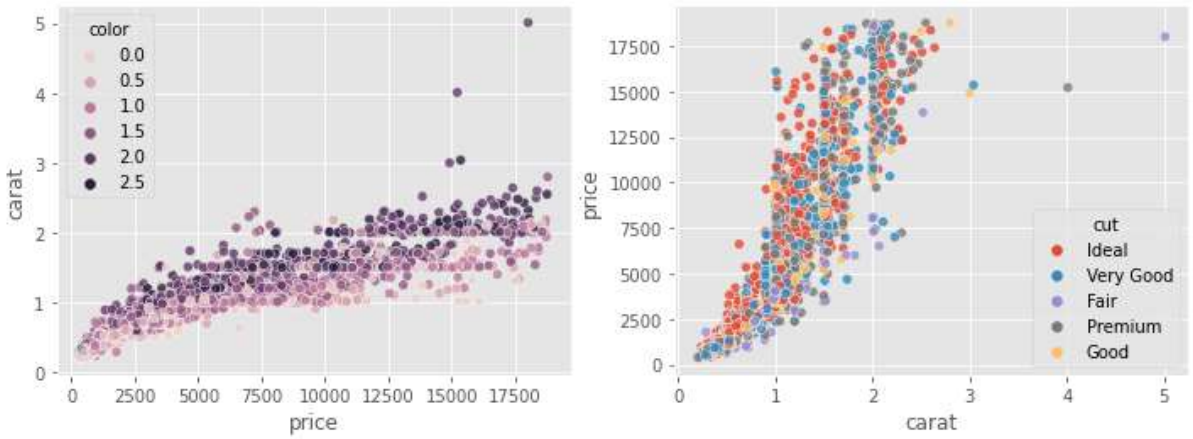


In [15]:

```
sample = df.sample(frac=0.1, random_state=42)
```

In [16]:

```
fig, ax = plt.subplots(1,2, figsize=(12,4))

sns.scatterplot(ax=ax[0], data=sample, x='price', y='carat', hue='color', alpha=
0.8)

sns.scatterplot(ax=ax[1], data=sample, x='carat', y='price', hue='cut', alpha=0.
8)

plt.show()
```



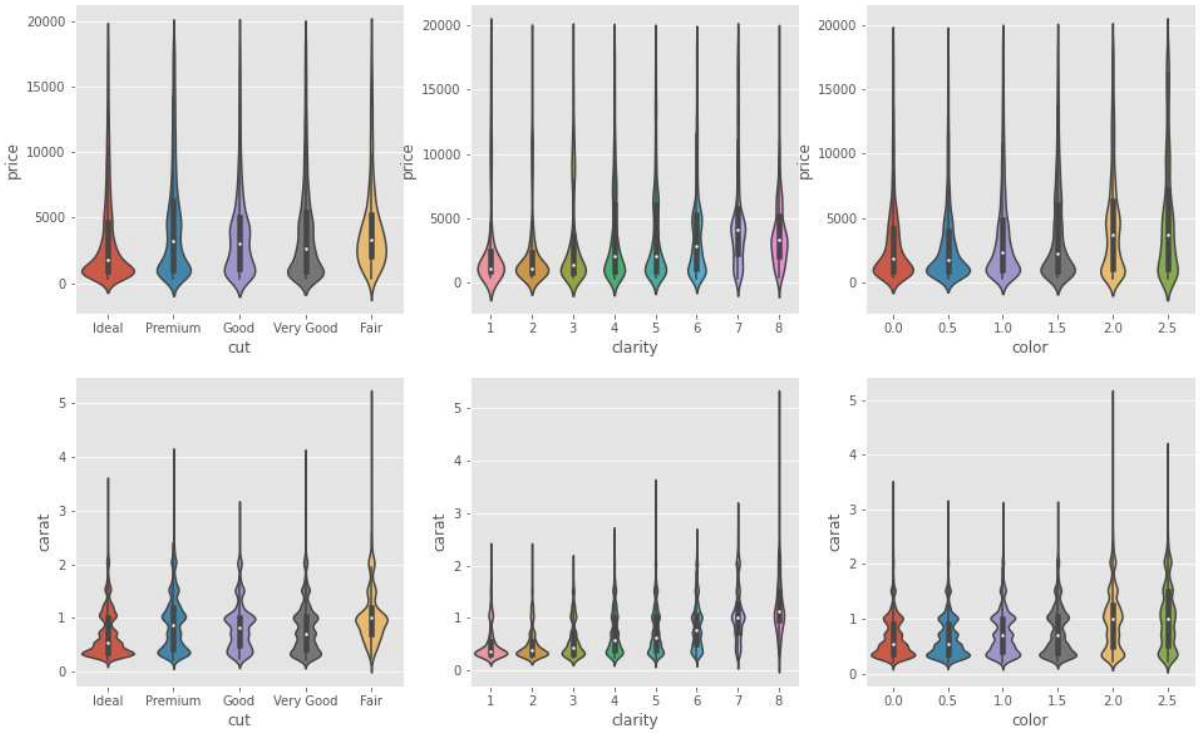
Because of the larger number of data, the plot was a bit blurry when using the full dataframe. We extract a random 10% sample of it using DataFrame.random()

The price to carat relationship is not linear. Above 1 carat, the prices start to skyrocket

Colored diamonds are also more valuable than their colorless counterparts

In [17]:

```
fig, ax = plt.subplots(2,3, figsize=(16,10))

sns.violinplot(ax=ax[0,0], data=df, y='price', x='cut')

sns.violinplot(ax=ax[0,1], data=df, y='price', x='clarity')

sns.violinplot(ax=ax[0,2], data=df, y='price', x='color')

sns.violinplot(ax=ax[1,0], data=df, y='carat', x='cut')

sns.violinplot(ax=ax[1,1], data=df, y='carat', x='clarity')

sns.violinplot(ax=ax[1,2], data=df, y='carat', x='color')

plt.show()
```

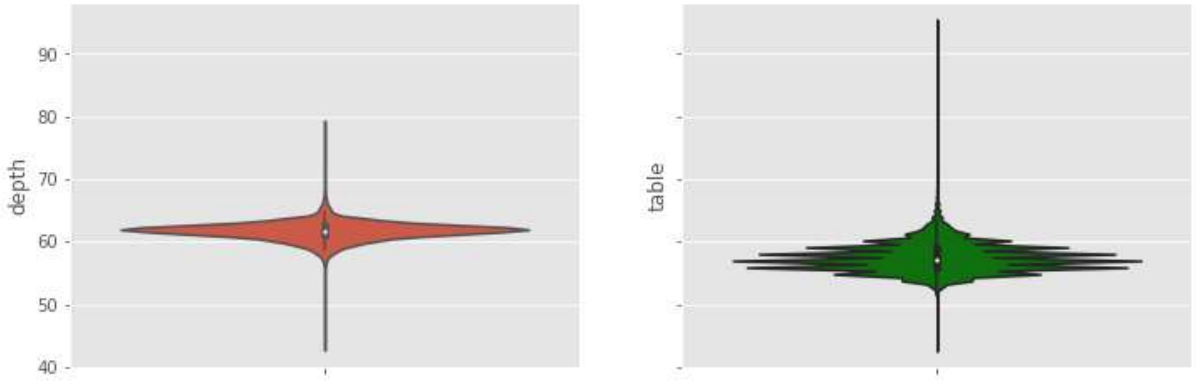


The diamonds with the best quality often come with a smaller size.

But because carat is the parameter that influence the price the most, we have lower quality diamonds at a higher price than high quality ones.

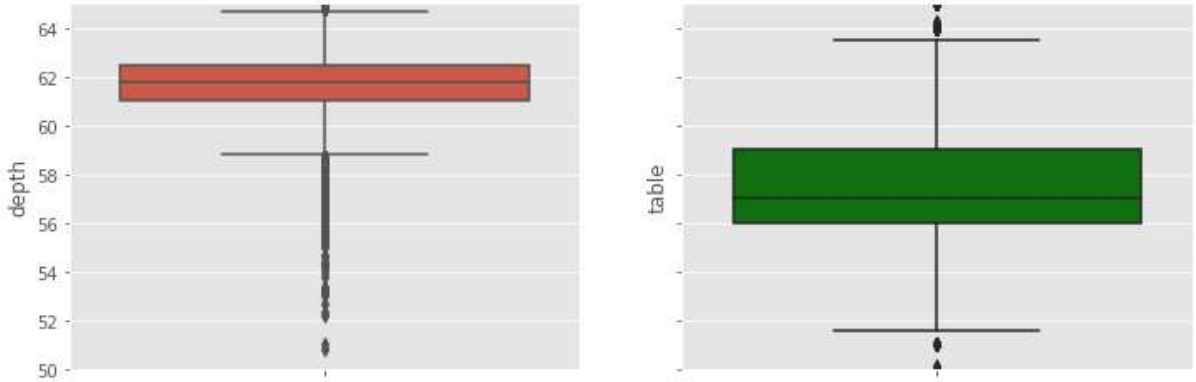
In [18]:

```
fig, ax = plt.subplots(1,2, figsize=(12,4), sharey=True)

sns.violinplot(ax=ax[0], data=df, y='depth')

sns.violinplot(ax=ax[1], data=df, y='table', color='green')

plt.show()
```



In [19]:

```
fig, ax = plt.subplots(1,2, figsize=(12,4), sharey=True)

sns.boxplot(ax=ax[0], data=df, y='depth')

sns.boxplot(ax=ax[1], data=df, y='table', color='green')


plt.ylim([50, 65])
plt.show()
```



In the diamond prices EDA we learn about that factors that gives diamonds their value:

- bigger diamond are much more expensive than smaller
- a nice color
- a fine cut
- diamond clarity

3. Machine Learning Model Comparison

For this dataset, we are going to compare three machine learning models:

- Linear Regression
- Decision Tree
- XGBoost

In [20]:

```
cut_map = {'Ideal': 5, 'Premium': 4, 'Very Good': 3, 'Good': 2, 'Fair': 1}
df['cut'] = df.cut.map(cut_map)
df.head()
```

Out[20]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	5	0.5	7	61.5	55.0	326	3.95	3.98	2.43
1	0.21	4	0.5	6	59.8	61.0	326	3.89	3.84	2.31
2	0.23	2	0.5	4	56.9	65.0	327	4.05	4.07	2.31
3	0.29	4	2.5	5	62.4	58.0	334	4.20	4.23	2.63
4	0.31	2	2.0	7	63.3	58.0	335	4.34	4.35	2.75

In [21]:

```
y = df['price']
X = df[['carat', 'cut', 'color', 'clarity', 'depth', 'table']]
```

Importing the three models we are going to try

In [22]:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_absolute_percentage_error as mape
```

In [23]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random
_state=42)
print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
```

```
(43154, 6) (10789, 6)
(43154,) (10789,)
```

In [24]:

```
model = LinearRegression()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
mae_lr = mean_absolute_error(y_test, y_pred)
mape_lr = mape(y_test, y_pred)

print(f'the mean absolute error for linear regression is {round(mae_lr, 2)}')
print(f'the mean absolute percentage error for linear regression is {round(mape_
lr, 2)}')
```

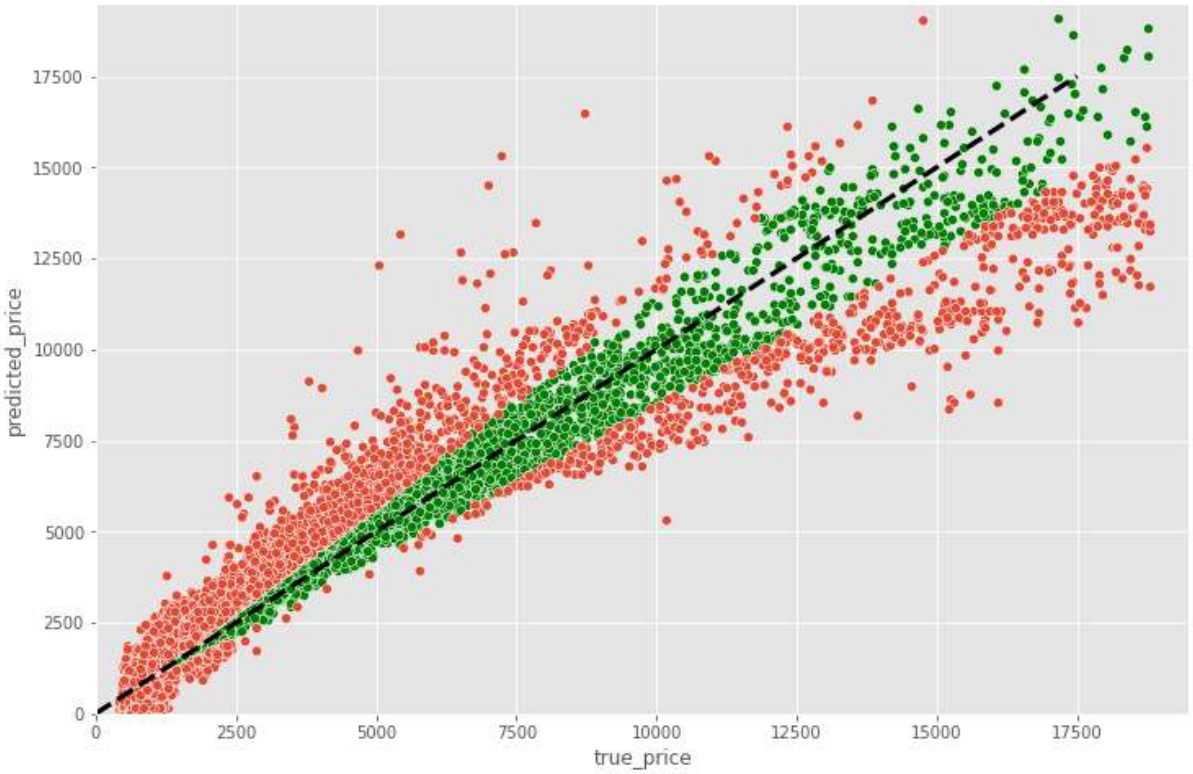
```
the mean absolute error for linear regression is 853.63
the mean absolute percentage error for linear regression is 0.48
```

In [25]:

```
graph = X_test.copy()
graph['true_price'] = y_test.iloc[:]
graph['predicted_price'] = y_pred
graph['percent_error'] = np.abs(graph['true_price'] - graph['predicted_price'])
* 100 / graph['true_price']
```

In [26]:

```
plt.figure(figsize=(12,8))

sns.scatterplot(data=graph[graph.percent_error <= 15], x='true_price', y='predicted_price', color='green')
sns.scatterplot(data=graph[graph.percent_error > 15], x='true_price', y='predicted_price')
sns.lineplot(x=[0,17500], y=[0,17500], color='black', linestyle='--', linewidth=3)
plt.xlim([0, 19500])
plt.ylim([0, 19500])
plt.show()
```



In [27]:

```
from sklearn.tree import DecisionTreeRegressor

model = DecisionTreeRegressor()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
mae_tree = mean_absolute_error(y_test, y_pred)
mape_tree = mape(y_test, y_pred)

print(f'the mean absolute error for decision tree is {round(mae_tree, 2)}')
print(f'the mean absolute percent error for decision tree is {round(mape_tree, 2
)}')
```

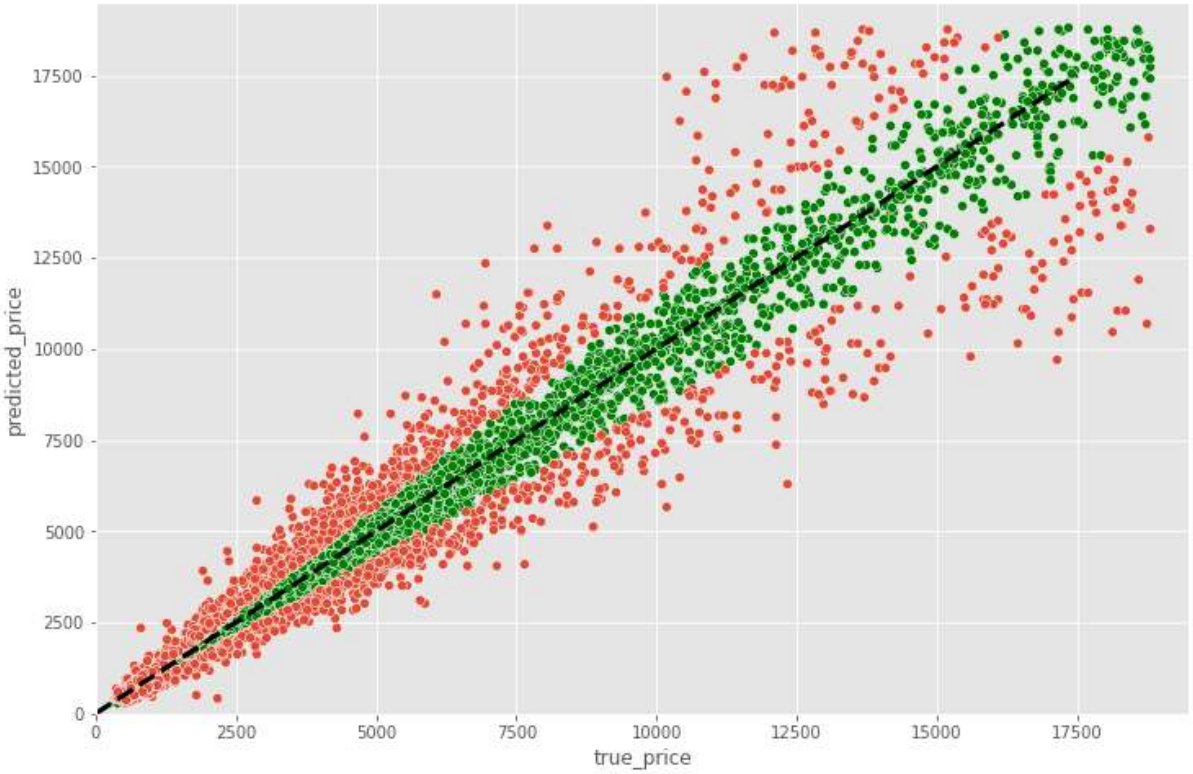
```
the mean absolute error for decision tree is 401.28
the mean absolute percent error for decision tree is 0.11
```

In [28]:

```
tree = X_test.copy()
tree['true_price'] = y_test.iloc[:]
tree['predicted_price'] = y_pred
tree['percent_error'] = np.abs(tree.true_price - tree.predicted_price) * 100 / t
ree.true_price
```

In [29]:

```
plt.figure(figsize=(12,8))

sns.scatterplot(data=tree[tree.percent_error <= 15], x='true_price', y='predicted_price', color='green')
sns.scatterplot(data=tree[tree.percent_error > 15], x='true_price', y='predicted_price')
sns.lineplot(x=[0,17500], y=[0,17500], color='black', linestyle='--', linewidth=3)
plt.xlim([0, 19500])
plt.ylim([0, 19500])
plt.show()
```



In [30]:

```
from xgboost.sklearn import XGBRegressor

model = XGBRegressor()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
mae_xgb = mean_absolute_error(y_test, y_pred)
mape_xgb = mape(y_test, y_pred)

print(f'the mean absolute error for XGB regressor is {round(mae_xgb, 2)}')
print(f'the mean absolute percent error for XGB regressor is {round(mape_xgb, 2)
}')
```

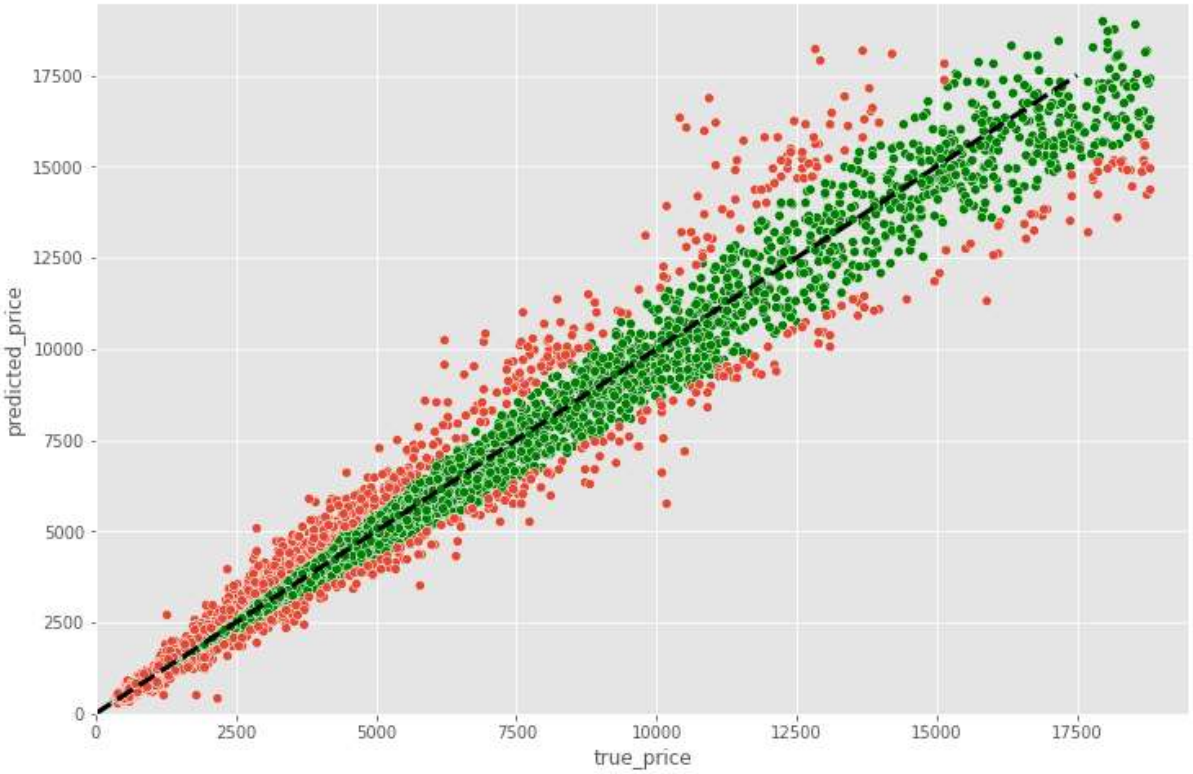
```
the mean absolute error for XGB regressor is 321.37
the mean absolute percent error for XGB regressor is 0.09
```

In [31]:

```
xgb = X_test.copy()
xgb['true_price'] = y_test.iloc[:]
xgb['predicted_price'] = y_pred
xgb['percent_error'] = np.abs(xgb.true_price - xgb.predicted_price) * 100 / xgb.
true_price
```

In [32]:

```
plt.figure(figsize=(12,8))

sns.scatterplot(data=xgb[xgb.percent_error <=15], x='true_price', y='predicted_price', color='green')
sns.scatterplot(data=xgb[xgb.percent_error > 15], x='true_price', y='predicted_price')
sns.lineplot(x=[0,17500], y=[0,17500], color='black', linestyle='--', linewidth=3)
plt.xlim([0, 19500])
plt.ylim([0, 19500])
plt.show()
```



In [33]:

```
error = {'Linear Regression': [mae_lr, mape_lr],
         'Decision Tree': [mae_tree, mape_tree],
         'XGBoost': [mae_xgb, mape_xgb]}
error = pd.DataFrame.from_dict(error, orient='index', columns=['Mean Absolute Er
ror', 'Mean Percentage Error'])
error['Mean Percentage Error'] = error['Mean Percentage Error'] * 100
error.style.background_gradient(subset=['Mean Absolute Error', 'Mean Percentage
 Error'], cmap='Reds')
```

Out[33]:

	Mean Absolute Error	Mean Percentage Error
Linear Regression	853.625279	47.973371
Decision Tree	401.278193	11.096070
XGBoost	321.366196	8.661559

Summary

- XGB boost had the best score but it took a lot of time to train
- Decision Tree had a good score
- Linear Regression doesn't work well for this problem because it cannot take into consideration the exponential increase in price