In [54]:
```
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import RobustScaler
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import SGDRegressor, Lasso, Ridge
from sklearn.svm import SVR
from catboost import CatBoostRegressor
from xgboost import XGBRegressor
from sklearn.model_selection import train_test_split, learning_curve, RandomizedSearch
from sklearn.metrics import r2_score,make_scorer,mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
from lightgbm import LGBMRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
```

In [28]:
```
# Load the dataset
file_path = 'diamonds.csv'
diamonds_df = pd.read_csv(file_path)

# Display the first few rows of the dataset for an overview
diamonds_df.info()
diamonds_df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 219703 entries, 0 to 219702
Data columns (total 26 columns):
 #   Column                        Non-Null Count   Dtype  
---  ------                        --------------   -----  
 0   Unnamed: 0                    219703 non-null  int64  
 1   cut                           219703 non-null  object 
 2   color                         219703 non-null  object 
 3   clarity                       219703 non-null  object 
 4   carat_weight                  219703 non-null  float64
 5   cut_quality                   219703 non-null  object 
 6   lab                           219703 non-null  object 
 7   symmetry                      219703 non-null  object 
 8   polish                        219703 non-null  object 
 9   eye_clean                     219703 non-null  object 
 10  culet_size                    219703 non-null  object 
 11  culet_condition               219703 non-null  object 
 12  depth_percent                 219703 non-null  float64
 13  table_percent                 219703 non-null  float64
 14  meas_length                   219703 non-null  float64
 15  meas_width                    219703 non-null  float64
 16  meas_depth                    219703 non-null  float64
 17  girdle_min                    219703 non-null  object 
 18  girdle_max                    219703 non-null  object 
 19  fluor_color                   219703 non-null  object 
 20  fluor_intensity               219703 non-null  object 
 21  fancy_color_dominant_color    219703 non-null  object 
 22  fancy_color_secondary_color   219703 non-null  object 
 23  fancy_color_overtone          219703 non-null  object 
 24  fancy_color_intensity         219703 non-null  object 
 25  total_sales_price             219703 non-null  int64  
dtypes: float64(6), int64(2), object(18)
memory usage: 43.6+ MB
```

Out[28]:

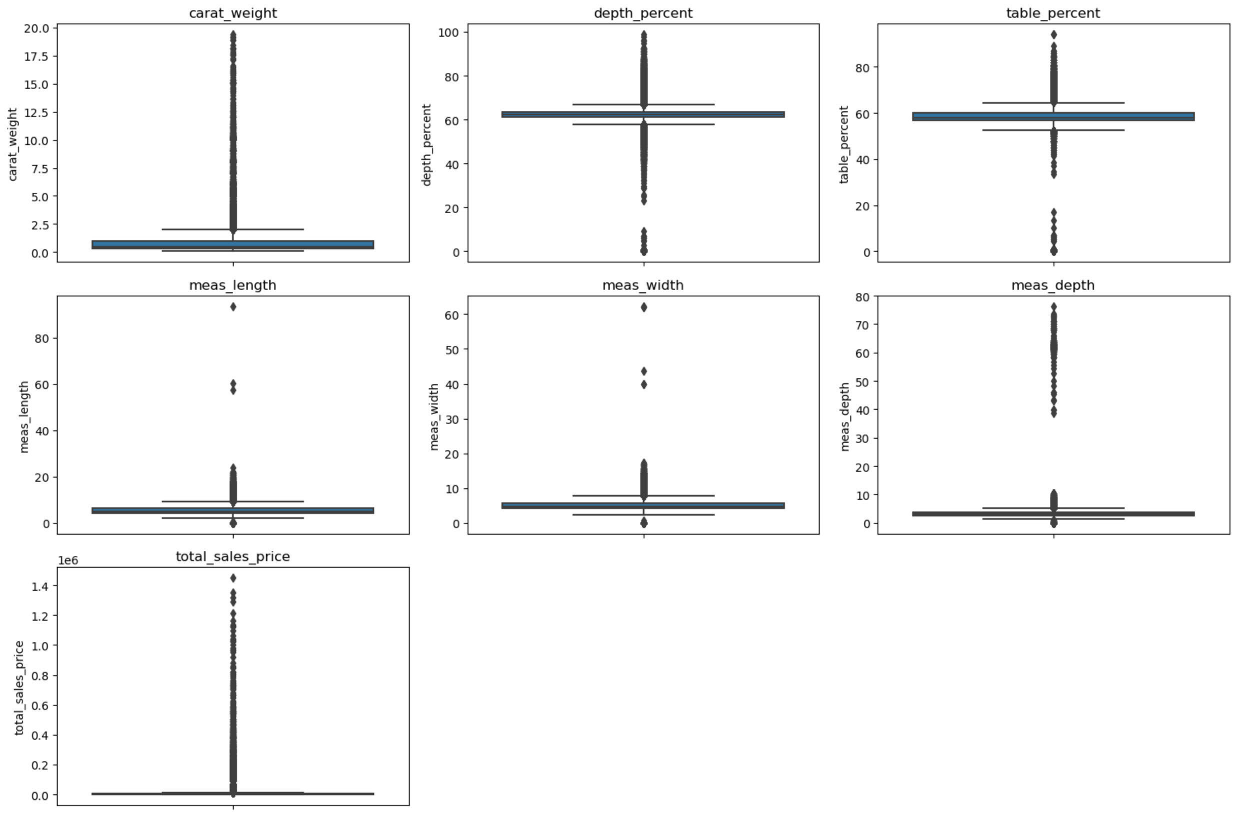
	Unnamed: 0	cut	color	clarity	carat_weight	cut_quality	lab	symmetry	polish	eye_clean	...
0	0	Round	E	VVS2	0.09	Excellent	IGI	Very Good	Very Good	unknown	...
1	1	Round	E	VVS2	0.09	Very Good	IGI	Very Good	Very Good	unknown	...
2	2	Round	E	VVS2	0.09	Excellent	IGI	Very Good	Very Good	unknown	...
3	3	Round	E	VVS2	0.09	Excellent	IGI	Very Good	Very Good	unknown	...
4	4	Round	E	VVS2	0.09	Very Good	IGI	Very Good	Excellent	unknown	...

5 rows × 26 columns

In [ ]:

Outlier Analysis

In [29]:
```
# Plotting boxplots for each numerical column to visually identify outliers
plt.figure(figsize=(15, 10))
for i, col in enumerate(numerical_cols):
    plt.subplot(3, 3, i+1)
    sns.boxplot(y=diamonds_df[col])
    plt.title(col)
plt.tight_layout()
plt.show()
```



The boxplots for each numerical column provide a visual representation of potential outliers. Here's what we can observe:

Carat Weight: There are visible outliers, with some diamonds having significantly higher carat weights than the majority. Depth Percent: This column also shows outliers, particularly on the higher end. Table Percent: There are outliers present on both lower and higher ends. Measurements (Length, Width, Depth): Each of these columns has outliers, especially on the higher end, indicating some diamonds are much larger in size than most. Total Sales Price: There are significant outliers, with some diamonds having exceptionally high sales prices.

In [30]:
```
def iqr_outliers(data):
    Q1 = data.quantile(0.25)
    Q3 = data.quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return ((data < lower_bound) | (data > upper_bound))

iqr_outliers_df = diamonds_df[numerical_cols].apply(iqr_outliers)

# Function to calculate Median Absolute Deviation (MAD)-based outliers
```

```
def mad_outliers(data):
    median = np.median(data)
    mad = np.median(np.abs(data - median))
    mad_based_z_score = 0.6745 * (data - median) / mad
    return np.abs(mad_based_z_score) > 3


mad_outliers_df = diamonds_df[numerical_cols].apply(mad_outliers)

# Count of outliers using IQR and MAD
iqr_outlier_counts = iqr_outliers_df.sum()
mad_outlier_counts = mad_outliers_df.sum()

iqr_outlier_counts, mad_outlier_counts
```

Out[30]:

```
(carat_weight          9447
 depth_percent        34801
 table_percent        26721
 meas_length           9882
 meas_width           11456
 meas_depth            9115
 total_sales_price    27330
 dtype: int64,
 carat_weight         27148
 depth_percent        33391
 table_percent        20714
 meas_length          15768
 meas_width           16361
 meas_depth           14302
 total_sales_price    42474
 dtype: int64)
```

Observations: Both methods identify a significant number of outliers across the dataset, although the counts vary between methods. The IQR method, which is based on quartiles, tends to be less sensitive than the MAD method, which is evident from the generally lower counts of outliers. The MAD method, which is less influenced by extreme values, identifies a larger number of outliers in most columns. This could be due to the presence of extreme values or heavy tails in the data distribution. Depth Percent, Table Percent, and Total Sales Price show a high number of outliers in both methods, indicating these attributes might have a wide range of values or distributions that deviate from normality. Now, let's use boxplots and domain-specific criteria to further investigate these outliers. For the domain-specific analysis, we can focus on certain thresholds or unusual combinations of attributes that might be considered rare or atypical in the diamond industry

In [31]:

```
# Domain-specific analysis:
# For instance, we can consider extremely high carat weights or very low/high depth an
high_carat_threshold = 2.5 # Carat weight above 2.5 is relatively rare
low_high_depth_threshold = (55, 70) # Typical depth percent range
low_high_table_threshold = (53, 65) # Typical table percent range

# Identifying domain-specific outliers
domain_outliers = diamonds_df[(diamonds_df['carat_weight'] > high_carat_threshold) |
                              (diamonds_df['depth_percent'] < low_high_depth_threshold
                              (diamonds_df['depth_percent'] > low_high_depth_threshold
                              (diamonds_df['table_percent'] < low_high_table_threshold
                              (diamonds_df['table_percent'] > low_high_table_threshold
```

```
# Count of domain-specific outliers
domain_outlier_count = domain_outliers.shape[0]
domain_outlier_count
```

Out[31]:

```
35130
```

Domain-Specific Outlier Analysis I identified domain-specific outliers based on certain industry thresholds: Carat Weight: Outliers were considered for weights above 2.5 carats, as such large diamonds are relatively rare. Depth Percent: Typical depth percent ranges between 55% and 70%. Values outside this range were considered outliers. Table Percent: A typical range is between 53% and 65%. Values outside this were also flagged as outliers.

ref : https://www.gemsociety.org/article/diamond-carat-weight/

ref : https://beyond4cs.com/grading/depth-and-table-values/

ref : https://beyond4cs.com/grading/depth-and-table-values/#:~:text=For%20round%20cut%20diamonds%2C%20I,53%E2%80%A0here%E3%80%91

Exploratory Data Analysis

In [35]:

```
# Descriptive Statistics for numerical features
descriptive_stats = diamonds_df.describe()
descriptive_stats
```

Out[35]:

	Unnamed: 0	carat_weight	depth_percent	table_percent	meas_length	meas_width
count	219703.000000	219703.000000	219703.000000	219703.000000	219703.000000	219703.000000
mean	109851.747418	0.755176	61.683768	57.747585	5.548853	5.135626
std	63423.264419	0.845894	9.915266	9.959928	1.763924	1.374529
min	0.000000	0.080000	0.000000	0.000000	0.000000	0.000000
25%	54925.500000	0.310000	61.200000	57.000000	4.350000	4.310000
50%	109852.000000	0.500000	62.400000	58.000000	5.060000	4.800000
75%	164777.500000	1.000000	63.500000	60.000000	6.350000	5.700000
max	219703.000000	19.350000	98.700000	94.000000	93.660000	62.300000

In [36]:

```
# Distribution of Categorical Features
categorical_features = diamonds_df.select_dtypes(include=['object']).columns
categorical_distribution = diamonds_df[categorical_features].describe()

categorical_distribution
```

Out[36]:

	cut	color	clarity	cut_quality	lab	symmetry	polish	eye_clean	culet_size	culet
count	219703	219703	219703	219703	219703	219703	219703	219703	219703	
unique	11	11	11	6	3	5	5	5	9	
top	Round	E	SI1	Excellent	GIA	Excellent	Excellent	unknown	N	
freq	158316	33103	38627	124861	200434	131619	175806	156916	131899	

In [37]:

```
#Skewness
numeric_columns = diamonds_df.select_dtypes(include=['float64', 'int64']).columns
numeric_skewness = diamonds_df[numeric_columns].skew()

numeric_skewness
```

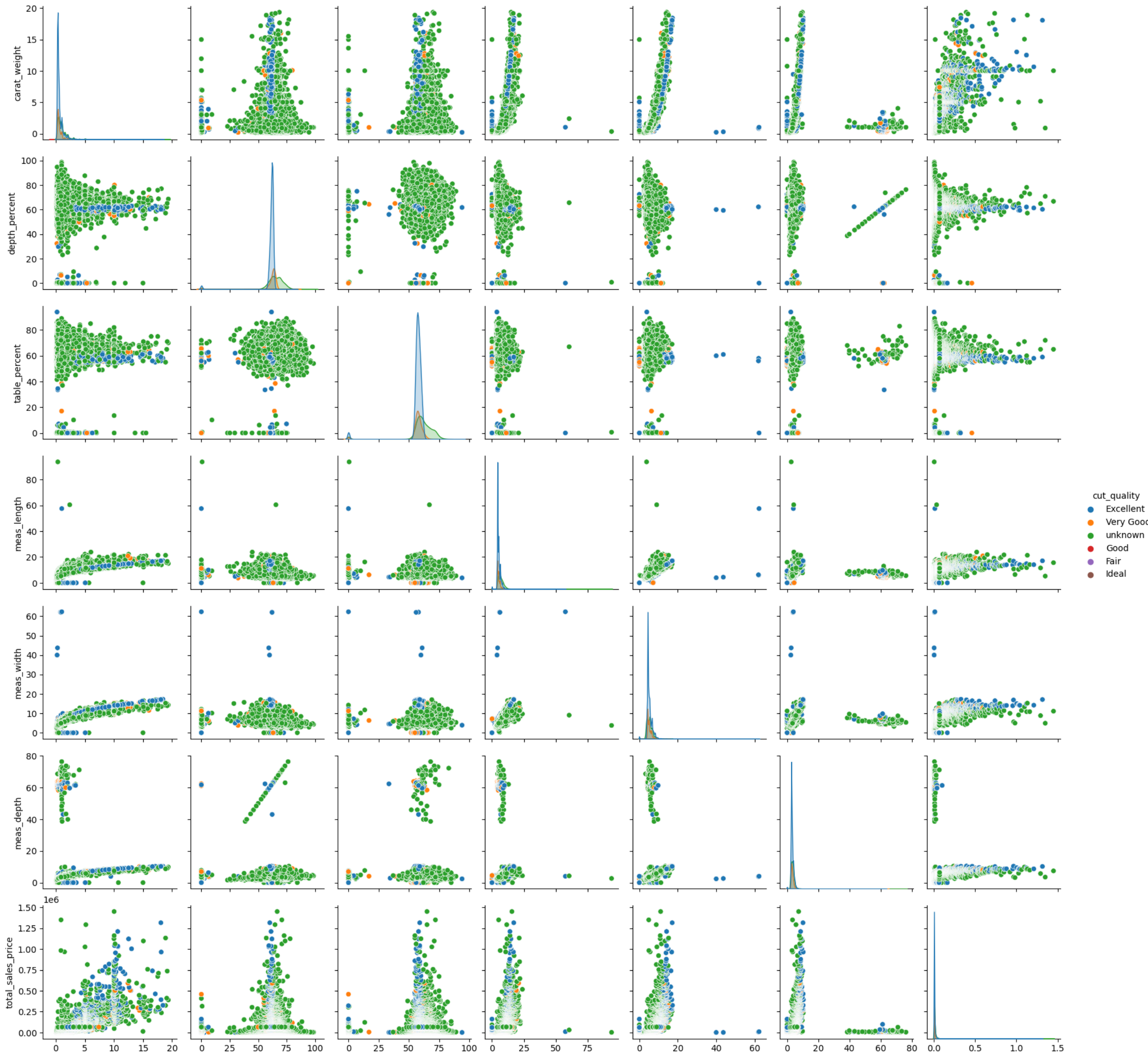
Out[37]:

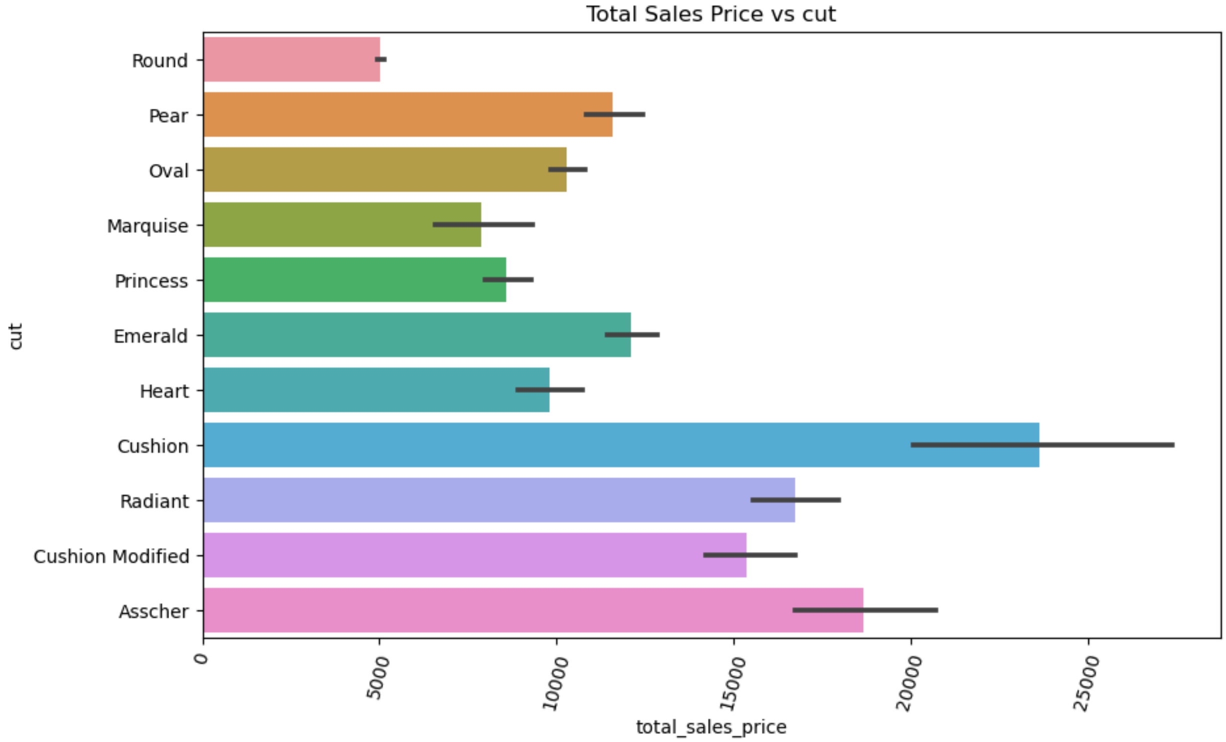
```
Unnamed: 0          -0.000009
carat_weight         6.044752
depth_percent       -5.133846
table_percent       -4.537950
meas_length          2.295008
meas_width           2.269753
meas_depth          24.153615
total_sales_price   19.409831
dtype: float64
```

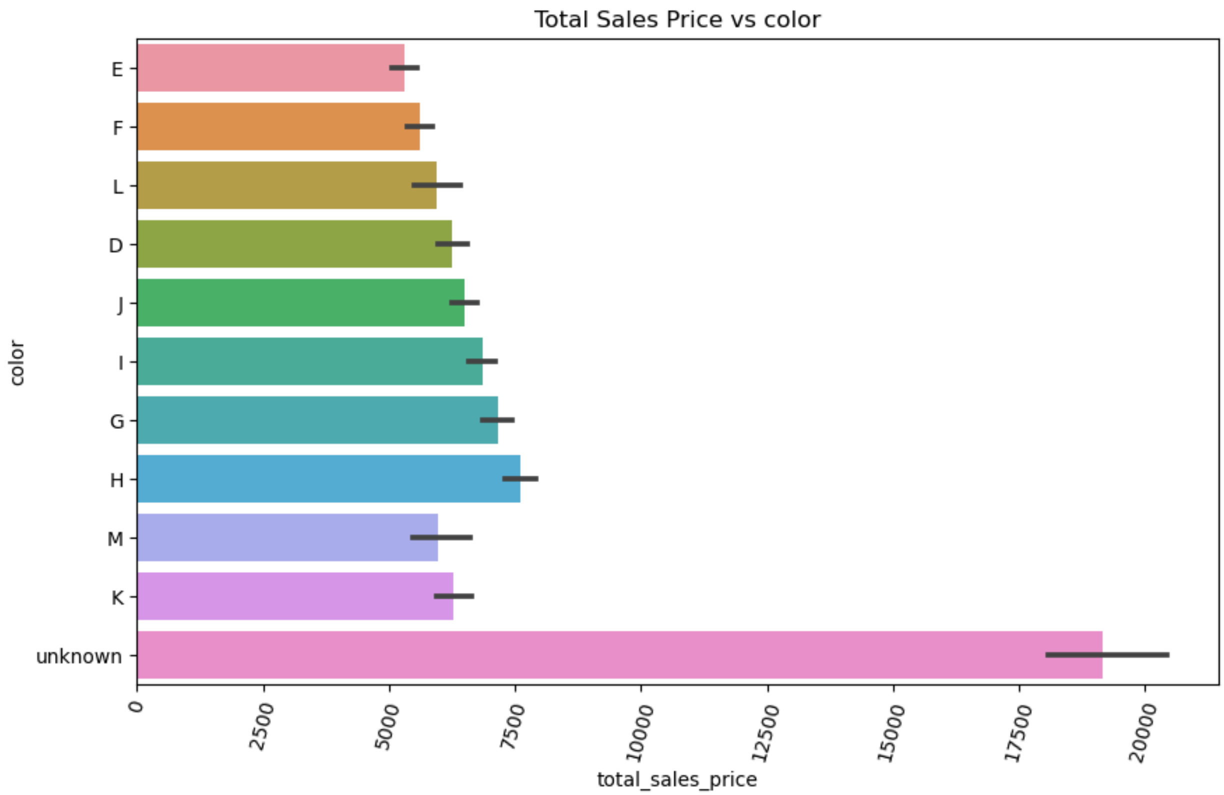
In [38]:

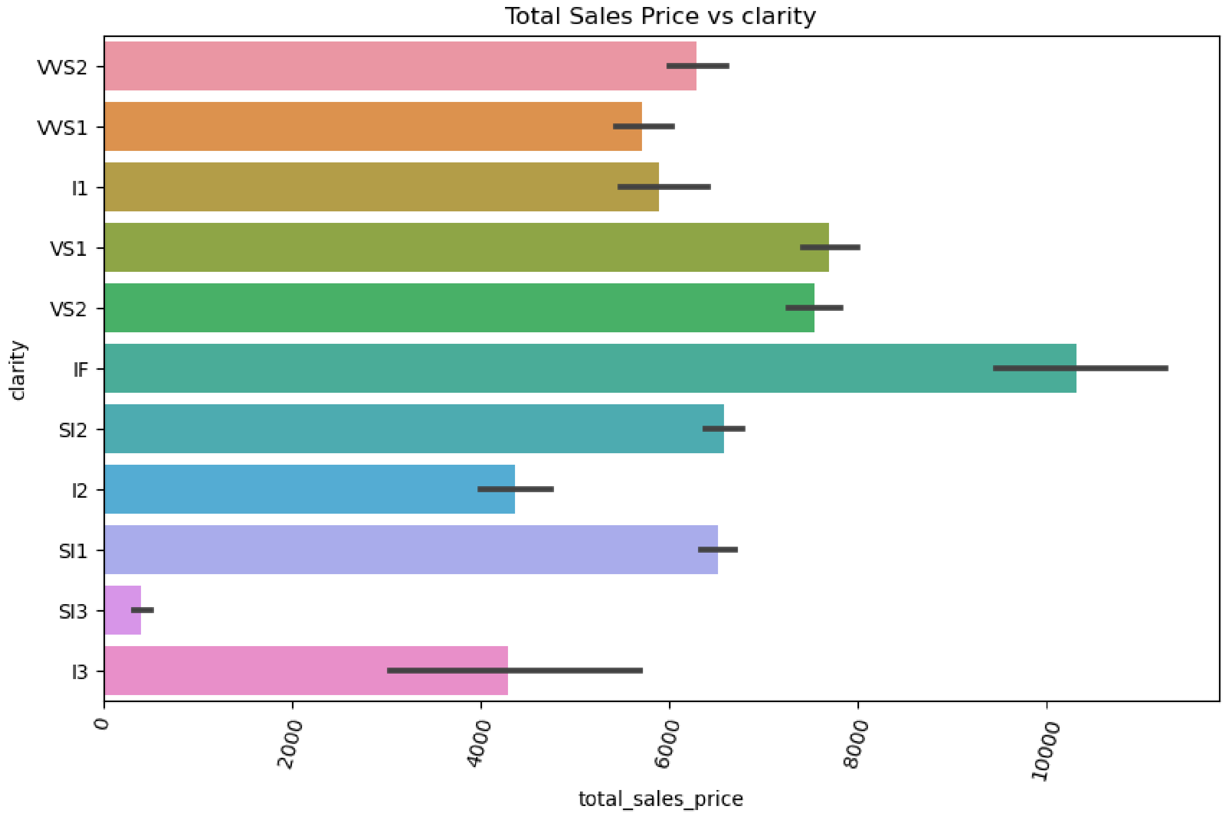
```
import seaborn as sns
import matplotlib.pyplot as plt

# Extracting categorical and numerical columns from the dataset
categorical_columns = [feature for feature in diamonds_df.columns if diamonds_df[featu
numerical_columns = [feature for feature in diamonds_df.columns if diamonds_df[feature
numerical_columns.remove('Unnamed: 0')  # Removing the 'Unnamed: 0' column as it's jus


features = numerical_columns + categorical_columns
target = ['total_sales_price']

#Pairplot for numerical features with hue based on 'cut_quality'
sns.pairplot(diamonds_df[numerical_columns + ['cut_quality']], hue="cut_quality")
plt.show()

#Bar plots for categorical features against total sales price
for cat in categorical_columns:
    plt.figure(figsize=(10, 6))
    sns.barplot(x='total_sales_price', y=cat, data=diamonds_df)
    plt.xticks(rotation=75)
    plt.title("Total Sales Price vs " + cat)
    plt.show()

#Scatter plots for numerical features against total sales price with hue based on 'cut
for num in numerical_columns:
    if num != 'total_sales_price':  # Avoiding plotting the target against itself
        sns.relplot(x='total_sales_price', y=num, hue='cut_quality', data=diamonds_df)
        plt.xticks(rotation=75)
        plt.title("Total Sales Price vs " + num)
        plt.show()
```

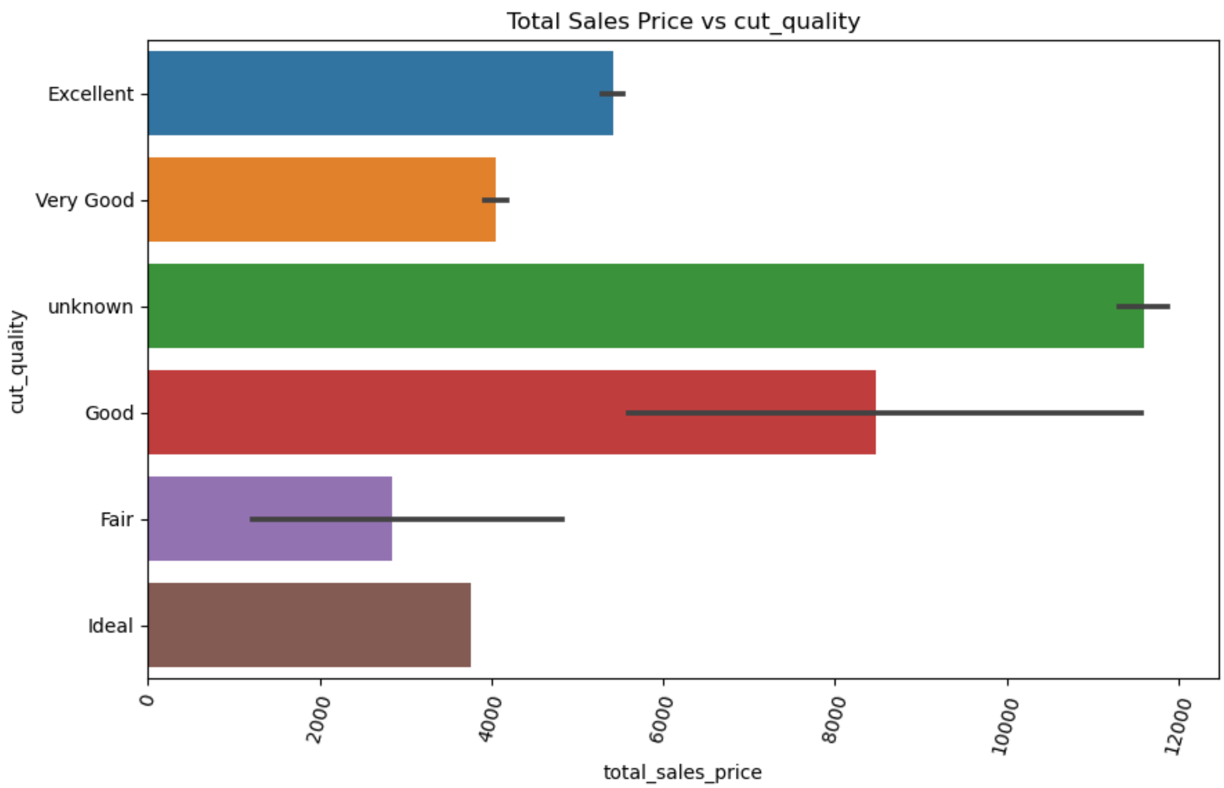
```python
#Distribution plots for numerical features
for num in numerical_columns:
    sns.kdeplot(diamonds_df[num])
    plt.xticks(rotation=75)
    plt.title("Distribution of " + num)
    plt.show()

# Distribution of the target variable
sns.kdeplot(diamonds_df['total_sales_price'], gridsize=100)
plt.title('Distribution of Total Sales Price')
plt.show()
```

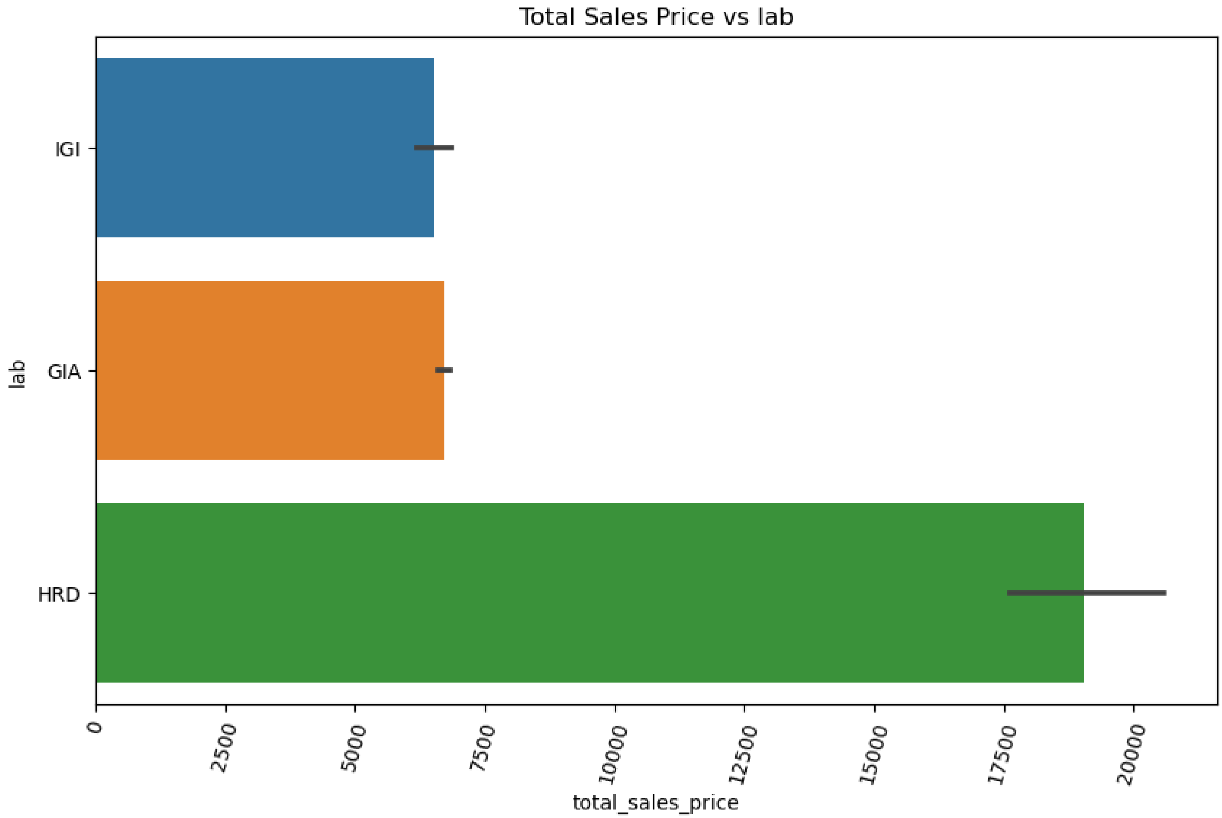


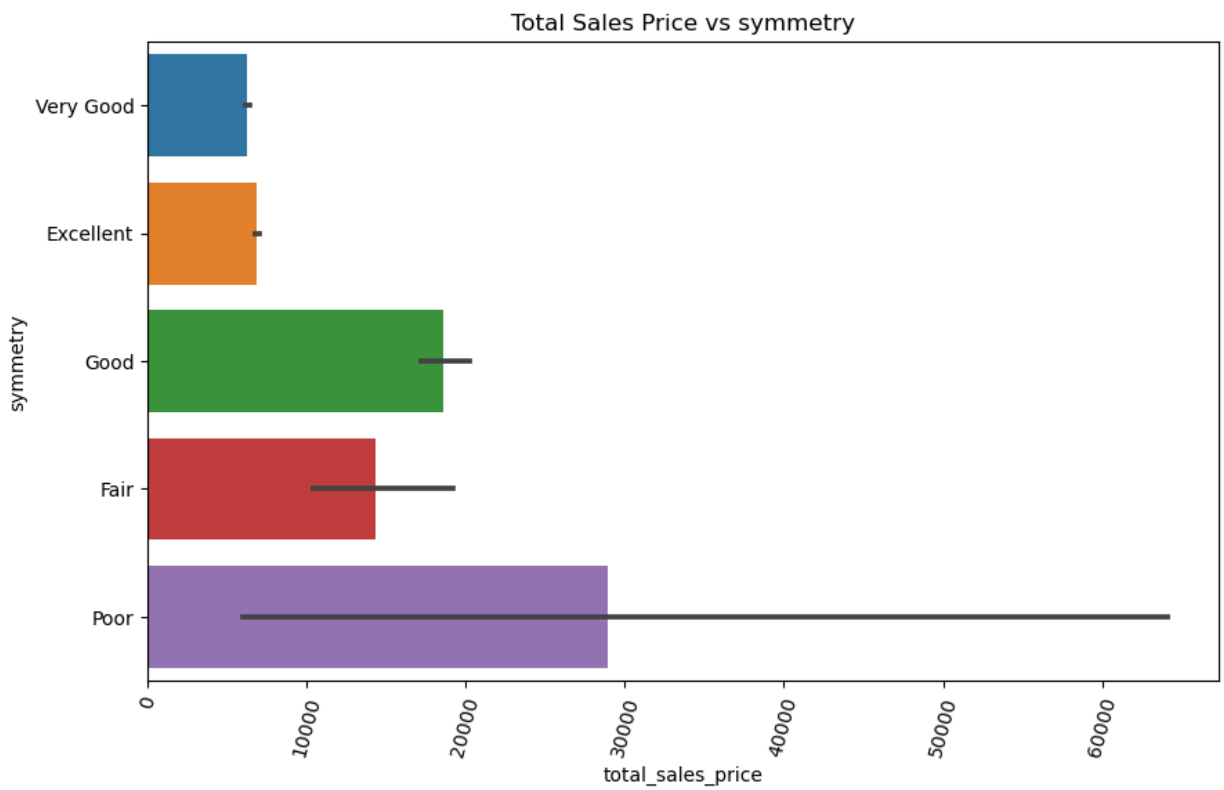

Total Sales Price vs cut
Round
Pear
Oval
Marquise
Princess
Emerald
Heart
Cushion
Radiant
Cushion Modified
Asscher
cut
0
5000
10000
15000
20000
25000
total_sales_price

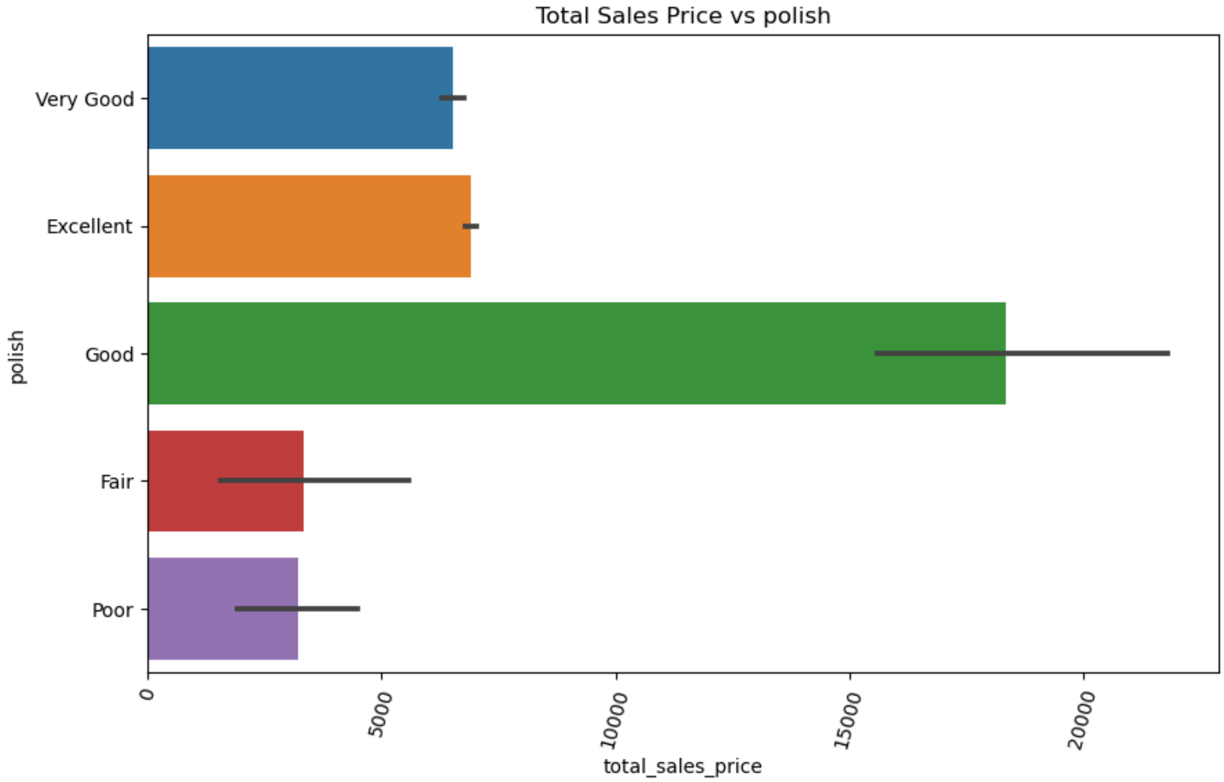

Total Sales Price vs color
E
F
L
D
J
I
G
H
M
K
unknown
color
0
2500
5000
7500
10000
12500
15000
17500
20000
total_sales_price

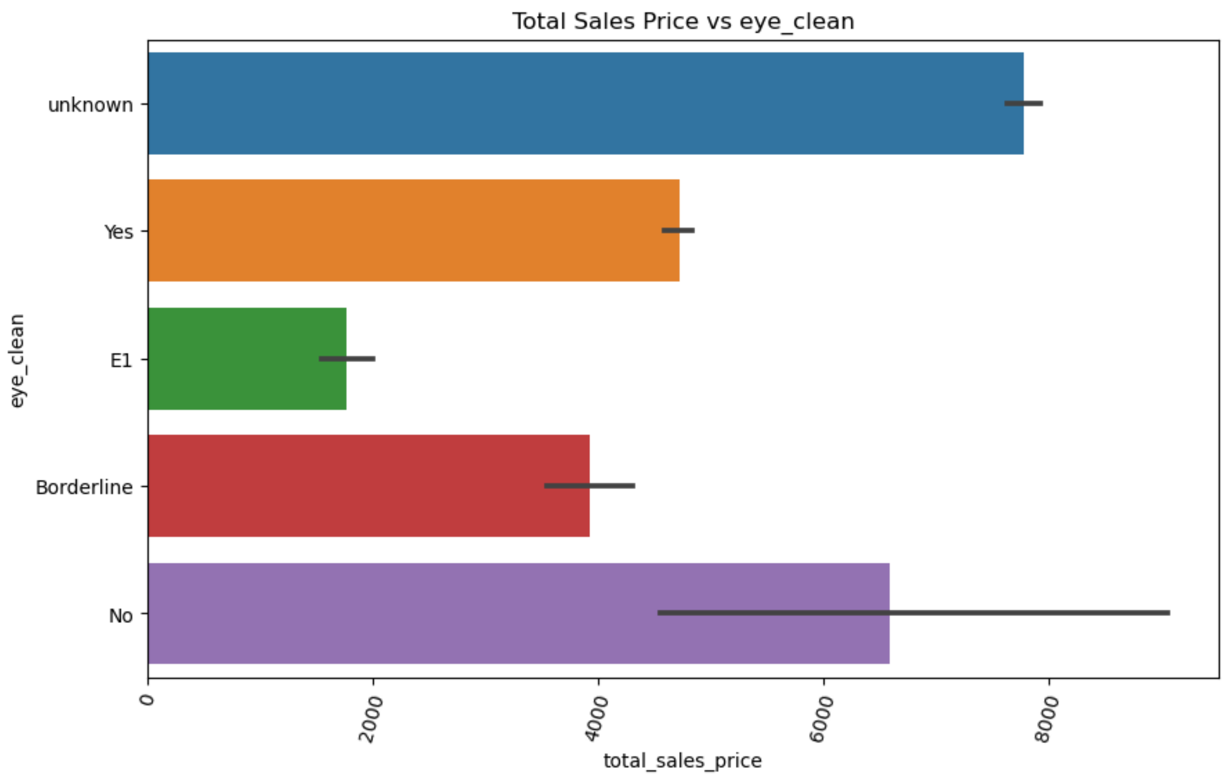

Total Sales Price vs clarity
clarity
VVS2
VVS1
I1
VS1
VS2
IF
SI2
I2
SI1
SI3
I3
0
2000
4000
6000
8000
10000
total_sales_price

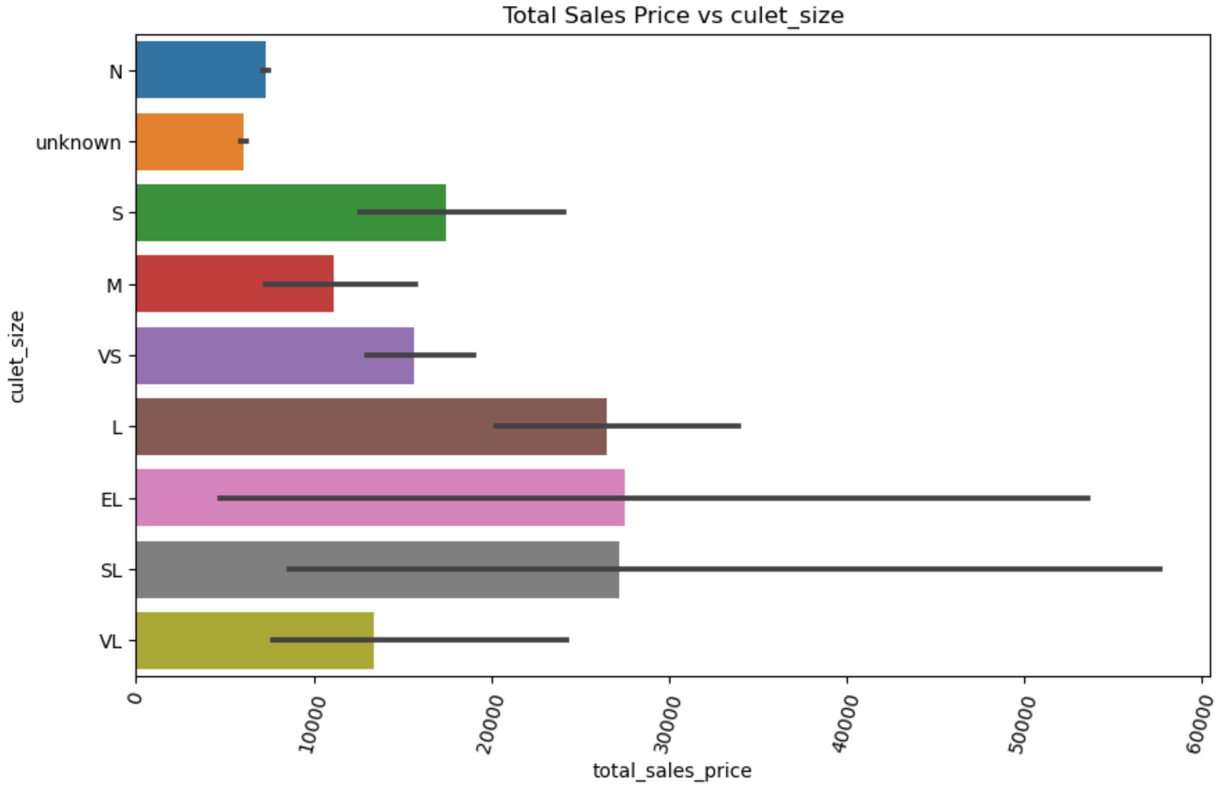

Total Sales Price vs cut_quality
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal
0
2000
4000
6000
8000
10000
12000
total_sales_price

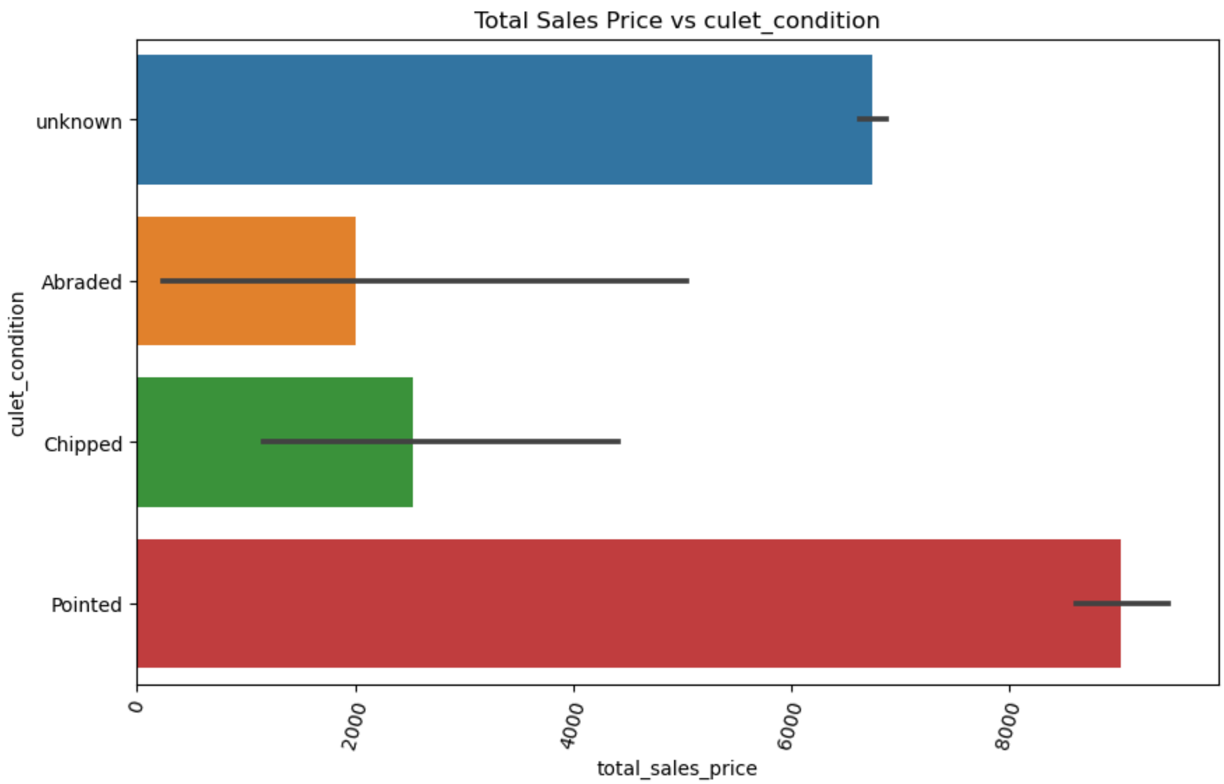

Total Sales Price vs lab
lab
IGI
GIA
HRD
total_sales_price
0
2500
5000
7500
10000
12500
15000
17500
20000

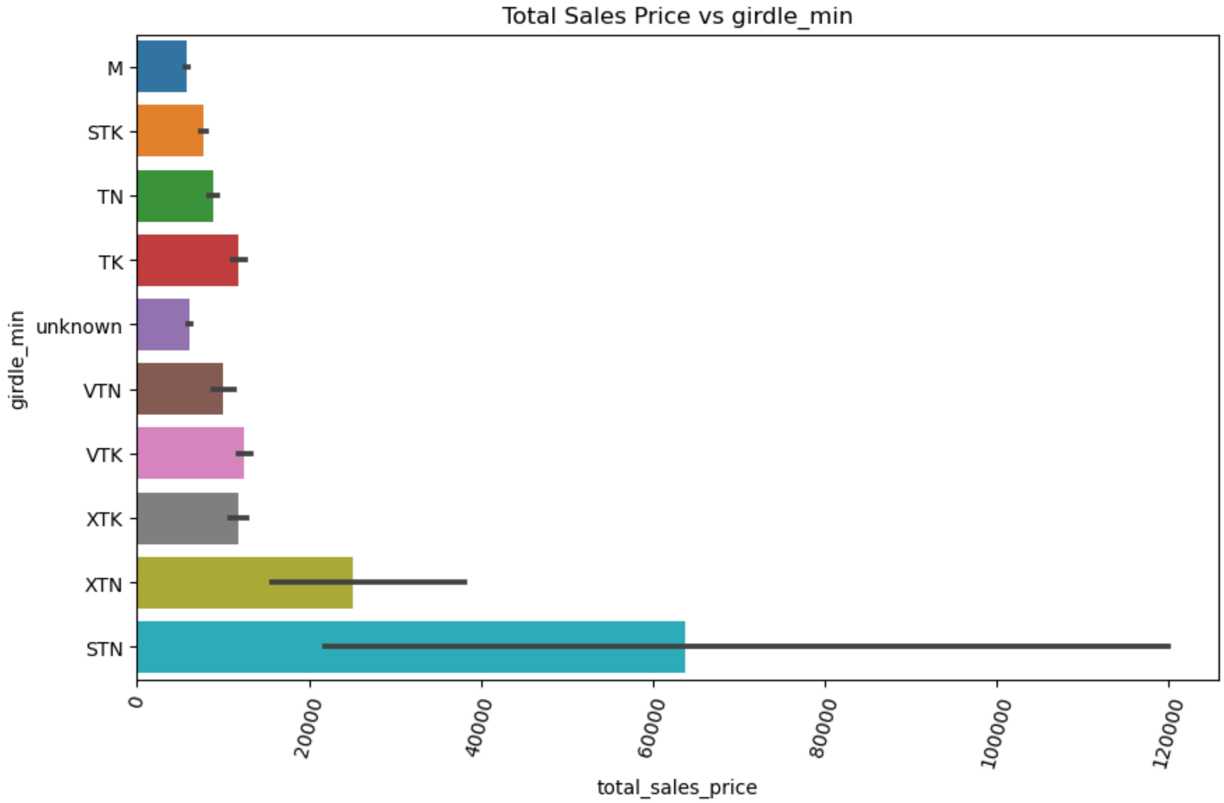

Total Sales Price vs symmetry
symmetry
Very Good
Excellent
Good
Fair
Poor
total_sales_price
0
10000
20000
30000
40000
50000
60000

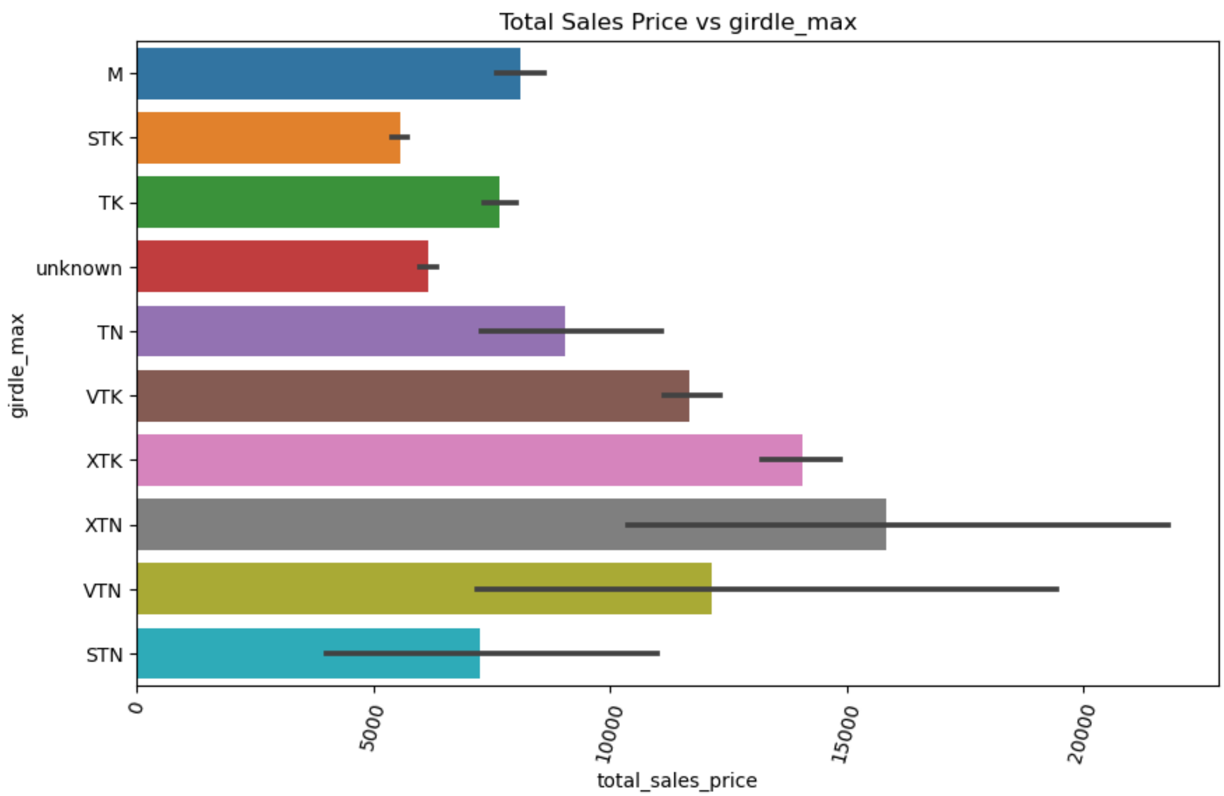

Total Sales Price vs polish
Very Good
Excellent
Good
Fair
Poor
polish
0
5000
10000
15000
20000
total_sales_price

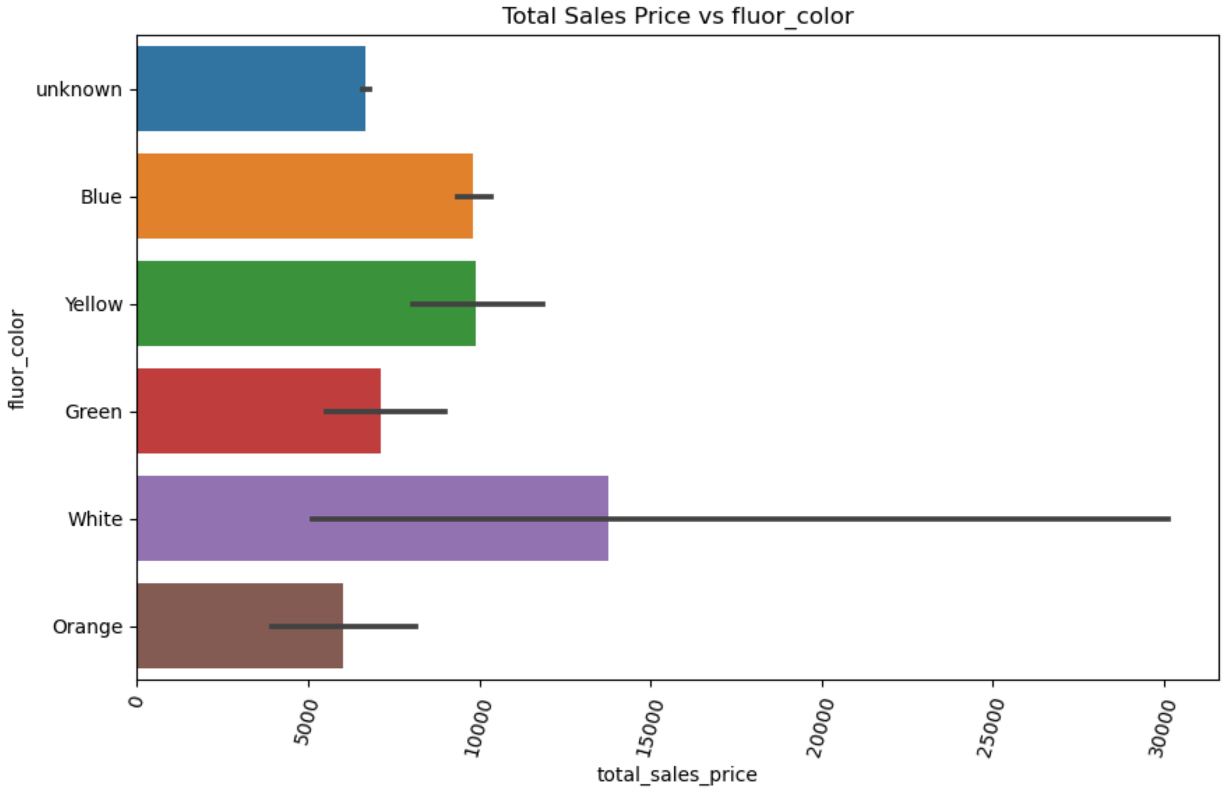

Total Sales Price vs eye_clean
unknown
Yes
E1
Borderline
No
eye_clean
0
2000
4000
6000
8000
total_sales_price

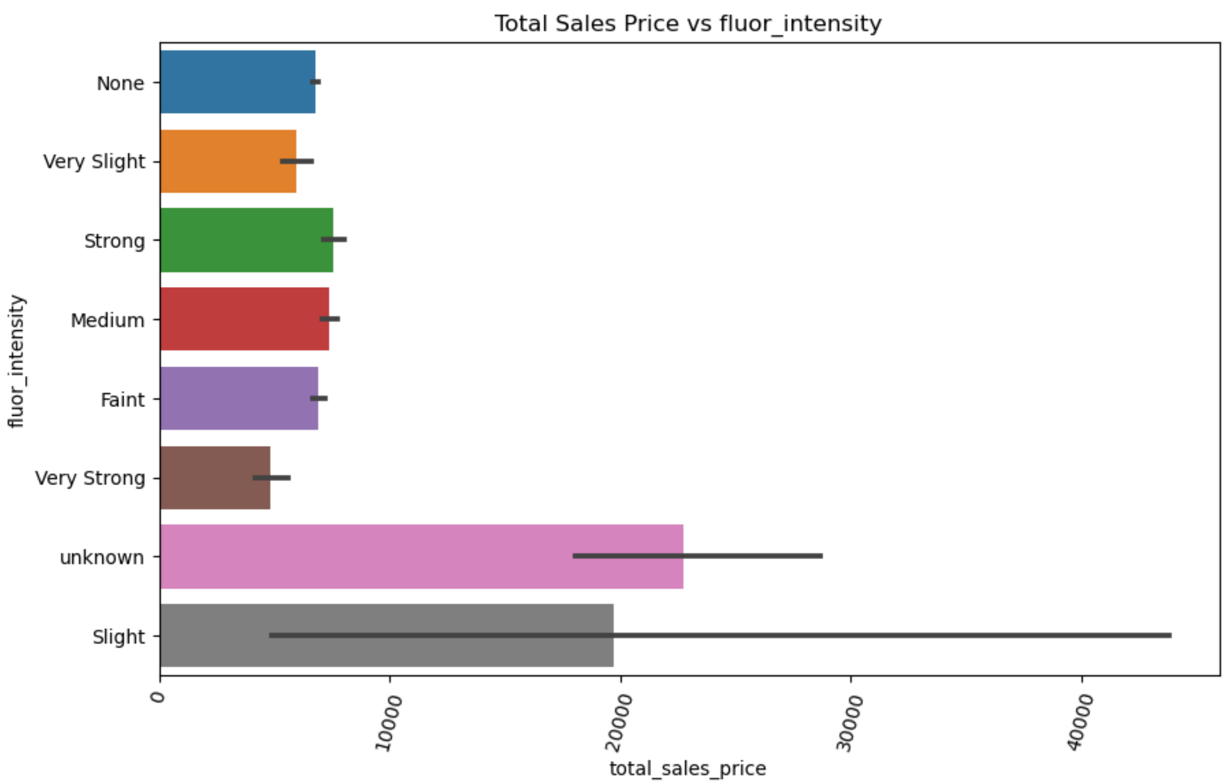

Total Sales Price vs culet_size
N
unknown
S
M
VS
L
EL
SL
VL
culet_size
0
10000
20000
30000
40000
50000
60000
total_sales_price

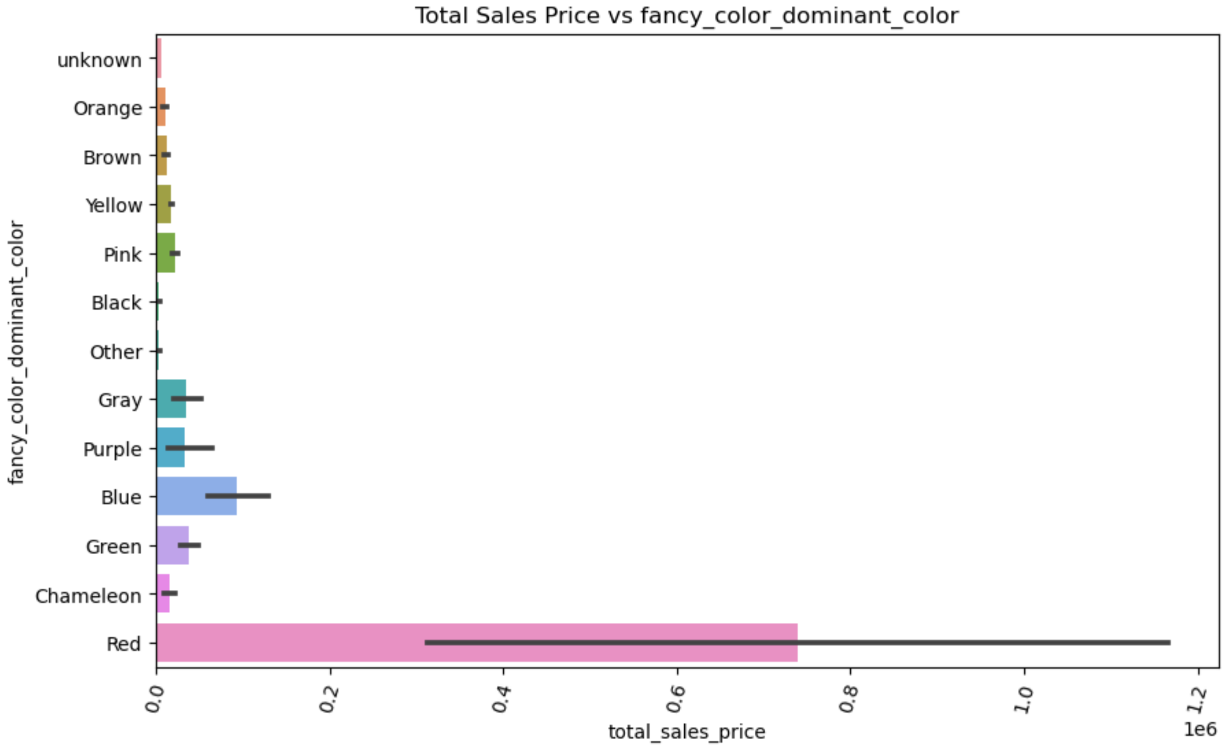

Total Sales Price vs culet_condition
unknown
Abraded
Chipped
Pointed
culet_condition
0
2000
4000
6000
8000
total_sales_price

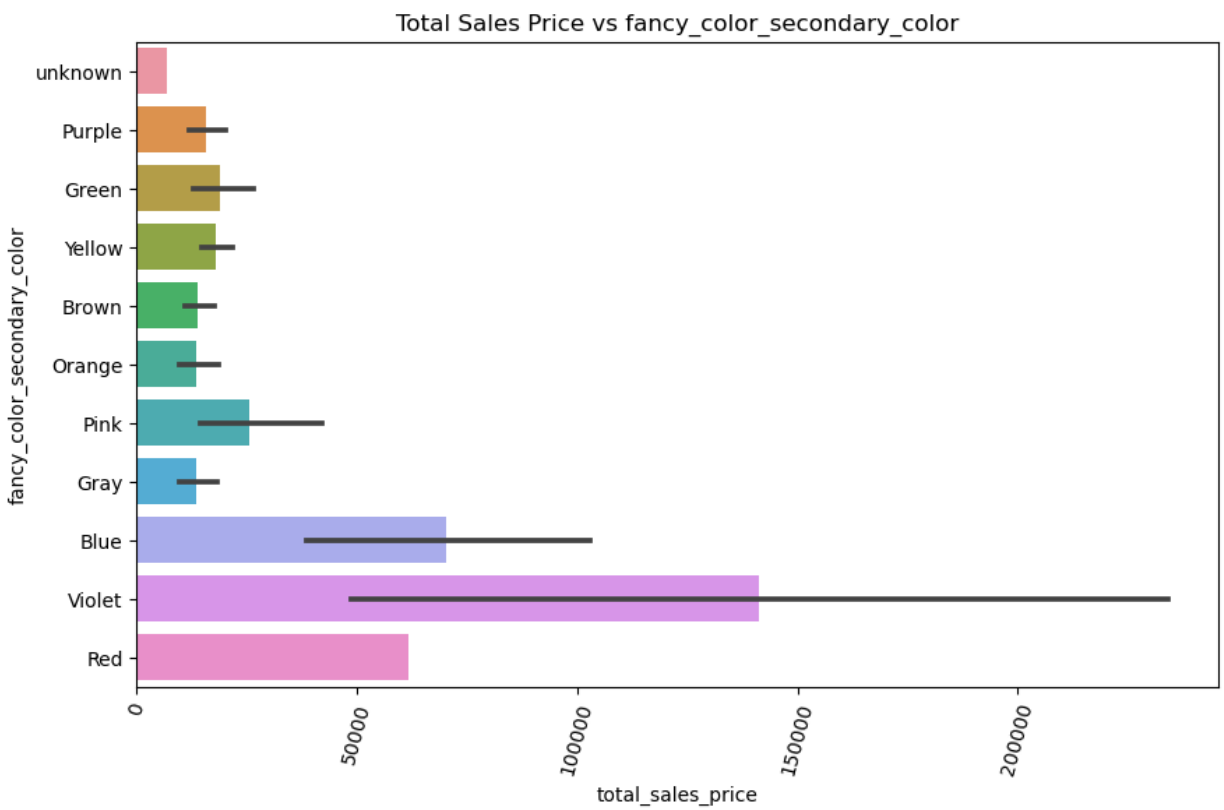

Total Sales Price vs girdle_min
girdle_min
M
STK
TN
TK
unknown
VTN
VTK
XTK
XTN
STN
total_sales_price
0
20000
40000
60000
80000
100000
120000

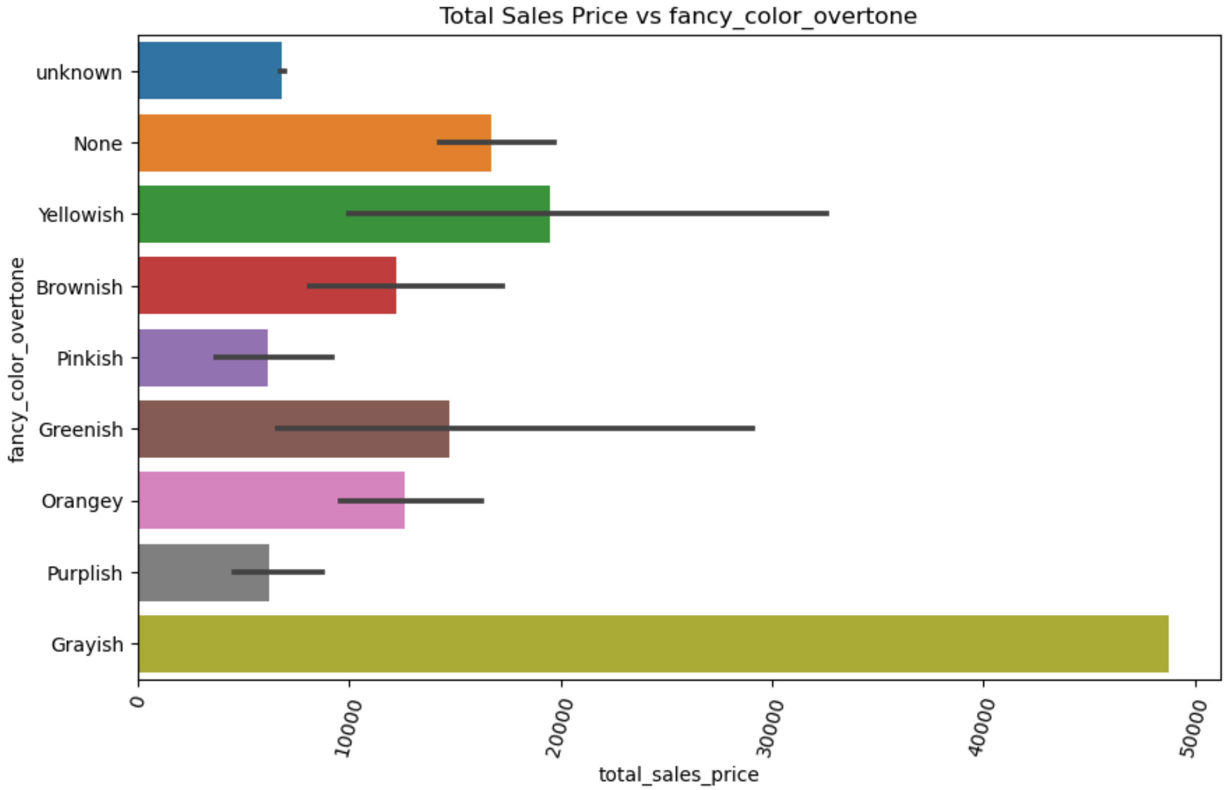

Total Sales Price vs girdle_max
girdle_max
M
STK
TK
unknown
TN
VTK
XTK
XTN
VTN
STN
total_sales_price
0
5000
10000
15000
20000

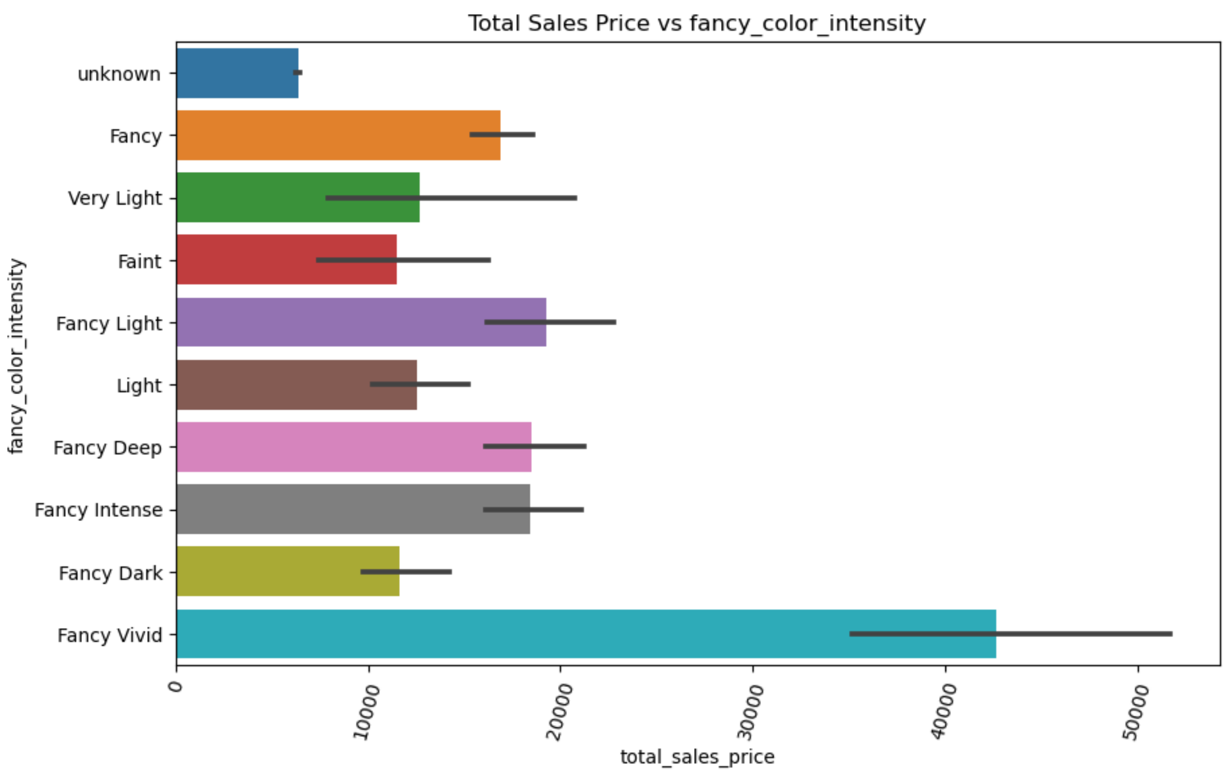

Total Sales Price vs fluor_color
unknown
Blue
Yellow
Green
White
Orange
fluor_color
0
5000
10000
15000
20000
25000
30000
total_sales_price

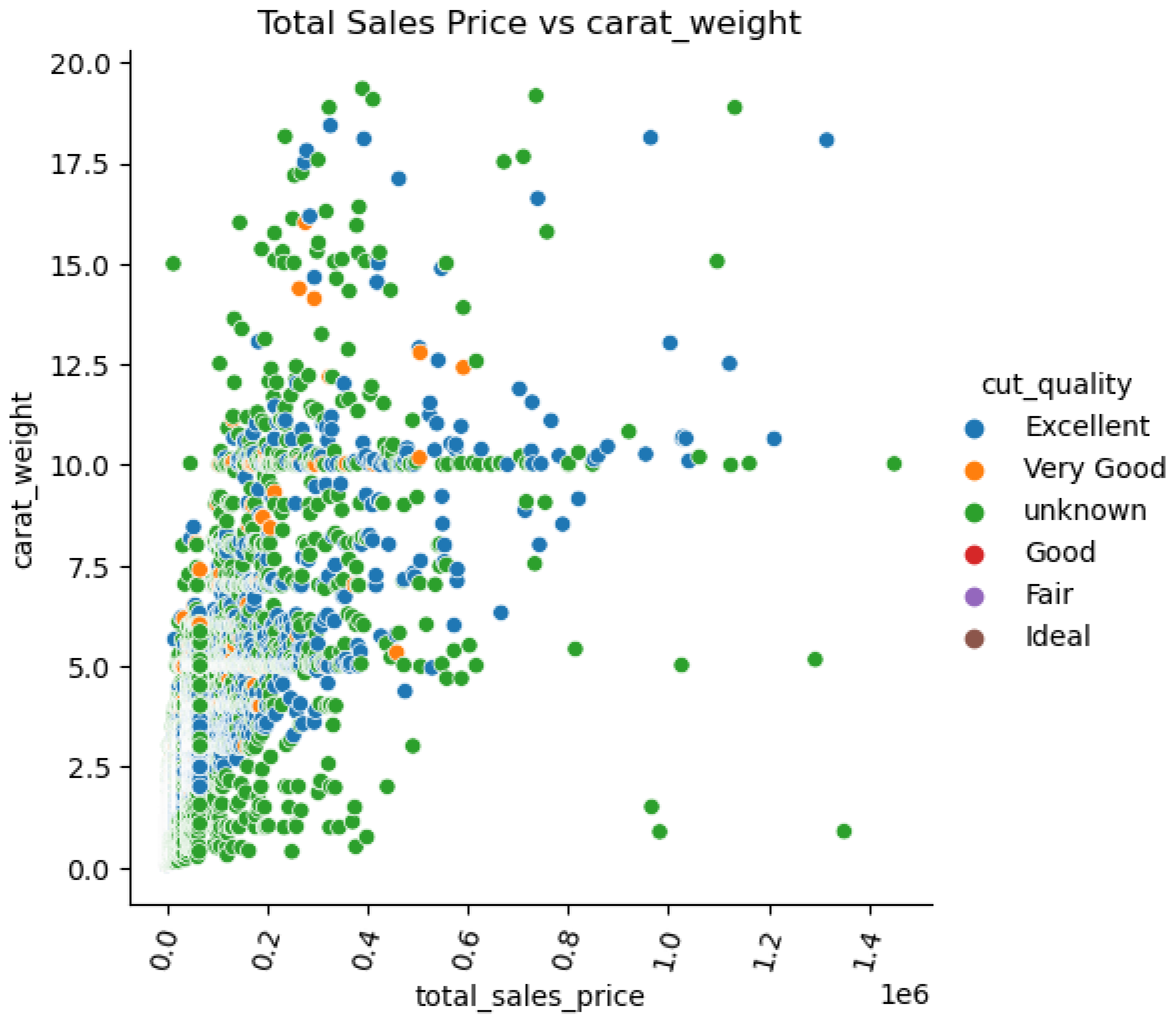

Total Sales Price vs fluor_intensity
None
Very Slight
Strong
Medium
Faint
Very Strong
unknown
Slight
fluor_intensity
0
10000
20000
30000
40000
total_sales_price


Total Sales Price vs fancy_color_dominant_color
fancy_color_dominant_color
unknown
Orange
Brown
Yellow
Pink
Black
Other
Gray
Purple
Blue
Green
Chameleon
Red
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e6
total_sales_price

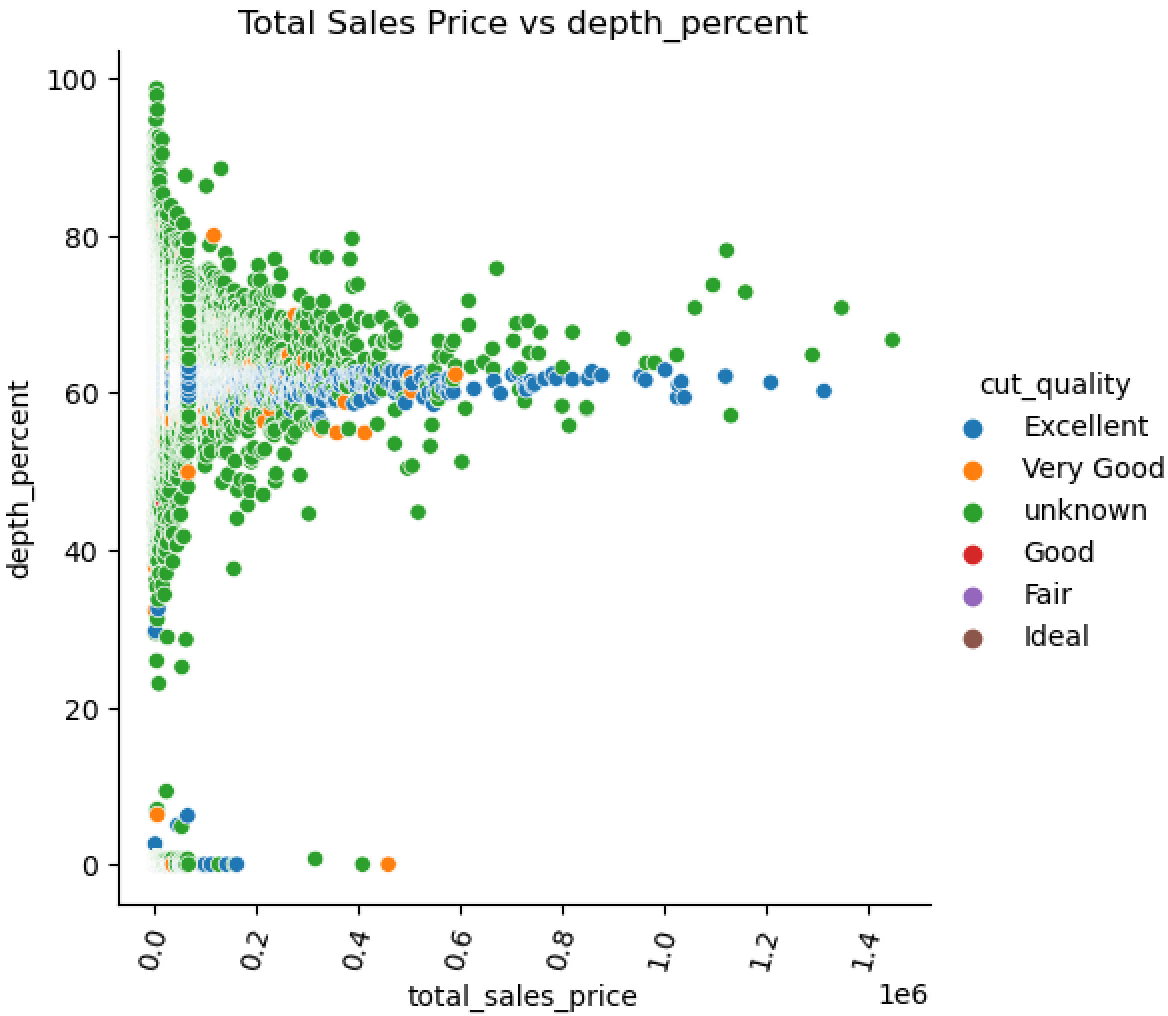

Total Sales Price vs fancy_color_secondary_color
fancy_color_secondary_color
unknown
Purple
Green
Yellow
Brown
Orange
Pink
Gray
Blue
Violet
Red
0
50000
100000
150000
200000
total_sales_price


Total Sales Price vs fancy_color_overtone
unknown
None
Yellowish
Brownish
Pinkish
Greenish
Orangey
Purplish
Grayish
fancy_color_overtone
0
10000
20000
30000
40000
50000
total_sales_price

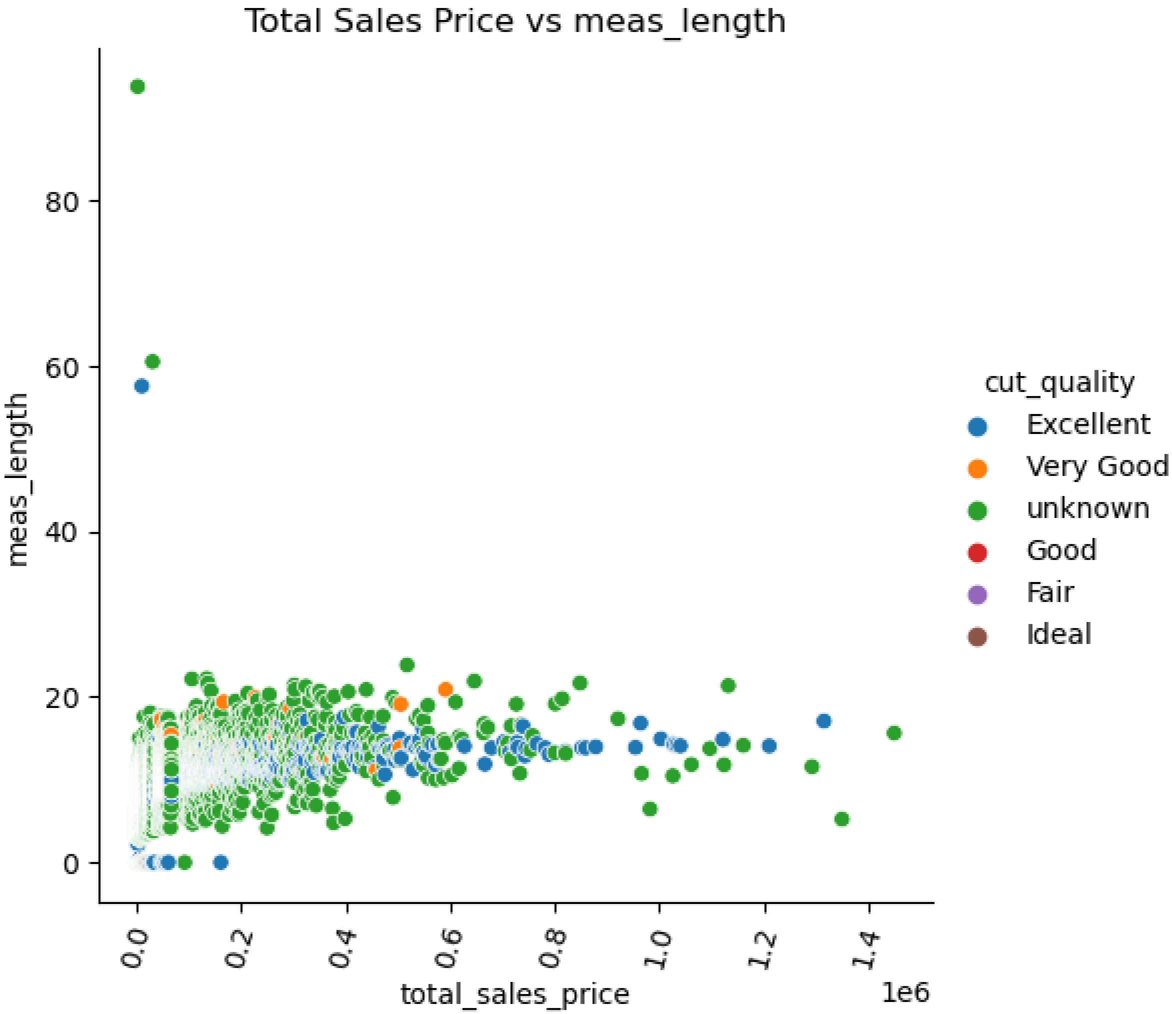

Total Sales Price vs fancy_color_intensity
unknown
Fancy
Very Light
Faint
Fancy Light
Light
Fancy Deep
Fancy Intense
Fancy Dark
Fancy Vivid
fancy_color_intensity
0
10000
20000
30000
40000
50000
total_sales_price



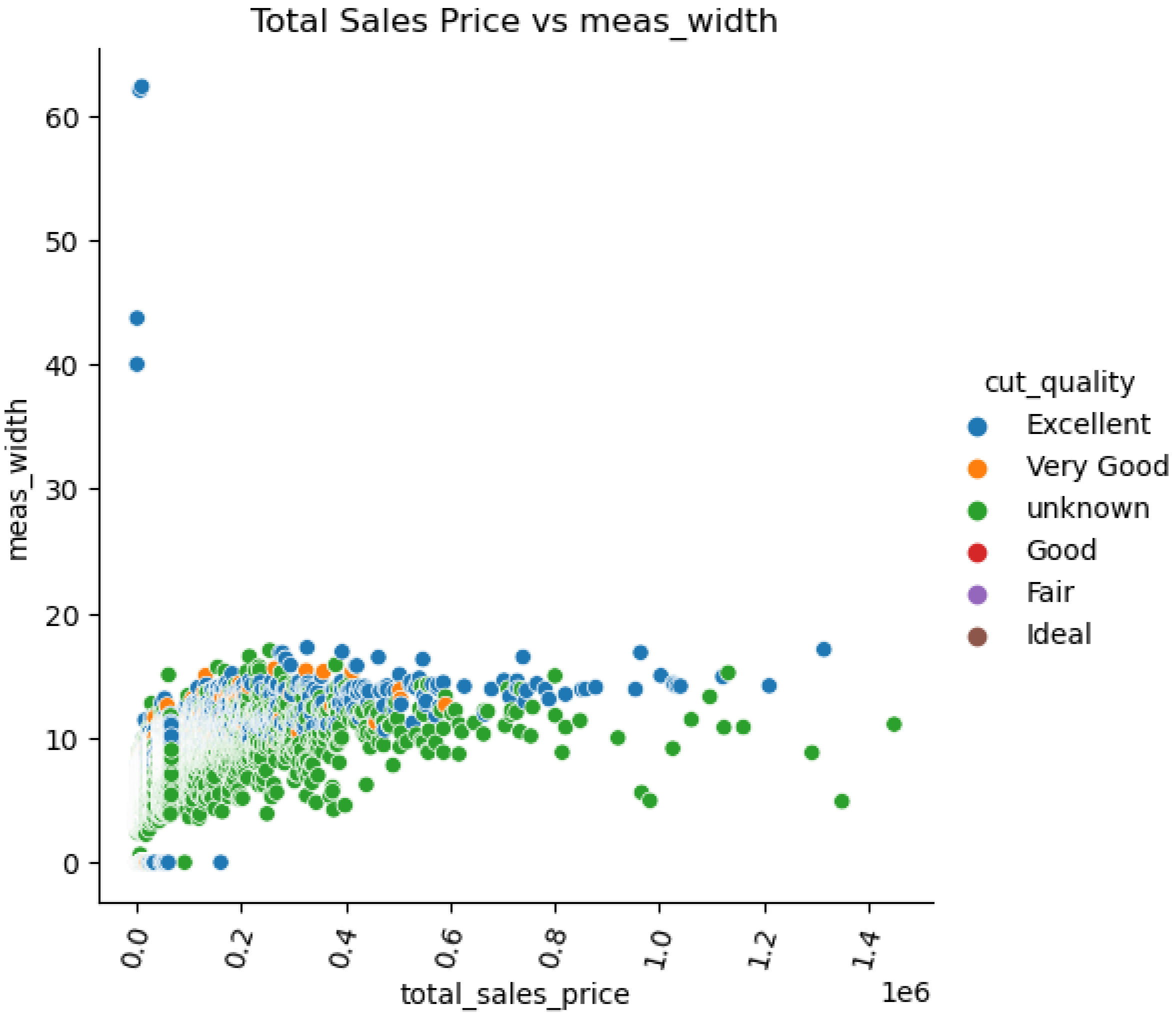
Total Sales Price vs carat_weight
carat_weight
total_sales_price
1e6
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal

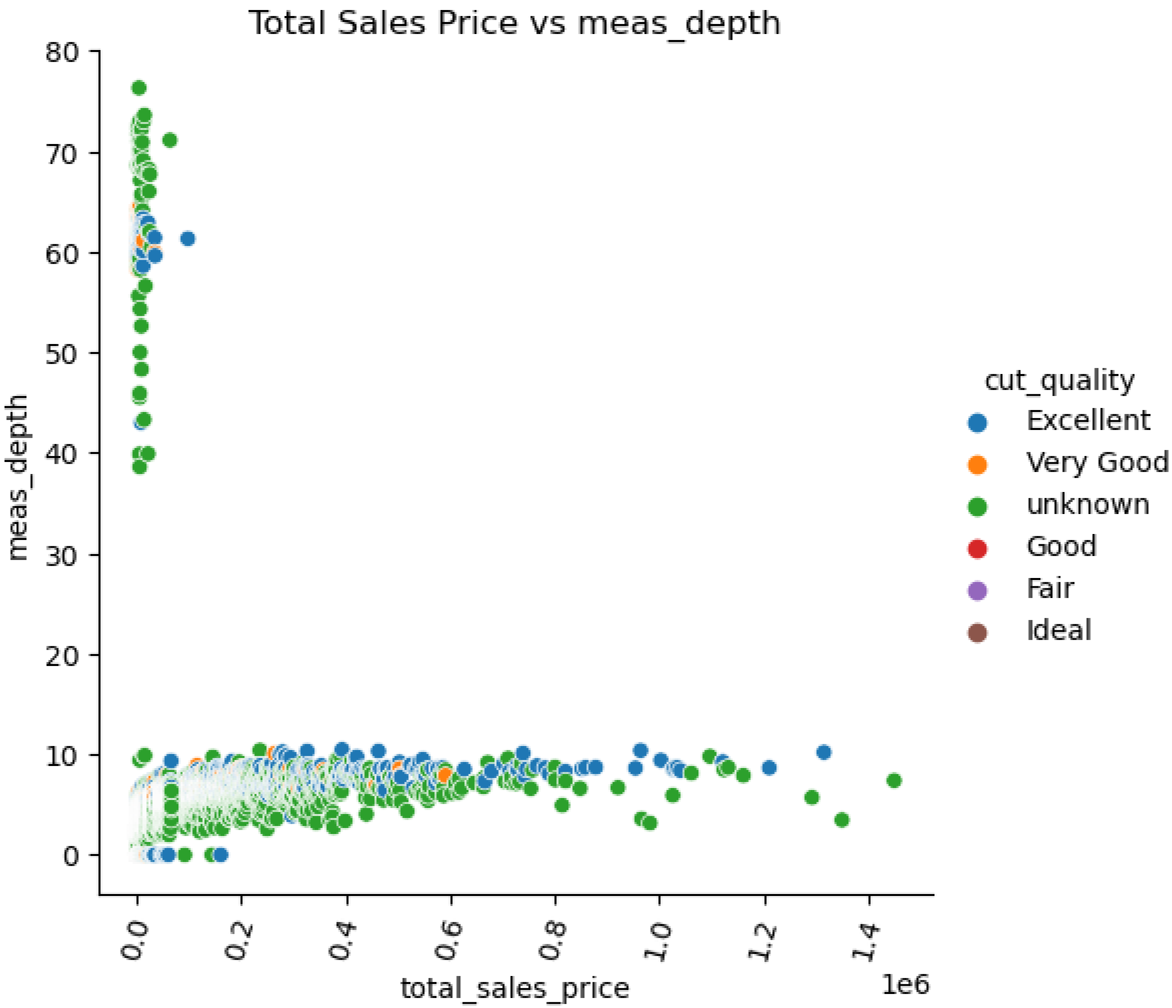

Total Sales Price vs depth_percent
100
80
60
40
20
0
depth_percent
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
total_sales_price
1e6
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal

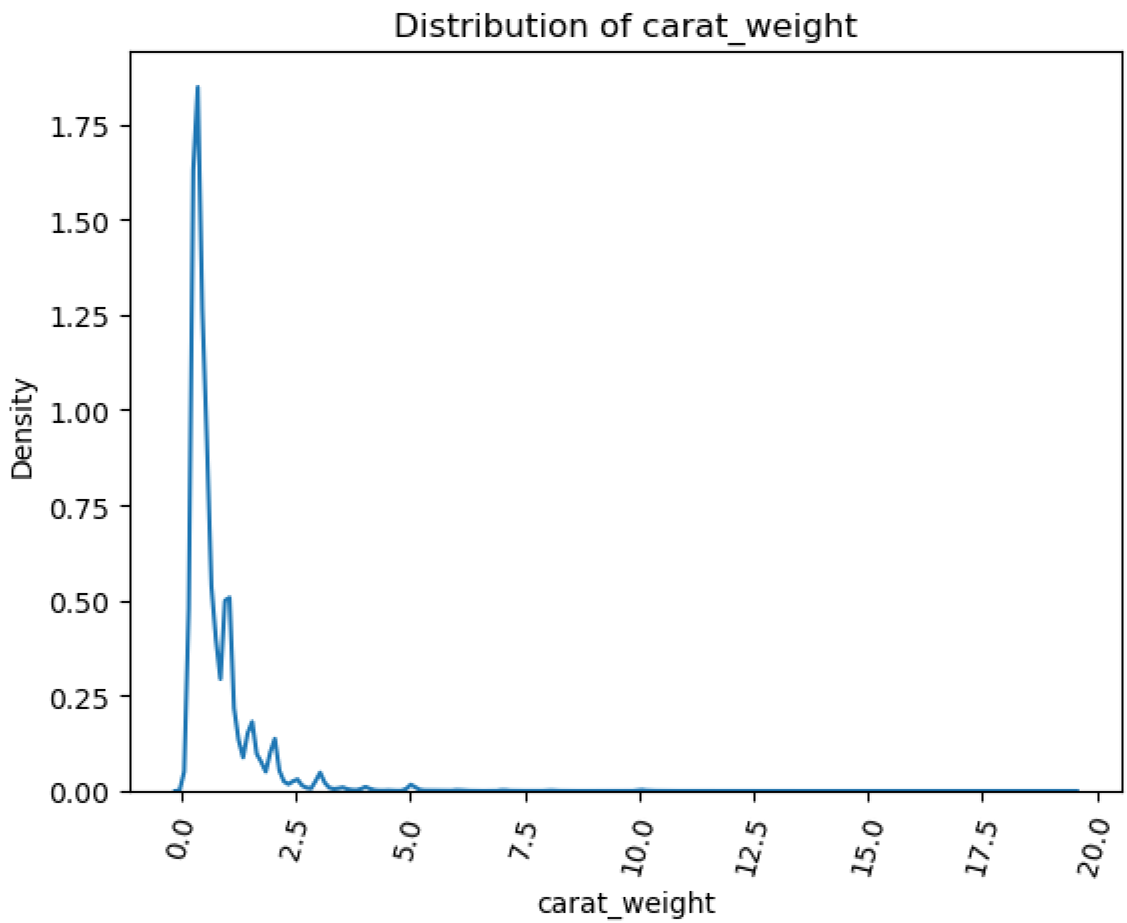

Total Sales Price vs table_percent
table_percent
total_sales_price
1e6
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
20
40
60
80
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal

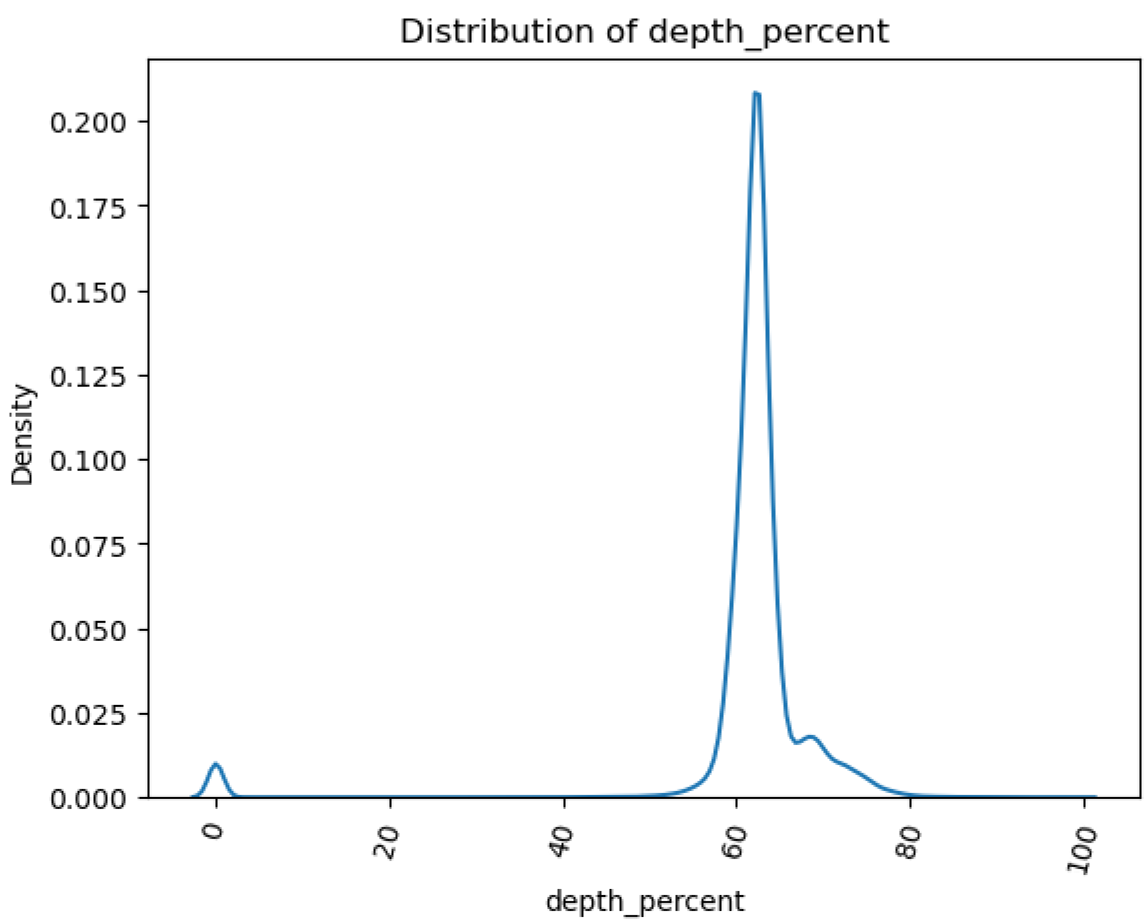


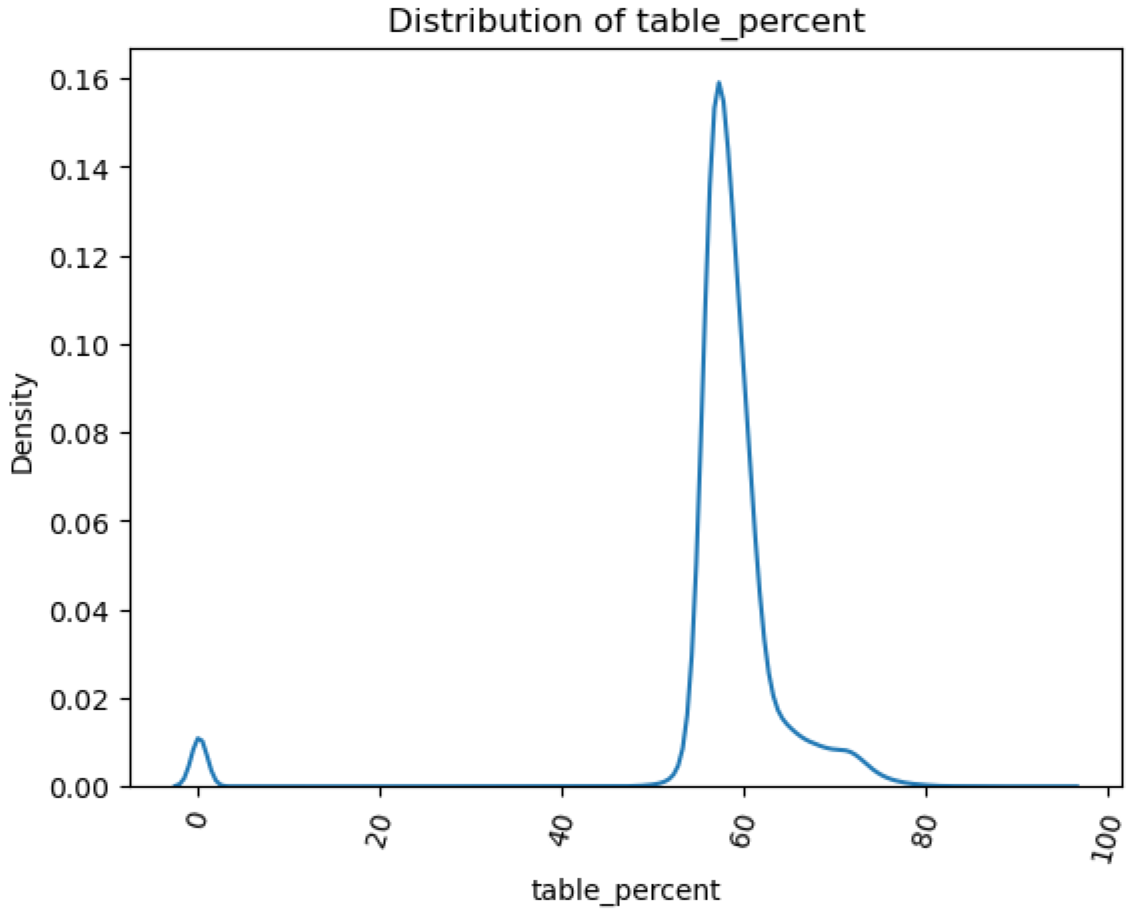
Total Sales Price vs meas_length
meas_length
0
20
40
60
80
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1e6
total_sales_price
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal

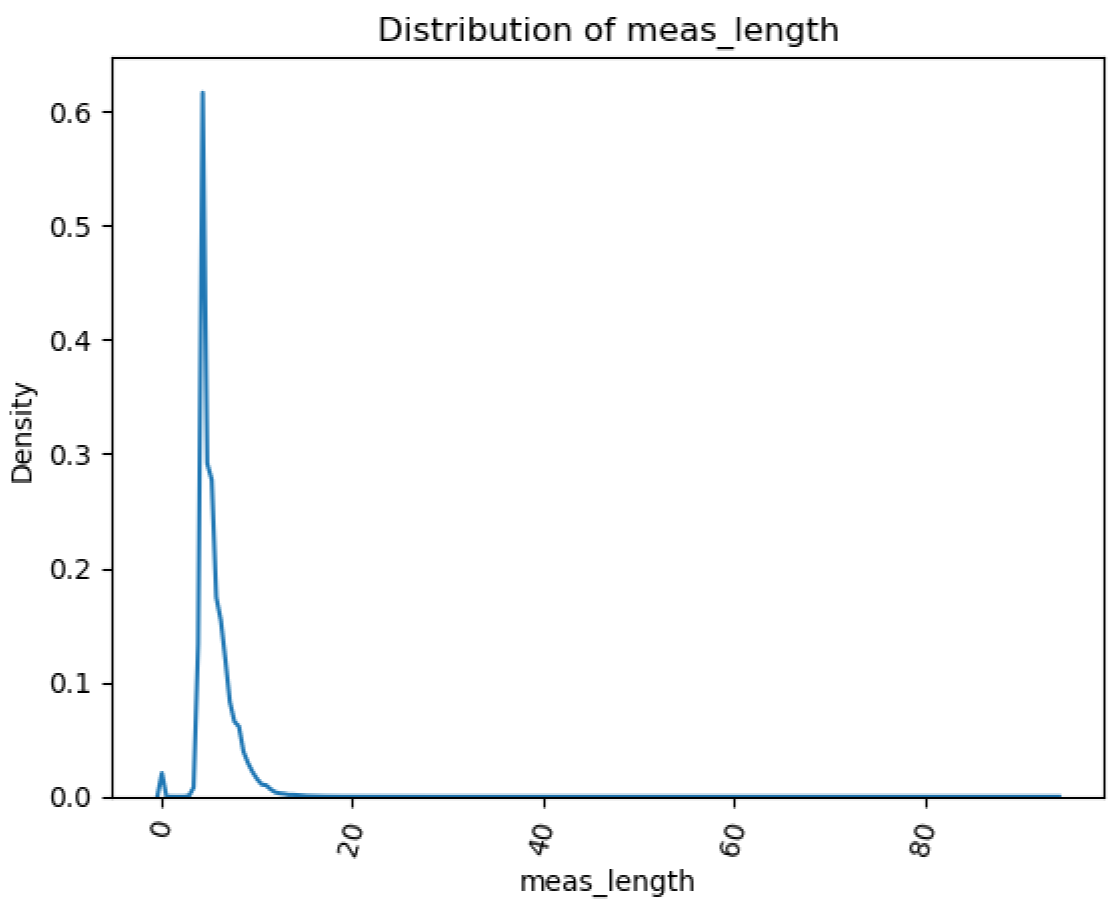

Total Sales Price vs meas_width
meas_width
total_sales_price
1e6
0
10
20
30
40
50
60
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal

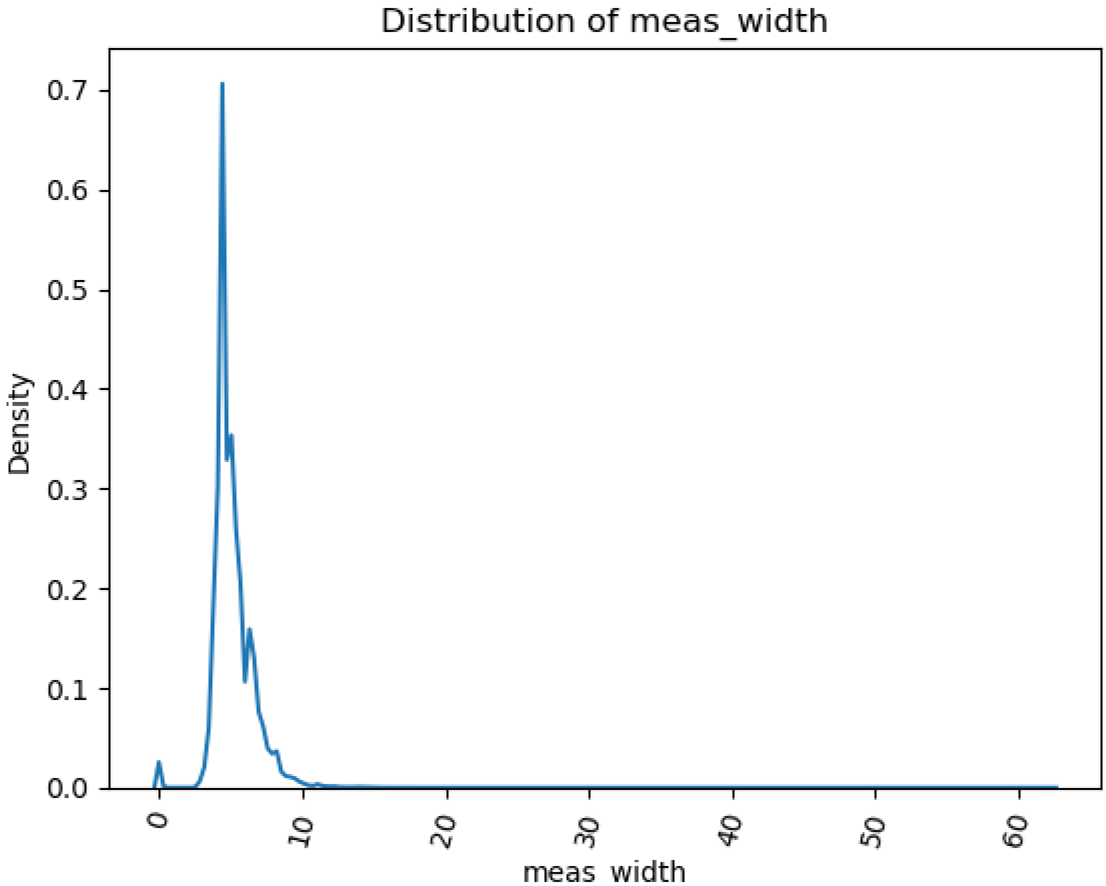

Total Sales Price vs meas_depth
80
70
60
50
40
30
20
10
0
meas_depth
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1e6
total_sales_price
cut_quality
Excellent
Very Good
unknown
Good
Fair
Ideal

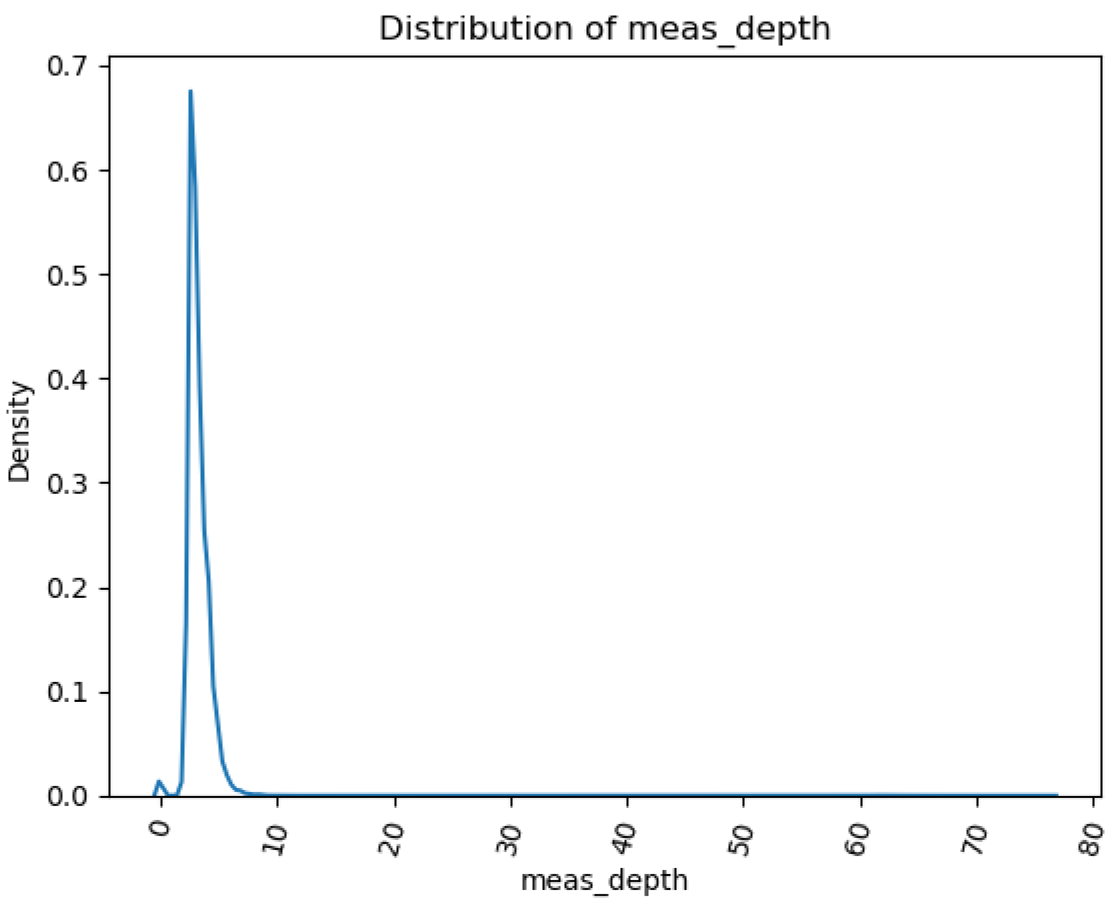

Distribution of carat_weight
Density
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
carat_weight

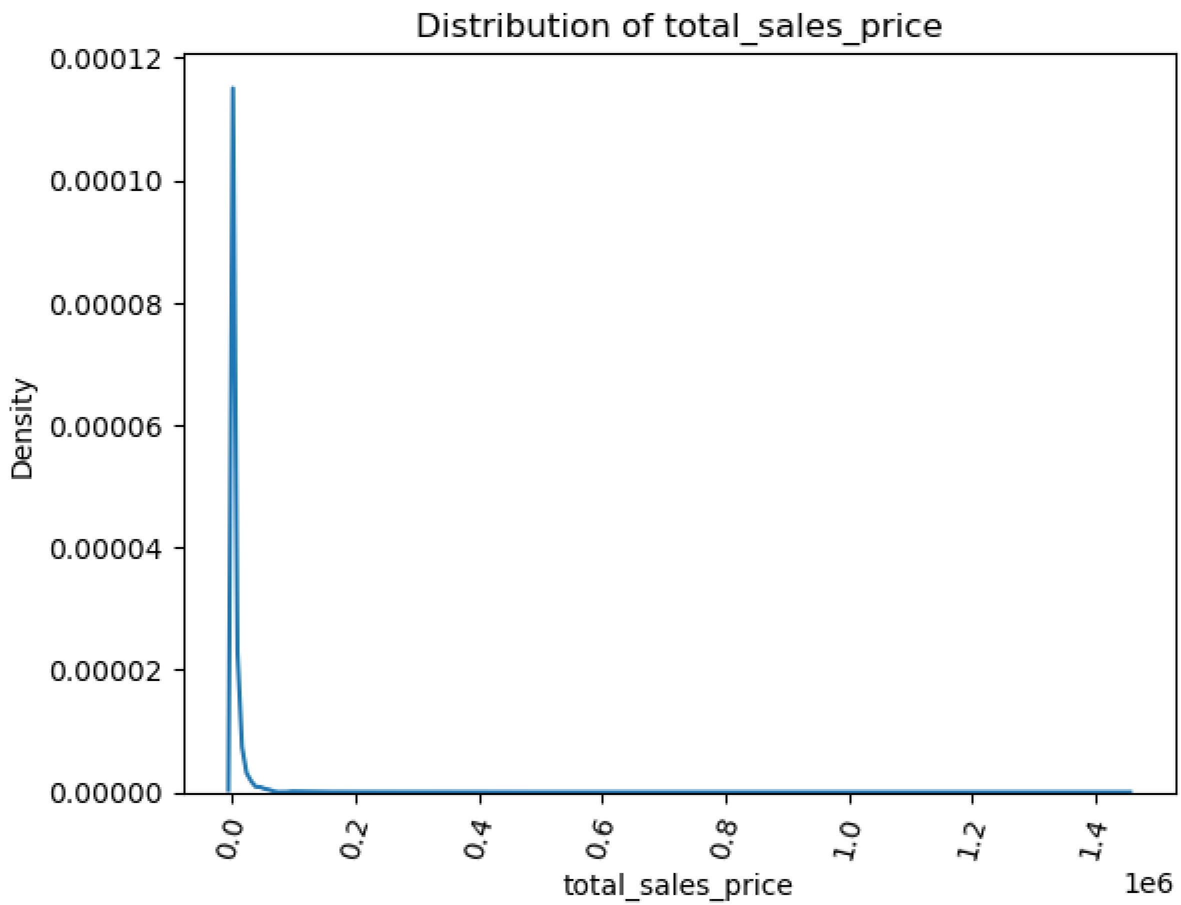

Distribution of depth_percent
Density
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
20
40
60
80
100
depth_percent

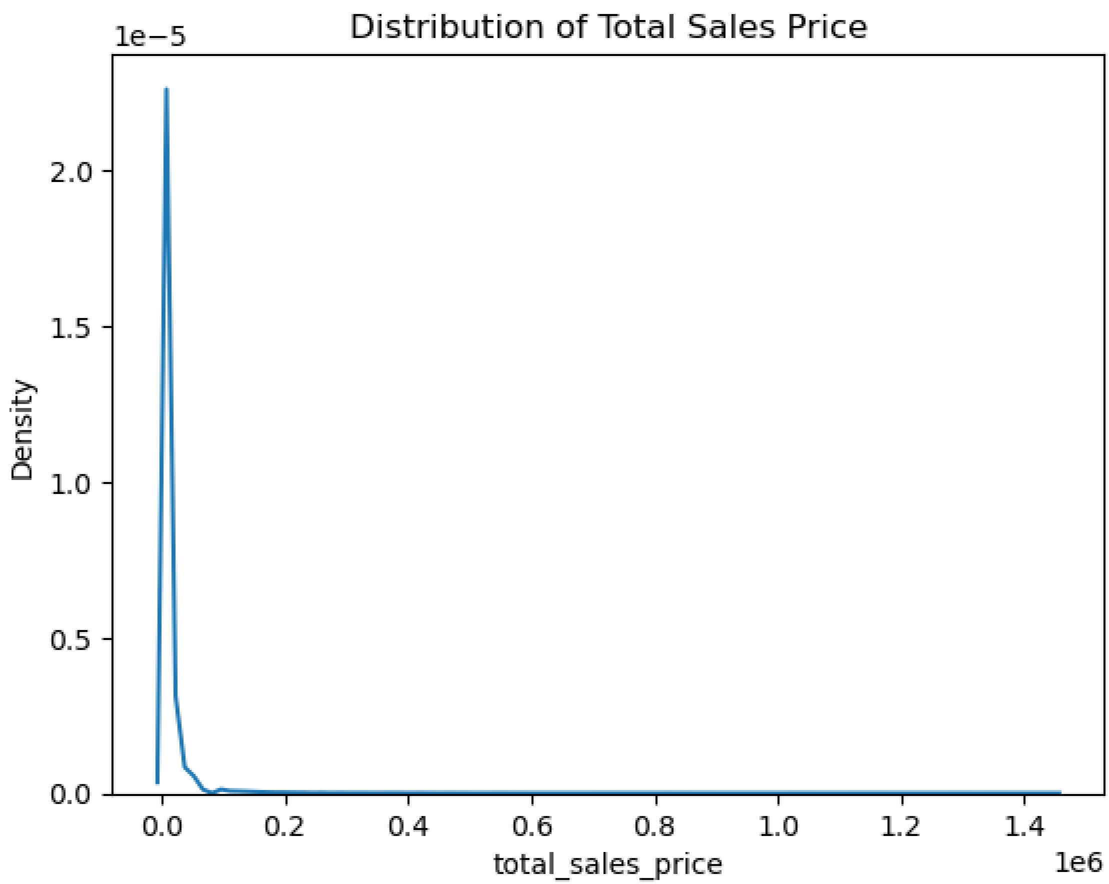

Distribution of table_percent
Density
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100
table_percent


Distribution of meas_length
Density
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
20
40
60
80
meas_length


Distribution of meas_width
Density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
10
20
30
40
50
60
meas_width

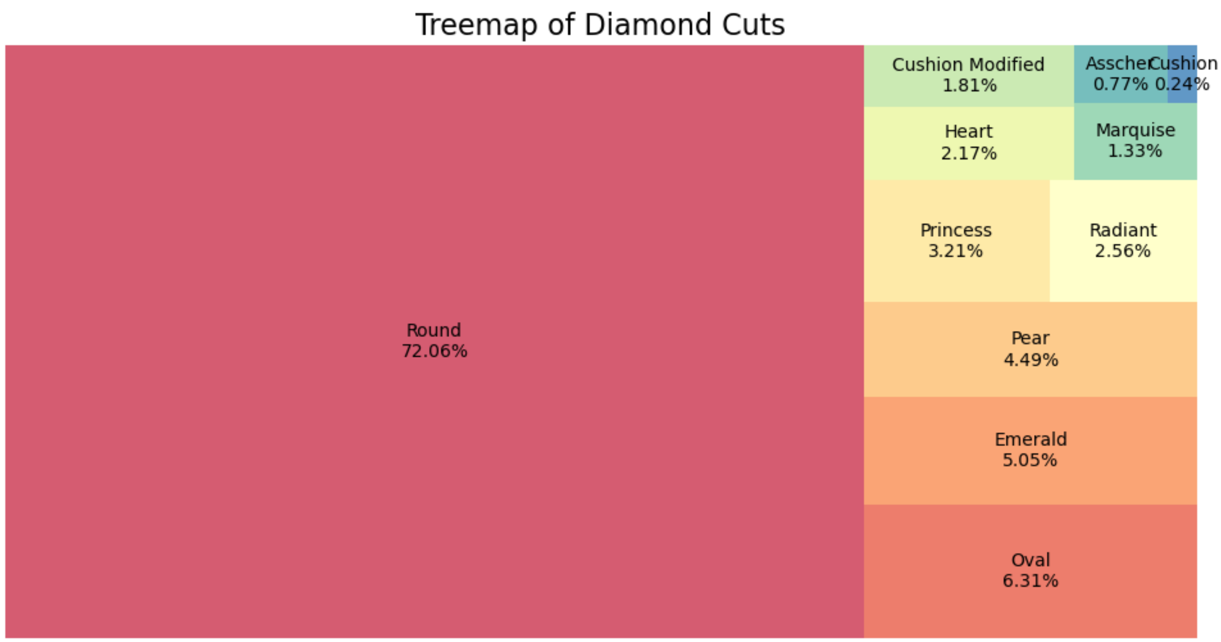

Distribution of meas_depth
Density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
10
20
30
40
50
60
70
80
meas_depth





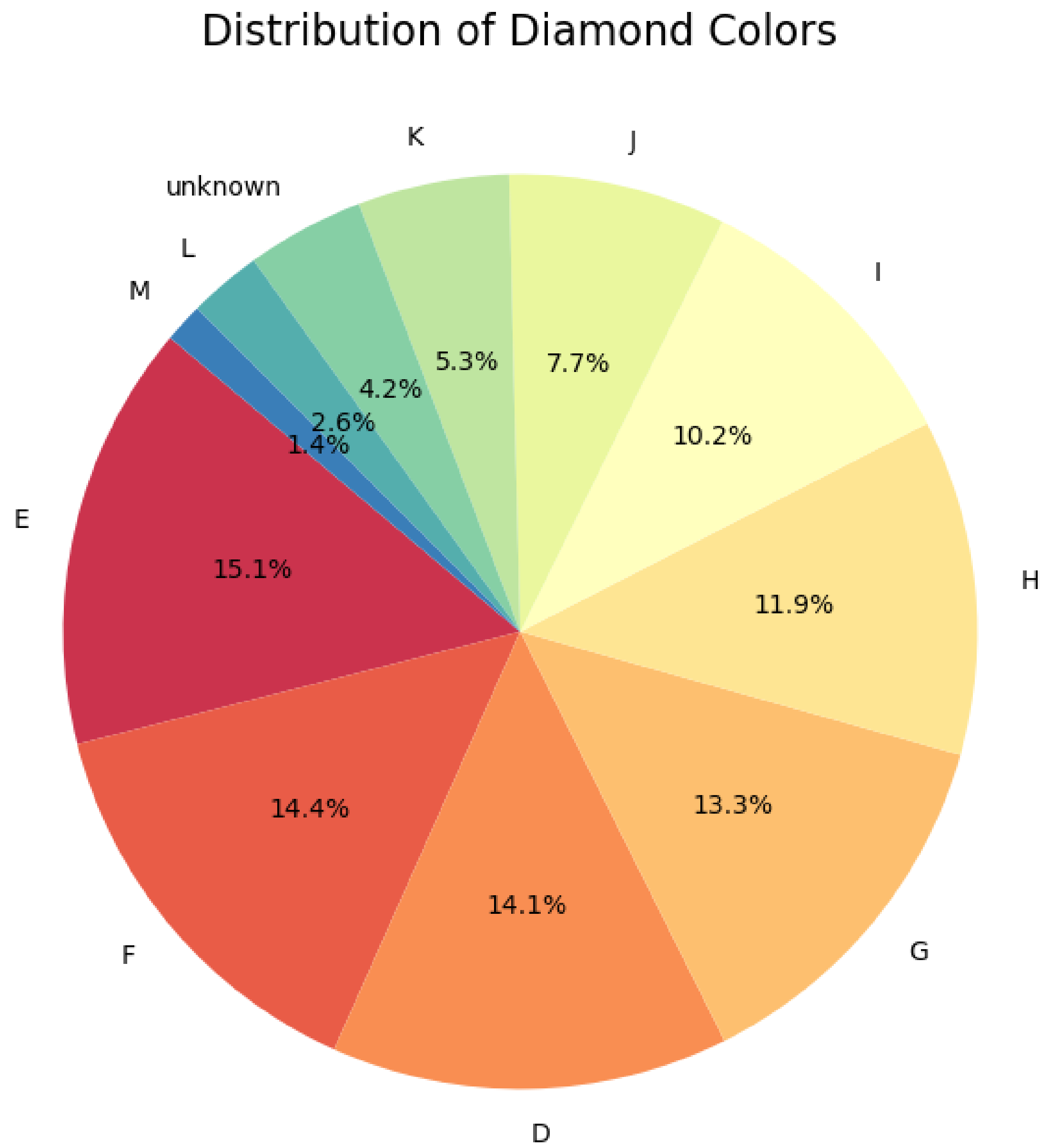
In [40]:
```
import squarify  # For treemap
```

```
# Calculating the percentage of each cut type
cut_counts = diamonds_df['cut'].value_counts()
total_diamonds = len(diamonds_df)
cut_percentages = (cut_counts / total_diamonds) * 100

# Creating a treemap for the distribution of diamond cuts
plt.figure(figsize=(12, 6))
colors = sns.color_palette("Spectral", n_colors=len(cut_counts))
squarify.plot(sizes=cut_counts, label=['%s\n%.2f%%' % (cut, percent) for cut, percent
plt.title('Treemap of Diamond Cuts', fontsize=16)
plt.axis('off')  # Hides the axes
plt.show()
```



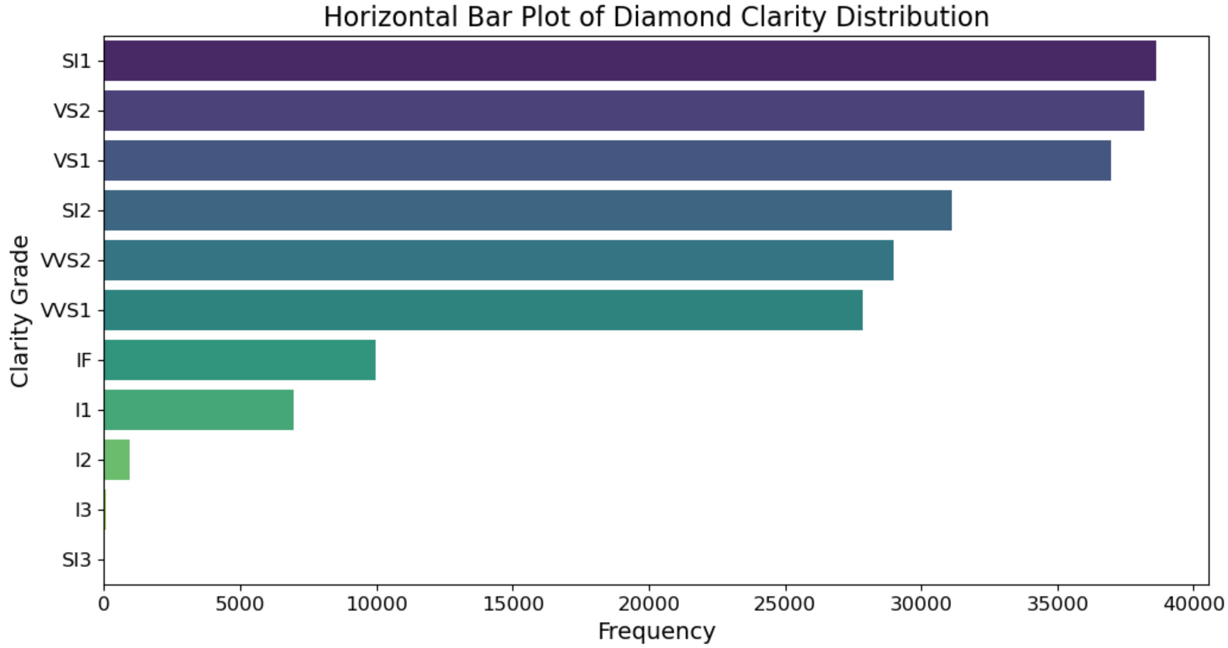
In [ ]:

In [41]:
```
# Creating a pie chart for the distribution of diamond colors

# Calculating color counts
color_counts = diamonds_df['color'].value_counts()

# Pie chart
plt.figure(figsize=(10, 8))
plt.pie(color_counts, labels=color_counts.index, autopct='%1.1f%%', startangle=140, co
plt.title('Distribution of Diamond Colors', fontsize=16)
plt.show()
```



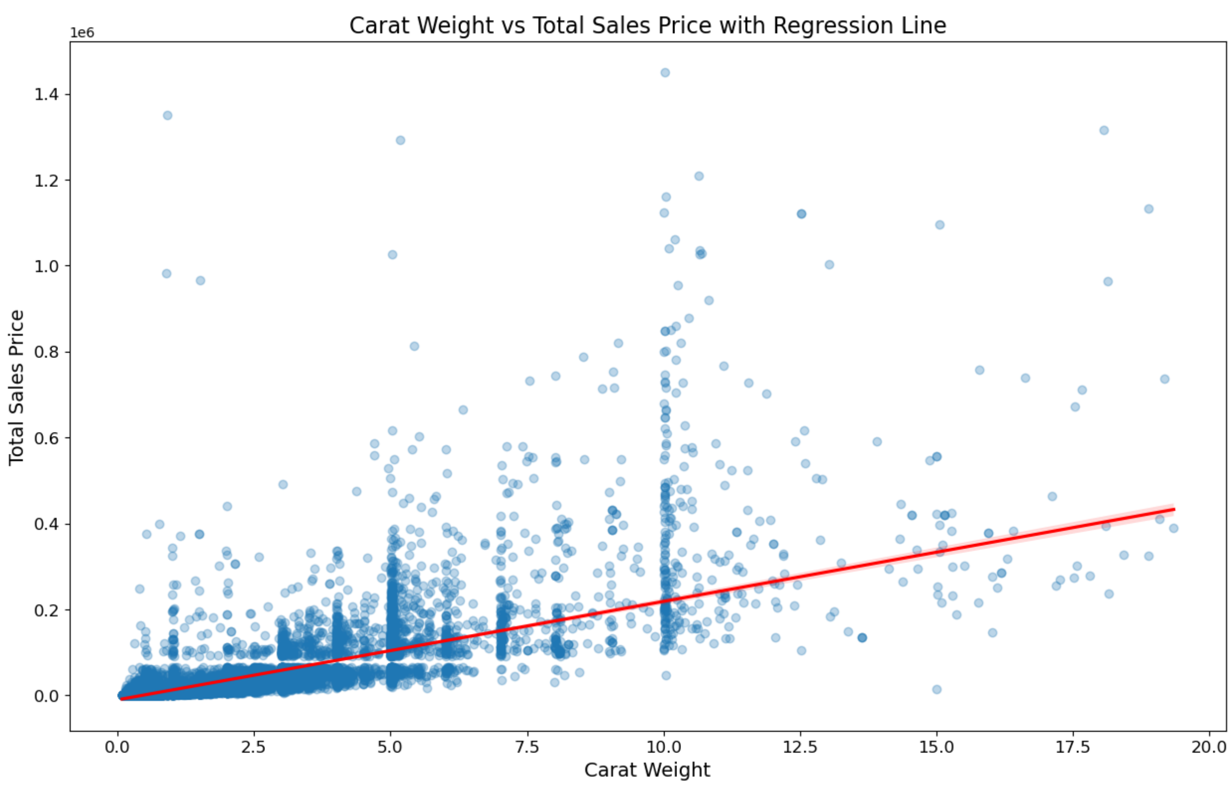
In [ ]:

In [42]:
```
# Creating a horizontal bar plot for the distribution of diamond clarity grades
clarity_counts = diamonds_df['clarity'].value_counts()
plt.figure(figsize=(12, 6))
sns.barplot(x=clarity_counts, y=clarity_counts.index, palette="viridis")
plt.title('Horizontal Bar Plot of Diamond Clarity Distribution', fontsize=16)
plt.xlabel('Frequency', fontsize=14)
plt.ylabel('Clarity Grade', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



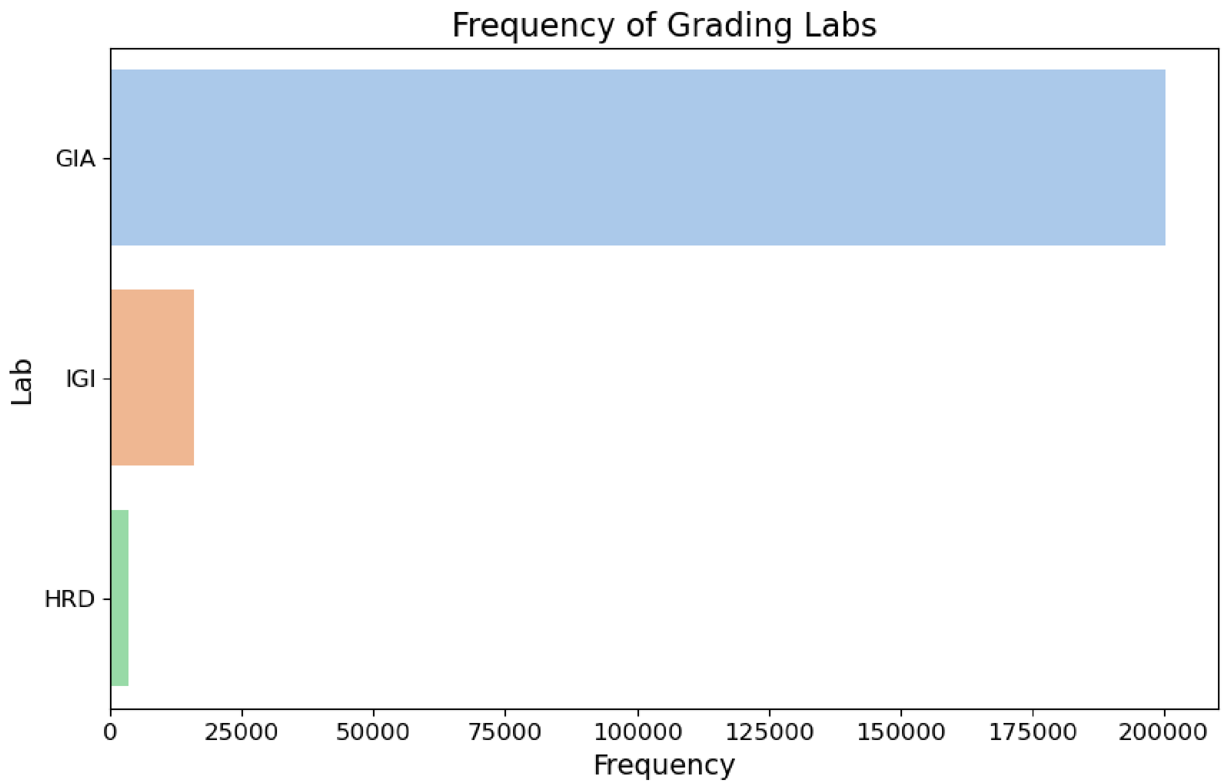
In [ ]:

In [43]:
```
# Scatter plot with regression line for carat weight vs total sales price
plt.figure(figsize=(15, 9))
sns.regplot(data=diamonds_df, x="carat_weight", y="total_sales_price", scatter_kws={'a
plt.title('Carat Weight vs Total Sales Price with Regression Line', fontsize=16)
plt.xlabel('Carat Weight', fontsize=14)
plt.ylabel('Total Sales Price', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



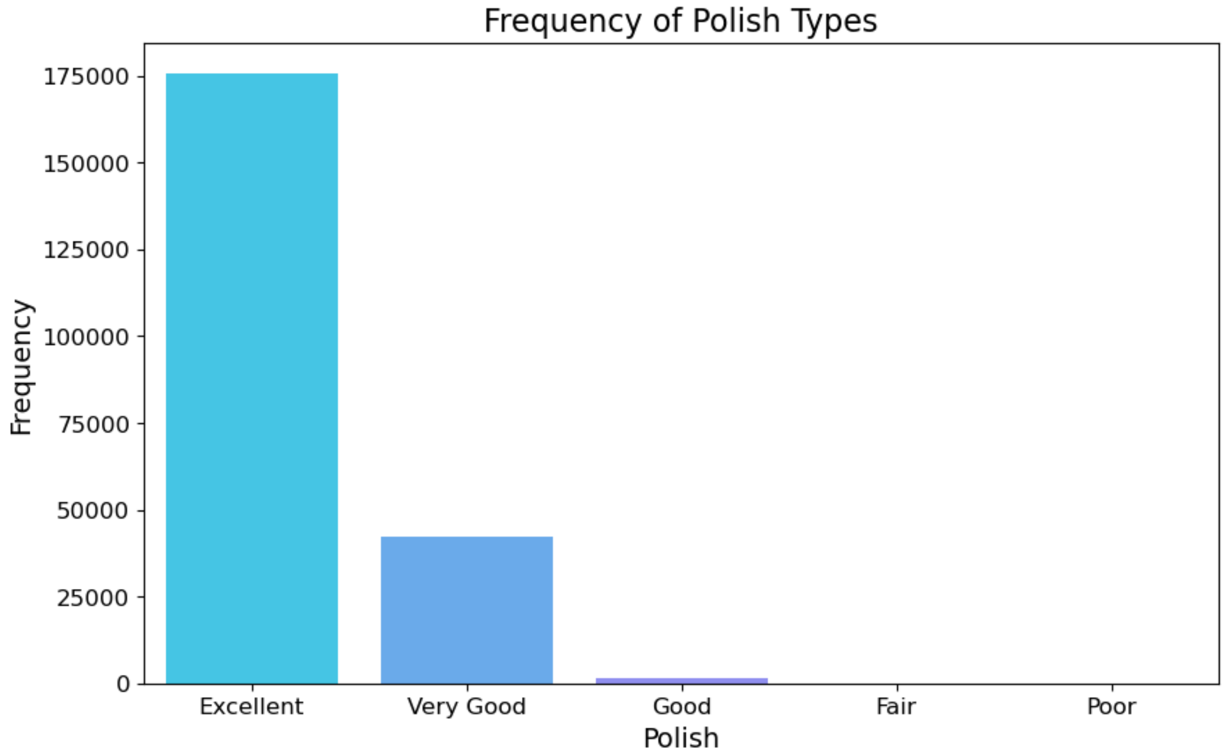
In [ ]:

In [44]:
```
# Grading Lab Analysis with Horizontal Bar Plot
lab_counts = diamonds_df['lab'].value_counts()
plt.figure(figsize=(10, 6))
sns.barplot(y=lab_counts.index, x=lab_counts.values, palette="pastel")
plt.title('Frequency of Grading Labs', fontsize=16)
plt.xlabel('Frequency', fontsize=14)
plt.ylabel('Lab', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



In [ ]:

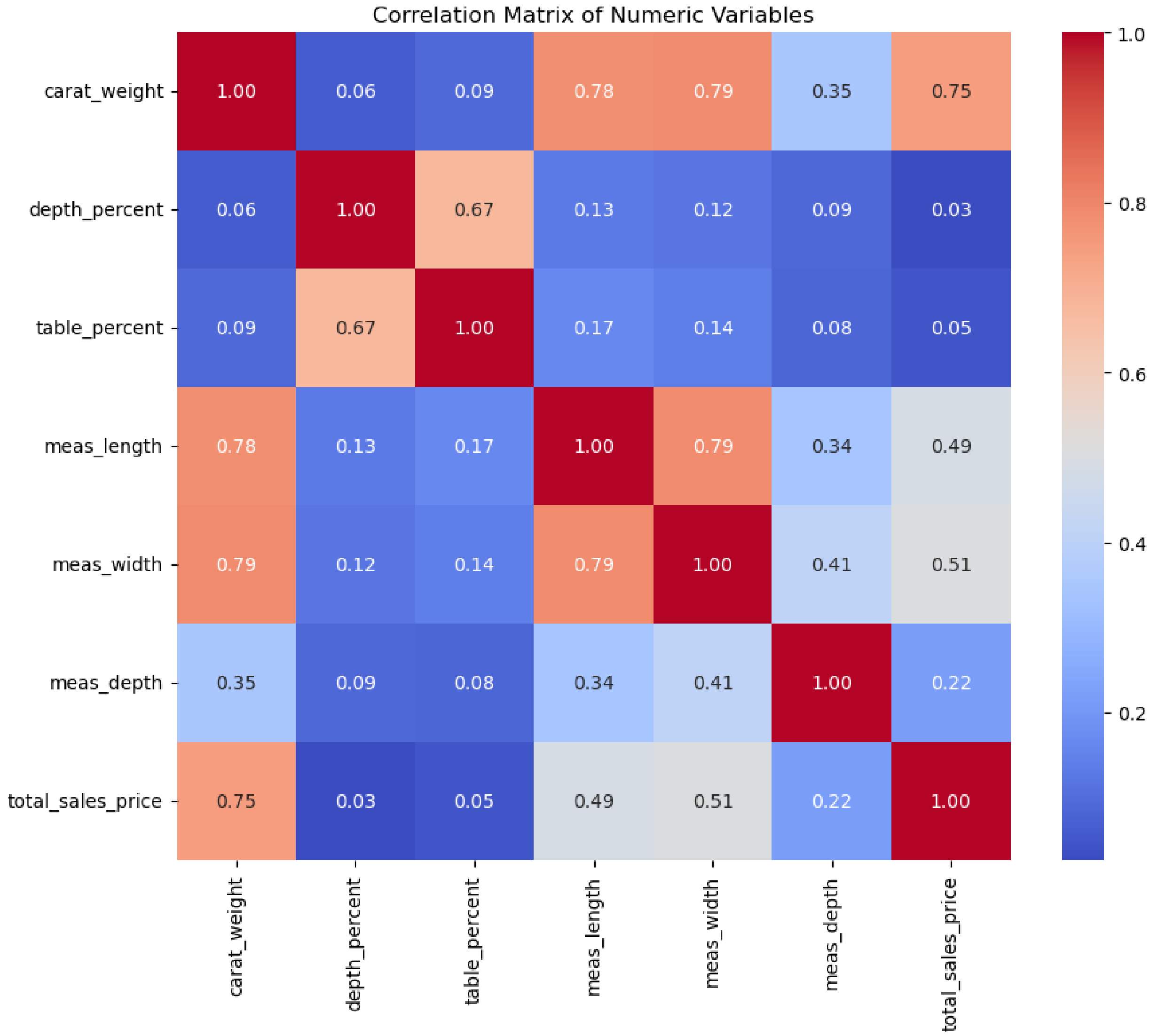
In [45]:
```
# Polish Analysis with Stacked Bar Plot
polish_counts = diamonds_df['polish'].value_counts()
plt.figure(figsize=(10, 6))
sns.barplot(x=polish_counts.index, y=polish_counts.values, palette="cool")
plt.title('Frequency of Polish Types', fontsize=16)
plt.xlabel('Polish', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



In [ ]:

Investigating Correlation and Collinearity

In [48]:
```
numeric_df = diamonds_df[numerical_columns]

# Calculating the correlation matrix
correlation_matrix = numeric_df.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Matrix of Numeric Variables")
plt.show()
```



The analysis reveals several pairs of features in the diamonds dataset that have high correlations, indicating potential collinearity:

Meas Width and Carat Weight: Correlation Coefficient = 0.789

Meas Width and Meas Length: Correlation Coefficient = 0.789

Meas Length and Carat Weight: Correlation Coefficient = 0.783

Total Sales Price and Carat Weight: Correlation Coefficient = 0.746

In [56]:
```
# Initialize the Robust Scaler
robust_scaler = RobustScaler()
numeric_features = diamonds_df.select_dtypes(include=['float64', 'int64']).columns
numeric_features = numeric_features.drop('Unnamed: 0')
# Select numeric features for scaling, excluding the target variable
numeric_features_to_scale = [feature for feature in numeric_features if feature != 'to

# Apply Robust Scaler to the numeric features
diamonds_df_robust_scaled = diamonds_df.copy()
diamonds_df_robust_scaled[numeric_features_to_scale] = robust_scaler.fit_transform(dia
```

```
# Check the first few rows of the scaled data
print(diamonds_df_robust_scaled[numeric_features_to_scale].head())
```

```
   carat_weight  depth_percent  table_percent  meas_length  meas_width  \
0     -0.594203       0.130435       0.333333       -1.105   -1.388489
1     -0.594203      -0.217391       0.333333       -1.110   -1.374101
2     -0.594203      -0.565217       0.333333       -1.090   -1.366906
3     -0.594203      -0.173913       0.333333       -1.100   -1.381295
4     -0.594203       1.086957       0.166667       -1.135   -1.417266

   meas_depth
0   -1.305263
1   -1.315789
2   -1.326316
3   -1.315789
4   -1.273684
```

Applying One Hot Encoding for Categorical Features

In [57]:

```
from sklearn.preprocessing import OneHotEncoder

# Identifying categorical columns
categorical_columns = diamonds_df.select_dtypes(include=['object']).columns

# Applying One-Hot Encoding to categorical variables
one_hot_encoder = OneHotEncoder(sparse=False, drop='first')
encoded_categorical_data = one_hot_encoder.fit_transform(diamonds_df[categorical_colum

# Creating a DataFrame for encoded categorical features
encoded_categorical_df = pd.DataFrame(encoded_categorical_data, columns=one_hot_encode

# Concatenating the encoded categorical features with the scaled numeric features
diamonds_df_preprocessed = pd.concat([diamonds_df_robust_scaled.drop(categorical_colum

# Displaying the first few rows of the preprocessed dataset
diamonds_df_preprocessed.head()
```

Out[57]:

	Unnamed: 0	carat_weight	depth_percent	table_percent	meas_length	meas_width	meas_depth	tota
0	0	-0.594203	0.130435	0.333333	-1.105	-1.388489	-1.305263	
1	1	-0.594203	-0.217391	0.333333	-1.110	-1.374101	-1.315789	
2	2	-0.594203	-0.565217	0.333333	-1.090	-1.366906	-1.326316	
3	3	-0.594203	-0.173913	0.333333	-1.100	-1.381295	-1.315789	
4	4	-0.594203	1.086957	0.166667	-1.135	-1.417266	-1.273684	

5 rows × 137 columns

Splitting and making a baseline Linear Regressor Model

In [58]:

```
from sklearn.model_selection import train_test_split

# Define the features and target variable
X = diamonds_df_preprocessed.drop(['total_sales_price', 'Unnamed: 0'], axis=1)  # Excl
```

```
y = diamonds_df_preprocessed['total_sales_price']

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

In [59]:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Initialize the Linear Regression model
linear_model = LinearRegression()

# Fit the model on the training data
linear_model.fit(X_train, y_train)
```

Out[59]:

```
LinearRegression()
```

In [60]:

```
# Predict on the test set
y_pred = linear_model.predict(X_test)

# Calculate RMSE and R^2
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

print("RMSE:", rmse)
print("R^2:", r2)
```

```
RMSE: 16669.820717825998
R^2: 0.6422636588386184
```

Ridge and Lasso Regression

In [62]:

```
from sklearn.linear_model import Ridge, Lasso
from sklearn.metrics import r2_score

# # Defining the hyperparameters for Ridge and Lasso
# # Typically, these values are selected via cross-validation
# # Here, we are using arbitrary values for demonstration
# alpha_ridge = 1.0  # Regularization strength for Ridge
# alpha_lasso = 0.01 # Regularization strength for Lasso

# Initializing Ridge and Lasso models
ridge_model = Ridge()
lasso_model = Lasso()

# Training Ridge model and making predictions
ridge_model.fit(X_train, y_train)
ridge_pred = ridge_model.predict(X_test)

# Training Lasso model and making predictions
lasso_model.fit(X_train, y_train)
lasso_pred = lasso_model.predict(X_test)

# Calculating R2 score for Ridge and Lasso models
ridge_r2 = r2_score(y_test, ridge_pred)
lasso_r2 = r2_score(y_test, lasso_pred)

ridge_r2, lasso_r2
```

Out[62]:
```
(0.6372745913646023, 0.6413961449649282)
```

Random Forest Regression

In [63]:
```
from sklearn.ensemble import RandomForestRegressor

# Initialize the Random Forest Regressor
random_forest_model = RandomForestRegressor(n_estimators=100, random_state=69)

# Fit the model on the training data
random_forest_model.fit(X_train, y_train)

# Predict on the test set
y_pred_rf = random_forest_model.predict(X_test)

# Calculate RMSE and R^2 for Random Forest
rmse_rf = mean_squared_error(y_test, y_pred_rf, squared=False)
r2_rf = r2_score(y_test, y_pred_rf)

print("Random Forest RMSE:", rmse_rf)
print("Random Forest R^2:", r2_rf)
```

```
Random Forest RMSE: 12134.982791926412
Random Forest R^2: 0.8104256688807343
```

In [64]:
```
# Initializing the models
rf_model = RandomForestRegressor(random_state=42)
xgb_model = XGBRegressor(random_state=42)
catboost_model = CatBoostRegressor(random_state=42, verbose=0)  # verbose=0 to suppres
lgbm_model = LGBMRegressor(random_state=42)
svm_model = SVR()

# Training and evaluating each model
models = [rf_model, xgb_model, catboost_model, lgbm_model, svm_model]
model_names = ['Random Forest', 'XGBoost', 'CatBoost', 'LightGBM', 'SVM']
results = []

for model, name in zip(models, model_names):
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    r2 = r2_score(y_test, predictions)
    mse = mean_squared_error(y_test, predictions)
    results.append((name, r2, mse))

results
```

Out[64]:
```
[('Random Forest', 0.8037195672501383, 152466982.12692732),
 ('XGBoost', 0.867930523413202, 102589108.0642432),
 ('CatBoost', 0.8570497859754068, 111041062.1241278),
 ('LightGBM', 0.8091420320301963, 148254912.5430041),
 ('SVM', 0.04316212723297086, 743253827.1986289)]
```

In [ ]: