Objective

This code focuses on preparing data for regression supervised learning for the price of diamonds. Feature selection and analysis will occur at a later stage.

Variables

Carat: The diamond's carat.

Cut: The type of diamond cut.

Color: The quality of the diamond's color.

Clarity: The clarity indicator for a diamond.

Depth: The distance from the table to the point of the diamond.

Table: The size of the flat edge of the diamonds.

x: Length of the diamond in millimeters.

y: Width of the diamond in millimeters.

z: Depth of the diamond in millimeters.

Price: The overall price of the diamond that we will be predicting.

Data Source: https://www.kaggle.com/datasets/nancyalaswad90/diamonds-prices?select=Diamonds+Prices2022.csv

In [3]:
```python
import pandas as pd
import numpy as np

# Load the data
diamonds = pd.read_csv(
    filepath_or_buffer = "C:\\Users\\brink\\OneDrive\\Desktop\\UCF Notes\\STA 5703\
    dtype = {
        'id': int,
        'carat': float,
        'cut': str,
        'color': str,
        'clarity': str,
        'depth': float,
        'table': float,
        'price': float,
        'x': float,
        'y': float,
        'z': float
```

```
    }
)
```

In [4]:
```
# copy data frame, data prep the copy
diamonds_prepared = diamonds.copy()
```

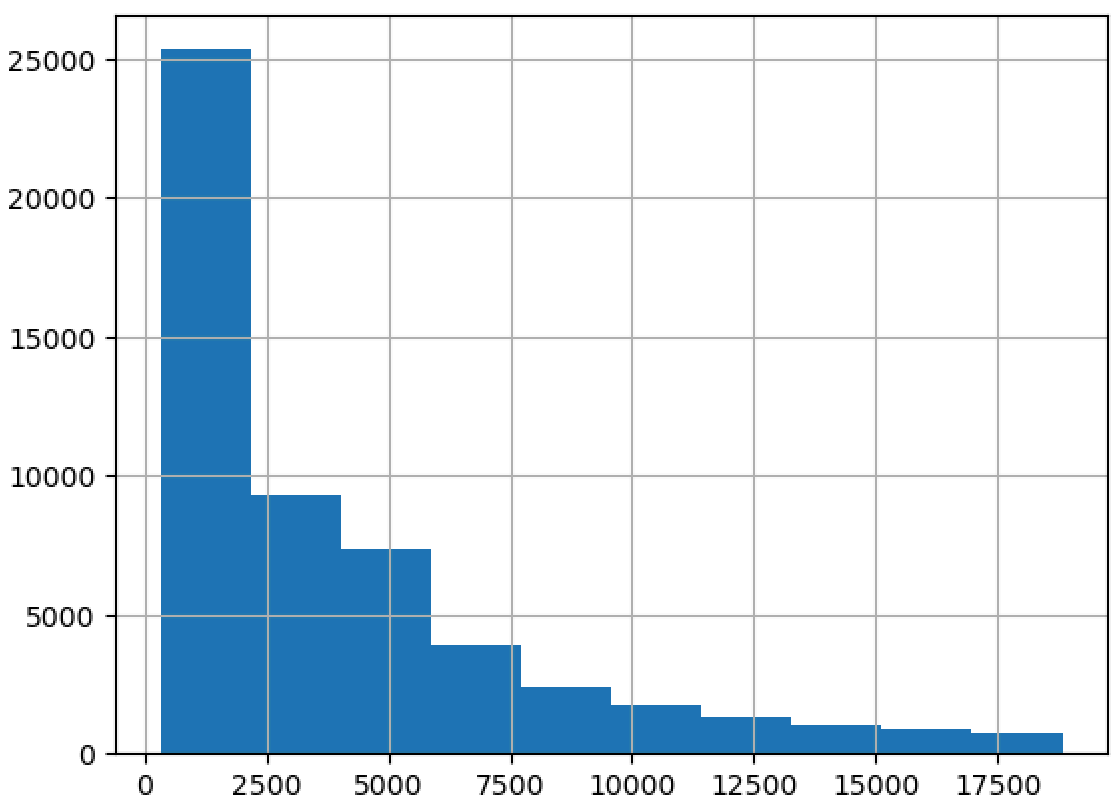
In [5]:
```
diamonds_prepared.head()
```

Out[5]:

	Unnamed: 0	carat	cut	color	clarity	depth	table	price	x	y	z
0	1	0.23	Ideal	E	SI2	61.5	55.0	326.0	3.95	3.98	2.43
1	2	0.21	Premium	E	SI1	59.8	61.0	326.0	3.89	3.84	2.31
2	3	0.23	Good	E	VS1	56.9	65.0	327.0	4.05	4.07	2.31
3	4	0.29	Premium	I	VS2	62.4	58.0	334.0	4.20	4.23	2.63
4	5	0.31	Good	J	SI2	63.3	58.0	335.0	4.34	4.35	2.75

In [6]:
```
#Visualize Target Variable
diamonds['price'].hist()
```

Out[6]: <Axes: >



In [7]:
```
list_categorical = [
    'cut','color','clarity'
]
list_numeric = [
```

```
    'carat','depth','x','y','z','table'
]
```

In [8]:
```
#Categorical Smoothing
for j in list_categorical:
    df_mean = diamonds_prepared.loc[~diamonds['price'].isna()][[j,'price']].groupby
    df_mean['proportion'] = diamonds_prepared.loc[~diamonds['price'].isna()][j].val
    df_mean = df_mean.sort_values('price',ascending = False)
    df_mean['cumulative_descending'] = 1 - np.cumsum(df_mean['proportion'])
    df_mean = df_mean.iloc[::-1]
    df_mean['cumulative_ascending'] = np.cumsum(df_mean['proportion'])
    df_mean['cumulative'] = (df_mean['cumulative_ascending'] + df_mean['cumulative_
    df_mean['bin'] = ''
    df_mean['bin'] = np.where(df_mean['cumulative'] <= 1/3,'1_low',df_mean['bin'])
    df_mean['bin'] = np.where((1/3 <= df_mean['cumulative']) & (df_mean['cumulative
    df_mean['bin'] = np.where(2/3 <= df_mean['cumulative'],'3_high',df_mean['bin'])
    print("-------------------------------------------------------------------------
    print(df_mean[['price','proportion','bin']])
    diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index
    diamonds_prepared[j] = df_mean['bin'].loc[diamonds_prepared[j]].to_list()
```

```
--------------------------------------------------------------------------------
-
                 price  proportion         bin
cut
Ideal      3457.541970    0.399514       1_low
Good       3928.864452    0.090948  2_moderate
Very Good  3981.658529    0.223996  2_moderate
Fair       4358.757764    0.029846      3_high
Premium    4583.992605    0.255696      3_high
--------------------------------------------------------------------------------
-
             price  proportion         bin
color
E      3076.687111    0.181655       1_low
D      3169.954096    0.125596       1_low
F      3724.784868    0.176909  2_moderate
G      3999.135671    0.209332  2_moderate
H      4486.669196    0.153940      3_high
I      5091.874954    0.100514      3_high
J      5323.818020    0.052055      3_high
--------------------------------------------------------------------------------
-
               price  proportion         bin
clarity
VVS1     2523.114637    0.067757       1_low
IF       2864.839106    0.033183       1_low
VVS2     3283.737071    0.093914       1_low
VS1      3839.455391    0.151475       1_low
I1       3924.168691    0.013737  2_moderate
VS2      3924.894119    0.227258  2_moderate
SI1      3995.811357    0.242237      3_high
SI2      5063.028606    0.170439      3_high
```

```
C:\Users\brink\AppData\Local\Temp\ipykernel_6192\2998296141.py:16: FutureWarning: Ch
ainedAssignmentError: behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in certain c
ases, but when using Copy-on-Write (which will become the default behaviour in panda
s 3.0) this will never work to update the original DataFrame or Series, because the
intermediate object on which we are setting values will behave as a copy.
A typical example is when you are setting values in a column of a DataFrame, like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment in a si
ngle step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy

  diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index[n
p.argmin(abs(df_mean['cumulative'] - 0.5))]
C:\Users\brink\AppData\Local\Temp\ipykernel_6192\2998296141.py:16: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index[n
p.argmin(abs(df_mean['cumulative'] - 0.5))]
C:\Users\brink\AppData\Local\Temp\ipykernel_6192\2998296141.py:16: FutureWarning: Ch
ainedAssignmentError: behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in certain c
ases, but when using Copy-on-Write (which will become the default behaviour in panda
s 3.0) this will never work to update the original DataFrame or Series, because the
intermediate object on which we are setting values will behave as a copy.
A typical example is when you are setting values in a column of a DataFrame, like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment in a si
ngle step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy

  diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index[n
p.argmin(abs(df_mean['cumulative'] - 0.5))]
C:\Users\brink\AppData\Local\Temp\ipykernel_6192\2998296141.py:16: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index[n
p.argmin(abs(df_mean['cumulative'] - 0.5))]
C:\Users\brink\AppData\Local\Temp\ipykernel_6192\2998296141.py:16: FutureWarning: Ch
ainedAssignmentError: behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in certain c
ases, but when using Copy-on-Write (which will become the default behaviour in panda
```

```
s 3.0) this will never work to update the original DataFrame or Series, because the
intermediate object on which we are setting values will behave as a copy.
A typical example is when you are setting values in a column of a DataFrame, like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment in a si
ngle step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy

  diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index[n
p.argmin(abs(df_mean['cumulative'] - 0.5))]
C:\Users\brink\AppData\Local\Temp\ipykernel_6192\2998296141.py:16: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  diamonds_prepared[j][~diamonds_prepared[j].isin(df_mean.index)] = df_mean.index[n
p.argmin(abs(df_mean['cumulative'] - 0.5))]
```

In [9]:

```
#Create dummy variables
diamonds_prepared[list_categorical]
diamonds_prepared = pd.get_dummies(
    data = diamonds_prepared,
    columns = list_categorical,
    dtype = float
)
```

In [10]:

```
diamonds_prepared.head()
```

Out[10]:

	Unnamed: 0	carat	depth	table	price	x	y	z	cut_1_low	cut_2_moderate	cut_
0	1	0.23	61.5	55.0	326.0	3.95	3.98	2.43	1.0	0.0	
1	2	0.21	59.8	61.0	326.0	3.89	3.84	2.31	0.0	0.0	
2	3	0.23	56.9	65.0	327.0	4.05	4.07	2.31	0.0	1.0	
3	4	0.29	62.4	58.0	334.0	4.20	4.23	2.63	0.0	0.0	
4	5	0.31	63.3	58.0	335.0	4.34	4.35	2.75	0.0	1.0	

In [11]:

```
from sklearn.preprocessing import PowerTransformer, StandardScaler

# Initialize the PowerTransformer for Yeo-Johnson transformation
PowerTransformer_yeo_johnson = PowerTransformer(method='yeo-johnson').fit(
    X = diamonds_prepared[list_numeric]
)

# transform the data
```

```
diamonds_prepared[list_numeric] = PowerTransformer_yeo_johnson.transform(
    X = diamonds_prepared[list_numeric]
)

# Initialize the StandardScaler for centering and scaling
StandardScaler_mean_with_std = StandardScaler().fit(
    X = diamonds_prepared[list_numeric],
    y = diamonds_prepared['price']
)

# Fit and scale the transformed data
diamonds_prepared[list_numeric] = StandardScaler_mean_with_std.transform(
    X = diamonds_prepared[list_numeric]
)
```

In [12]:
```
#Check transformation
diamonds_prepared[list_numeric].describe().loc[['mean','std']].round(1)
```

Out[12]:

	carat	depth	x	y	z	table
mean	0.0	-0.0	0.0	-0.0	-0.0	-0.0
std	1.0	1.0	1.0	1.0	1.0	1.0

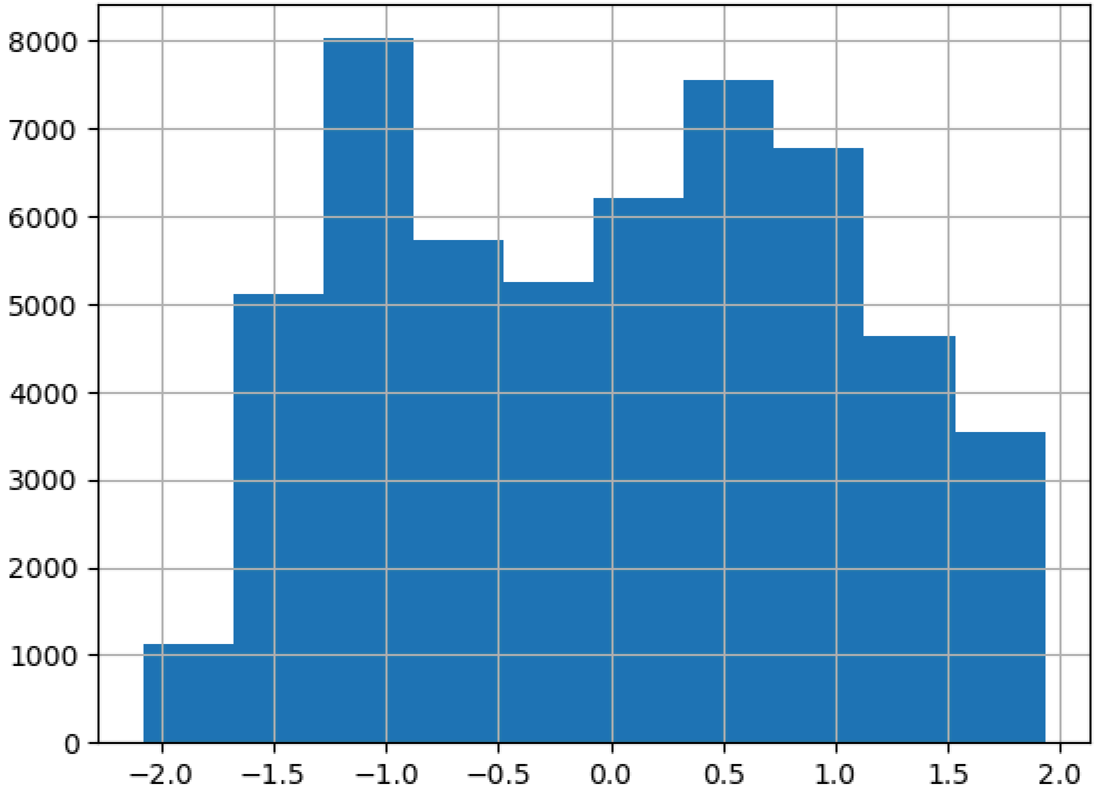
In [13]:
```
#Given the skewed nature of the target variable, we will transform the target varia
PowerTransformer_Price = PowerTransformer(method='yeo-johnson').fit(
    X = diamonds_prepared.loc[:, ['price']]
)
print(PowerTransformer_Price.lambdas_)
```

```
[-0.067386]
```

In [14]:
```
diamonds_prepared['price'] = PowerTransformer_Price.transform(
    X = diamonds_prepared.loc[:, ['price']]
)
```

In [15]:
```
diamonds_prepared['price'].hist()
```

Out[15]: <Axes: >



In [16]:
```
#Check that the transformation was a success.
np.mean(np.abs([
    j for k in PowerTransformer_Price.inverse_transform(
        X = diamonds_prepared.loc[:, ['price']]
   ) for j in k
] - diamonds['price']))
```

Out[16]: 3.5965288343262033e-12