X. ANNEXURE

Dataset: https://www.kaggle.com/datasets/austinreese/craigslist-carstrucks-data

In [ ]:
```
# Import libraries and loading the csv file
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
import missingno as msno
import plotly.express as px
import re
import sweetviz as sv
from kmodes.kprototypes import KPrototypes
import scipy.stats as stats
from scipy.stats import chi2
vehicles_data_initial = pd.read_csv(r"C:\Users\91886\OneDrive\QMUL Masterclass\vehicles.csv")
```

In [ ]:
```
vehicles_data = vehicles_data_initial.copy()
sample = vehicles_data_initial.copy()
```

In [ ]:
```
vehicles_data[vehicles_data["price"] < 500]
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['price'] < 500].index, inplace = True)
```

In [ ]:
```
vehicles_data.drop(['url', 'region_url','image_url','county','VIN','size'], axis=1,inplace = True)
```

In [ ]:
```
vehicles_data.shape
```

In [ ]:
```
msno.matrix(vehicles_data, color = (0.5, 0.5, 0.5))
```

In [ ]:
```
percent_missing = vehicles_data.isnull().sum() * 100 / len(vehicles_data)
missing_value_df = pd.DataFrame({'column_name': vehicles_data.columns,
                                 'percent_missing': percent_missing})
```

In [ ]:
```
missing_value_df
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['year'] < 1990].index, inplace = True)
```

In [ ]:
```
vehicles_data.dropna(axis = 0,thresh=17,inplace = True)
```

In [ ]:
```
vehicles_data.dropna(subset=['manufacturer', 'model','odometer','title_status','fuel',
                             'transmission','lat','long','year','description'], inplace = True)
```

In [ ]:
```
msno.matrix(vehicles_data, color = (0.5, 0.5, 0.5))
```

In [ ]:
```
percent_missing = vehicles_data.isnull().sum() * 100 / len(vehicles_data)
missing_value_df = pd.DataFrame({'column_name': vehicles_data.columns,
                                 'percent_missing': percent_missing})
missing value df
```

In [ ]:
```
vehicles_data.shape
```

In [ ]:
```
# To find records with fuel as electric and cylinders as NULL
condition1 = pd.isna(vehicles_data_initial["cylinders"])
condition2 = vehicles data["fuel"] == 'electric'
```

In [ ]:
```
# Use the & operator to combine the conditions
filtered df = vehicles data[condition1 & condition2]
```

In [ ]:
```
# imputing electric car cylinders to 0
vehicles_data['cylinders'] = np.where(pd.isna(vehicles_data["cylinders"]) & (vehicles_data["fuel"] == 'electric')
                                      ,0,vehicles data['cylinders'])
```

In [ ]:
```
# Data correction for fuel type as electric and number of cylinders as non zero
vehicles_data[vehicles_data["fuel"] == 'electric']["cylinders"].value_counts()
vehicles_data['cylinders'] = np.where((vehicles_data['fuel'] == 'electric') & (vehicles_data['cylinders'] != 0),
                                      0, vehicles data['cylinders'])
```

In [ ]:
```
# Dropping Null values from Type

vehicles data.dropna(subset=['type'], inplace = True)
```

In [ ]:
```
vehicles_data.shape
```

In [ ]:
```
percent_missing = vehicles_data.isnull().sum() * 100 / len(vehicles_data)
missing_value_df = pd.DataFrame({'column_name': vehicles_data.columns,
                                 'percent_missing': percent_missing})
```

In [ ]:
```
missing_value_df
```

msno.matrix(vehicles_data, color = (0.5, 0.5, 0.5))

Handing outliers

In [ ]:
```
df = vehicles_data[vehicles_data['year'] == 2007]["price"]
print(df.max())
```

In [ ]:
```
df = vehicles_data[vehicles_data['year'] == 1999]["price"]
print(df.max())
```

In [ ]:
```
filtered_df = vehicles_data[vehicles_data["price"] == 3736928711]
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 3736928711].index, inplace = True)
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 1111111111].index, inplace = True)
```

In [ ]:
```
df = vehicles_data[vehicles_data['year'] == 2015]["price"]
print(df.max())
```

In [ ]:
```
# removing Outlier prices
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 3736928711].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 1111111111].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 123456789].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 17000000].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 6995495].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 2000000].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 1234567].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 990000].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 135008900].index, inplace = True)
```

In [ ]:
```
fig = px.scatter(vehicles_data, x = 'odometer', y = 'price', color = 'fuel')
fig.show()
```

In [ ]:
```
# removing odometer > 5000000
vehicles_data.drop(vehicles_data[vehicles_data['odometer'] >= 5000000].index,inplace = True)
```

In [ ]:
```
fig = px.scatter(vehicles_data, x = 'odometer', y = 'price', color = 'fuel')
fig.show()
```

In [ ]:
```
# copy the data for analysis to find correlation for non missing cylinder values
vehicles_data_cyldrop = vehicles_data.copy()
```

In [ ]:
```
vehicles_data_cyldrop.dropna(subset=['cylinders','drive','condition','paint_color'], inplace = True)
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'cylinders', y = 'price', color = 'cylinders')
fig.show()
```

In [ ]:
```
# replace the huge name of model with a short value for a better graph
vehicles_data_cyldrop['model'] = np.where(vehicles_data_cyldrop['model'].str.contains('impreza sedan premium')
                                          , 'impreza', vehicles_data_cyldrop['model'])
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'paint_color', y = 'price', color = 'manufacturer')
fig.show()
```

In [ ]:
```
filtered_df = vehicles_data[vehicles_data['paint_color'].isnull()]
```

Null values analysis on basis of Description

In [ ]:
```
def extract_color(expression):
    pattern = r"Color: ([A-Za-z]+)"
    color = re.search(pattern, expression)
    if color:
        return color.group(1)
    return None
```

In [ ]:
```
filtered_df["color"] = filtered_df["description"].apply(extract_color)
```

In [ ]:
```
filtered_df["color"].value_counts()
```

In [ ]:
```
vehicles_data.isna().sum()
```

In [ ]:
```
vehicles_data_paint = vehicles_data.copy()
```

In [ ]:
```
def extract_color(expression):
    pattern = r"Color: ([A-Za-z]+)"
    color = re.search(pattern, expression)
    if color:
        return color.group(1)
    return None
```

In [ ]:
```
vehicles_data_paint["paint_color"] = vehicles_data_paint["paint_color"].where(~vehicles_data_paint["paint_color"].isnull(),
                                                                              vehicles_data_paint["description"].apply(extract_color))
```

In [ ]:
```
vehicles_data_paint["paint_color"].unique()
```

From Drive column

In [ ]:
```
def extract_drive_type(expression):
    pattern = r"Drive: ([A-Za-z0-9]+)"
    drive_type = re.search(pattern, expression)
    if drive_type:
        return drive_type.group(1)
    return None
```

In [ ]:
```
vehicles_data_paint["drive"] = vehicles_data_paint["drive"].where(~vehicles_data_paint["drive"].isnull(),
                                                                  vehicles_data_paint["description"].apply(extract_drive_type))
```

In [ ]:
```
vehicles_data_paint["drive"].value_counts()
```

In [ ]:
```
vehicles_data_paint.isna().sum()
```

In [ ]:
```
vehicles_data_cyldrop

fig = px.scatter(vehicles_data_cyldrop, x = 'drive', y = 'price', color = 'cylinders')
fig.show()
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'type', y = 'price', color = 'cylinders')
fig.show()
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'paint_color', y = 'price', color = 'model')
fig.show()
```

In [ ]:
```
# Import libraries and loading the csv file
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
import missingno as msno
import plotly.express as px
import sweetviz as sv
from kmodes.kprototypes import KPrototypes
import scipy.stats as stats
from scipy.stats import chi2
vehicles data initial = pd.read csv(r"C:\Users\91886\OneDrive\QMUL Masterclass\vehicles.csv")
```

In [ ]:
```
vehicles_data = vehicles_data_initial.copy()
sample = vehicles data initial.copy()
```

In [ ]:
```
vehicles_data[vehicles_data["price"] < 500]
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['price'] < 500].index, inplace = True)
```

In [ ]:
```
vehicles_data.drop(['url', 'region_url','image_url','county','VIN','size'], axis=1,inplace = True)
```

In [ ]:
```
vehicles_data.shape
```

In [ ]:
```
msno.matrix(vehicles_data, color = (0.5, 0.5, 0.5))
```

In [ ]:
```
percent_missing = vehicles_data.isnull().sum() * 100 / len(vehicles_data)
missing_value_df = pd.DataFrame({'column_name': vehicles_data.columns,
                                 'percent missing': percent missing})
```

In [ ]:
```
missing_value_df
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['year'] < 1990].index, inplace = True)
```

In [ ]:
```
vehicles_data.dropna(axis = 0,thresh=17,inplace = True)
```

In [ ]:
```
vehicles_data.dropna(subset=['manufacturer', 'model','odometer','title_status','fuel',
                             'transmission','lat','long','year','description'], inplace = True)
```

In [ ]:
```
msno.matrix(vehicles_data, color = (0.5, 0.5, 0.5))
```

In [ ]:
```
percent_missing = vehicles_data.isnull().sum() * 100 / len(vehicles_data)
missing_value_df = pd.DataFrame({'column_name': vehicles_data.columns,
                                 'percent_missing': percent_missing})
missing value df
```

In [ ]:
```
vehicles_data.shape
```

In [ ]:
```
# To find records with fuel as electric and cylinders as NULL
condition1 = pd.isna(vehicles_data_initial["cylinders"])
condition2 = vehicles data["fuel"] == 'electric'
```

In [ ]:
```
# Use the & operator to combine the conditions
filtered df = vehicles data[condition1 & condition2]
```

In [ ]:
```
# imputing electric car cylinders to 0
vehicles_data['cylinders'] = np.where(pd.isna(vehicles_data["cylinders"]) & (vehicles_data["fuel"] == 'electric')
                                      ,0,vehicles data['cylinders'])
```

In [ ]:
```
# Data correction for fuel type as electric and number of cylinders as non zero
vehicles_data[vehicles_data["fuel"] == 'electric']["cylinders"].value_counts()
vehicles_data['cylinders'] = np.where((vehicles_data['fuel'] == 'electric') & (vehicles_data['cylinders'] != 0),
                                      0, vehicles_data['cylinders'])
```

In [ ]:
```
# Dropping Null values from Type

vehicles_data.dropna(subset=['type'], inplace = True)
```

In [ ]:
```
vehicles_data.shape
```

In [ ]:
```
percent_missing = vehicles_data.isnull().sum() * 100 / len(vehicles_data)
missing_value_df = pd.DataFrame({'column_name': vehicles_data.columns,
                                 'percent missing': percent_missing})
```

In [ ]:
```
missing_value_df
```

In [ ]:
```
msno.matrix(vehicles_data, color = (0.5, 0.5, 0.5))
```

Handing outliers

In [ ]:
```
df = vehicles_data[vehicles_data['year'] == 2007]["price"]
print(df.max())
```

In [ ]:
```
df = vehicles_data[vehicles_data['year'] == 1999]["price"]
print(df.max())
```

In [ ]:
```
filtered_df = vehicles_data[vehicles_data["price"] == 3736928711]
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 3736928711].index, inplace = True)
```

In [ ]:
```
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 1111111111].index, inplace = True)
```

In [ ]:
```
df = vehicles_data[vehicles_data['year'] == 2015]["price"]
print(df.max())
```

In [ ]:
```
# removing Outlier prices
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 3736928711].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 1111111111].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 123456789].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 17000000].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 6995495].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 2000000].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 1234567].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 990000].index, inplace = True)
vehicles_data.drop(vehicles_data[vehicles_data['price'] == 135008900].index, inplace = True)
```

In [ ]:
```
fig = px.scatter(vehicles_data, x = 'year', y = 'price', color = 'fuel')
fig.show()
```

In [ ]:
```
fig = px.scatter(vehicles_data, x = 'odometer', y = 'price', color = 'fuel')
fig.show()
```

In [ ]:
```
# removing odometer > 5000000
vehicles_data.drop(vehicles_data[vehicles_data['odometer'] >= 5000000].index,inplace = True)
```

In [ ]:
```
fig = px.scatter(vehicles_data, x = 'odometer', y = 'price', color = 'fuel')
fig.show()
```

In [ ]:
```
# copy the data for analysis to find correlation for non missing cylinder values
vehicles_data_cyldrop = vehicles_data.copy()
```

In [ ]:
```

vehicles_data_cyldrop.dropna(subset=['cylinders','drive','condition','paint color'], inplace = True)
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'cylinders', y = 'price', color = 'cylinders')
fig.show()
```

In [ ]:
```
# replace the huge name of model with a short value for a better graph
vehicles_data_cyldrop['model'] = np.where(vehicles_data_cyldrop['model'].str.contains('impreza sedan premium')
                                          , 'impreza', vehicles_data_cyldrop['model'])
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'paint_color', y = 'price', color = 'manufacturer')
fig.show()
```

In [ ]:
```
filtered_df = vehicles_data[vehicles_data['paint_color'].isnull()]
```

Null values analysis on basis of Description

In [ ]:
```
def extract_color(expression):
    pattern = r"Color: ([A-Za-z]+)"
    color = re.search(pattern, expression)
    if color:
        return color.group(1)
    return None
```

In [ ]:
```
filtered_df["color"] = filtered_df["description"].apply(extract_color)
```

In [ ]:
```
filtered_df["color"].value_counts()
```

In [ ]:
```
vehicles_data.isna().sum()
```

In [ ]:
```
vehicles_data_paint = vehicles_data.copy()
```

In [ ]:
```
def extract_color(expression):
    pattern = r"Color: ([A-Za-z]+)"
    color = re.search(pattern, expression)
    if color:
        return color.group(1)
    return None
```

In [ ]:
```
vehicles_data_paint["paint_color"] = vehicles_data_paint["paint_color"].where(~vehicles_data_paint["paint_color"].isnull(),
                                                                               vehicles_data_paint["description"].apply(extract_color))
```

In [ ]:
```
vehicles_data_paint["paint_color"].unique()
```

From Drive column

In [ ]:
```
def extract_drive_type(expression):
    pattern = r"Drive: ([A-Za-z0-9]+)"
    drive_type = re.search(pattern, expression)
    if drive_type:
        return drive_type.group(1)
    return None
```

In [ ]:
```
vehicles_data_paint["drive"] = vehicles_data_paint["drive"].where(~vehicles_data_paint["drive"].isnull(),
                                                                  vehicles_data_paint["description"].apply(extract_drive_type))
```

In [ ]:
```
vehicles_data_paint["drive"].value_counts()
```

In [ ]:
```
vehicles_data_paint.isna().sum()
```

In [ ]:
```
vehicles_data_cyldrop

fig = px.scatter(vehicles_data_cyldrop, x = 'drive', y = 'price', color = 'cylinders')
fig.show()
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'type', y = 'price', color = 'cylinders')
fig.show()
```

In [ ]:
```
fig = px.scatter(vehicles_data_cyldrop, x = 'paint_color', y = 'price', color = 'model')
fig.show()
```

In [ ]:
```
# Import libraries and loading the csv file
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import string
import nltk
nltk.download('stopwords')
nltk.download('wordnet')
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('omw-1.4')
from sklearn.pipeline import Pipeline
from sklearn.base import TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import make_column_selector
```

In [ ]:
```
vehicles_clean = pd.read_csv(r"C:\Users\91886\OneDrive\QMUL Masterclass\vehicles_initialdatacleaning.csv")
```

In [ ]:
```
vehicles_clean = pd.DataFrame(vehicles_clean)
```

In [ ]:
```
vehicles_clean = vehicles_clean.head(5000)
```

In [ ]:
```
vehicles_clean['description'] = vehicles_clean['description'].astype('string')
# Replace 'other' with 1 and strip 'cylinders' string from other values
vehicles_clean['cylinders'] = vehicles_clean['cylinders'].str.replace('other', '1').str.rstrip('cylinders').str.strip()
# Convert to float data type and replace '<NA>' values with NaN
vehicles_clean['cylinders'] = pd.to_numeric(vehicles_clean['cylinders'], errors='coerce').astype(float)
# drop model and posting_date column for encoding
vehicles_clean.drop(['condition','id','posting date','model'], axis=1,inplace = True)
```

In [ ]:
```
from sklearn.model_selection import train_test_split
import pandas as pd

# define the features and target variables
X = vehicles_clean.drop('price', axis=1)
y = vehicles_clean['price']

# divide the data into train, validation, and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42)

# display the shapes of the resulting datasets
print(f"Training set shape: {X_train.shape}, {y_train.shape}")
print(f"Validation set shape: {X_val.shape}, {y_val.shape}")
print(f"Test set shape: {X_test.shape}, {y_test.shape}")
```

In [ ]:
```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import string
import numpy as np
from sklearn.base import TransformerMixin

class TokenizerTransformer(TransformerMixin):
    def transform(self, X, **transform_params):
        # Convert to lowercase
        X = X.apply(lambda x: x.lower())
        # Tokenize into words
        X = X.apply(lambda x: word_tokenize(x)[:3500])  # Limit tokens to 3500
        # Remove stop words
        stop_words = stopwords.words('english')
        X = X.apply(lambda x: [word for word in x if word not in stop_words])
        # Lemmatize words using WordNetLemmatizer
        lemmatizer = WordNetLemmatizer()
        X = X.apply(lambda x: [lemmatizer.lemmatize(word) for word in x])
        # Remove punctuation
        X = X.apply(lambda x: [word for word in x if word not in string.punctuation])
        # Return tokenized text
        return X

    def fit(self, X, y=None, **fit_params):
        return self
```

In [ ]:
```
# Define pipeline with for tokenization
token_pipeline = Pipeline([
    ('tokenizer', TokenizerTransformer())
])
```

In [ ]:
```
X_train['description'] = token_pipeline.fit_transform(X_train['description'])
X_val['description'] = token_pipeline.transform(X_val['description'])
X_test['description'] = token_pipeline.transform(X_test['description'])
```

In [ ]:
```
# vehicles_clean['description'] = token_pipeline.fit_transform(vehicles_clean['description'])
```

Cylinder

In [ ]:
```
# Define predefined lists
# Define the list of valid cylinders
cylinder_list = ['i2','i3','i4','i5','i6','i8','i10','i12',
                 'v2','v3','v4','v5','v6','v8','v10','v12',
                 '2cylinder','3cylinder','4cylinder','5cylinder','6cylinder','8cylinder','10cylinder','12cylinder',
                 '2cylinders','3cylinders','4cylinders','5cylinders','6cylinders','8cylinders','10cylinders',
                '12cylinders']
```

In [ ]:
```
class CylindersCleaning(BaseEstimator, TransformerMixin):
# ReplaceNaNWithCylinders
    def__init__(self, cylinder_list):
        self.cylinder_list = cylinder_list

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
# replace_nan_with_cylinder
        def cylinders_cleaning(row):
            cylinders = row['cylinders']
            desc = row['description']
            if pd.isnull(cylinders):
                for c in self.cylinder_list:
                    if c in desc:
                        stripped_c = c.strip('ivcylinders')
                        try:
                            cylinders = float(stripped_c)
                        except ValueError:
                            pass
            row['cylinders'] = cylinders
            return row

        X = X.apply(cylinders_cleaning, axis=1)
        return X
```

In [ ]:
```
cyl_pipeline = Pipeline([
    ('cylinders_cleaning', CylindersCleaning(cylinder_list)),
])
```

In [ ]:
```
X_train_t = cyl_pipeline.fit_transform(X_train)
```

In [ ]:
```
from sklearn.base import BaseEstimator, TransformerMixin

class ColumnSelector(BaseEstimator, TransformerMixin):
    '''select specific columns of a given dataset'''
    def__init__(self, subset):
        self.subset = subset

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.loc[:, self.subset]
```

In [ ]:
```
cylclean_pipeline = Pipeline(steps=[('replace_cylinders', CylindersCleaning(cylinder_list)),
                           ('ct', ColumnTransformer(transformers=[('imputer', SimpleImputer(strategy='mean'),
                                                                   ['cylinders'])],remainder='passthrough'))])
```

In [ ]:
```
X_train['cylinders'].isna().sum()
```

In [ ]:
```
X_train_t = cylclean_pipeline.fit_transform(X_train)
# X_val_t = cylclean_pipeline.transform(X_val)
# X_test_t = cylclean_pipeline.transform(X_test)
```

In [ ]:
```
X_train_t
```

In [ ]:
```
X_train = pd.DataFrame(X_train_t, columns=X_train.columns)
# X_val = pd.DataFrame(X_val_t, columns=X_val.columns)
# X_test = pd.DataFrame(X_test_t, columns=X_test.columns)
```

In [ ]:
```
X_train['cylinders'].isna().sum()
X_val['cylinders'].isna().sum()
X_test['cylinders'].isna().sum()
```

In [ ]:
```
X_train
```

Drive

In [ ]:
```python
# split drive
class SplitDrive(TransformerMixin):
    def transform(self, X):
        X_new = []
        for row in X:
            new_row = []
            for val in row:
                if 'drive' in val:
                    split_vals = val.split('drive')
                    for i in range(len(split_vals)):
                        if i == 0:
                            new_row.append(split_vals[i])
                        elif i == len(split_vals) - 1:
                            if split_vals[i] != '':
                                if new_row[-1] == '':
                                    new_row.pop()
                                new_row.append('drive')
                                new_row.append(split_vals[i])
                            else:
                                new_row.append('drive')
                        elif split_vals[i] != '':
                            if new_row[-1] == '':
                                new_row.pop()
                            new_row.extend(['drive', split_vals[i]])
                else:
                    new_row.append(val)
            X_new.append(new_row)
        return X_new

    def fit(self, X, y=None, **fit_params):
        return self
```

In [ ]:
```python
# First level of cleaning - check for 2 drive occurences
class DriveImputer(BaseEstimator, TransformerMixin):
    def__init__(self):
        self.mapping_dict = {'two wheel': 'rwd', 'all wheel': '4wd', '2 wheel': 'rwd', '4 wheel': '4wd','four wheel': '4wd',
                             'awd':'4wd','4x4':'4wd','xdrive':'4wd','quattro':'4wd'}
        self.drive_master = ['rwd','4wd','awd','xdrive','4x4','4matic','fwd','quattro']

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        nan_rows = X['drive'].isnull()
        X.loc[nan_rows, 'drive'] = X.loc[nan_rows, 'description'].apply(lambda x: self.get_drive(x))
        X['drive'] = X['drive'].map(self.mapping_dict).fillna(X['drive'])
        X['drive'] = X['drive'].apply(lambda x: self.check_drive(x))
        return X

    def get_drive(self, description):
        drive_idxs = [i for i, x in enumerate(description) if x == 'drive']
        if len(drive_idxs) >= 2:
            start_idx = drive_idxs[0]
            end_idx = drive_idxs[1]
            drive = ' '.join(description[start_idx+1:end_idx]).lower()
            return drive
        else:
            return np.nan

    def check_drive(self, drive):
        if drive in self.drive_master:
            return drive
        else:
            return np.nan
```

In [ ]:
```
# Second level of cleaning - check for first drive occurence


class DriveTransformer(TransformerMixin):

    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        # list of possible drive values
        drive_master = ['rwd', '4wd', 'awd', 'xdrive', '4x4', '4matic', 'fwd','awdtransmission','quattro']
        # dictionary mapping common drive phrases to standard values
        mapping_dict = {'two wheel': 'rwd', 'all wheel': '4wd', '2 wheel': 'rwd',
                        '4 wheel': '4wd', 'four wheel': '4wd', 'awd': '4wd','awdtransmission':'4wd',
                       '4x4':'4wd','4x4':'4wd','xdrive':'4wd','quattro':'4wd'}

        # loop through the rows of the dataframe
        for i, row in X.iterrows():
            # check if the 'drive' value is NaN
            if pd.isna(row['drive']):
                # loop through the 'description' list to find the first occurrence of 'drive'
                if 'drive' in row['description']:
                    j = row['description'].index('drive')
                    # if 'drive' is found, replace the NaN value with the next non-empty token in the list
                    for k in range(j+1, len(row['description'])):
                        if row['description'][k] != '':
                            # check if the token is in the drive_master list
                            if row['description'][k] in drive_master:
                                # map the token to the standard value using the mapping_dict
                                X.at[i, 'drive'] = mapping_dict.get(row['description'][k], row['description'][k])
                            break
        return X
```

In [ ]:
```
from transformers import YearTransformer
year_transformer = YearTransformer()
drive_imputer = DriveImputer()
vehicles_clean = year_transformer.fit_transform(vehicles_clean)
vehicles_clean = drive_imputer.fit_transform(vehicles_clean)
clean_data = pipeline.fit_transform(vehicles_clean)
```

In [ ]:
```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('split_drive', SplitDrive()),
    ('impute_drive', DriveImputer()),
    ('transform_drive', DriveTransformer())
])
```

In [ ]:
```
class DriveImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.mapping_dict = {'4wd': 'four_wheel_drive',
                             'fwd': 'front_wheel_drive',
                             'rwd': 'rear_wheel_drive',
                             'awd': 'all_wheel_drive'}

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        X = pd.DataFrame(X)  # Convert X to a pandas DataFrame
        nan_rows = X['drive'].isnull()
        X.loc[nan_rows, 'drive'] = X.loc[nan_rows, 'description'].apply(lambda x: self.get_drive(x))
        X['drive'] = X['drive'].map(self.mapping_dict).fillna(X['drive'])
        return X.values.tolist()  # Convert X back to a list
```

In [ ]:
```
clean_data = pipeline.fit_transform(vehicles_clean)
```

Paint

In [ ]:
```
X_train['description'].iloc[12]
```

In [ ]:
```
X_val['description'].iloc[100]
```

In [ ]:
```
class SplitExteriorInterior(TransformerMixin):
    def transform(self, X):
        X_new = []
        for row in X:
            new_row = []
            for val in row:
                if 'exterior' in val:
                    split_vals = val.split('exterior')
                    for i in range(len(split_vals)):
                        if i == 0:
                            new_row.append(split_vals[i])
                        elif i == len(split_vals) - 1:
                            if split_vals[i] != '':
                                if new_row[-1] == '':
                                    new_row.pop()
                                new_row.append('exterior')
                                new_row.append(split_vals[i])
                            else:
                                new_row.append('exterior')
                        elif split_vals[i] != '':
                            if new_row[-1] == '':
                                new_row.pop()
                            new_row.extend(['exterior', split_vals[i]])
                elif 'interior' in val:
                    split_vals = val.split('interior')
                    for i in range(len(split_vals)):
                        if i == 0:
                            new_row.append(split_vals[i])
                        elif i == len(split_vals) - 1:
                            if split_vals[i] != '':
                                if new_row[-1] == '':
                                    new_row.pop()
                                new_row.append('interior')
                                new_row.append(split_vals[i])
                            else:
                                new_row.append('interior')
                        elif split_vals[i] != '':
                            if new_row[-1] == '':
                                new_row.pop()
                            new_row.extend(['interior', split_vals[i]])
                else:
                    new_row.append(val)
            X_new.append(new_row)
        return X_new

    def fit(self, X, y=None, **fit_params):
        return self
```

In [ ]:
```
split_ext_int_pipeline = Pipeline([
    ('split_ext_int', SplitExteriorInterior())
])
```

In [ ]:
```
vehicles_clean['description'] = split_ext_int_pipeline.fit_transform(vehicles_clean['description'])
```

In [ ]:
```
X_train['description'] = split_ext_int_pipeline.fit_transform(X_train['description'])
```

In [ ]:
```
X_train['description'].iloc[12]
```

In [ ]:
```
X_val['description'] = split_ext_int_pipeline.transform(X_val['description'])
```

In [ ]:
```
X_val['description'].iloc[100]
```

In [ ]:
```
X_test['description'] = split_ext_int_pipeline.transform(X_test['description'])
```

paint_color cleaning

In [ ]:
```
paint_master = ['white', 'blue', 'red', 'black', 'silver', 'grey', 'beige','brown', 'burgundy',
                'gold', 'yellow', 'orange', 'green','purple', 'tan', 'charcoal','anvil',
                'maroon','gray','champagne','olive','darkblue','darkgreen','lightblue','lightgray',
                'lightgrey','darkgray','darkgrey','teal','sapphireblue','midnightbblue','charcoalgray',
                'bronze','copper','pearlwhite','pearlblack','rossored','brilliantsilve','cyan','magenta',
                'aliceblue','antiquewhite']
```

In [ ]:
```
mapping_dict = {'gray':'grey','whiteinterior':'white','brilliantsilve':'silver','pearlwhite':'white',
                'darkgray':'grey','lightgray':'grey','sapphireblue':'blue','darkblue':'blue','lightblue':'blue',
                'darkgreen':'green','aliceblue':'blue','antiquewhite':'white'}
```

In [ ]:
```
from sklearn.base import TransformerMixin

class PaintColorImputer(TransformerMixin):
    def __init__(self, paint_master, mapping_dict):
        self.paint_master = paint_master
        self.mapping_dict = mapping_dict

    def transform(self, X):
        X_new = X.copy()
        for i, row in X_new.iterrows():
            if pd.isna(row['paint_color']):
                description_tokens = row['description']
                try:
                    color_token_idx = description_tokens.index('color')
                    if 'exterior' in description_tokens[color_token_idx-1]:
                        color = description_tokens[color_token_idx+1]
                        if color in self.paint_master:
                            if color in self.mapping_dict:
                                X_new.at[i, 'paint_color'] = self.mapping_dict[color]
                            else:
                                X_new.at[i, 'paint_color'] = color
                except (ValueError, IndexError):
                    continue
        return X_new

    def fit(self, X, y=None):
        return self
```

In [ ]:
```
pipeline = Pipeline(steps=[('paint_color_imputer', PaintColorImputer(paint_master, mapping_dict))])
```

In [ ]:
```
vehicles_clean['paint_color'].isna().sum()
```

In [ ]:
```
vehicles_clean = pipeline.fit_transform(vehicles_clean)
```

In [ ]:
```
vehicles_clean['paint_color'].isna().sum()
```

In [ ]:
```
# color_pipeline = Pipeline([
#     ('paint_color_imputer', PaintColorImputer(paint_master, mapping_dict))
# ])
```

In [ ]:
```
from sklearn.base import BaseEstimator, TransformerMixin

class ColumnSelector(BaseEstimator, TransformerMixin):
    '''select specific columns of a given dataset'''
    def __init__(self, subset):
        self.subset = subset

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return X.loc[:, self.subset]
```

In [ ]:
```
# full_pipeline = Pipeline(steps=[('paint_color_imputer', PaintColorImputer(paint_master, mapping_dict)),
#                           ('ct', ColumnTransformer(transformers=[('imputer', SimpleImputer(strategy='mean'),
#                                                                   ['paint color'])],remainder='passthrough'))])
```

In [ ]:
```
full_pipeline = Pipeline(steps=[
    ('paint_color_imputer', PaintColorImputer(paint_master, mapping_dict)),
    ('ct', ColumnTransformer(
        transformers=[
            ('imputer', SimpleImputer(strategy='most_frequent'), ['paint_color'])],
        remainder='drop'))
])
```

In [ ]:
```
vehicles_clean['paint_color'].isna().sum()
```

In [ ]:
```
vehicles_clean = full_pipeline.fit_transform(vehicles_clean)
```

In [ ]:
```
vehicles_clean['paint_color'].isna().sum()
```

In [ ]:
```
vehicles_clean
```

In [ ]:
```
pipeline = Pipeline(steps=[('paint_color_imputer', PaintColorImputer(paint_master, mapping_dict)),
                           ('ct', ColumnTransformer(transformers=[('imputer', SimpleImputer(strategy='most_frequent'),
                                                                   ['paint color'])],remainder='passthrough'))])
```

In [ ]:
```
vehicles_clean_t = pipeline.fit_transform(vehicles_clean)
```

In [ ]:
```
vehicles_clean_t = pd.DataFrame(vehicles_clean_t, columns=vehicles_clean.columns)
```

In [ ]:
```
vehicles_clean_t['paint_color'].isna().sum()
```

In [ ]:
```
# Import libraries and loading the csv file
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import string
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('omw-1.4')
from sklearn.pipeline import Pipeline
from sklearn.base import TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,r2_score
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import PolynomialFeatures
from sklearn.tree import DecisionTreeRegressor
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from math import sqrt
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
import seaborn as sns
from sklearn.model_selection import RandomizedSearchCV
```

In [ ]:
```
vehicles_clean = pd.read_csv(r"C:\Users\91886\Downloads\vehicles_allclean_imputer.csv")
```

In [ ]:
```
vehicles_clean.info()
```

In [ ]:
```
vehicles_clean.isna().sum()
```

In [ ]:
```
vehicles_clean.drop(['condition','id','posting_date','model','description'], axis=1,inplace = True)
```

In [ ]:
```
from sklearn.model_selection import train_test_split
import pandas as pd

# define the features and target variables
X = vehicles_clean.drop('price', axis=1)
y = vehicles_clean['price']

# divide the data into train, validation, and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42)

# display the shapes of the resulting datasets
print(f"Training set shape: {X_train.shape}, {y_train.shape}")
print(f"Validation set shape: {X_val.shape}, {y_val.shape}")
print(f"Test set shape: {X_test.shape}, {y_test.shape}")
```

In [ ]:
```
type(X_train)
```

In [ ]:
```
type(X_val)
```

In [ ]:
```
X_train.dtypes
```

In [ ]:
```
cat_cols = ['region', 'manufacturer', 'fuel', 'title_status', 'transmission', 'type', 'paint_color', 'state', 'drive']
num_cols = ['year','cylinders','odometer','lat','long','price']
# define the pipeline to perform one-hot encoding
ohe_pipeline = Pipeline([
    ('ohe_transformer', ColumnTransformer(
        transformers=[
            ('one_hot_encoder', OneHotEncoder(handle_unknown='ignore', sparse=False), cat_cols)
        ],
        remainder='passthrough'
    ))
])
```

In [ ]:
```
X_train = ohe_pipeline.fit_transform(X_train)
X_val = ohe_pipeline.transform(X_val)
X_test = ohe_pipeline.transform(X_test)
```

In [ ]:
```
# class DataFrameSelector(BaseEstimator, TransformerMixin):
#     def __init__(self, feature_names):
#         self.attribute_names = feature_names
#     def fit(self, X, y=None):
#         return self
#     def transform(self, X):
#         return X[self.attribute_names].values
```

In [ ]:
```
# # Impute fills missing numerals by the median of the remaining data
# imputer = SimpleImputer(strategy="median")

# # The imputer is fit on the training data. It can then also be applied to the test data (without a refit)
# num_pipeline = Pipeline([
#         ('selector', DataFrameSelector(num_cols)),
#         ('imputer', SimpleImputer(strategy="median")),
#         ('std_scaler', StandardScaler())
#     ])
```

In [ ]:
```
# X_train = num_pipeline.fit_transform(X_train)
```

In [ ]:
```
type(X_val)
```

Linear Regression

In [ ]:
```
linreg_model = LinearRegression()
linreg model.fit(X train, y train)
```

In [ ]:
```
y_val_pred = linreg_model.predict(X_val)
```

In [ ]:
```
mse_val = mean_squared_error(y_val, y_val_pred)
r2 = r2 score(y val, y val pred)
```

In [ ]:
```
print('Mean squared error on validation data:', np.sqrt(mse_val))
print("R-squared:", r2)
```

In [ ]:
```
y_val_pred = pd.DataFrame(y_val_pred, columns = ['Predicted Output'])
lin reg results = pd.concat([y val pred, y val.to frame().reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = lin_reg_results, y = 'Predicted Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

In [ ]:
```
#Lin Reg on log transformed target variable
```

In [ ]:
```
# Log transform the target variable
y train log = np.log(y train)
```

In [ ]:
```
# Fit the model to the log-transformed target variable
linreg model.fit(X train, y train log)
```

In [ ]:
```
# Predict the log-transformed car prices for the validation data
y pred log = linreg model.predict(X val)
```

In [ ]:
```
# Transform the predicted values back to the original scale
y pred = np.exp(y pred log)
```

In [ ]:
```
# Evaluate the model's performance on the validation data
mse = mean_squared_error(y_val, y_pred)
r2 = r2 score(y val, y pred)
```

In [ ]:
```
print('Mean squared error on validation data:', mse_val)
print("R-squared:", r2)
```

In [ ]:
```
y_pred = pd.DataFrame(y_pred, columns = ['Predicted Output'])
lin reg results = pd.concat([y pred, y val.to frame().reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = lin_reg_results, y = 'Predicted Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

Decision Tree

In [ ]:
```
dtr_model = DecisionTreeRegressor(splitter = 'random')
dtr model.fit(X train, y train)
```

In [ ]:
```
dtr_predict_train = dtr_model.predict(X_train)
```

In [ ]:
```
dtr_predict_val = dtr_model.predict(X_val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, dtr_predict_train))
r2_train = r2_score(y_train, dtr_predict_train)
print(rmse_train)
print(r2 train)
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, dtr_predict_val))
r2_val = r2_score(y_val, dtr_predict_val)
print(rmse_val)
print(r2 val)
```

In [ ]:
```
dtr_predict_val = pd.DataFrame(dtr_predict_val, columns = ['Predicted Val Output'])
```

In [ ]:
```
dtr_results = pd.concat([dtr_predict_val, y_val.to_frame().reset_index(drop = True)], axis = 1, ignore_index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = dtr_results, y = 'Predicted Val Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

In [ ]:
```
#Decision Treee on log transformed target variable
```

In [ ]:
```
# Log transform the target variable
y_train_log = np.log(y_train)
# y val log = np.log(y val)
```

In [ ]:
```
# Fit the model to the log-transformed target variable
dtr model.fit(X train, y train log)
```

In [ ]:
```
dtr_predict_train_log = dtr_model.predict(X_train)
```

In [ ]:
```
dtr_predict_val_log = dtr_model.predict(X_val)
```

In [ ]:
```
predict_train = np.exp(dtr_predict_train_log)
```

In [ ]:
```
predict_val = np.exp(dtr_predict_val_log)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, predict_train))
r2_train = r2_score(y_train, predict_train)
print(rmse_train)
print(r2 train)
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, predict_val))
r2_val = r2_score(y_val, predict_val)
print(rmse_val)
print(r2_val)

# W/o log data
# 6124.774719287382
# 0.8253020280442003
```

In [ ]:
```
dtr_predict_val = pd.DataFrame(predict_val, columns = ['Predicted Val Output'])
```

In [ ]:
```
dtr_results = pd.concat([dtr_predict_val, y_val.to_frame().
                         reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = dtr_results, y = 'Predicted Val Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

Random Forest

In [ ]:
```
# Create a random forest regressor with 100 trees
rfr_model = RandomForestRegressor(n_estimators=500, max_depth=10, bootstrap=True, random_state=42)

# max depth -> 10
```

In [ ]:
```
# fit the model to the training data
rfr model.fit(X train, y train)
```

In [ ]:
```
rfr_predict_train = rfr_model.predict(X_train)
rfr predict val = rfr model.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, rfr_predict_train))
r2_train = r2_score(y_train, rfr_predict_train)
print(rmse_train)
print(r2 train)
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, rfr_predict_val))
r2_val = r2_score(y_val, rfr_predict_val)
print(rmse_val)
print(r2 val)
```

In [ ]:
```
rfr_predict_val = pd.DataFrame(rfr_predict_val, columns = ['Predicted Val Output'])
rfr_results = pd.concat([rfr_predict_val, y_val.to_frame().
                         reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = rfr_results, y = 'Predicted Val Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

In [ ]:
```
# On Log transformed
```

In [ ]:
```
# Log transform the target variable
y_train_log = np.log(y_train)
# y val log = np.log(y val)
```

In [ ]:
```
# Create a random forest regressor with 100 trees
rfr_model_log = RandomForestRegressor(n_estimators=500, max_depth=10, bootstrap=True, random_state=42)

# max depth -> 10
```

In [ ]:
```
rfr_model_log.fit(X_train, y_train_log)
```

In [ ]:
```
rfr_predict_train_log = rfr_model_log.predict(X_train)
rfr predict val log = rfr model log.predict(X val)
```

In [ ]:
```
predict_train = np.exp(rfr_predict_train_log)
predict val = np.exp(rfr predict val log)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, predict_train))
r2_train = r2_score(y_train, predict_train)
print(rmse_train)
print(r2 train)
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, predict_val))
r2_val = r2_score(y_val, predict_val)
print(rmse_val)
print(r2 val)
```

In [ ]:
```
rfr_results = pd.concat([rfr_predict_val, y_val.to_frame().
                         reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = rfr_results, y = 'Predicted Val Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

XGBoost

In [ ]:
```
from xgboost import XGBRegressor
```

In [ ]:
```
xgb_model = XGBRegressor()
```

In [ ]:
```
# fit the model to the training data
xgb model.fit(X train, y train)
```

In [ ]:
```
xgb_predict_train = xgb_model.predict(X_train)
xgb predict val = xgb model.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2 train)
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2 val)
```

In [ ]:
```
xgb_predict_val = pd.DataFrame(xgb_predict_val, columns = ['Predicted Val Output'])
xgb_results = pd.concat([xgb_predict_val, y_val.to_frame().
                         reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = xgb_results, y = 'Predicted Val Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

In [ ]:
```
# Log transform the target variable
y_train_log = np.log(y_train)
# y val log = np.log(y val)
```

In [ ]:
```
xgb_model.fit(X_train, y_train_log)
```

In [ ]:
```
xgb_predict_train_log = xgb_model.predict(X_train)
xgb predict val log = xgb model.predict(X val)
```

In [ ]:
```
predict_train = np.exp(xgb_predict_train_log)
predict val = np.exp(xgb predict val log)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, predict_train))
r2_train = r2_score(y_train, predict_train)
print(rmse_train)
print(r2 train)
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, predict_val))
r2_val = r2_score(y_val, predict_val)
print(rmse_val)
print(r2 val)
```

In [ ]:
```
predict_val = pd.DataFrame(predict_val, columns = ['Predicted Val Output'])
xgb_results = pd.concat([predict_val, y_val.to_frame().
                         reset index(drop = True)], axis = 1, ignore index = False)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
sns.regplot(data = xgb_results, y = 'Predicted Val Output', x = 'price', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```

In [ ]:
```
# Hyperparameters
params = {'max_depth': [5,8,10],
          'learning_rate': [0.01, 0.05, 0.1],
          'n_estimators': [100, 500, 1000],
          'gamma': [0, 0.2, 0.4],
          'reg_alpha': [0, 0.5, 5],
          'reg_lambda': [1, 10, 100]
          }
# params = {
#     'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
#     'learning_rate': [0.01, 0.05, 0.1, 0.15, 0.2],
#     'n_estimators': [100, 500, 1000, 2000, 3000],
#     'colsample_bytree': [0.3, 0.4, 0.5, 0.6, 0.7],
#     'gamma': [0, 0.1, 0.2, 0.3, 0.4],
#     'subsample': [0.5, 0.6, 0.7, 0.8, 0.9],
#     'reg_alpha': [0, 0.1, 0.5, 1, 10],
#     'reg_lambda': [0.01, 0.1, 1, 10, 100]
# }
```

In [ ]:
```
# Define the randomized search
random_search = RandomizedSearchCV(
    xgb_model, param_distributions=params,
    n_iter=50, cv=5, verbose=1, n_jobs=-1)
```

In [ ]:
```
random_search.fit(X_train, y_train)
```

In [ ]:
```
# DEFAULT ONE

# XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
#              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
#              gamma=0, gpu_id=-1, importance_type=None,
#              interaction_constraints='', learning_rate=0.300000012,
#              max_delta_step=0, max_depth=6, min_child_weight=1, missing=nan,
#              monotone_constraints='()', n_estimators=100, n_jobs=4,
#              num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
#              reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
#              validate_parameters=1, verbosity=None)
```

In [ ]:
```
# HYPERPARAMETER 1 --> Overfitting observed

xgb_model1 = XGBRegressor(learning_rate = 0.5, n_estimators=500,max_depth=10)
```

In [ ]:
```
xgb_model1.fit(X_train, y_train)
```

In [ ]:
```
xgb_predict_train = xgb_model1.predict(X_train)
xgb_predict_val = xgb_model1.predict(X_val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# HYPERPARAMETER 2
# (with regularization parameters)
# reduced learning rate
# max depeth reduced

## overfitting

xgb_model2 = XGBRegressor(learning_rate = 0.3, n_estimators=500,max_depth=8, reg_alpha=0.01, reg_lambda=1)
```

In [ ]:
```
xgb_model2.fit(X_train, y_train)
```

In [ ]:
```
xgb_predict_train = xgb_model2.predict(X_train)
xgb_predict_val = xgb_model2.predict(X_val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# HYPERPARAMETER 3


xgb model3 = XGBRegressor(learning rate = 0.4, n estimators=500,max depth=6, reg alpha=0.05, reg lambda=1)
```

In [ ]:
```
xgb_model3.fit(X_train, y_train)
```

In [ ]:
```
xgb_predict_train = xgb_model3.predict(X_train)
xgb predict val = xgb model3.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```

```

In [ ]:
```
# HYPERPARAMETER 4
#. increase reg_alpha

xgb model4 = XGBRegressor(learning rate = 0.4, n estimators=500,max depth=6, reg alpha=0.1, reg lambda=1)
```

In [ ]:
```
xgb_model4.fit(X_train, y_train)
```

In [ ]:
```
xgb_predict_train = xgb_model4.predict(X_train)
xgb predict val = xgb model4.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```

```

In [ ]:
```
# XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
#              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
#              gamma=0, gpu_id=-1, importance_type=None,
#              interaction_constraints='', learning_rate=0.300000012,
#              max_delta_step=0, max_depth=6, min_child_weight=1, missing=nan,
#              monotone_constraints='()', n_estimators=100, n_jobs=4,
#              num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
#              reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
#              validate_parameters=1, verbosity=None)
```

In [ ]:
```
# HYPERPARAMETER 5
#. n_estimators

xgb model5 = XGBRegressor(learning rate = 0.4, n estimators=200,max depth=6, reg alpha=0.1, reg lambda=1)
```

In [ ]:
```
xgb_model5.fit(X_train, y_train)
```

In [ ]:
```
xgb_predict_train = xgb_model5.predict(X_train)
xgb predict val = xgb model5.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# HYPERPARAMETER 6


xgb_model6 = XGBRegressor(max_depth=8,learning_rate = 0.5)
xgb model6.fit(X train, y train)
```

In [ ]:
```
xgb_predict_train = xgb_model6.predict(X_train)
xgb predict val = xgb model6.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# HYPERPARAMETER 7
xgb_model7 = XGBRegressor(learning_rate = 0.38, n_estimators=150,max_depth=8)
xgb model7.fit(X train, y train)
```

In [ ]:
```
xgb_predict_train = xgb_model7.predict(X_train)
xgb predict val = xgb model7.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# Hyperparameter 8
#  max_depth is set to 6 and min_child_weight is set to 5,

xgb_model8 = XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
             gamma=0, gpu_id=-1, importance_type=None,
             interaction_constraints='', learning_rate=0.300000012,
             max_delta_step=0, max_depth=6, min_child_weight=5,
             monotone_constraints='()', n_estimators=100, n_jobs=4,
             num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
             reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
xgb model8.fit(X train, y train)
```

In [ ]:
```
xgb_predict_train = xgb_model8.predict(X_train)
xgb predict val = xgb model8.predict(X val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# Hyperparameter 9
# Learning rate increased

xgb_model9 = XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
             gamma=0, gpu_id=-1, importance_type=None,
             interaction_constraints='', learning_rate=0.35,
             max_delta_step=0, max_depth=6, min_child_weight=5,
             monotone_constraints='()', n_estimators=100, n_jobs=4,
             num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
             reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
xgb model9.fit(X train, y train)
```

In [ ]:
```
xgb_predict_train = xgb_model9.predict(X_train)
xgb_predict_val = xgb_model9.predict(X_val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```

In [ ]:
```
# Hyperparameter 10
# max_depth increased

xgb_model10 = XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
             gamma=0, gpu_id=-1, importance_type=None,
             interaction_constraints='', learning_rate=0.35,
             max_delta_step=0, max_depth=8, min_child_weight=5,
             monotone_constraints='()', n_estimators=100, n_jobs=4,
             num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
             reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
xgb_model10.fit(X_train, y_train)
```

In [ ]:
```
xgb_predict_train = xgb_model10.predict(X_train)
xgb_predict_val = xgb_model10.predict(X_val)
```

In [ ]:
```
rmse_train = sqrt(mean_squared_error(y_train, xgb_predict_train))
r2_train = r2_score(y_train, xgb_predict_train)
print(rmse_train)
print(r2_train)

# 4706.4166370845305
# 0.8950213517845189
```

In [ ]:
```
rmse_val = sqrt(mean_squared_error(y_val, xgb_predict_val))
r2_val = r2_score(y_val, xgb_predict_val)
print(rmse_val)
print(r2_val)

# 5530.851800951771
# 0.8575404171042413
```