Data Preparation Project

Name: Sameer Ahamed Rizwan Basha

Student No: 202381922

Importing essential libraries and packages for data manipulation, analysis, and visualization.

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
```

In [2]:
```
def train_random_forest(X,y):
    # Use the command train_test_split to divide the dataset
    X_train, X_test, y_train, y_test = train_test_split(X,y,
                                                        test_size=0.2,
                                                        shuffle=True,
                                                        random_state=42)
    # initialize the random forest classifier
    clf = RandomForestRegressor(random_state=42)
    # Fit the model
    clf.fit(X_train,y_train)
    # Check the model score
    score = clf.score(X_test,y_test)
    print(score)
    return score
```

The dataset you have chosen, its variables, in particular the target variable.

Loading Dataset

In [3]:
```
# Import the Pandas library and read the CSV file into a DataFrame
import pandas as pd
df = pd.read_csv("archive\CAR DETAILS FROM CAR DEKHO - Copy.csv")

# Display the first few rows of the DataFrame
df.head()
```

Out[3]:

	name	yearkm_driven	fuel	seller_type	transmission	owner	selling_price
0	Maruti 800 AC	200770000	Petrol	Individual	Manual	First Owner	60000
1	Maruti Wagon R LXI Minor	200750000	Petrol	Individual	Manual	First Owner	135000
2	Hyundai Verna 1.6 SX	2012100000	Diesel	Individual	Manual	First Owner	600000
3	Datsun RediGO T Option	201746000	Petrol	Individual	Manual	First Owner	250000
4	Honda Amaze VX i-DTEC	2014141000	Diesel	Individual	Manual	Second Owner	450000

The dataset "CAR DETAILS FROM CAR DEKHO" contains information about various cars, with the following features:

Name: The model and make of the car (e.g., 'Maruti 800 AC', 'Hyundai Verna 1.6 SX').

Year: The year of manufacture of the car (e.g., 2007, 2012).

Km Driven: The total kilometers driven by the car (e.g., 70000 km, 100000 km).

NOTE: The year and Km Driven are combined. we need to address that in the data preperation process.

Selling Price: The price at which the car is being sold (e.g., 60000, 600000). This is likely in a local currency, though the dataset does not specify.

Fuel: The type of fuel used by the car (e.g., Petrol, Diesel).

Seller Type: The type of seller (e.g., 'Individual', 'Dealer'). This field classifies who is selling the car.

Transmission: The type of transmission in the car (e.g., 'Manual', 'Automatic').

Owner: The ownership status of the car (e.g., 'First Owner', 'Second Owner').

This dataset is used to analyze the used car market, with details that are crucial for determining the value and appeal of a car, such as its age, condition (indicated by km driven), type of fuel, and ownership history.

EDA of the dataset

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6364 entries, 0 to 6363
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   name           6364 non-null   object
 1   yearkm_driven  6364 non-null   int64 
 2   fuel           6364 non-null   object
 3   seller_type    6364 non-null   object
 4   transmission   4688 non-null   object
 5   owner          6364 non-null   object
 6   selling_price  6364 non-null   int64 
dtypes: int64(2), object(5)
memory usage: 348.2+ KB
```

In [5]:
```
df.shape
```

Out[5]: (6364, 7)

In [6]:
```
df.describe()
```

Out[6]:

	yearkm_driven	selling_price
count	6.364000e+03	6.364000e+03
mean	7.518344e+08	5.107396e+05
std	2.028224e+09	5.943842e+05
min	2.014100e+04	2.000000e+04
25%	2.012500e+08	2.000000e+05
50%	2.015770e+08	3.515000e+05
75%	2.018700e+08	6.000000e+05
max	2.019100e+10	8.900000e+06

Step 0 are the collection of data preperation I did to run get a baseline estimate. These are not counted as data preperation steps for the project

Step 0 - Calculating base performance with initial data preperation

In [7]:
```
# Create a train test split in the dataset
y = df['selling_price']
X = df.drop(['selling_price'],axis=1)

# Convert the 'name' column to a categorical data type
df['name'] = pd.Categorical(df['name'])
# Assign the encoded variable to a new column
df['name'] = df['name'].cat.codes

# Convert the 'fuel' column to a categorical data type
df['fuel'] = pd.Categorical(df['fuel'])
# Assign the encoded variable to a new column
df['fuel'] = df['fuel'].cat.codes

# Convert the 'seller_type' column to a categorical data type
df['seller_type'] = pd.Categorical(df['seller_type'])
# Assign the encoded variable to a new column
df['seller_type'] = df['seller_type'].cat.codes

# Convert the 'transmission' column to a categorical data type
df['transmission'] = pd.Categorical(df['transmission'])
# Assign the encoded variable to a new column
df['transmission'] = df['transmission'].cat.codes

# Convert the 'owner' column to a categorical data type
df['owner'] = pd.Categorical(df['owner'])
# Assign the encoded variable to a new column
df['owner'] = df['owner'].cat.codes

# Create a train test split in the dataset
y = df['selling_price']
X = df.drop(['selling_price'],axis=1)

score_0 = train_random_forest(X,y)
```

```
0.5723483216745441
```

I encountered a "ValueError" while working with some code. The error message I received was "could not convert string to float: 'Mahindra Scorpio S11 BSIV'". It appears that I was attempting to convert a string containing the text "Mahindra Scorpio S11 BSIV" into a floating-point number, but this conversion wasn't possible because the string didn't represent a numerical value.

To resolve this issue, I need to check the context in which I was trying to convert this string to a float. It seems like I might have been trying to perform a conversion on data that wasn't intended to be a numerical value.

I faced the same issue with 'name', 'fuel', 'seller_type', 'transmission', 'owner'. So i converted them from non-numeric to numeric.

The baseline score is 0.5723

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

Out[8]:
```
name             int16
yearkm_driven    int64
fuel              int8
seller_type       int8
transmission      int8
owner             int8
selling_price    int64
dtype: object
```

Duplicate Instances

Handling duplicate instances in a dataset is crucial for several reasons, particularly in data analysis, machine learning, and statistical modeling:

Data Accuracy: Duplicate entries can lead to inaccuracies in the dataset. They might give a skewed view of the data, leading to incorrect conclusions or insights. For example, if a dataset of car sales erroneously lists the same transaction multiple times, it may appear as though more cars were sold than actually were.

Biased Results: In machine learning and statistical analysis, duplicate data can bias the results. Models may overfit to these repeated instances, thinking they are more significant than they actually are. This can reduce the model's ability to generalize to new, unseen data.

Efficiency: Duplicate records can increase the size of the dataset unnecessarily, leading to inefficiencies in storage and computation. Processing larger datasets requires more memory and computational power, which can be optimized by removing duplicates.

Statistical Analysis: In statistics, duplicates can skew the results of descriptive statistics (like mean, median, mode) and inferential statistics, leading to invalid conclusions.

Therefore, I have identified and appropriately handled duplicates to maintain the quality and reliability my data analysis.

Experiment 1 : Handling Duplicate Instances

In [9]:
```
# Removing duplicates
print(f'Shape of df before the duplicates were removed: {df.shape}')
df = df.drop_duplicates()
print(f'Shape of df after the duplicates were removed: {df.shape}')

# Create a train test split in the dataset
y = df['selling_price']
X = df.drop(['selling_price'],axis=1)

score_1 = train_random_forest(X,y)
```

```
Shape of df before the duplicates were removed: (6364, 7)
Shape of df after the duplicates were removed: (5096, 7)
0.6708761921059816
```

Missingness in features

Missingness in data can pose significant challenges and potentially lead to negative impacts in data analysis and decision-making processes for several reasons:

- **Reduced Data Quality**: Missing data can significantly reduce the overall quality and reliability of a dataset. It can lead to biased estimates and distort the true picture that the complete data is supposed to represent.
- **Compromised Statistical Analysis**: Many statistical techniques assume complete data. Missing values can lead to incorrect conclusions or statistical inferences. For instance, calculating means, medians, or variances becomes problematic when data points are missing. Random Forest ignores the missing data when training.
- **Model Accuracy**: In machine learning, missing data can affect the performance of models. Models trained on incomplete data might not accurately represent the underlying patterns and relationships, leading to poor predictions or classifications.
- **Limited Representativeness**: Missing data can result in a dataset that does not accurately represent the population it's supposed to. This is particularly crucial in surveys and research studies, where the goal is to understand broader trends or patterns.

Therefore, it is crucial to appropriately address missing data in any analysis to ensure the validity and reliability of the results. The strategies to handle missing data depend on the nature and extent of the missingness and the specific context of the analysis.

Check the missing values in the df

In [10]:
```
import pandas as pd

# Replace empty strings (or any other placeholder) with NaN
df['transmission'].replace(0, np.nan, inplace=True)

# Counting the missing values in each column
missing_values_count = df.isna().sum()
print(missing_values_count)
```

```
name              0
yearkm_driven     0
fuel              0
seller_type       0
transmission    319
owner             0
selling_price     0
dtype: int64
```

Experiment 2: Handling Missing Data - Mean substitution

In [11]:
```
# Count the frequency of each category in the 'category' column
frequency = df['transmission'].value_counts()

# First, calculate the mean of the non-missing values in the column
mean_value = df['transmission'].mean()

# Deep copy
mean_df = df.copy()

# Now, fill the missing (NaN) values with the mean value
mean_df['transmission'].fillna(mean_value, inplace=True)

# Create a train test split in the dataset
y = mean_df['selling_price']
X = mean_df.drop(['selling_price'],axis=1)

score_2 = train_random_forest(X,y)
```

```
0.6697825383425331
```

Experiment 3: Handling Missing Data - Median substitution

In [12]:
```
from sklearn.impute import SimpleImputer

# Initialize the SimpleImputer with median strategy
imputer = SimpleImputer(strategy='median')

# Impute missing values
median_df = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)

# Create a train test split in the dataset
y = mean_df['selling_price']
X = mean_df.drop(['selling_price'],axis=1)

score_3 = train_random_forest(X,y)
```

```
0.6697825383425331
```

Experiment 4 : Handling Missing Data - Frequency Substitution

In [13]:
```
# Initialize the SimpleImputer with frequency strategy
imputer = SimpleImputer(strategy='most_frequent')

# Impute missing values
median_df = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)

# Create a train test split in the dataset
y = mean_df['selling_price']
X = mean_df.drop(['selling_price'],axis=1)

score_4 = train_random_forest(X,y)
```

```
0.6697825383425331
```

Experiment 5 : Handling Missing Data - Multiple Imputation

In [14]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

imp = IterativeImputer(max_iter=10, random_state=0)

# Create a train test split in the dataset
y = df['selling_price']
```

```
X = df.drop(['selling_price'],axis=1)

imp.fit(X)  # X is your data with missing values

X_imputed = imp.transform(X)

score_5 = train_random_forest(X_imputed,y)
```

```
0.6746201735634997
```

Experiment 6 : Handling Missing Data - knn_imputer

In [15]:
```
from sklearn.impute import KNNImputer
# Create KNN imputer object
# You can specify the number of neighbors to use for imputing missing values
imputer = KNNImputer(n_neighbors=2)

# Fit the imputer to the data and transform it
df_imputed = imputer.fit_transform(df)

# The result is a NumPy array, so you may want to convert it back to a DataFrame
df_imputed = pd.DataFrame(df_imputed, columns=df.columns)

# Create a train test split in the dataset
y = df_imputed['selling_price']
X = df_imputed.drop(['selling_price'],axis=1)

score_6 = train_random_forest(X,y)
```

```
0.6941655544588718
```

In the course of addressing missing values within the dataset, I employed several imputation techniques, namely mean substitution, median substitution, frequency substitution, multiple imputation, and K-Nearest Neighbors (KNN) imputation. After a thorough evaluation of these methods, it was determined that the KNN imputation technique yielded the most favorable results, attaining a score of 0.69. This superior performance underscores the effectiveness of KNN imputation in dealing with the missing data in my dataset, making it the most suitable method for this specific application.

Compound Variables

In data analysis, the presence of concatenated or compound variables, where multiple pieces of information are merged into a single variable, presents a significant challenge. This is because such compound variables can obscure the true relationships and effects of the individual components they contain. Analyzing these combined variables without separating them can lead to misleading interpretations and conclusions, as the granular detail and specific influences of each element are lost in the amalgamation. Consequently, it's crucial to address and decompose these compound variables in any dataset. Doing so ensures that each variable is analyzed distinctly, leading to more accurate, reliable, and meaningful insights from the data. This approach is fundamental for maintaining the integrity and efficacy of data analysis.

Experiment 7 : Handling Compound Variables

In [16]:
```
# One the visual inspection of the project we can see that the features 'year' and 'km_driven are merged

# We will attempt to seperate them here.
# Convert the column to string
df_imputed['yearkm_driven'] = df_imputed['yearkm_driven'].astype(str)

# Now you can use string slicing
df_imputed['year'] = df_imputed['yearkm_driven'].str[:4].astype(int)
df_imputed['km_driven'] = df_imputed['yearkm_driven'].str[4:].astype(float)

del df_imputed['yearkm_driven']
print(df_imputed)
# Create a train test split in the dataset
y = df_imputed['selling_price']
X = df_imputed.drop(['selling_price'],axis=1)

score_7 = train_random_forest(X,y)
```

```
        name  fuel  seller_type  transmission  owner  selling_price  year  \
0      775.0   4.0          1.0           1.0    0.0        60000.0  2007
1     1041.0   4.0          1.0           1.0    0.0       135000.0  2007
2      505.0   1.0          1.0           1.0    0.0       600000.0  2012
3      118.0   4.0          1.0           1.0    0.0       250000.0  2017
4      279.0   1.0          1.0           1.0    2.0       450000.0  2014
...      ...   ...          ...           ...    ...            ...   ...
5091  1035.0   4.0          1.0           1.0    0.0       250000.0  2011
5092   971.0   1.0          1.0           1.0    0.0       700000.0  2018
5093   238.0   4.0          1.0           1.0    4.0       185000.0  2011
5094   928.0   1.0          1.0           1.0    0.0       200000.0  2011
5095   630.0   1.0          1.0           1.0    4.0       315000.0  2004

      km_driven
0       70000.0
1       50000.0
2      100000.0
3       46000.0
4      141000.0
...         ...
5091    50000.0
5092    38217.0
5093    50000.0
5094   120000.0
5095   110000.0

[5096 rows x 8 columns]
0.7719603775408748
```

Separating the 'year' and 'km driven' columns in the dataset significantly enhanced the model's effectiveness. This decision proved to be a step in the right direction, as it allowed the model to more accurately interpret and learn from these distinct aspects of the data. By treating 'year' and 'km driven' as separate variables, the model could better understand their individual contributions and relationships to the target variable, leading to improved performance and more reliable predictions. This approach exemplifies the importance of thoughtful feature engineering in data analysis and model development.

Outlier Detection

Detecting outliers in a dataset is crucial for several reasons, particularly in data analysis, statistics, and machine learning:

- Data Quality Assessment: Outliers can indicate errors in data collection or entry. For example, a data point that significantly deviates from the norm may be the result of a measurement error or a typo. Identifying outliers helps in assessing and improving the quality of data.
- Robust Statistical Analysis: Outliers can skew statistical measures like the mean, standard deviation, and correlations. This can lead to misleading conclusions. By identifying and handling outliers appropriately, you can ensure more robust and reliable statistical analysis.
- Improved Model Performance: In machine learning, outliers can adversely affect the performance of models. Algorithms can be unduly influenced by outliers, leading to models that are less generalizable to typical data. Detecting and managing outliers can lead to better model performance and predictive accuracy.

Experiment 8 : Outlier Detection using K Nearest Neighbours

In [17]:
```python
from sklearn.neighbors import NearestNeighbors

y = df_imputed['selling_price']
X = df_imputed.drop(['selling_price'],axis=1)

# KNN for outlier detection
K = 5  # Number of neighbors
neigh = NearestNeighbors(n_neighbors=K)
neigh.fit(X)
distances, indices = neigh.kneighbors(X)

# Determine an outlier threshold
outlier_threshold = np.mean(distances[:, -1]) + 2 * np.std(distances[:, -1])

# Identify outliers
outliers = distances[:, -1] > outlier_threshold

# Remove outliers
X_cleaned = X[~outliers]
y_cleaned = y[~outliers]

print(f'Number of outliers removed: {len(outliers) - X_cleaned.shape[0]}')

score_8 = train_random_forest(X_cleaned,y_cleaned)
```

```
Number of outliers removed: 11
0.734151870484302
```

Experiment 9: Isolation Forest Outlier Detection

In [18]:
```
from sklearn.ensemble import IsolationForest

y = df_imputed['selling_price']
X = df_imputed.drop(['selling_price'],axis=1)

# Apply Isolation Forest for Outlier Detection
iso_forest = IsolationForest(contamination=0.5,random_state=2) # contamination is the proportion of outliers in the dataset
outliers = iso_forest.fit_predict(X)
outlier_index = np.where(outliers == -1) # -1 indicates outliers

# Remove Outliers
X_clean = X.drop(outlier_index[0])
y_clean = y.drop(outlier_index[0])

score_9 = train_random_forest(X_clean,y_clean)
```

```
0.7797388368084834
```

Feature Transformations - Discretization

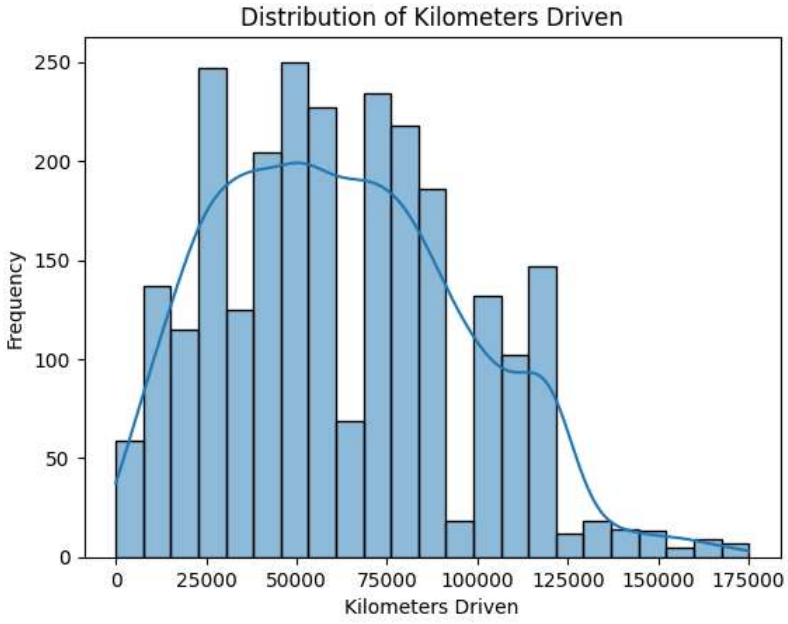
Discretization of continuous variables can be highly useful in certain data analysis and machine learning scenarios for:

- Simplification of Relationships: Discretization can simplify the relationships between variables by converting a complex continuous variable into a simpler, categorical one. This can make patterns more apparent and easier to interpret, especially in cases where the relationship is non-linear.

However, it's important to note that discretization should be used judiciously, as it involves the loss of information. The choice of bins (i.e., how to discretize) is crucial and can significantly affect the analysis outcomes. The benefits of discretization need to be weighed against the potential loss of granularity and information. But in our case, descritization helped the model to get a better understanding of accuracy.

Visualize the distribution of a feature named 'km_driven'

In [19]:
```
sns.histplot(X_clean['km_driven'], kde=True) # kde for Kernel Density Estimate
plt.xlabel('Kilometers Driven')
plt.ylabel('Frequency')
plt.title('Distribution of Kilometers Driven')
plt.show()
```



Observation: The distribution is slightly normal, and slightly right skewed

Experiment 10: Interval Based Binning

In [20]:
```
'''
In interval based binning, we are going to -
    1) Create numerical bins without any labels
    2) Use the mid-point of each bin as the value
'''

interval_bin_X = X_clean.copy()
interval_bin_y = y_clean.copy()

bin_edges = list(range(0,175001, 25000))  # Bin edges from 0 to 175,000 with a step of 25,000

# Use pd.cut without labels to get bin codes (integers)
interval_bin_X['km_driven_bin_codes'] = pd.cut(interval_bin_X['km_driven'], bins=bin_edges, labels=False, include_lowest=True, right=True)
```

```
# Calculate mid-points of bins
bin_midpoints = [(bin_edges[i] + bin_edges[i+1])/2 for i in range(len(bin_edges)-1)]

# Map bin codes to mid-points
interval_bin_X['km_driven_binned'] = interval_bin_X['km_driven_bin_codes'].map(dict(enumerate(bin_midpoints)))

del interval_bin_X['km_driven']
del interval_bin_X['km_driven_bin_codes']

score_10 = train_random_forest(interval_bin_X,interval_bin_y)
```

```
0.7828672900046549
```

Experiment 11: Frequency Based Binning

In [21]:
```
freq_bin_X = X_clean.copy()
freq_bin_y = y_clean.copy()

num_bins = 10  # You can choose the number of bins
freq_bin_X['km_driven_binned'], bins = pd.qcut(freq_bin_X['km_driven'], q=num_bins, labels=False, retbins=True, precision=0, duplicates='drop

del freq_bin_X['km_driven']

score_11 = train_random_forest(freq_bin_X,freq_bin_y)
```

```
0.7756906917404713
```

Experiment 12: Threshold Based Binning

In [22]:
```
thresh_bin_X = X_clean.copy()
thresh_bin_y = y_clean.copy()

bins = [0, 175000/2, 175000, float('inf')]  # float('inf') for upper bound
labels = [0, 1, 2] # 0 = low, 1 = medium, 2 = high
thresh_bin_X['km_driven_category'] = pd.cut(thresh_bin_X['km_driven'], bins=bins, labels=labels, include_lowest=True)

del thresh_bin_X['km_driven']

score_12 = train_random_forest(thresh_bin_X,thresh_bin_y)
```

```
0.763326775046851
```

The effectiveness of outlier detection methods was evaluated, and it was observed that using Isolation Forest for outlier detection yielded better results compared to the K-Nearest Neighbors (KNN) approach. Specifically, the Isolation Forest method achieved a score of 0.78, which surpassed the performance of KNN, which scored 0.73. This outcome indicates that the Isolation Forest method was more efficient and accurate in identifying outliers within the dataset, making it a preferable choice for this particular task.

A 1:1 feature transformation can be highly beneficial in data analysis and machine learning for 2 reasons:

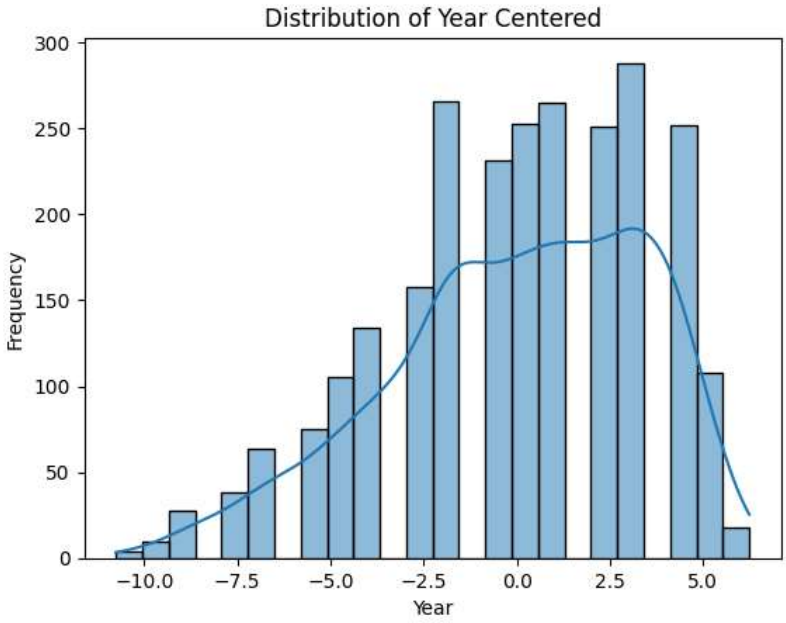
- **Data Standardization**: Transformations can standardize data to a common scale without distorting differences in the ranges of values. This is essential for many machine learning algorithms that assume features are on a similar scale.
- **Enhancing Feature Importance**: In some cases, a transformation can help in highlighting the importance of a feature that might not be apparent in its original form. This can lead to the discovery of new insights and more effective models.

1:1 Feature Tranformation

Experiment 13: Centering

In [23]:
```
# I am going to perform centering for year and see the affect it has on the model

center_X = interval_bin_X.copy()
center_y = interval_bin_y.copy()

year_mean = center_X['year'].mean()
center_X['year_centered'] = center_X['year'] - year_mean

del center_X['year']

score_13 = train_random_forest(center_X,center_y)

sns.histplot(center_X['year_centered'], kde=True) # kde for Kernel Density Estimate
plt.xlabel('Year')
plt.ylabel('Frequency')
plt.title('Distribution of Year Centered')
plt.show()
```

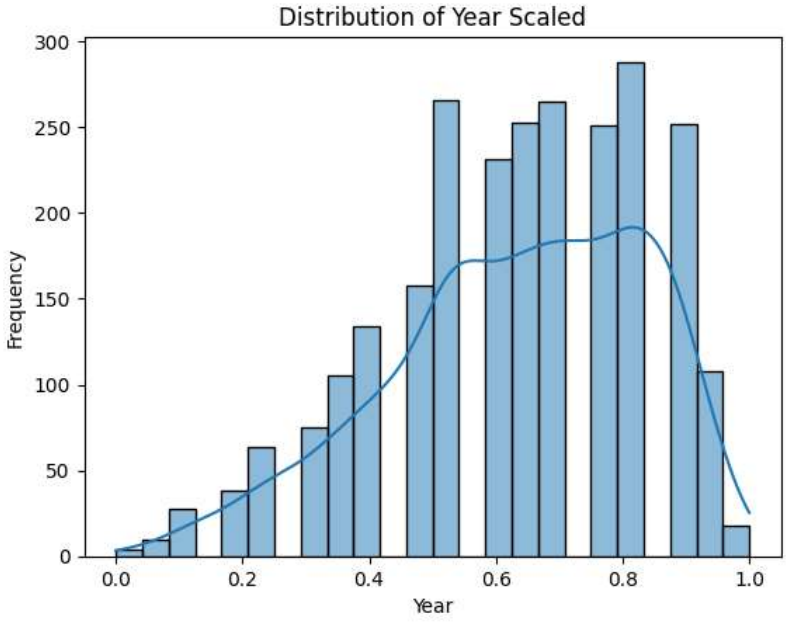
```
0.7826613552993049
```



Experiment 14: Scaling

In [24]:
```
from sklearn.preprocessing import MinMaxScaler

minmax_X = interval_bin_X.copy()
minmax_y = interval_bin_y.copy()

scaler = MinMaxScaler()
minmax_X['year_scaled'] = scaler.fit_transform(minmax_X[['year']])

del minmax_X['year']

score_14 = train_random_forest(minmax_X,minmax_y)

sns.histplot(minmax_X['year_scaled'], kde=True) # kde for Kernel Density Estimate
plt.xlabel('Year')
plt.ylabel('Frequency')
plt.title('Distribution of Year Scaled')
plt.show()
```

```
0.7825124658462738
```

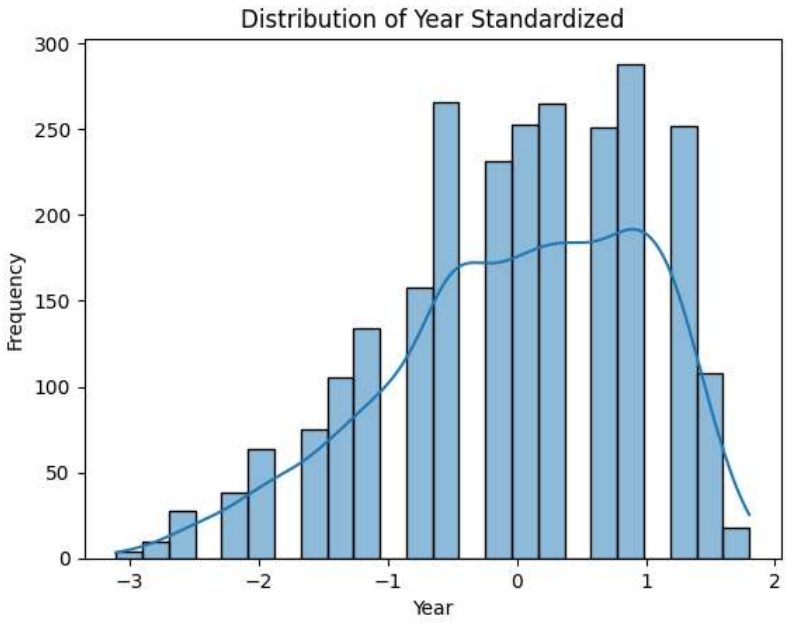


Experiment 15: Standard Scaling

In [25]:
```
from sklearn.preprocessing import StandardScaler

stdscale_X = interval_bin_X.copy()
stdscale_y = interval_bin_y.copy()

scaler = StandardScaler()
stdscale_X['year_standardized'] = scaler.fit_transform(stdscale_X[['year']])

del stdscale_X['year']
```

```
score_15 = train_random_forest(stdscale_X,stdscale_y)

sns.histplot(stdscale_X['year_standardized'], kde=True) # kde for Kernel Density Estimate
plt.xlabel('Year')
plt.ylabel('Frequency')
plt.title('Distribution of Year Standardized')
plt.show()
```

```
0.7828459394305441
```



There are several reasons why 1:1 feature transformations like scaling, centering, and standardization might not have a significant impact when applied to a feature in a Random Forest model:

- **Inherent Nature of Random Forests**: Random Forest, as an ensemble of decision trees, is relatively insensitive to the scale of features. Decision trees split data based on thresholds and are not influenced by the absolute scale of variables. Hence, transformations that primarily affect scale, like standardization and normalization, might not lead to substantial performance improvements in Random Forest models.
- **Feature Importance**: The specific feature I transformed might not be very influential in predicting the outcome in my model. Random Forest models can inherently handle a mix of relevant and irrelevant features, and if the transformed feature has limited predictive power, its transformation won't significantly impact the model's performance.

1:many Feature Transformations

Experiment 16 : Basis Expansion

In [26]:
```
from sklearn.preprocessing import PolynomialFeatures

BE_X = stdscale_X.copy()
BE_y = stdscale_y.copy()

features = BE_X[['year_standardized', 'km_driven_binned']]

# Create a PolynomialFeatures object with the desired degree of expansion
poly = PolynomialFeatures(degree=2)  # This will create quadratic features

# Fit and transform the features
expanded_features = poly.fit_transform(features)

# Manually create feature names for the expanded dataset
feature_names = ['1']
for i in range(1, len(features.columns) + 1):
    feature_names.append(f'x{i}')
    feature_names += [f'x{i}^2']
    for j in range(i+1, len(features.columns) + 1):
        feature_names.append(f'x{i}x{j}')

# Create a DataFrame for the expanded features
expanded_df = pd.DataFrame(expanded_features, columns=feature_names)

# Drop the '1' column as it's just the intercept term
expanded_df = expanded_df.drop('1', axis=1)

BE_X.reset_index(drop=True, inplace=True)
expanded_df.reset_index(drop=True, inplace=True)
```

```
# Concatenate the expanded features back to the original dataframe
BE_X = pd.concat([BE_X, expanded_df], axis=1)

del BE_X['km_driven_binned']
del BE_X['year_standardized']

score_16 = train_random_forest(BE_X,BE_y)
BE_X.head()
```

```
0.782660370098961
```

Out[26]:

	name	fuel	seller_type	transmission	owner	x1	x1^2	x1x2	x2	x2^2
0	775.0	4.0	1.0	1.0	0.0	-1.950918	62500.0	3.806081	-121932.368847	3.906250e+09
1	1041.0	4.0	1.0	1.0	0.0	-1.950918	37500.0	3.806081	-73159.421308	1.406250e+09
2	505.0	1.0	1.0	1.0	0.0	-0.508142	87500.0	0.258209	-44462.457458	7.656250e+09
3	118.0	4.0	1.0	1.0	0.0	0.934633	37500.0	0.873539	35048.743487	1.406250e+09
4	279.0	1.0	1.0	1.0	2.0	0.068968	137500.0	0.004757	9483.078179	1.890625e+10

Experiment 17 : One Hot Encoding

In [27]:

```
import pandas as pd

OHE_X = stdscale_X.copy()
OHE_y = stdscale_y.copy()

# If you have a single column to encode:
OHE_X = pd.get_dummies(OHE_X, columns=['fuel'])

score_17 = train_random_forest(OHE_X,OHE_y)

OHE_X.head()
```

```
0.7816371315209341
```

Out[27]:

	name	seller_type	transmission	owner	km_driven_binned	year_standardized	fuel_1.0	fuel_3.0	fuel_4.0
0	775.0	1.0	1.0	0.0	62500.0	-1.950918	False	False	True
1	1041.0	1.0	1.0	0.0	37500.0	-1.950918	False	False	True
2	505.0	1.0	1.0	0.0	87500.0	-0.508142	True	False	False
3	118.0	1.0	1.0	0.0	37500.0	0.934633	False	False	True
4	279.0	1.0	1.0	2.0	137500.0	0.068968	True	False	False

Experiment 18: Hashing

In [28]:

```
hash_X = stdscale_X.copy()
hash_y = stdscale_y.copy()

# Apply the hash function to the column
hash_X['hashed_fuel'] = hash_X['fuel'].apply(lambda x: hash(str(x)))
del hash_X['fuel']

score_18 = train_random_forest(hash_X,hash_y)
hash_X.head()
```

```
0.7833874469141888
```

Out[28]:

	name	seller_type	transmission	owner	km_driven_binned	year_standardized	hashed_fuel
0	775.0	1.0	1.0	0.0	62500.0	-1.950918	7551575224084988209
1	1041.0	1.0	1.0	0.0	37500.0	-1.950918	7551575224084988209
2	505.0	1.0	1.0	0.0	87500.0	-0.508142	-4396501268508799832
3	118.0	1.0	1.0	0.0	37500.0	0.934633	7551575224084988209
4	279.0	1.0	1.0	2.0	137500.0	0.068968	-4396501268508799832

Experiment 19: Encoding Interval/Ratio Variables

In [29]:

```
IRV_X = stdscale_X.copy()
IRV_y = stdscale_y.copy()

# Km_driven binned was descretized in experiment

# If you have a single column to encode:
IRV_X = pd.get_dummies(IRV_X, columns=['km_driven_binned'])

# If you have multiple columns to encode, pass a list of column names:
# encoded_df = pd.get_dummies(df, columns=['category_column1', 'category_column2', ...])
```

```
score_19 = train_random_forest(OHE_X,OHE_y)
IRV_X.head()
```

```
0.7816371315209341
```

Out[29]:

	name	fuel	seller_type	transmission	owner	year_standardized	km_driven_binned_12500.0	km_driven_binned_37500.0	km_driven_binned_62500.0	km
0	775.0	4.0	1.0	1.0	0.0	-1.950918	False	False	True	
1	1041.0	4.0	1.0	1.0	0.0	-1.950918	False	True	False	
2	505.0	1.0	1.0	1.0	0.0	-0.508142	False	False	False	
3	118.0	4.0	1.0	1.0	0.0	0.934633	False	True	False	
4	279.0	1.0	1.0	1.0	2.0	0.068968	False	False	False	

Many:Many Transformations - Feature Expansion

Experiment 20: Feature Expansion - Kernel-Induced Feature Expansion

In [30]:

```
from sklearn.metrics.pairwise import rbf_kernel

kernel_X = BE_X.copy()
kernel_y = BE_y.copy()

# Apply RBF kernel
gamma = 0.1  # This is a parameter we can tune
X_kernel = rbf_kernel(kernel_X, gamma=gamma)

score_20 = train_random_forest(X_kernel,kernel_y)
```

```
0.4614450819521898
```

Many:Many Transformations - Feature Contraction

Experiment 21: PCA

In [31]:

```
from sklearn.decomposition import PCA

PCA_X = IRV_X.copy()
PCA_y = IRV_y.copy()

X = PCA_X
y = PCA_y

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Applying PCA
# You can choose the number of components, e.g., n_components=2 for 2D, or a variance ratio like 0.95
pca = PCA(n_components=0.4)
X_pca = pca.fit_transform(X_scaled)

# The result is in X_pca, and you can create a new DataFrame from it
df_pca = pd.DataFrame(data=X_pca, columns=[f'PC{i+1}' for i in range(X_pca.shape[1])])

score_21 = train_random_forest(df_pca,y)
```

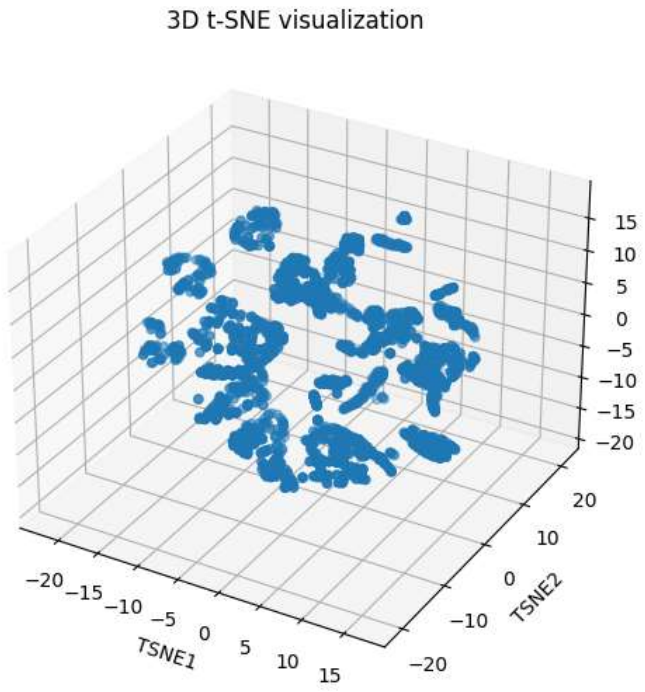
```
0.495150131976824
```

Experiment 22: t-SNE

In [32]:

```
from sklearn.manifold import TSNE

tSNE_X = IRV_X.copy()
tSNE_y = IRV_y.copy()

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(tSNE_X)

# Applying t-SNE
tsne = TSNE(n_components=3, random_state=42)  # Use n_components=3 for 3D
X_tsne = tsne.fit_transform(X_scaled)

# Converting to DataFrame for easier plotting
```

```
df_tsne = pd.DataFrame(data=X_tsne, columns=['TSNE1', 'TSNE2', 'TSNE3'])

# Plotting in 3D
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(df_tsne['TSNE1'], df_tsne['TSNE2'], df_tsne['TSNE3'])
ax.set_xlabel('TSNE1')
ax.set_ylabel('TSNE2')
ax.set_zlabel('TSNE3')
plt.title('3D t-SNE visualization')
plt.show()

score_22 = train_random_forest(X_tsne,tSNE_y)
```




```
0.49791465246877054
```

Experiment 23: Isometric Mapping

In [33]:
```
from sklearn.manifold import Isomap

import warnings
warnings.filterwarnings("ignore")

isomap_X = IRV_X.copy()
isomap_y = IRV_y.copy()

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(isomap_X)

# Applying Isomap
# Choose the number of components and neighbors
isomap = Isomap(n_components=7, n_neighbors=5)
X_isomap = isomap.fit_transform(X_scaled)

score_23 = train_random_forest(X_isomap,isomap_y)
```

```
0.5921998328051177
```

Feature selection

Both SFS and SBS are particularly useful in scenarios where the number of features is large compared to the number of observations, or when the features are highly correlated with each other. By carefully selecting the most relevant features, these techniques can enhance model performance, reduce computational cost, and increase the interpretability of the model.

Experiment 24: Sequential Forward Generation

In [34]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SequentialFeatureSelector

# Load a sample dataset
X, y = IRV_X,IRV_y

# Initialize your classifier
```

```
regressor = RandomForestRegressor(random_state=42)

# Initialize Sequential Forward Selection
sfs = SequentialFeatureSelector(regressor, n_features_to_select='auto', direction='forward')

# Fit SFS
sfs.fit(X, y)

# Get the indices of the selected features
selected_features = sfs.get_support(indices=True)

# Use these features for further modeling
X_selected = sfs.transform(X)

# Get column names
feature_names = X.columns

print('selected_features')
for feature in selected_features:
    print(feature_names[feature],end=', ')
print()
score_24 = train_random_forest(X_selected,y)
```

```
selected_features
name, seller_type, year_standardized, km_driven_binned_12500.0, km_driven_binned_62500.0, km_driven_binned_87500.0,
0.8480610045000848
```

Experiment 25: Sequential Backward Generation

In [35]:
```
# Load a sample dataset
X, y = IRV_X,IRV_y

# Initialize your classifier
regressor = RandomForestRegressor(random_state=42)

# Initialize Sequential Forward Selection
sfs = SequentialFeatureSelector(regressor, n_features_to_select='auto', direction='backward')

# Fit SFS
sfs.fit(X, y)

# Get the indices of the selected features
selected_features = sfs.get_support(indices=True)

# Use these features for further modeling
X_selected = sfs.transform(X)

# Get column names
feature_names = X.columns

print('selected_features')
for feature in selected_features:
    print(feature_names[feature],end=', ')
print()
score_24 = train_random_forest(X_selected,y)
```

```
selected_features
name, seller_type, year_standardized, km_driven_binned_12500.0, km_driven_binned_62500.0, km_driven_binned_87500.0, km_driven_binned_137500.0,
0.8554198728891687
```

Summary

Using data preperation we imporoved the model from 57.23 to 85.54

There is a **28.31 points increase from baseline to best model**

Experiment	Description	Score	Improvement	Comment
Baseline	Baseline	**57.23**	-	Baseline 1
Duplicate Instances				
Exp 1	Handling Duplicate Instances	67.09	+9.86	Baseline 2
Missingness in features				
Exp 2	Mean substitution	66.98	-0.11	
Exp 3	Median substitution	66.98	-0.11	
Exp 4	Frequency Substitution	66.98	-0.11	
Exp 5	Multiple Imputation	67.46	+0.37	
Exp 6	kNN imputer	69.42	+2.33	Baseline 3
Compound Variables				
Exp 7	Handling Compound Variables	77.12	+7.7	Baseline 4

Experiment	Description	Score	Improvement	Comment
Outlier Detection				
Exp 8	kNN outlier detection	73.42	-3.7	
Exp 9	Isolation Forest	77.97	+0.85	Baseline 5
Feature Transformation - Discretization				
Exp 10	Interval Based Binning	78.29	+0.32	Baseline 6
Exp 11	Frequency Based Binning	77.57	-0.4	
Exp 12	Threshold Based Binning	76.33	-1.64	
1:1 Feature Transformation				
Exp 13	Centering	78.27	-0.02	
Exp 14	Scaling	78.25	-0.04	
Exp 15	Standard Scaling	78.28	-0.01	
1:many Feature Transformation				
Exp 16	Basis Expansion	78.27	-0.02	
Exp 17	One Hot Encoding	78.16	-0.13	
Exp 18	Hashing	78.16	-0.13	
Exp 19	Encoding Interval/Ratio Variables	78.16	-0.13	
Many:Many Transformation - Feature Expansion				
Exp 20	Kernel-Induced Feature Expansion	46.14	-32.15	
Many:Many Transformation - Feature Contraction				
Exp 21	PCA	49.52	-28.77	
Exp 22	t-SNE	49.79	-28.5	
Exp 23	Isometric Mapping	59.22	-19.07	
Feature Selection				
Exp 24	Sequential Forward Generation	84.81	+6.52	
Exp 25	Sequential Backward Generation	**85.54**	+7.25	**BEST MODEL**