Car_price_Prediction

May 2, 2025

0.1 Problem Statement

The goal of this project is to build a robust machine learning model to predict the selling prices of used cars based on key vehicle attributes and historical sales data. The dataset, sourced from Mobil123 through web scraping, contains both numerical and categorical features that describe a car's specifications, usage, and ownership history.

The primary business problem is to enable more accurate, data-driven car valuations that benefit both buyers and sellers in the used car market. To address this, the project focuses on:

1. Identifying which features significantly influence car prices through exploratory data analysis and statistical testing.
2. Managing multicollinearity among highly correlated variables.
3. Selecting and engineering the most relevant features to improve model performance.
4. Comparing and evaluating multiple regression models (XGBoost, Gradient Boosting, Random Forest) to determine the most accurate and efficient approach for price prediction.

The project concludes with the deployment of the best-performing model (XGBoost), integrated with full preprocessing via a pipeline. This model is packaged and ready for real-world application through an interactive Streamlit-based web app.

0.2 Summary of Insights for Car Selling Price Prediction

Statistical analysis was conducted to understand the factors influencing car selling prices and prepare the data for predictive modeling.

Key Findings from EDA and Correlation Analysis:

- **Strong Positive Drivers of Price:** Numerical features like `max_power`, `torque`, `engine` size, and the manufacturing `year` showed strong positive correlations with the selling price. This indicats that cars with higher specifications and newer models tend to command higher prices.
- **Negative Impact of Usage and Age:** Numerical features such as `car_age`, `mileage`, and `km_driven` exhibited negative correlations, suggesting that older and more used vehicles generally have lower selling prices.

Categorical Feature Importance: 1. Welch's ANOVA analysis indicated that categorical features like `fuel_type` and `transmission` have a statistically significant impact on the mean selling price.

2. The number of previous owners (`owner`) also showed a moderate influence.

3. Despite the relatively low F-statistic, (`name`) is included in model training because it is a high-cardinality categorical feature (many unique car names) and is highly imbalanced. These characteristics dilute the overall variance captured by ANOVA yet the specific car make and model are strong determinants of car pricee.

4. `seller_type` is considered less likely to contribute meaningfully to the model and was excluded. This is because it reflects the context of the transaction rather than intrinsic characteristics of the car.

Multicollinearity Management:

1. A strong positive correlation has been identified between `engine`, `max_power`, and `torque`.

2. A perfect negative correlation between `car_age` and `year`.

One representative feature from each correlated group (`engine` or `max_power`, and `year`) was chosen for model training to avoid redundancy.

3. Given that `mileage` and `km_driven` represent the same concept of car usage, `km_driven` has been used for the model training due to its slightly stronger negative correlation with price.

- **Feature Exclusion** `seller_type` is considered less likely to contribute meaningfully to the model and was excluded. This is because it reflects the context of the transaction rather than intrinsic characteristics of the car.

Model Evaluation:

Model	MAE	MSE	RMSE	R^2 Score
XGBoost (no outliers)	65,224.45	8,681,822,530.02	93,176.30	0.86
XGBoost	87,513.79	32,608,928,997.74	180,579.43	0.85
Gradient Boosting	88,424.50	36,732,389,083.97	191,656.96	0.83
Random Forest	91,316.73	39,347,158,917.14	198,361.18	0.82

XGBoost delivered the best performance across all evaluation metrics, making it the ideal model for predicting car prices. Removing outliers greatly improved its performance by cutting RMSE nearly in half and reducing MSE from 32.6 billion to 8.7 billion. This optimized version of XGBoost will be deployed as the final prediction tool.

[ ]:
```python
# Importing Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import levene
import statsmodels.api as sm
```

```
from statsmodels.formula.api import ols
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
import xgboost as xgb
from xgboost import XGBRegressor
import pickle
import joblib
from matplotlib.ticker import ScalarFormatter

import warnings
warnings.filterwarnings('ignore')
```

[ ]:
```
pd.set_option('display.float_format', '{:,.2f}'.format)
```

[ ]:
```
# Reading the data from Google sheets
from google.colab import auth
auth.authenticate_user()
import gspread
from google.auth import default
creds, _ = default()
gc = gspread.authorize(creds)
worksheet = gc.open('Cardetails').sheet1
# get_all_values gives a list of rows.
rows = worksheet.get_all_values()
# Convert to a DataFrame and render.
import pandas as pd
df =pd.DataFrame.from_records(rows)
```

0.3 Data Inspection and cleaning

[ ]:
```
# setting first row as headers
df.columns = df.iloc[0]
df = df.iloc[1:]
df.head()
```

[ ]:
```
0                          name  year selling_price km_driven    fuel  \
1         Maruti Swift Dzire VDI  2014        450000    145500  Diesel
2   Skoda Rapid 1.5 TDI Ambition  2014        370000    120000  Diesel
3       Honda City 2017-2020 EXi  2006        158000    140000  Petrol
```

```
4     Hyundai i20 Sportz Diesel  2010          225000    127000  Diesel
5        Maruti Swift VXI BSIII  2007          130000    120000  Petrol

0 seller_type transmission         owner     mileage   engine   max_power  \
1  Individual       Manual   First Owner   23.4 kmpl  1248 CC      74 bhp
2  Individual       Manual  Second Owner  21.14 kmpl  1498 CC  103.52 bhp
3  Individual       Manual   Third Owner   17.7 kmpl  1497 CC      78 bhp
4  Individual       Manual   First Owner   23.0 kmpl  1396 CC      90 bhp
5  Individual       Manual   First Owner   16.1 kmpl  1298 CC    88.2 bhp

0                    torque seats
1            190Nm@ 2000rpm     5
2       250Nm@ 1500-2500rpm     5
3     12.7@ 2,700(kgm@ rpm)     5
4  22.4 kgm at 1750-2750rpm     5
5     11.5@ 4,500(kgm@ rpm)     5
```

[ ]:
```python
# size of the data
df.shape
```

[ ]: (8128, 13)

[ ]:
```python
# Checking for duplicates
df[df.duplicated(keep=False)]

# droping duplicates
df.drop_duplicates(inplace=True)
```

[ ]:
```python
df.head()
```

[ ]:
```
0                          name  year selling_price km_driven    fuel  \
1        Maruti Swift Dzire VDI  2014         450000    145500  Diesel
2  Skoda Rapid 1.5 TDI Ambition  2014         370000    120000  Diesel
3      Honda City 2017-2020 EXi  2006         158000    140000  Petrol
4     Hyundai i20 Sportz Diesel  2010         225000    127000  Diesel
5        Maruti Swift VXI BSIII  2007         130000    120000  Petrol

0 seller_type transmission         owner     mileage   engine   max_power  \
1  Individual       Manual   First Owner   23.4 kmpl  1248 CC      74 bhp
2  Individual       Manual  Second Owner  21.14 kmpl  1498 CC  103.52 bhp
3  Individual       Manual   Third Owner   17.7 kmpl  1497 CC      78 bhp
4  Individual       Manual   First Owner   23.0 kmpl  1396 CC      90 bhp
5  Individual       Manual   First Owner   16.1 kmpl  1298 CC    88.2 bhp

0                    torque seats
1            190Nm@ 2000rpm     5
2       250Nm@ 1500-2500rpm     5
```

```
3     12.7@ 2,700(kgm@ rpm)     5
4  22.4 kgm at 1750-2750rpm     5
5     11.5@ 4,500(kgm@ rpm)     5
```

[ ]:
```
# function for cleaning data
def get_clean_car_detail(car_detail):
  car_detail = car_detail.split(" ")[0]
  return car_detail
```

[ ]:
```
df.name = df.name.apply(get_clean_car_detail)
df.mileage = df.mileage.apply(get_clean_car_detail)
df.engine = df.engine.apply(get_clean_car_detail)
df.max_power = df.max_power.apply(get_clean_car_detail)
```

[ ]:
```
import re

def extract_torque_value(torque):
    if pd.isnull(torque):
        return None
    # Search for the first float or integer in the string
    match = re.search(r'\d+\.\d+|\d+', str(torque))
    if match:
        return float(match.group())
    return None
```

[ ]:
```
df.torque = df.torque.apply(extract_torque_value)
```

[ ]:
```
# setting year to type int
df.year = df.year.astype(int)

df['car_age'] = 2025 - df['year']
```

[ ]:
```
# checking for missing values
df.isnull().sum()

# replacing missing values with nan
df.replace('', np.nan, inplace=True)
```

[ ]:
```
df.head(2)
```

[ ]:
```
0    name  year selling_price km_driven    fuel seller_type transmission  \
1  Maruti  2014        450000    145500  Diesel  Individual       Manual
2   Skoda  2014        370000    120000  Diesel  Individual       Manual

0         owner mileage engine max_power  torque seats  car_age
1   First Owner    23.4   1248        74   190.0     5       11
2  Second Owner   21.14   1498    103.52   250.0     5       11
```

[ ]:
```
df[df['max_power'].str.contains('bhp', na = False)]['max_power']

# Replacing bhp with nothing and replace with nan
df['max_power'] = df['max_power'].str.replace('bhp', '')
df['max_power'] = df.max_power.replace('', np.nan)
```

0.4 Statistical Analysis

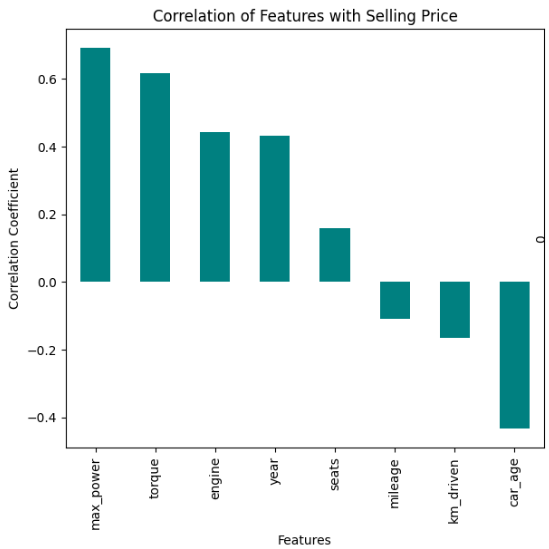
0.4.1 Figure: Correlation of Numerical Variables with Target Variable (selling_price)

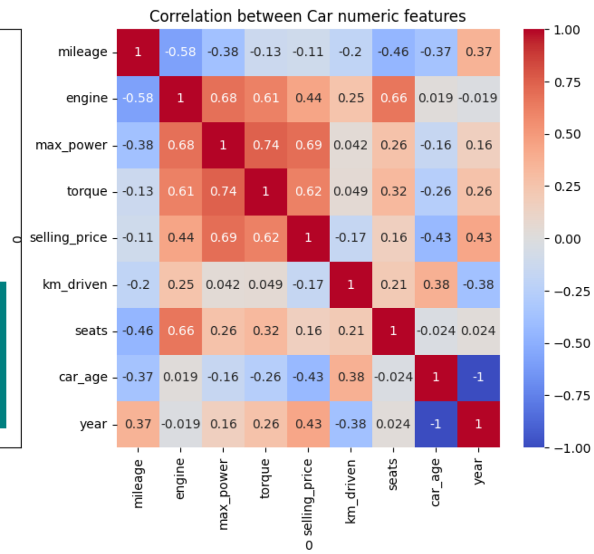
[ ]:
```
nums_float = ['mileage', 'engine', 'max_power', 'torque', 'selling_price',␣
↪'km_driven', 'seats', 'car_age']
nums_int = ['year']
df[nums_float] = df[nums_float].astype(float)
df[nums_int] = df[nums_int].astype(int)
```

[ ]:
```
nums = nums_float + nums_int
len(nums)
```

[ ]: 9

[ ]:
```
# Subplot
correlation_matrix = df[nums].corr()
# Focus on correlation of each feature with 'selling_price'
price_corr = correlation_matrix['selling_price']

fig, ax = plt.subplots(1, 2, figsize=(15, 6))
# Visualize the correlation with a bar plot
plt.figure(figsize=(8, 4))
price_corr.drop('selling_price').sort_values(ascending=False).plot(kind='bar',␣
↪color='teal', ax=ax[0])
ax[0].set_title("Correlation of Features with Selling Price")
ax[0].set_ylabel("Correlation Coefficient")
ax[0].set_xlabel("Features")

# Heat Map
df[nums].corr()

# Visualizing correlation between numeric features
plt.figure(figsize=(10,4))
sns.heatmap(df[nums].corr(), annot=True, cmap = 'coolwarm', ax=ax[1])
ax[1].set_title("Correlation between Car numeric features")

plt.tight_layout()
plt.show()
```






```
&lt;Figure size 800x400 with 0 Axes&gt;
&lt;Figure size 1000x400 with 0 Axes&gt;
```

Interpretation of the above graphs

- engine (0.69): Cars with larger engines tend to have significantly higher selling prices.
- max_power (0.69): Cars with higher maximum power are strongly associated with higher selling prices.
- torque (0.62): Higher torque values also correlate well with higher selling prices.
- year (0.43): Newer cars tend to have higher selling prices.
- car_age (-0.43): As cars get older, their selling price tends to decrease. This is the inverse of the correlation with year, which makes sense.
- mileage (-0.11): Higher mileage shows a weak negative correlation with selling price, suggesting that cars with more miles tend to be priced slightly lower.
- km_driven (-0.17): Similar to mileage, it shows a weak negative correlation.
- seats (0.16): The number of seats has a weak positive correlation with selling price.

Strong Positive Correlations Among Features (Potential Multicollinearity)

- engine, max_power, and torque show very strong positive correlations with each other (around 0.74 - 0.8). This suggests that these features are highly related. Either engine or max_power will be used for the model training, as their correlation with selling price is the same (0.69) and slightly greater than that of torque (0.62). Torque might be considered as an alternative if needed.
- car_age and year have a perfect negative correlation (-1), which is expected since one is directly derived from the other. Only year will be kept for the model training to avoid

redundancy.

In summary, the selling price of a car is most strongly correlated with the **engine** (or max_power), manufacturing **year**, and **km_driven** (negatively). The number of **seats** will also be considered in the model despite its weak positive correlation.

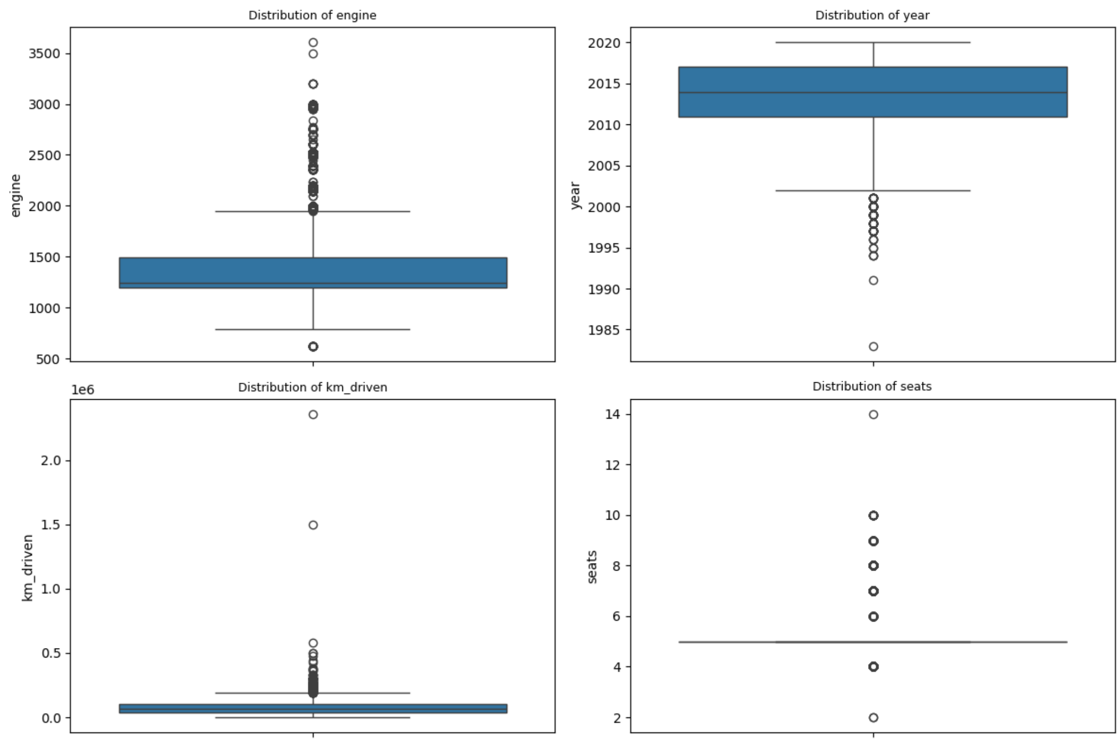
The highly correlated features (engine, max_power, torque and car_age, year) will be handled by selecting one representative from each group (**engine** is prefered).

Both mileage and km_driven measure car usage by total distance traveled. Mileage is in miles, while km_driven is in kilometers. km_driven shows a slightly stronger negative correlation with selling price (-0.17 vs. -0.11), meaning higher usage leads to a larger price drop when using km_driven, assuming consistent units.

Final Selection

- The model will include the following numerical features: **engine**, **year**, **km_driven** and **seats**.

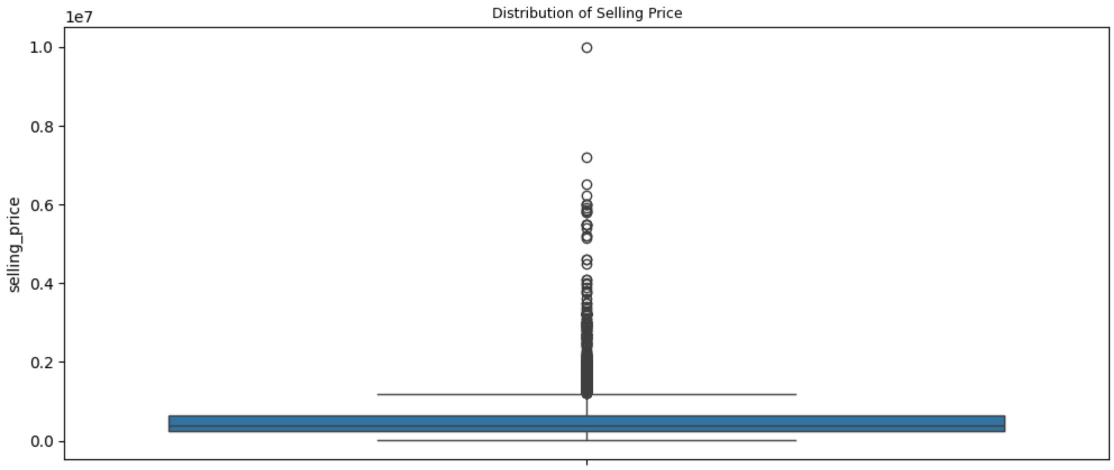
[ ]:
```python
num_features = ['engine', 'year', 'km_driven', 'seats']

# Box Plots
plt.figure(figsize=(12, 8))
for i, feature in enumerate(num_features, 1):
    plt.subplot(2, 2, i)
    sns.boxplot(y=df[feature])
    plt.title(f'Distribution of {feature}', fontsize=9)
    plt.xlabel('', fontsize=9)
plt.tight_layout()
plt.show()
```



[ ]:
```
df.km_driven.max()
```

[ ]: 2360457.0

[ ]:
```
# box plost showing distribution of selling price
plt.figure(figsize=(12, 5))
sns.boxplot(y=df['selling_price'])
plt.title('Distribution of Selling Price', fontsize=9)
plt.xlabel('', fontsize=9)
plt.show()
```



Distribution of Selling Price

Key Observations:

- **Median Inclination:** The median line within the box is noticeably closer to the lower quartile, confirming that the central tendency of the data leans towards lower selling prices.
- **Long Right Tail:** A significant number of outliers are present on the higher end of the price range, forming a long tail extending to the right. These represent a smaller proportion of cars with considerably higher selling prices.

Impact of Distribution on Predictions:

The right-skewed distribution of the selling price is likely to influence the model's predictions in the following ways:

1. **Better Performance on Lower Prices:** Due to the higher frequency of lower-priced cars in the training data, the model is expected to perform more accurately when predicting selling prices within this dominant range.

2. **Challenges with Higher Prices:** The relatively fewer examples of high-priced cars in the training data may lead to less accurate and more variable predictions for these vehicles.

3. **Potential for Underprediction:** Models trained on right-skewed data can sometimes exhibit a tendency to underpredict values in the longer, higher-value tail.

Understanding this distribution is crucial for interpreting the model's performance metrics and the "Predicted vs Actual Prices" plot later.

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

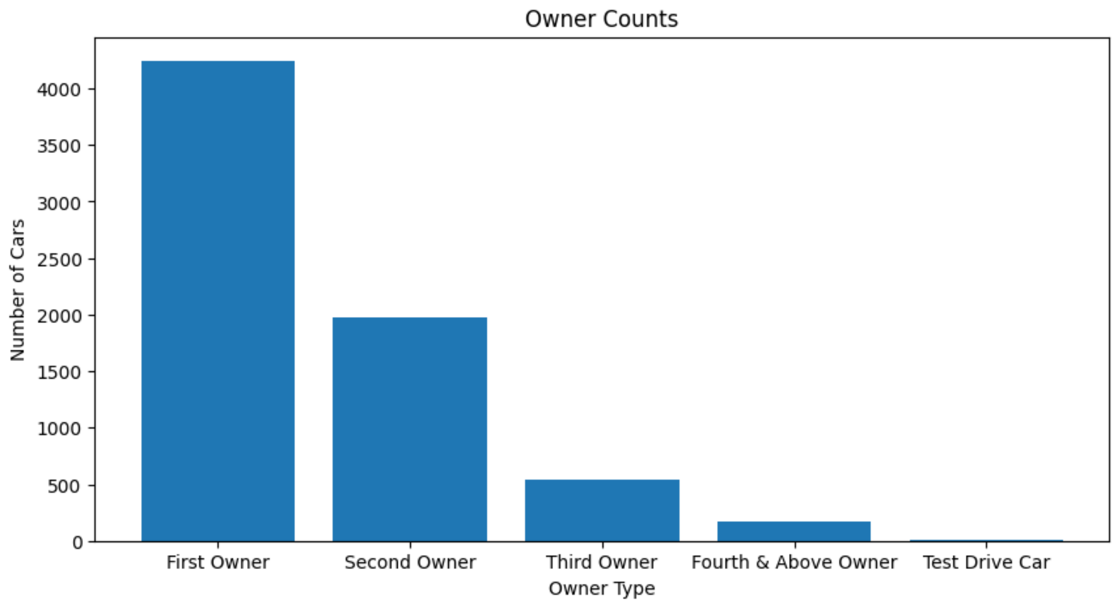
[ ]:
```
0             year  selling_price     km_driven     mileage       engine  \
count  6926.000000   6.926000e+03  6.926000e+03  6718.00000  6718.000000
mean   2013.420300   5.172707e+05  7.399568e+04    19.46531  1430.891337
std       4.078286   5.197670e+05  5.835810e+04     4.04915   493.493277
min    1983.000000   2.999900e+04  1.000000e+00     0.00000   624.000000
```

```
25%     2011.000000   2.500000e+05  4.000000e+04    16.80000  1197.000000
50%     2014.000000   4.000000e+05  7.000000e+04    19.44000  1248.000000
75%     2017.000000   6.335000e+05  1.000000e+05    22.50000  1498.000000
max     2020.000000   1.000000e+07  2.360457e+06    42.00000  3604.000000

0        max_power       torque        seats      car_age
count  6720.000000  6717.000000  6718.000000  6926.000000
mean     87.726919   160.854853     5.434653    11.579700
std      31.771619    91.630280     0.984230     4.078286
min       0.000000     4.800000     2.000000     5.000000
25%      67.100000    96.000000     5.000000     8.000000
50%      81.830000   146.000000     5.000000    11.000000
75%     100.000000   200.000000     5.000000    14.000000
max     400.000000   789.000000    14.000000    42.000000
```

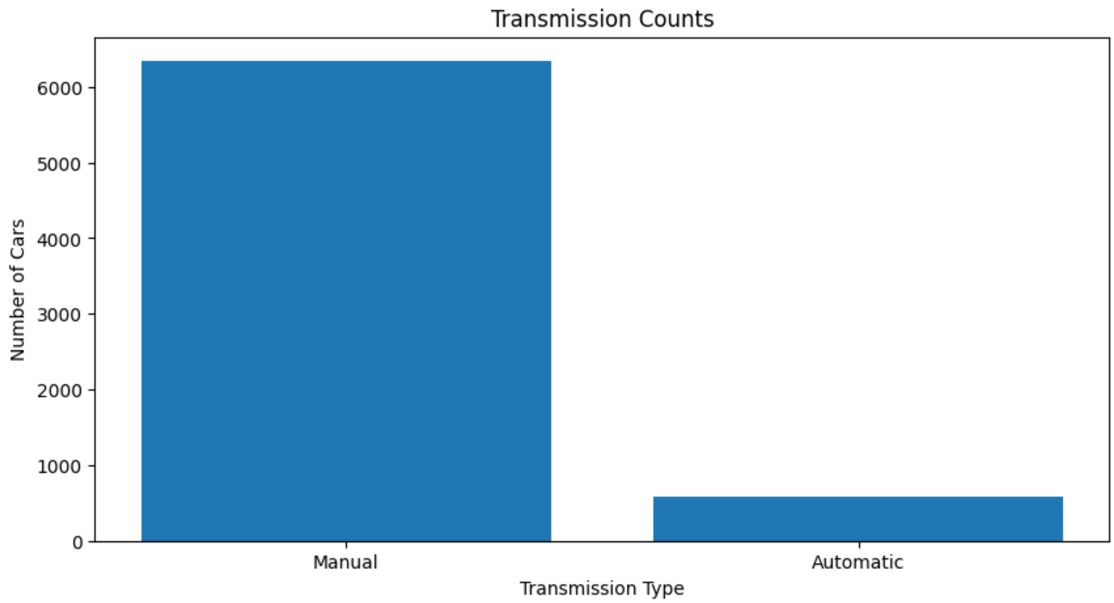
[ ]:
```
cat = ['name', 'fuel', 'seller_type', 'transmission', 'owner']
target = ['selling_price']
cat_target = cat + target
df[cat_target].head()
```

[ ]:
```
0     name    fuel seller_type transmission         owner  selling_price
1   Maruti  Diesel  Individual       Manual   First Owner       450000.0
2    Skoda  Diesel  Individual       Manual  Second Owner       370000.0
3    Honda  Petrol  Individual       Manual   Third Owner       158000.0
4  Hyundai  Diesel  Individual       Manual   First Owner       225000.0
5   Maruti  Petrol  Individual       Manual   First Owner       130000.0
```

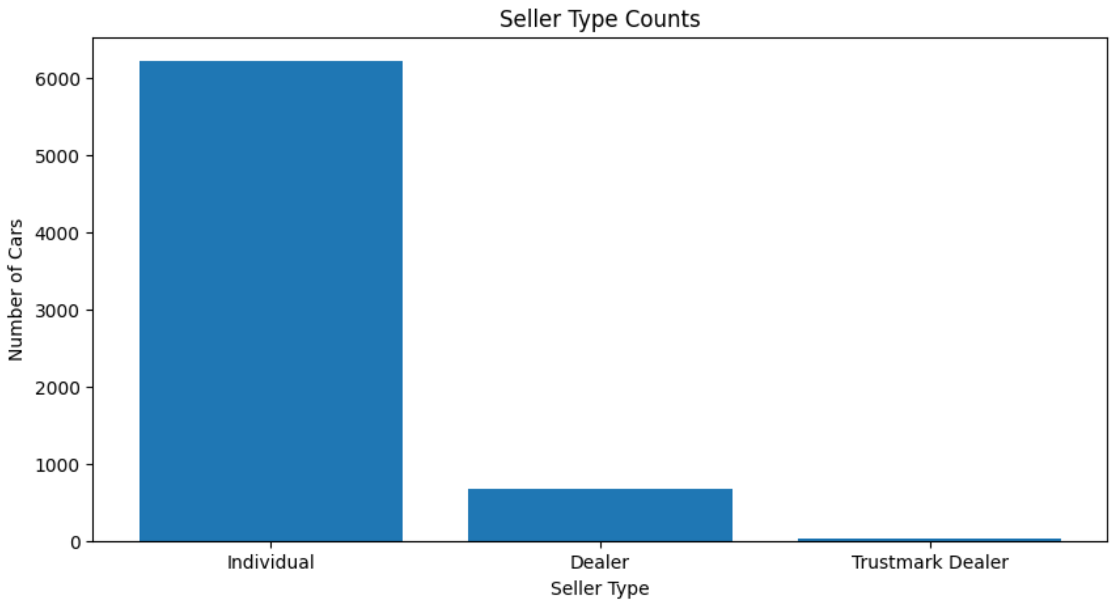
[ ]:
```
# counts for Owner
owner_counts = df['owner'].value_counts()

# Plot
plt.figure(figsize=(10,5))
plt.bar(owner_counts.index, owner_counts.values)
plt.title('Owner Counts')
plt.xlabel('Owner Type')
plt.ylabel('Number of Cars')
plt.show()
```



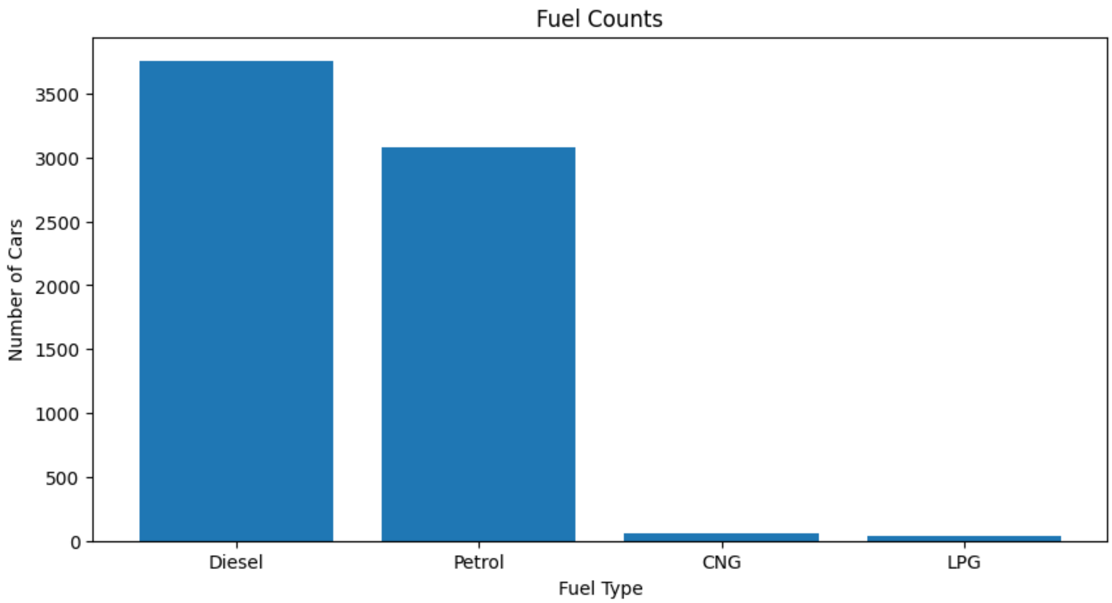
[ ]:
```
# Transmission counts
transmission_counts = df['transmission'].value_counts()

# Plot
plt.figure(figsize=(10,5))
plt.bar(transmission_counts.index, transmission_counts.values)
plt.title('Transmission Counts')
plt.xlabel('Transmission Type')
plt.ylabel('Number of Cars')
plt.show()
```



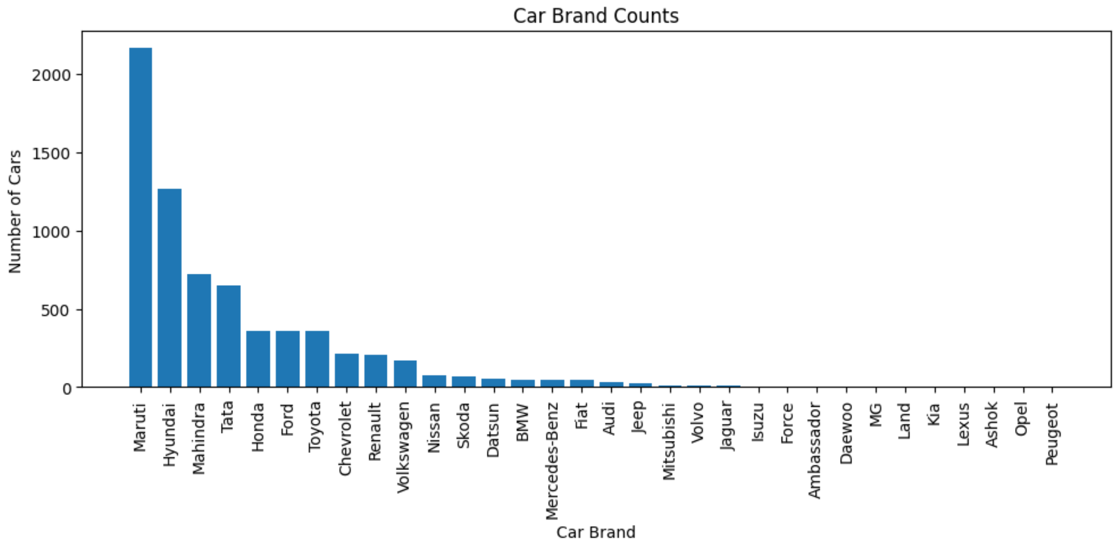
[ ]:
```python
# Counts (seller_type)
seller_counts = df['seller_type'].value_counts()

# Plot
plt.figure(figsize=(10,5))
plt.bar(seller_counts.index, seller_counts.values)
plt.title('Seller Type Counts')
plt.xlabel('Seller Type')
plt.ylabel('Number of Cars')
plt.show()
```



[ ]:
```
# Counts for Fuel
fuel_counts = df['fuel'].value_counts()

# Plot
plt.figure(figsize=(10,5))
plt.bar(fuel_counts.index, fuel_counts.values)
plt.title('Fuel Counts')
plt.xlabel('Fuel Type')
plt.ylabel('Number of Cars')
plt.show()
```



[ ]:
```
# Get counts of each car brand
name_counts = df['name'].value_counts()

# Plot
plt.figure(figsize=(10,5))
plt.bar(name_counts.index, name_counts.values)
plt.xticks(rotation=90)  # Rotate x-axis labels for readability
plt.title('Car Brand Counts')
plt.xlabel('Car Brand')
plt.ylabel('Number of Cars')
plt.tight_layout()
plt.show()
```



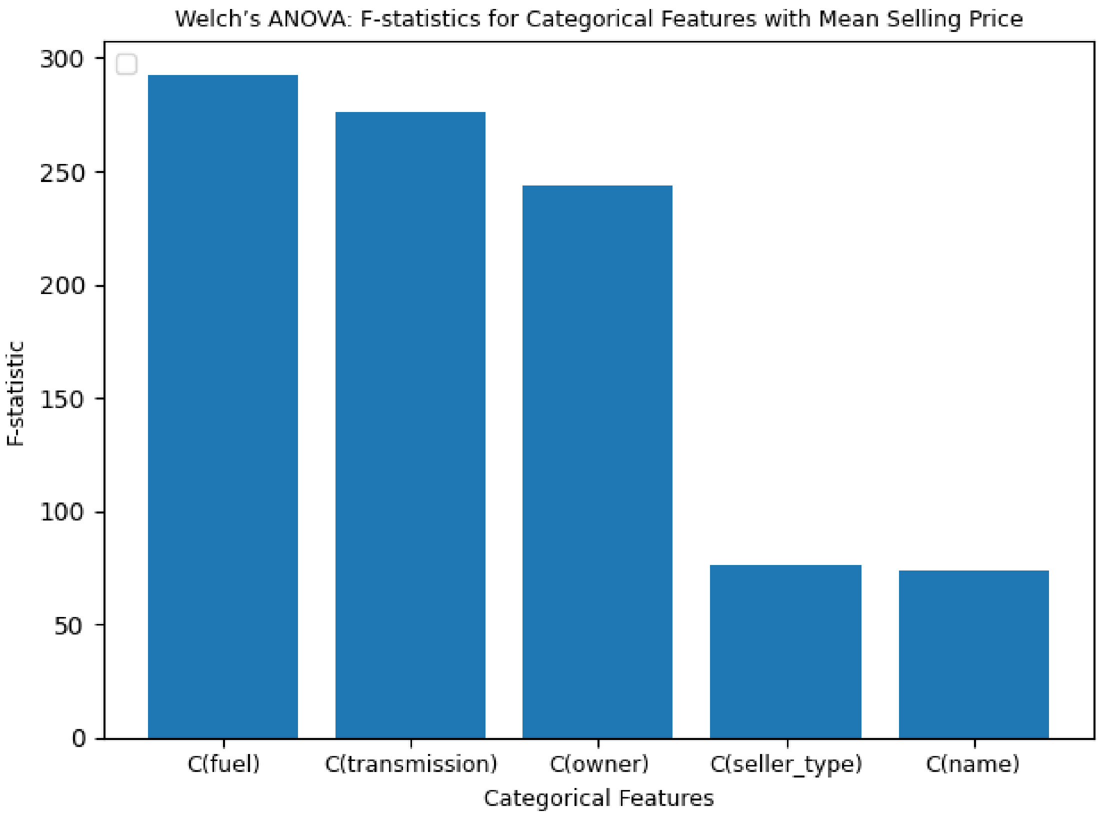
[ ]:
```python
anova_results = []
for col in cat:
  # performing welch anova
  formula = f'selling_price ~ C({col})'
  model = ols(formula, data=df).fit()
  anova_table = sm.stats.anova_lm(model, typ=2, robust='hc3')
  anova_table = anova_table[anova_table.index != 'Residual']
  anova_results.append(anova_table)

anova_results_df = pd.concat(anova_results, axis=0)
anova_results_df = anova_results_df.sort_values(by='F', ascending=False)
anova_results_df
```

[ ]:

	sum_sq	df	F	PR(>F)
C(fuel)	2.214430e+14	3.0	292.567986	8.080008e-179
C(transmission)	5.860624e+13	1.0	275.983306	8.374010e-61
C(owner)	2.361284e+14	4.0	243.779556	3.437094e-196
C(seller_type)	3.844585e+13	2.0	76.369348	1.562395e-33
C(name)	3.289485e+14	31.0	73.405500	0.000000e+00

[ ]:
```python
plt.bar(anova_results_df.index, anova_results_df['F'])
plt.xlabel('Categorical Features', fontsize = 9)
plt.ylabel('F-statistic', fontsize = 9)
plt.title('Welch's ANOVA: F-statistics for Categorical Features with Mean␣
  ↪Selling Price', fontsize = 9)
plt.xticks(fontsize = 9)
plt.legend(loc='upper left')
plt.tight_layout()
```

```
plt.show()
```



Understanding Welch's ANOVA F-statistic

Welch's ANOVA is a statistical test used to compare the means of two or more groups when the population variances are not assumed to be equal (unlike the standard ANOVA). Here, the F-statistic indicates the ratio of variance between group means to average variance within group means.

A higher F-statistic generally suggests a greater difference in the mean selling price across the different categories in that group. In other words, the feature is likely a better predictor of selling price.

1. **C(fuel)**: This feature has the highest F-statistic. This strongly suggests that the type of fuel a car uses has a substantial impact on its selling price. There are likely statistically significant differences in the average selling prices of cars with different fuel types.
2. **C(transmission)**: The transmission type also shows a relatively high F-statistic, although lower than fuel. This indicates that the type of transmission (e.g., manual, automatic) is also a significant factor influencing the selling price. We can infer that cars with different transmission types tend to have different average selling prices.
3. **C(owner)**: The F-statistic for the number of previous owners is moderate. This suggests that the number of owners has a noticeable, but less pronounced than fuel or transmission,

effect on the selling price. Cars with fewer previous owners likely have different average selling prices compared to those with more owners.

4. **C(seller_type)**: While seller_type shows statistical significance with a notable F-statistic, it has a lower impact on selling price compared to features like fuel type, transmission, and number of previous owners. This variable reflects the context of the transaction rather than characteristics of the car. Since the model is intended for use by car sellers to estimate prices, including seller_type could introduce data leakage or bias.Therefore, despite statistical relevance, seller_type is intentionally excluded to ensure fairness, robustness, and cleaner interpretation of results.

5. **C(name)**: The F-statistic for the car's name (make and model) is also relatively low. The F-statistic here might be lower because this is a high-cardinality categorical feature (many unique car names) and is highly imbalanced as seen from the graph above (Car brands count). These characteristics dilute the overall variance captured by ANOVA, even if individual brands have meaningful differences in selling price. The make and model of a car often encapsulate key pricing determinants such as brand reputation, performance, and market segment. Therefore, despite statistical challenges, the name feature holds substantial predictive value and is to be included in the modeling phase with appropriate encoding techniques.

Final Selection

- The model will include the following categorical features: **fuel**, **transmission**, **owner** and **name**.

0.5 Model Training

[ ]:
```
df_model = df.drop(columns = ['seller_type', 'mileage', 'max_power', 'torque',␣
↪'car_age'])
```

[ ]:
```
df_model.head()
```

[ ]:
```
0      name  year  selling_price  km_driven    fuel transmission         owner  \
1    Maruti  2014       450000.0   145500.0  Diesel       Manual   First Owner
2     Skoda  2014       370000.0   120000.0  Diesel       Manual  Second Owner
3     Honda  2006       158000.0   140000.0  Petrol       Manual   Third Owner
4   Hyundai  2010       225000.0   127000.0  Diesel       Manual   First Owner
5    Maruti  2007       130000.0   120000.0  Petrol       Manual   First Owner

0  engine  seats
1  1248.0    5.0
2  1498.0    5.0
3  1497.0    5.0
4  1396.0    5.0
5  1298.0    5.0
```

[ ]:
```
# defining features and the target
x = df_model.drop(columns = ['selling_price'])
y = df_model['selling_price']
```

[ ]:
```
# selecting the numerical features in x datatype intand float
num_model_features = ['year', 'km_driven', 'engine', 'seats']
x[num_model_features].head()
```

[ ]:
```
0  year  km_driven  engine  seats
1  2014   145500.0  1248.0    5.0
2  2014   120000.0  1498.0    5.0
3  2006   140000.0  1497.0    5.0
4  2010   127000.0  1396.0    5.0
5  2007   120000.0  1298.0    5.0
```

[ ]:
```
cat_model_features = ['name', 'fuel', 'transmission', 'owner']
x[cat_model_features].head()
```

[ ]:
```
0      name    fuel transmission          owner
1    Maruti  Diesel       Manual   First Owner
2     Skoda  Diesel       Manual  Second Owner
3     Honda  Petrol       Manual   Third Owner
4   Hyundai  Diesel       Manual   First Owner
5    Maruti  Petrol       Manual   First Owner
```

[ ]:
```
# Features pipeline
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

cat_pipeline = Pipeline([
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])
```

[ ]:
```
# Column transformer
preprocessor = ColumnTransformer([
    ('num', num_pipeline, num_model_features),
    ('cat', cat_pipeline, cat_model_features)
])

# display the preprocessor
preprocessor
```

[ ]:
```
ColumnTransformer(transformers=[('num',
                                 Pipeline(steps=[('imputer', SimpleImputer()),
                                                 ('scaler', StandardScaler())]),
                                 ['year', 'km_driven', 'engine', 'seats']),
                                ('cat',
                                 Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
```

```
                                ['name', 'fuel', 'transmission', 'owner'])])
```

```
[ ]: # spliting the data
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,␣
 ↪random_state=42)
```

```
[ ]: x_test.head()
```

```
[ ]: 0          name  year  km_driven    fuel transmission        owner  engine  \
8078    Toyota  2009   250000.0  Diesel       Manual  First Owner  2494.0
4095  Mahindra  2015    39000.0  Diesel       Manual  First Owner  2179.0
6494      Ford  2018    90000.0  Diesel       Manual  First Owner  1498.0
4340    Maruti  2008   100000.0  Petrol       Manual  Third Owner  1061.0
2536     Honda  2018    40000.0  Diesel       Manual  First Owner  1498.0

0     seats
8078    7.0
4095    7.0
6494    5.0
4340    5.0
2536    5.0
```

0.5.1 Random Forest Model

```
[ ]: # Rf pipeline
rf_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('rf', RandomForestRegressor(n_estimators=100, random_state=42))
])

# displaying the reg pipeline
rf_pipeline
```

```
[ ]: Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer()),
                                                                  ('scaler',
StandardScaler())]),
                                                  ['year', 'km_driven',
                                                   'engine', 'seats']),
                                                 ('cat',
                                                  Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
                                                  ['name', 'fuel',
                                                   'transmission',
                                                   'owner'])])),
```

```
                    ('rf', RandomForestRegressor(random_state=42))])
```

[ ]:
```python
# Training the data uisng Random forest Regressor
rf_model = rf_pipeline.fit(x_train, y_train)
rf_model
```

[ ]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer()),
                                                                  ('scaler',
StandardScaler())]),
                                                  ['year', 'km_driven',
                                                   'engine', 'seats']),
                                                 ('cat',
                                                  Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
                                                  ['name', 'fuel',
                                                   'transmission',
                                                   'owner'])])),
                ('rf', RandomForestRegressor(random_state=42))])
```

[ ]:
```python
# Predict
y_pred_rf = rf_model.predict(x_test)
```

[ ]:
```python
print("MAE:", round(mean_absolute_error(y_test, y_pred_rf),2))
print("MSE:", round(mean_squared_error(y_test, y_pred_rf),2))
print("RMSE:", round(np.sqrt(mean_squared_error(y_test, y_pred_rf)),2))
print("R² Score:",r2_score(y_test, y_pred_rf))
```

```
MAE: 91316.73
MSE: 39347158917.14
RMSE: 198361.18
R² Score: 0.8205960840365168
```

0.5.2 Gradient Boosting and GridSearchCV

[ ]:
```python
# Hyperparameters
param_grid = {
    'gsb__n_estimators': [100, 200],
    'gsb__learning_rate': [0.01, 0.1, 0.2],
    'gsb__max_depth': [3, 4, 5]
}
```

[ ]:
```python
# gsb pipeline
gb_pipeline = Pipeline([
    ('preprocessor', preprocessor),
```

```python
    ('gsb', GradientBoostingRegressor(random_state=42))
])

# gsb parameters
grid_search = GridSearchCV(gb_pipeline, param_grid, cv=5,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1)
grid_search
```

[ ]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preprocessor',
                                        ColumnTransformer(transformers=[('num',
Pipeline(steps=[('imputer',
          SimpleImputer()),
         ('scaler',
          StandardScaler())]),
['year',
'km_driven',
'engine',
'seats']),
                                                                         ('cat',
Pipeline(steps=[('onehot',
          OneHotEncoder(handle_unknown='ignore'))]),
['name',
'fuel',
'transmission',
'owner'])])),
                                       ('gsb',
GradientBoostingRegressor(random_state=42))]),
             n_jobs=-1,
             param_grid={'gsb__learning_rate': [0.01, 0.1, 0.2],
                         'gsb__max_depth': [3, 4, 5],
                         'gsb__n_estimators': [100, 200]},
             scoring='neg_mean_squared_error')
```

[ ]:
```python
# Training
gs_model = grid_search.fit(x_train, y_train)
```

[ ]:
```python
# Best params
best_params = gs_model.best_params_
best_params
```

[ ]:
```
{'gsb__learning_rate': 0.2, 'gsb__max_depth': 5, 'gsb__n_estimators': 100}
```

[ ]:
```python
# best model
best_model = gs_model.best_estimator_
```

[ ]:
```
# predicting
y_pred_gb = best_model.predict(x_test)
```

[ ]:
```
print("MAE:", round(mean_absolute_error(y_test, y_pred_gb),2))
print("MSE:", round(mean_squared_error(y_test, y_pred_gb),2))
print("RMSE:", round(np.sqrt(mean_squared_error(y_test, y_pred_gb)),2))
print("R² Score:",r2_score(y_test, y_pred_gb))
```

```
MAE: 88424.5
MSE: 36732389083.97
RMSE: 191656.96
R² Score: 0.8325181633002938
```

0.5.3 Xgboost

[ ]:
```
xgb_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('xgb', XGBRegressor(
        objective='reg:squarederror',
        eval_metric='rmse',
        random_state=42
    ))
])

# display the pipeline
xgb_pipeline
```

[ ]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer()),
                                                                  ('scaler',
StandardScaler())]),
                                                  ['year', 'km_driven',
                                                   'engine', 'seats']),
                                                 ('cat',
                                                  Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
                                                  ['name', 'fuel',
                                                   'transmission',
                                                   'owner'])])),
                ('xgb',
                 XGBRegressor(base_score=None, booster=None…
                              feature_types=None, gamma=None, grow_policy=None,
                              importance_type=None,
                              interaction_constraints=None, learning_rate=None,
                              max_bin=None, max_cat_threshold=None,
```

```
                                                max_cat_to_onehot=None, max_delta_step=None,
                                                max_depth=None, max_leaves=None,
                                                min_child_weight=None, missing=nan,
                                                monotone_constraints=None, multi_strategy=None,
                                                n_estimators=None, n_jobs=None,
                                                num_parallel_tree=None, random_state=42, …))])
```

[ ]:
```
# Train the model
xgb_model = xgb_pipeline.fit(x_train, y_train)
```

[ ]:
```
# Predicting
y_pred_xgb = xgb_model.predict(x_test)
```

[ ]:
```
print("MAE:", round(mean_absolute_error(y_test, y_pred_xgb),2))
print("MSE:", round(mean_squared_error(y_test, y_pred_xgb),2))
print("RMSE:", round(np.sqrt(mean_squared_error(y_test, y_pred_xgb)),2))
print("R² Score:",r2_score(y_test, y_pred_xgb))
```

```
MAE: 87513.79
MSE: 32608928997.74
RMSE: 180579.43
R² Score: 0.8513191366652719
```

[ ]:
```
# Model Performance Visualization

plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred_xgb, alpha=0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],␣
 ↪color='red', linestyle='--')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Predicted vs Actual Prices')
plt.grid(True)
plt.show()
```

Interpretation

- **X-axis (Actual Price)**: This represents the true selling prices of the cars.
- **Y-axis (Predicted Price)**: This represents the selling prices predicted by `xgb_model`.
- **Red Dashed Line (Ideal Prediction)**: This diagonal line represents a scenario where the predicted price is exactly equal to the actual price. If all the points fell perfectly on this line, the model would be making perfect predictions.
- **Blue Scatter Points**: Each blue dot represents a single car. Its position on the graph shows the actual selling price (x-coordinate) and the price predicted (y-coordinate).
- **Clustering Around the Red Line**: Notice that a significant number of the blue points are clustered relatively close to the red dashed line, especially for cars with lower actual selling prices (below roughly 2 million). This indicates that the model is generally performing well for cars in this price range.
- As the actual selling price increases (moving to the right on the x-axis), the scatter of the blue points tends to become wider around the red line. This suggests that the model's predictions become less precise for higher-priced cars. The errors (the vertical distance between a point and the red line) are larger for these more expensive vehicles.

- Points above the red line represent cases where your model overpredicted the selling price (the predicted price is higher than the actual price).
- Points below the red line represent cases where your model underpredicted the selling price (the predicted price is lower than the actual price).
- In conclusion, the model is reasonably good at predicting the selling prices of cars, particularly those in the lower to mid-price range. However, it exhibits a higher degree of error and less precision when predicting the prices of more expensive vehicles.

0.5.4 Saving the xbg model

[ ]:
```
# displaying
xgb_model
```

[ ]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer()),
                                                                  ('scaler',
StandardScaler())]),
                                                  ['year', 'km_driven',
                                                   'engine', 'seats']),
                                                 ('cat',
                                                  Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))]),
                                                  ['name', 'fuel',
                                                   'transmission',
                                                   'owner'])])),
                ('xgb',
                 XGBRegressor(base_score=None, booster=None…
                              feature_types=None, gamma=None, grow_policy=None,
                              importance_type=None,
                              interaction_constraints=None, learning_rate=None,
                              max_bin=None, max_cat_threshold=None,
                              max_cat_to_onehot=None, max_delta_step=None,
                              max_depth=None, max_leaves=None,
                              min_child_weight=None, missing=nan,
                              monotone_constraints=None, multi_strategy=None,
                              n_estimators=None, n_jobs=None,
                              num_parallel_tree=None, random_state=42, …))])
```

[ ]:
```
# saving the xgb model
joblib.dump(xgb_model, 'xgb_model.pkl')
```

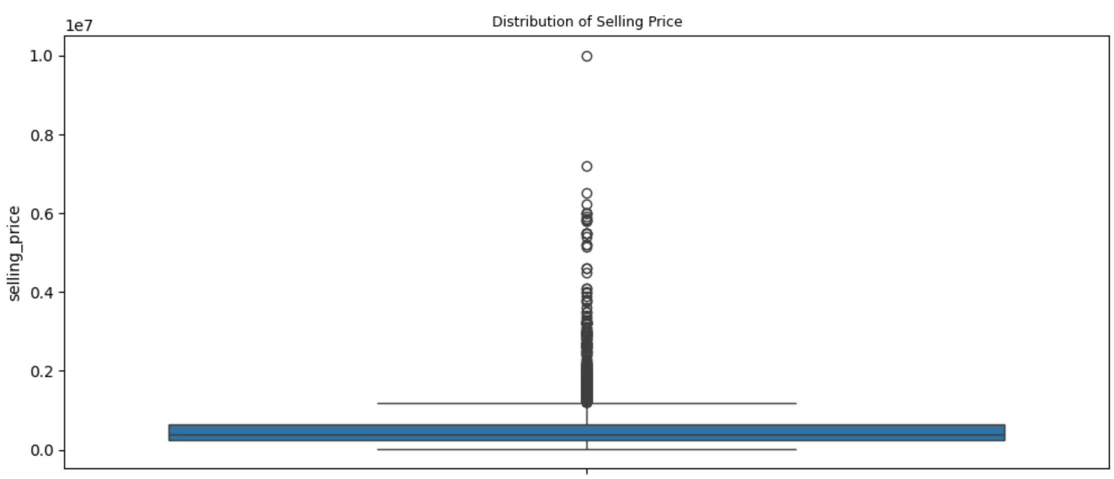
[ ]:
```
['xgb_model.pkl']
```

[ ]:
```
x_train.head()
```

[ ]:
```
0        name  year  km_driven    fuel transmission         owner  engine  \
5587     Ford  2018    25000.0  Diesel       Manual   First Owner  1498.0
3469   Maruti  2010   170000.0  Petrol       Manual  Second Owner     NaN
4936  Hyundai  2011    75500.0  Diesel       Manual  Second Owner  1396.0
2486   Maruti  2009   138000.0  Diesel       Manual  Second Owner  1248.0
4492  Hyundai  2003   200000.0  Diesel       Manual  Second Owner  1493.0

0     seats
5587    5.0
3469    NaN
4936    5.0
2486    5.0
4492    5.0
```

1 Model 2 - Removed extreme outliers

[ ]:
```python
# box plost showing distribution of selling price
plt.figure(figsize=(12, 5))
sns.boxplot(y=df['selling_price'])
plt.title('Distribution of Selling Price', fontsize=9)
plt.xlabel('', fontsize=9)
plt.show()
```



Removing data points beyond the upper whisker (i.e., high outliers).

[ ]:
```python
# q1 and q3
q1 = df["selling_price"].quantile(0.25)
q3 = df['selling_price'].quantile(0.75)
```

```
# inter-quantitle range
iqr = q3 - q1

# upper whisker
upper_whisker = q3 + 1.5 * iqr

# removing the outliers
df_no_outliers  = df[df['selling_price'] < upper_whisker]
```

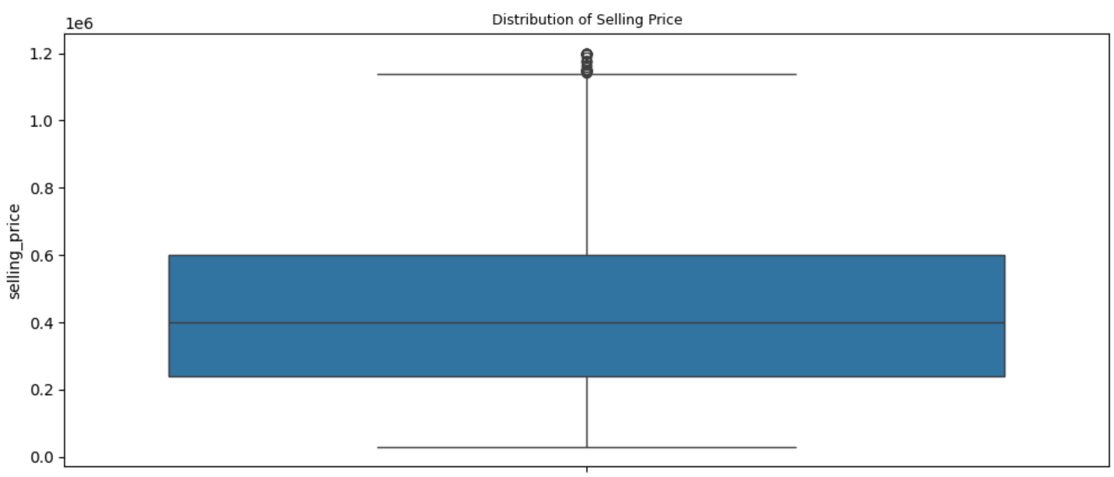
[ ]:
```
df.shape
```

[ ]: (6926, 14)

[ ]:
```
print(df.shape)
print(df_no_outliers.shape)

df_rows = df.shape[0]
df_no_outliers_srows = df_no_outliers.shape[0]

lost_data = df_rows - df_no_outliers_srows
lost_data_percent = round((lost_data / df_rows * 100), 2)
print (f'Data lost is {lost_data_percent}%')
```

```
(6926, 14)
(6598, 14)
Data lost is 4.74%
```

[ ]:
```
# distribution of selling price without outliers
plt.figure(figsize=(12, 5))
sns.boxplot(y=df_no_outliers['selling_price'])
plt.title('Distribution of Selling Price', fontsize=9)
plt.xlabel('', fontsize=9)
plt.show()
```



[ ]:
```
df_no_outliers.drop(columns = ['seller_type', 'mileage', 'max_power', 'torque',␣
↪'car_age'], inplace=True)
df_no_outliers.head()
```

[ ]:
```
0     name  year  selling_price  km_driven    fuel transmission         owner  \
1   Maruti  2014       450000.0   145500.0  Diesel       Manual   First Owner
2    Skoda  2014       370000.0   120000.0  Diesel       Manual  Second Owner
3    Honda  2006       158000.0   140000.0  Petrol       Manual   Third Owner
4  Hyundai  2010       225000.0   127000.0  Diesel       Manual   First Owner
5   Maruti  2007       130000.0   120000.0  Petrol       Manual   First Owner

0  engine  seats
1  1248.0    5.0
2  1498.0    5.0
3  1497.0    5.0
4  1396.0    5.0
5  1298.0    5.0
```

[ ]:
```
# Features and target
x_no_outliers = df_no_outliers.drop(columns = ['selling_price'])
y_no_outliers = df_no_outliers['selling_price']
```

[ ]:
```
# spliting the not outliers data
x_train_no_outliers, x_test_no_outliers, y_train_no_outliers,␣
↪y_test_no_outliers = train_test_split(x_no_outliers, y_no_outliers,␣
↪test_size = 0.2, random_state=42)
```

[ ]:
```
xgb_pipeline_no_outliers = Pipeline([
    ('preprocessor', preprocessor),
    ('xgb', XGBRegressor(
        objective='reg:squarederror',
        eval_metric='rmse',
        random_state=42
    ))
])
```

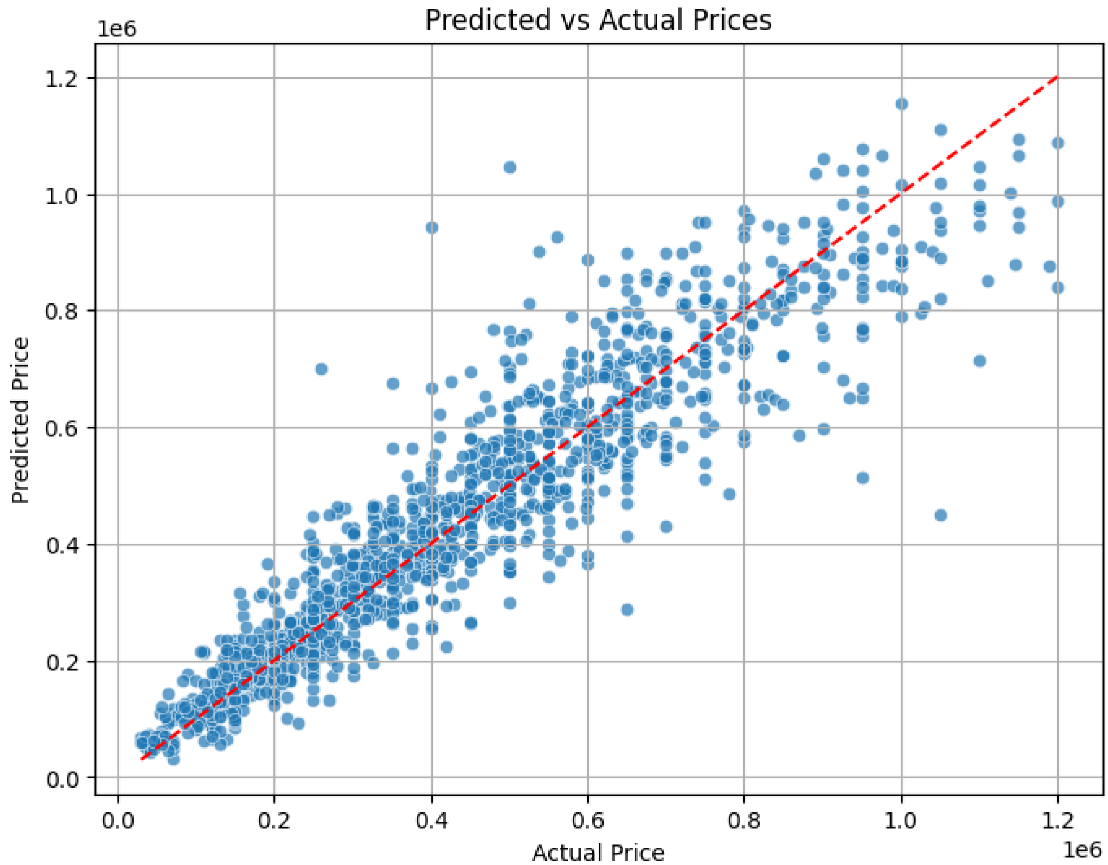
[ ]:
```
# training the no outliers data
xgb_model_no_outliers = xgb_pipeline_no_outliers.fit(x_train_no_outliers,␣
↪y_train_no_outliers)
```

[ ]:
```
# Predictins on no outliers
y_pred_xgb_no_outliers = xgb_model_no_outliers.predict(x_test_no_outliers)
```

[ ]:
```python
print("MAE:", round(mean_absolute_error(y_test_no_outliers,␣
 ↪y_pred_xgb_no_outliers),2))
print("MSE:", round(mean_squared_error(y_test_no_outliers,␣
 ↪y_pred_xgb_no_outliers),2))
print("RMSE:", round(np.sqrt(mean_squared_error(y_test_no_outliers,␣
 ↪y_pred_xgb_no_outliers)),2))
print("R² Score:", round(r2_score(y_test_no_outliers, y_pred_xgb_no_outliers),␣
 ↪2))
```

```
MAE: 65224.45
MSE: 8681822530.02
RMSE: 93176.3
R² Score: 0.86
```

[ ]:
```python
# Graphical Representation of the model
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test_no_outliers, y=y_pred_xgb_no_outliers, alpha=0.7)
# Perfect prediction line
plt.plot([y_test_no_outliers.min(), y_test_no_outliers.max()],␣
 ↪[y_test_no_outliers.min(), y_test_no_outliers.max()], color='red',␣
 ↪linestyle='--')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Predicted vs Actual Prices')
plt.grid(True)
plt.show()
```




```
[ ]: # Results for each model
results = {
    'Model': ['XGBoost', 'XGBoost (no outliers)', 'Gradient Boosting', 'Random␣
 ↪Forest'],
    'MAE': [
        round(mean_absolute_error(y_test, y_pred_xgb), 2),
        round(mean_absolute_error(y_test_no_outliers, y_pred_xgb_no_outliers),␣
 ↪2),
        round(mean_absolute_error(y_test, y_pred_gb), 2),
        round(mean_absolute_error(y_test, y_pred_rf), 2)
    ],
    'MSE': [
        round(mean_squared_error(y_test, y_pred_xgb), 2),
        round(mean_squared_error(y_test_no_outliers, y_pred_xgb_no_outliers),␣
 ↪2),
        round(mean_squared_error(y_test, y_pred_gb), 2),
        round(mean_squared_error(y_test, y_pred_rf), 2)
    ],
    'RMSE': [
```

```
        round(np.sqrt(mean_squared_error(y_test, y_pred_xgb)), 2),
        round(np.sqrt(mean_squared_error(y_test_no_outliers,␣
 ↪y_pred_xgb_no_outliers)), 2),
        round(np.sqrt(mean_squared_error(y_test, y_pred_gb)), 2),
        round(np.sqrt(mean_squared_error(y_test, y_pred_rf)), 2)
    ],
    'R² Score': [
    round(r2_score(y_test, y_pred_xgb), 4),
    round(r2_score(y_test_no_outliers, y_pred_xgb_no_outliers), 4),
    round(r2_score(y_test, y_pred_gb), 4),
    round(r2_score(y_test, y_pred_rf), 4)
]
}
# data frame
results_df = pd.DataFrame(results)
```

[ ]:
```
results_df = results_df.sort_values(by='R² Score', ascending=False)
results_df
```

[ ]:

	Model	MAE	MSE	RMSE	R² Score
1	XGBoost (no outliers)	65,224.45	8,681,822,530.02	93,176.30	0.86
0	XGBoost	87,513.79	32,608,928,997.74	180,579.43	0.85
2	Gradient Boosting	88,424.50	36,732,389,083.97	191,656.96	0.83
3	Random Forest	91,316.73	39,347,158,917.14	198,361.18	0.82

Explanation (Model Evaluation)

- Model: The name of the machine learning model.
- MAE: Mean Absolute Error. This measures the average absolute difference between the predicted selling price and the actual selling price. Lower is better.
- MSE: Mean Squared Error. This measures the average squared difference between the predicted and actual selling prices. Lower is better.
- RMSE: Root Mean Squared Error. This is the square root of the MSE. It has the same units as the target variable (selling price), making it more interpretable than MSE. Lower is better.

Performance Comparison:

- XGBoost appears to be the best-performing model based on these metrics. It has the lowest MAE, MSE, and RMSE, and the highest R^2 score. This indicates that, on average, its predictions are closest to the actual selling prices, with the least amount of error and the highest proportion of variance explained.
- Gradient Boosting performed second best, with MAE and RMSE values slightly higher than XGBoost but better than Random forest. Its R^2 score is also higher than Random forest.
- Random forest shows the weakest performance among the three, with the highest MAE, MSE, and RMSE, and the lowest R^2 score.

XGBoost Vs XGBoost (No Outliers)

1. **XGBoost (No Outliers)**: Shows Noticeable Performance Gains. Removing outliers from the training data led to a clear and measurable improvement in all evaluation metrics for the XGBoost model.
2. **Drastic Drop in RMSE Reflects Higher Accuracy**: The significant reduction in RMSE indicates that the model's predictions are now much closer to the actual selling prices. Notice that the he Mean Squared Error (MSE) also dropped heavily from 32.6 billion to just 8.7 billion. Outliers previously distorted the model's learning process, but filtering them out allowed the model to better capture the general patterns in the data and make more reliable predictions.

1.0.1 Saving the xgb_model_no_outliers

[ ]:
```
# saving the model
joblib.dump(xgb_model_no_outliers, 'xgb_model_no_outliers.pkl')
```

[ ]: ['xgb_model_no_outliers.pkl']

1.0.2 Deployment of Streamlit App

The app was developed and tested in VSCode, then deployed using Streamlit Cloud for easy access and sharing.

[ ]:
```
import pandas as pd
import streamlit as st
import xgboost as xgb
from xgboost import XGBRegressor
import joblib

st.set_page_config(layout="wide")

model = joblib.load('xgb_model.pkl')

st.header("Car Price Prediction (XGBRegressor)")
df = pd.read_csv(r"C:\Users\PC\Desktop\car_model\Cardetails - Cardetails.csv")

# function for cleaning data
def get_clean_car_detail(car_detail):
  car_detail = car_detail.split(" ")[0]
  return car_detail
df.name = df.name.apply(get_clean_car_detail)

# creating columns
col1, col2 = st.columns(2)
```

```python
# Car form
with col1:
  name = st.selectbox('Select Car Brand', df['name'].unique())
  year = st.selectbox('Select Car Year', df['year'].unique())
  km_driven = st.slider("Km_driven", 0, 2500000)
  fuel = st.selectbox('Select Fuel Type', df['fuel'].unique())

with col2:
  transmission = st.selectbox('Transmission type', df['transmission'].unique())
  owner = st.selectbox('Owner', df['owner'].unique())
  engine = st.slider("Engine Size (cc)", 200, 10000)
  seats = st.slider("Number of Seats", 2, 15)


# after getting information from user
if st.button("Predict"):
    input_data_model = pd.DataFrame(
        [[name, year, km_driven, fuel,transmission,
          owner, engine, seats]],
    columns = ['name', 'year', 'km_driven', 'fuel','transmission',
              'owner', 'engine', 'seats']
    )

    # st.write(input_data_model)

    # Passing values to the model to predict
    car_price = model.predict(input_data_model)

    st.markdown('Car price is {:,.0f}'.format(car_price[0]))
```