🚗 Car Price Prediction – Project Links

🔗 **GitHub Repository**: github.com/saugatshakya/predicting_car_price-3

🌐 **Live Demo (Flask App)**: st125986.ml.brain.cs.ait.ac.th

INIT MLFLOW

In [1]:
```
import os
import mlflow
import mlflow.pyfunc


os.environ["MLFLOW_TRACKING_USERNAME"] = "admin"
os.environ["MLFLOW_TRACKING_PASSWORD"] = "password"
```

```
/Users/saugatshakya/Library/Python/3.9/lib/python/site-packages/urll
ib3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports Open
SSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.
8.3'. See: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
```

In [2]:
```
mlflow.set_tracking_uri("https://mlflow.ml.brain.cs.ait.ac.th/")
mlflow.set_experiment("st125986-a3")
```

Out[2]:
```
<Experiment: artifact_location='mlflow-artifacts:/2265838743149410
70', creation_time=1759506644227, experiment_id='22658387431494107
0', last_update_time=1759506644227, lifecycle_stage='active', name
='st125986-a3', tags={'mlflow.experimentKind': 'custom_model_devel
opment'}>
```

IMPORT LIBRARIES

In [3]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pickle
import warnings
import time
warnings.filterwarnings('ignore')
```

LOAD DATA

In [4]:
```
# Load the dataset
```

```python
data = pd.read_csv("cars.csv")
print("Dataset loaded successfully!")
print(f"Original dataset shape: {data.shape}")
```

```
Dataset loaded successfully!
Original dataset shape: (8128, 13)
```

PREPROCESSING DATA

In [5]:
```python
# Display basic information about the dataset
print("\nDataset Info:")
data.info()
```

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8128 entries, 0 to 8127
Data columns (total 13 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   name           8128 non-null   object 
 1   year           8128 non-null   int64  
 2   selling_price  8128 non-null   int64  
 3   km_driven      8128 non-null   int64  
 4   fuel           8128 non-null   object 
 5   seller_type    8128 non-null   object 
 6   transmission   8128 non-null   object 
 7   owner          8128 non-null   object 
 8   mileage        7907 non-null   object 
 9   engine         7907 non-null   object 
 10  max_power      7913 non-null   object 
 11  torque         7906 non-null   object 
 12  seats          7907 non-null   float64
dtypes: float64(1), int64(3), object(9)
memory usage: 825.6+ KB
```

In [6]:
```python
# Display first few rows
print("\nFirst 5 rows:")
data.head()
```

```
First 5 rows:
```

Out[6]:

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

In [7]:

```
# Check for missing values
print("\nMissing values in each column:")
print(data.isnull().sum())
```

```
Missing values in each column:
name               0
year               0
selling_price      0
km_driven          0
fuel               0
seller_type        0
transmission       0
owner              0
mileage          221
engine           221
max_power        215
torque           222
seats            221
dtype: int64
```

In [8]:

```
# Check for duplicates
print(f"\nNumber of duplicates: {data.duplicated().sum()}")
```

```
Number of duplicates: 1202
```

In [9]:

```
# Remove duplicates
data = data.drop_duplicates()
print(f"Dataset shape after removing duplicates: {data.shape}")
```

```
Dataset shape after removing duplicates: (6926, 13)
```

In [10]:
```python
# Remove CNG and LPG fuel types as instructed
data = data[~data['fuel'].isin(['CNG', 'LPG'])]
print(f"Dataset shape after removing CNG/LPG: {data.shape}")
```

```
Dataset shape after removing CNG/LPG: (6832, 13)
```

In [11]:
```python
# Clean mileage column
data['mileage'] = (
    data['mileage']
    .str.replace('kmpl', '', regex=False)
    .str.replace('km/kg', '', regex=False)
    .str.strip()
)
data['mileage'] = pd.to_numeric(data['mileage'], errors='coerce')
```

In [12]:
```python
# Clean engine column
data['engine'] = (
    data['engine']
    .str.replace('CC', '', regex=False)
    .str.strip()
)
data['engine'] = pd.to_numeric(data['engine'], errors='coerce')
```

In [13]:
```python
# Clean max_power column
data['max_power'] = (
    data['max_power']
    .str.replace('bhp', '', regex=False)
    .str.strip()
)
data['max_power'] = pd.to_numeric(data['max_power'], errors='coerce
```

In [14]:
```python
# Fill missing values in seats with mode
data['seats'] = data['seats'].fillna(data['seats'].mode()[0])
```

In [15]:
```python
# Fill other missing values with median
for col in ['mileage', 'engine', 'max_power']:
    data[col] = data[col].fillna(data[col].median())
```

In [16]:
```python
# Drop torque column as instructed
data.drop(columns=['torque'], inplace=True)
```

In [17]:
```python
# Map owner values as instructed
owner_map = {
    'First Owner': 1,
    'Second Owner': 2,
    'Third Owner': 3,
    'Fourth & Above Owner': 4,
    'Test Drive Car': 5
}
data['owner'] = data['owner'].map(owner_map)
```

In [18]:
```python
# Remove test drive cars as instructed
data = data[data['owner'] != 5]
print(f"Dataset shape after removing test drive cars: {data.shape}"
```

```
Dataset shape after removing test drive cars: (6827, 12)
```

In [19]:
```python
# Extract brand from name
data['brand'] = data['name'].str.split().str[0]
data['brand'] = data['brand'].fillna('Unknown')
data.drop(columns=['name'], inplace=True)
```

In [20]:
```python
#see all brand column unique values
print("\nUnique brands in the dataset:")
print(data['brand'].unique())
```

```
Unique brands in the dataset:
['Maruti' 'Skoda' 'Honda' 'Hyundai' 'Toyota' 'Ford' 'Renault' 'Mahin
dra'
 'Tata' 'Chevrolet' 'Fiat' 'Datsun' 'Jeep' 'Mercedes-Benz' 'Mitsubis
hi'
 'Audi' 'Volkswagen' 'BMW' 'Nissan' 'Lexus' 'Jaguar' 'Land' 'MG' 'Vo
lvo'
 'Daewoo' 'Kia' 'Force' 'Ambassador' 'Ashok' 'Isuzu' 'Opel' 'Peugeo
t']
```

In [21]:
```python
# Apply log transformation to selling price as instructed
data['selling_price'] = np.log(data['selling_price'])
```

In [22]:
```python
# Display final dataset info
print("\nFinal dataset info:")
data.info()
```

```
Final dataset info:
<class 'pandas.core.frame.DataFrame'>
Index: 6827 entries, 0 to 8125
Data columns (total 12 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   year           6827 non-null   int64  
 1   selling_price  6827 non-null   float64
 2   km_driven      6827 non-null   int64  
 3   fuel           6827 non-null   object 
 4   seller_type    6827 non-null   object 
 5   transmission   6827 non-null   object 
 6   owner          6827 non-null   int64  
 7   mileage        6827 non-null   float64
 8   engine         6827 non-null   float64
 9   max_power      6827 non-null   float64
 10  seats          6827 non-null   float64
 11  brand          6827 non-null   object 
dtypes: float64(5), int64(3), object(4)
memory usage: 693.4+ KB
```

In [23]:
```python
print("\nFirst 5 rows of cleaned data:")
data.head()
```

```
First 5 rows of cleaned data:
```

Out[23]:

	year	selling_price	km_driven	fuel	seller_type	transmission	owner
0	2014	13.017003	145500	Diesel	Individual	Manual	1
1	2014	12.821258	120000	Diesel	Individual	Manual	2
2	2006	11.970350	140000	Petrol	Individual	Manual	3
3	2010	12.323856	127000	Diesel	Individual	Manual	1
4	2007	11.775290	120000	Petrol	Individual	Manual	1

In [24]:

```
# Check for remaining missing values
print("\nRemaining missing values:")
print(data.isnull().sum())
```

```
Remaining missing values:
year             0
selling_price    0
km_driven        0
fuel             0
seller_type      0
transmission     0
owner            0
mileage          0
engine           0
max_power        0
seats            0
brand            0
dtype: int64
```

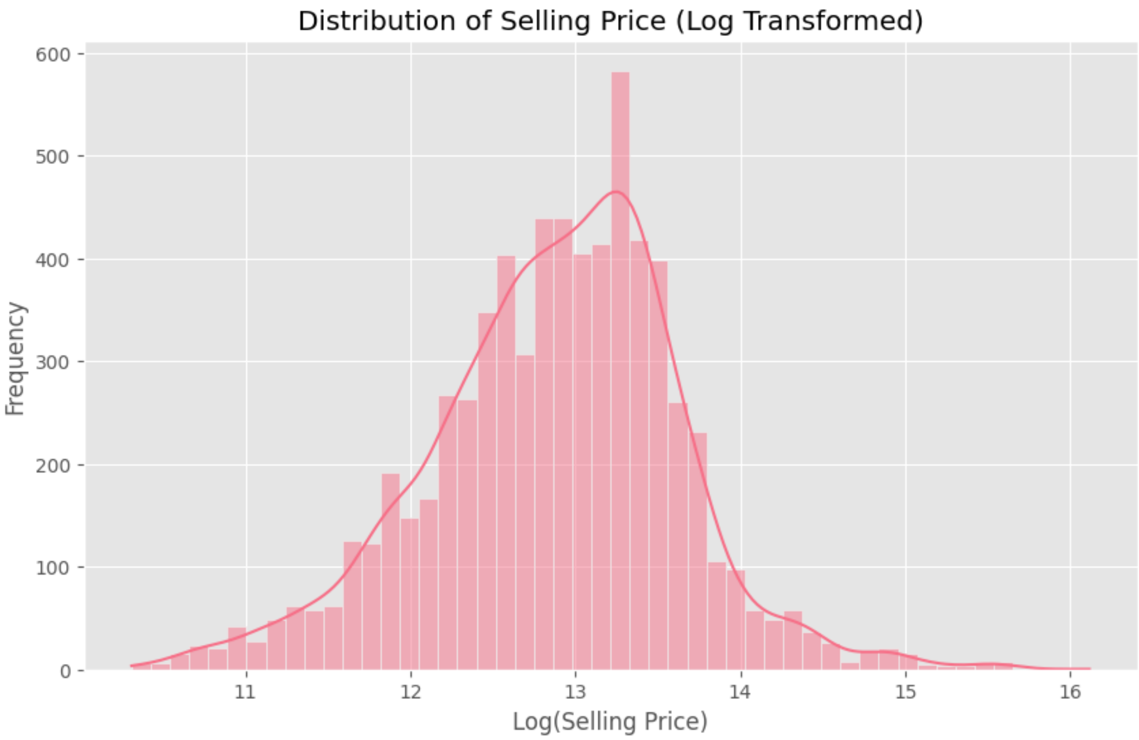
EDA

In [25]:

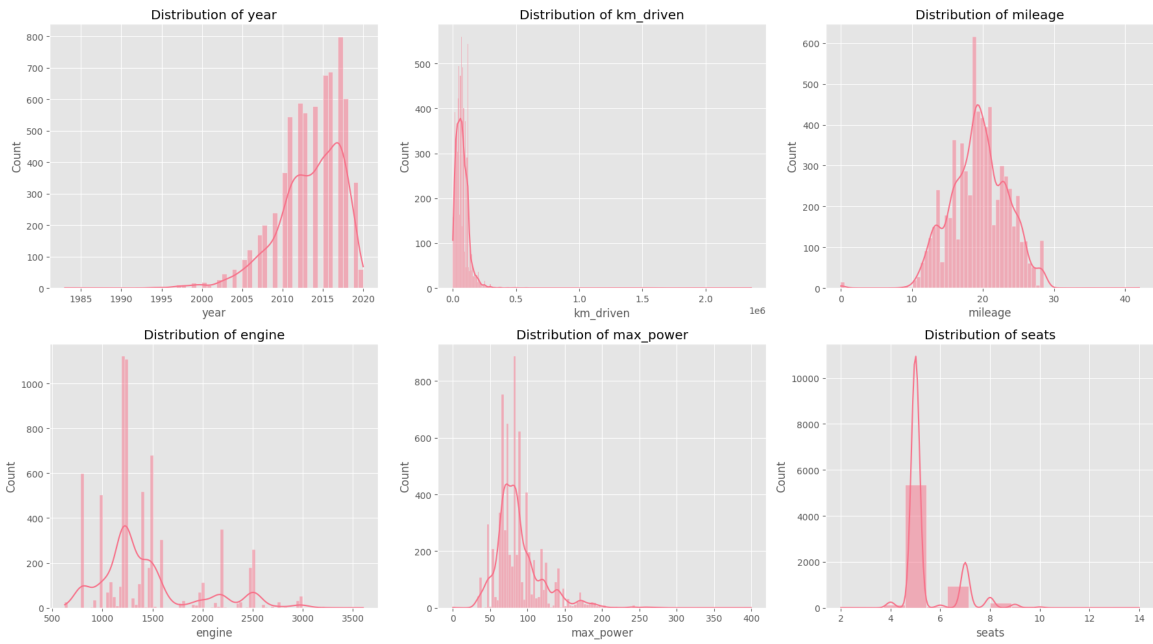
```
# Set style for plots
plt.style.use('ggplot')
sns.set_palette("husl")
```

In [26]:

```
# Visualize the distribution of selling price (log transformed)
plt.figure(figsize=(10, 6))
sns.histplot(data['selling_price'], bins=50, kde=True)
plt.title("Distribution of Selling Price (Log Transformed)")
plt.xlabel("Log(Selling Price)")
plt.ylabel("Frequency")
plt.show()
```

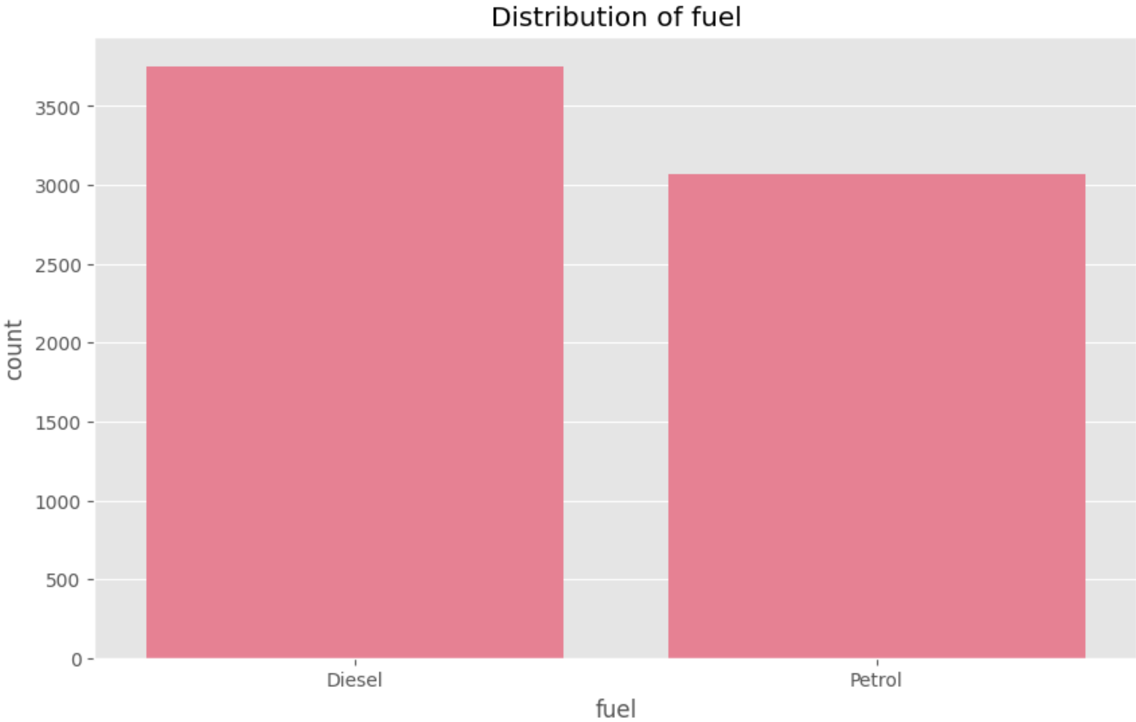


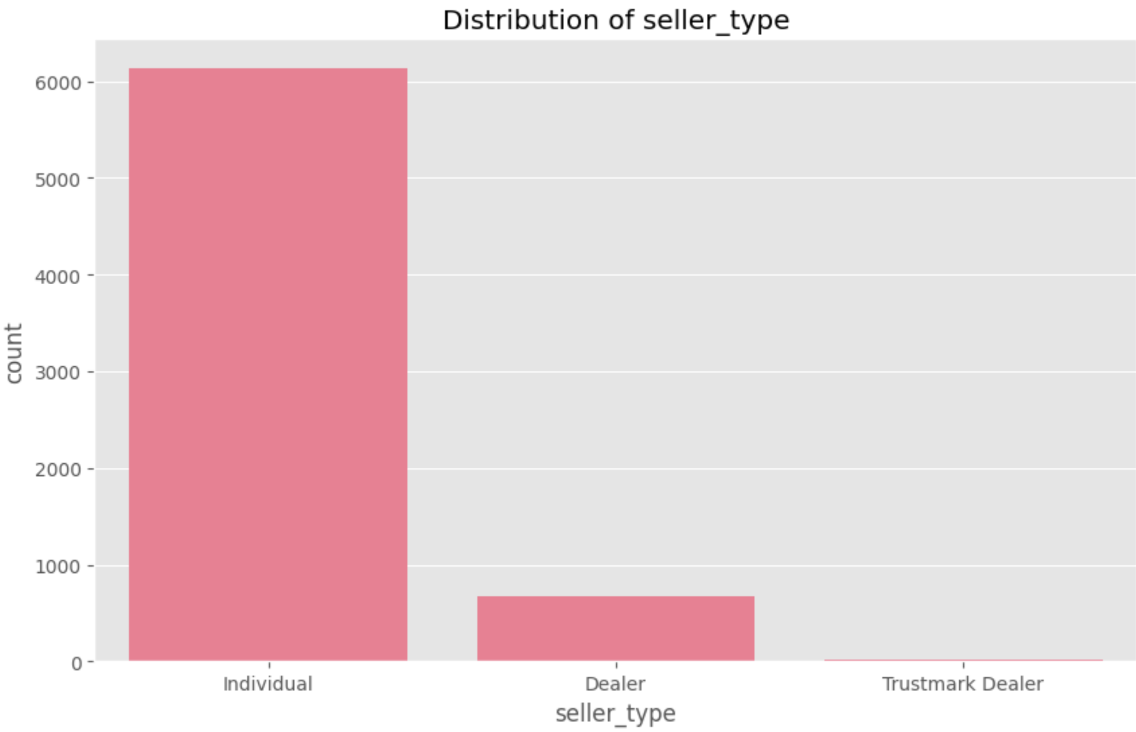
In [27]:
```
# Visualize numerical features
num_cols = ['year', 'km_driven', 'mileage', 'engine', 'max_power',

fig, axes = plt.subplots(2, 3, figsize=(18, 10))
axes = axes.ravel()

for i, col in enumerate(num_cols):
    sns.histplot(data[col], kde=True, ax=axes[i])
    axes[i].set_title(f"Distribution of {col}")

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

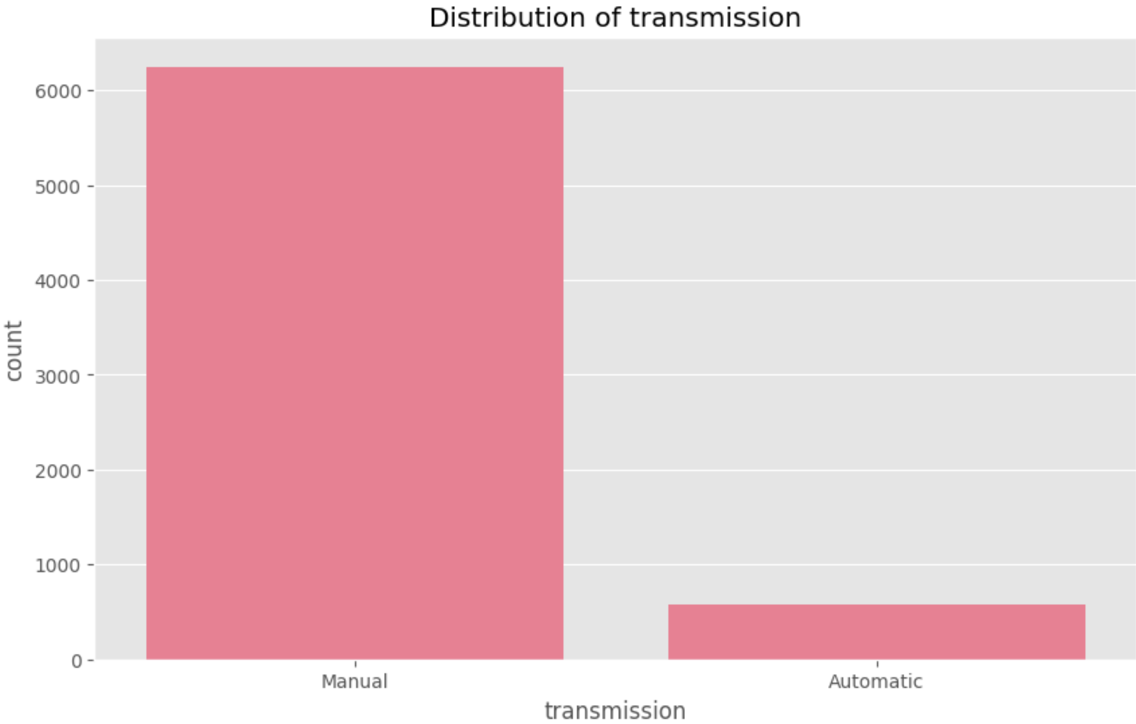


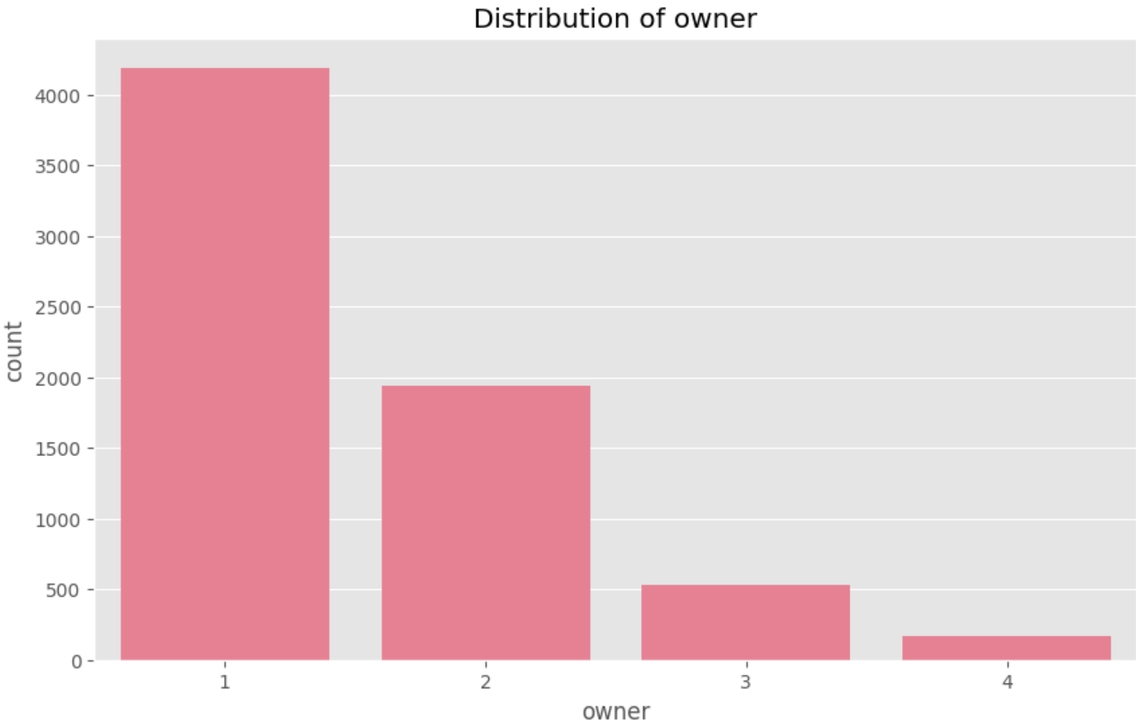
In [28]:
```
# Visualize categorical features
cat_cols = ['fuel', 'seller_type', 'transmission', 'owner', 'brand'
```

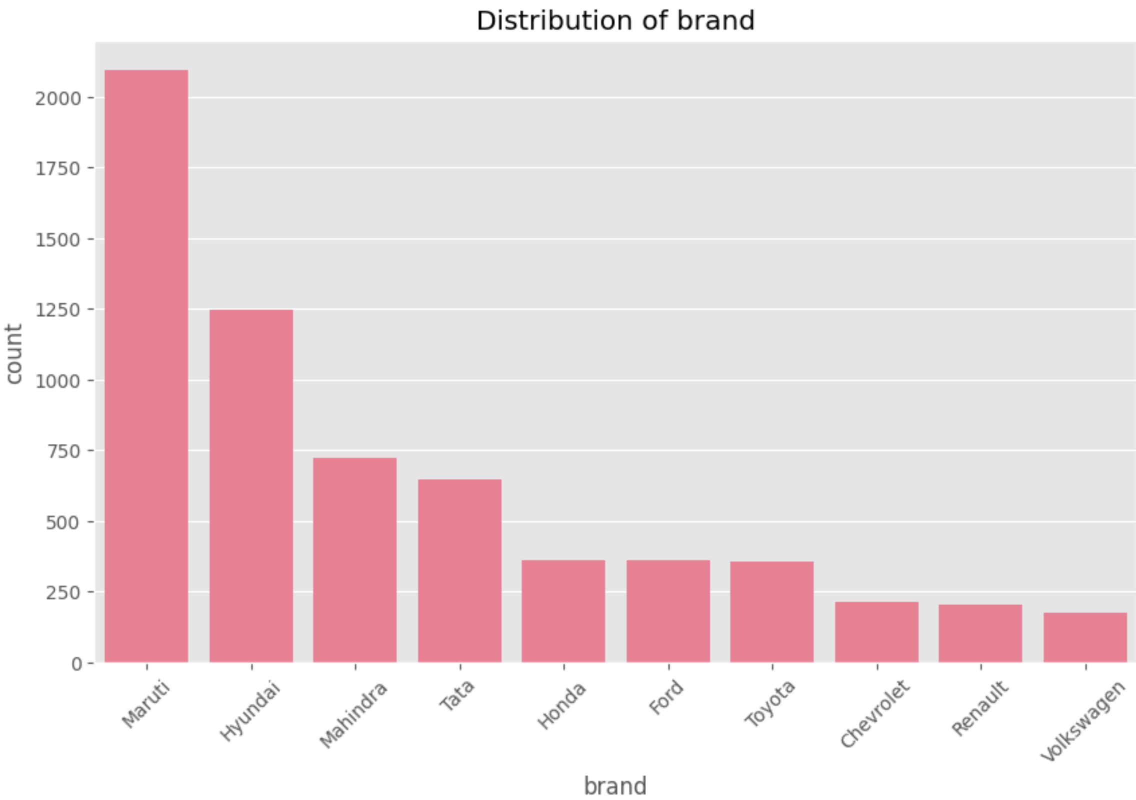
```
for col in cat_cols:
    plt.figure(figsize=(10, 6))
    if col == 'brand':  # For brand, show only top 10
        top_brands = data['brand'].value_counts().nlargest(10).inde
        sns.countplot(data=data[data['brand'].isin(top_brands)], x=
        plt.xticks(rotation=45)
    else:
        sns.countplot(data=data, x=col)
    plt.title(f"Distribution of {col}")
    plt.show()
```



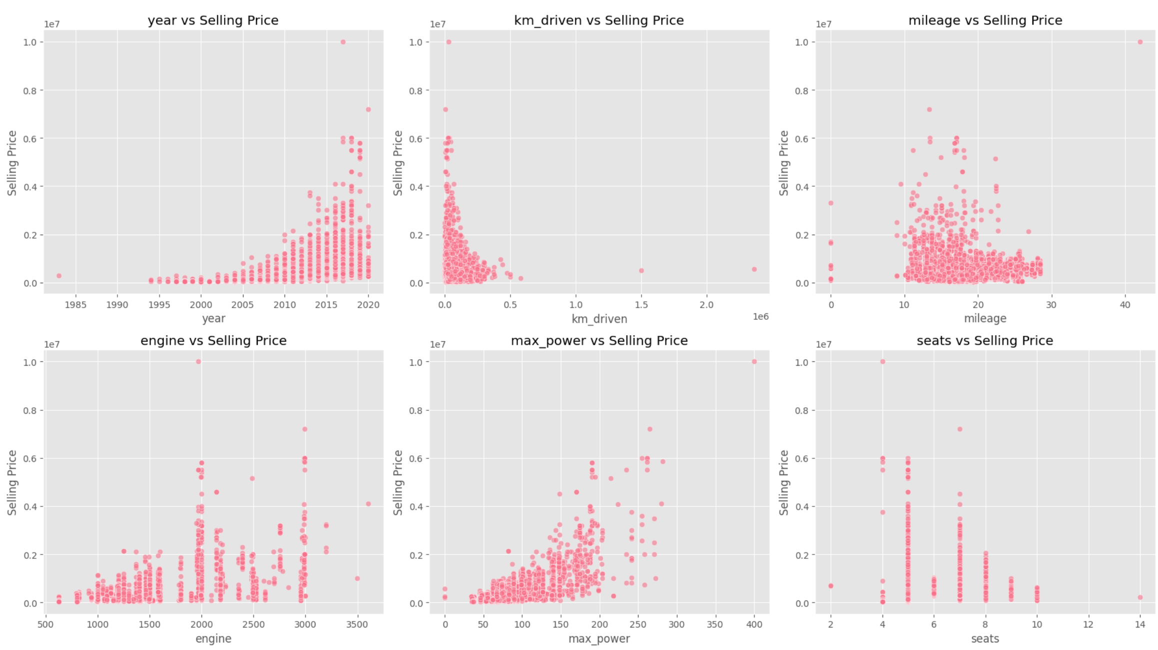



Distribution of transmission
count
0
1000
2000
3000
4000
5000
6000
Manual
Automatic
transmission


Distribution of owner
count
0
500
1000
1500
2000
2500
3000
3500
4000
1
2
3
4
owner

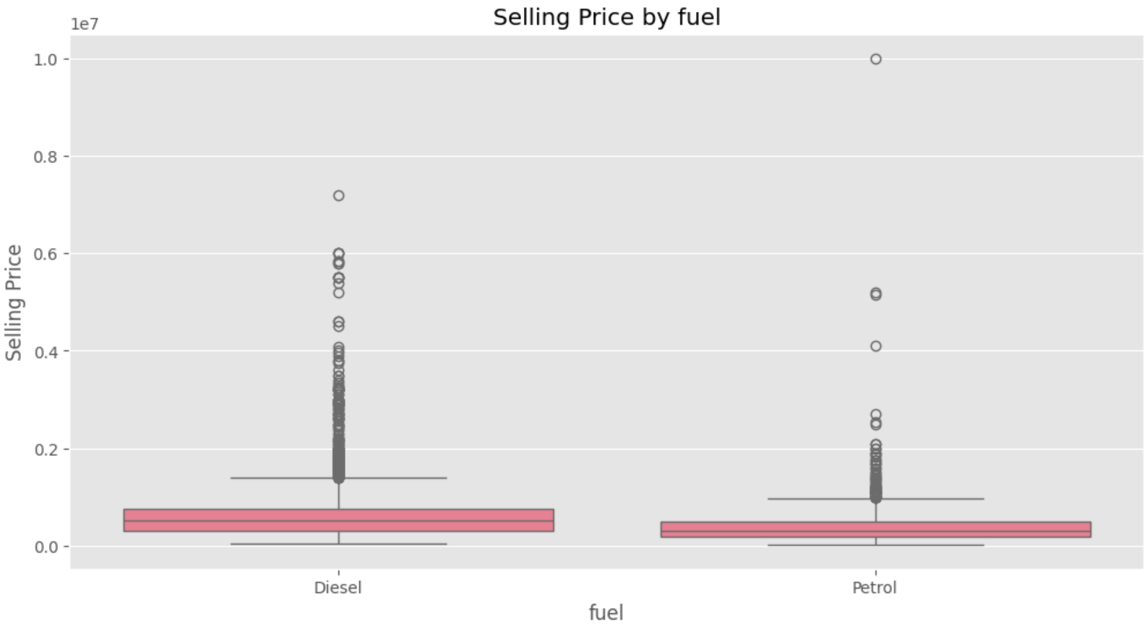


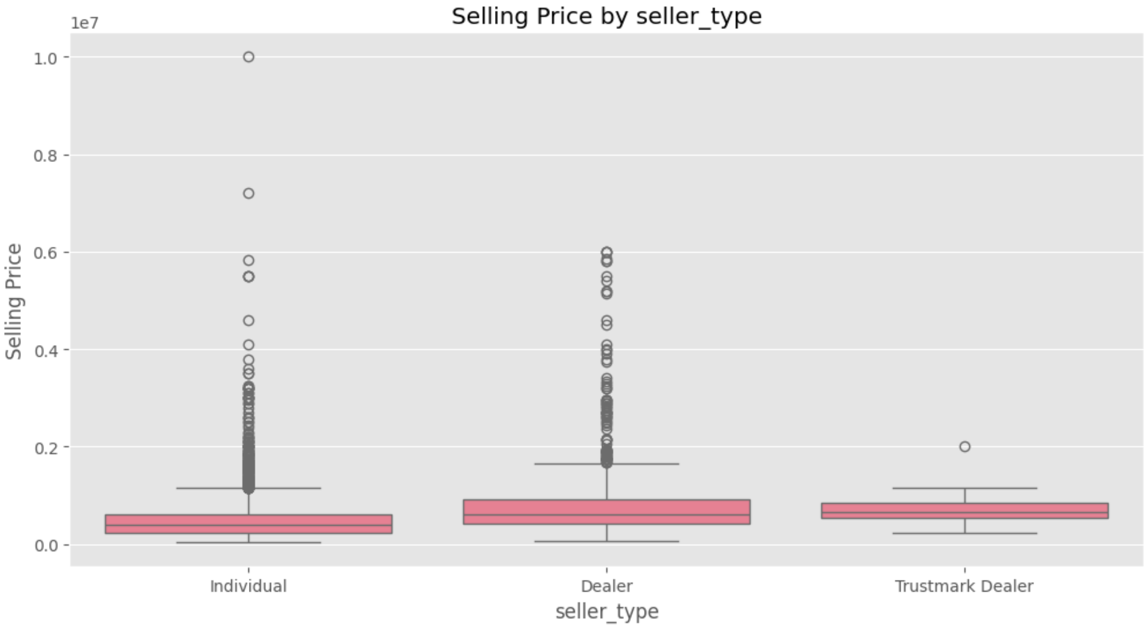
In [29]:
```
# Analyze relationship between features and selling price
# Numerical features vs selling price
fig, axes = plt.subplots(2, 3, figsize=(18, 10))
axes = axes.ravel()

for i, col in enumerate(num_cols):
    sns.scatterplot(data=data, x=col, y=np.exp(data['selling_price'
    axes[i].set_title(f"{col} vs Selling Price")
    axes[i].set_ylabel("Selling Price")

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



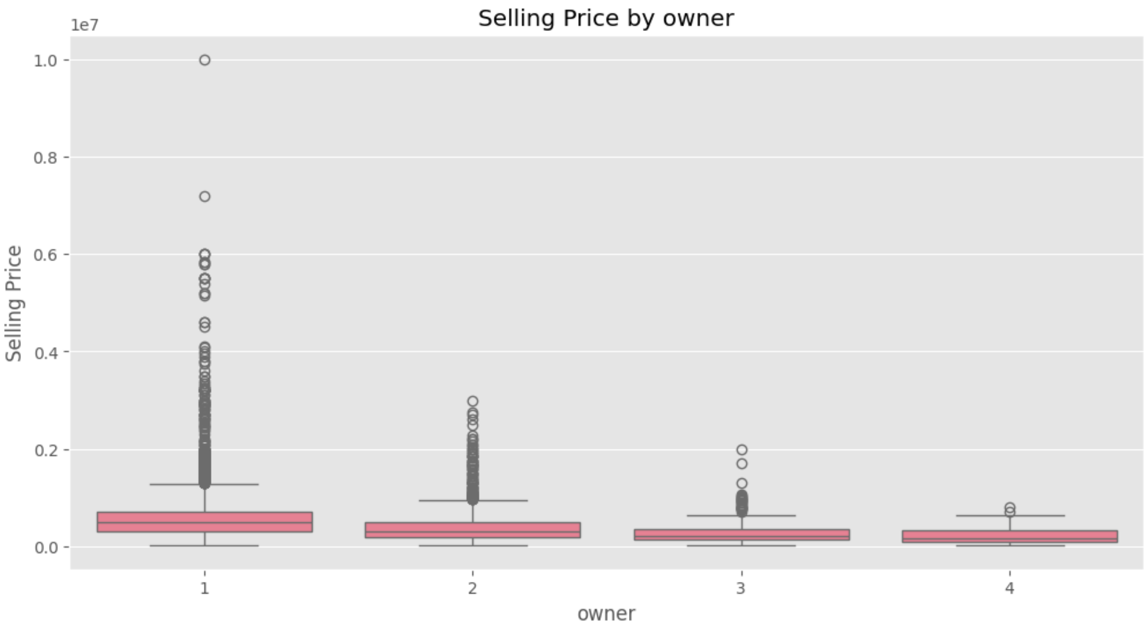
In [30]:
```
# Categorical features vs selling price
```

```
for col in cat_cols:
    plt.figure(figsize=(12, 6))
    if col == 'brand':  # For brand, show only top 10
        top_brands = data['brand'].value_counts().nlargest(10).inde
        sns.boxplot(data=data[data['brand'].isin(top_brands)], x=co
        plt.xticks(rotation=45)
    else:
        sns.boxplot(data=data, x=col, y=np.exp(data['selling_price'
    plt.title(f"Selling Price by {col}")
    plt.ylabel("Selling Price")
    plt.show()
```






Selling Price by transmission
1e7
Selling Price
1.0
0.8
0.6
0.4
0.2
0.0
Manual
Automatic
transmission

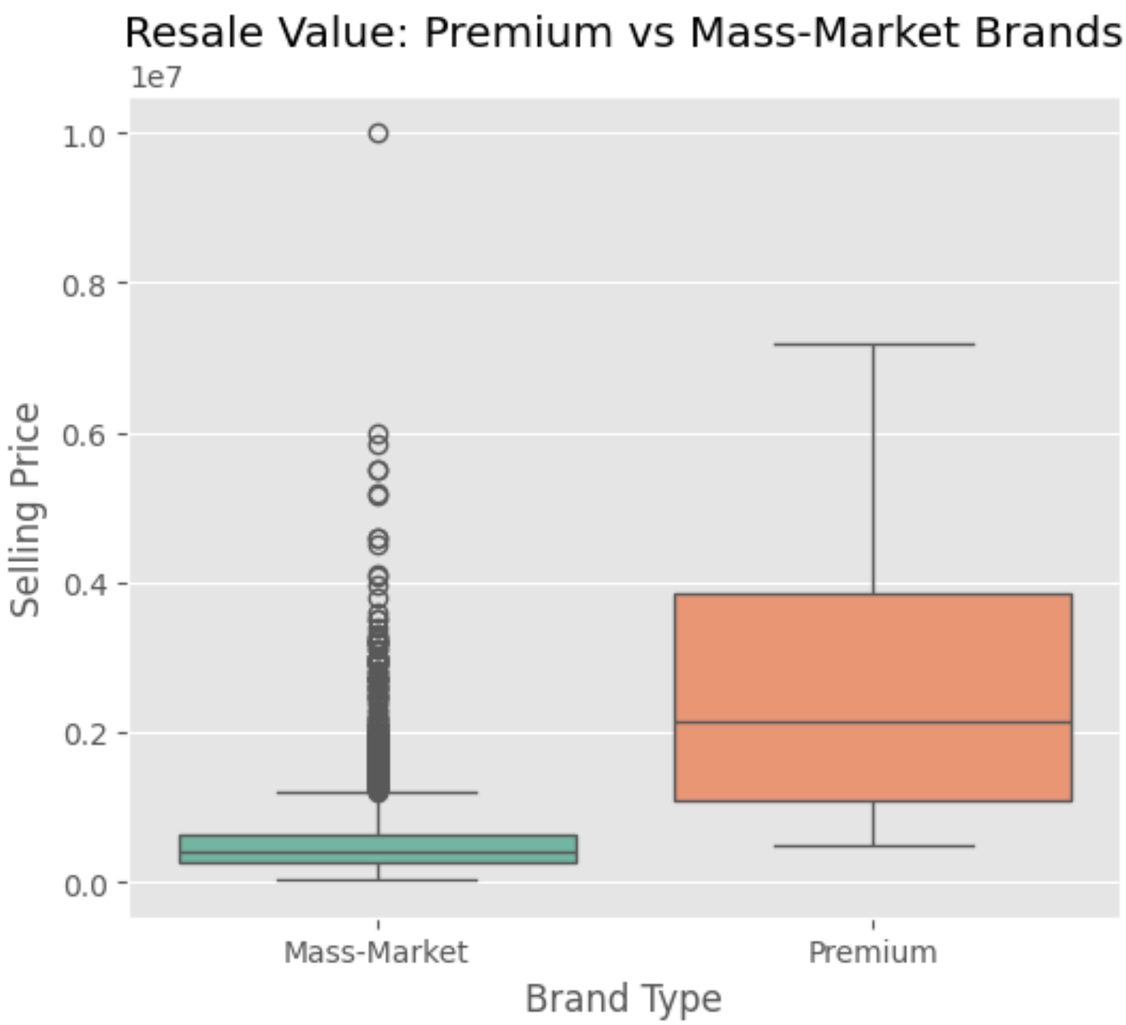

Selling Price by owner
1e7
Selling Price
1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
owner


Selling Price by brand
1e6
Selling Price
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Maruti
Hyundai
Mahindra
Tata
Honda
Ford
Toyota
Chevrolet
Renault
Volkswagen
brand

FEATURE ENGINEERING

In [31]:

```
# Focus only on premium vs non-premium
data["brand_type"] = data["brand"].apply(lambda x: "Premium" if x i

plt.figure(figsize=(6, 5))
sns.boxplot(
    data=data,
    x="brand_type",
    y=np.exp(data["selling_price"]),
    palette="Set2"
)

plt.title("Resale Value: Premium vs Mass-Market Brands")
plt.ylabel("Selling Price")
plt.xlabel("Brand Type")
plt.show()
```

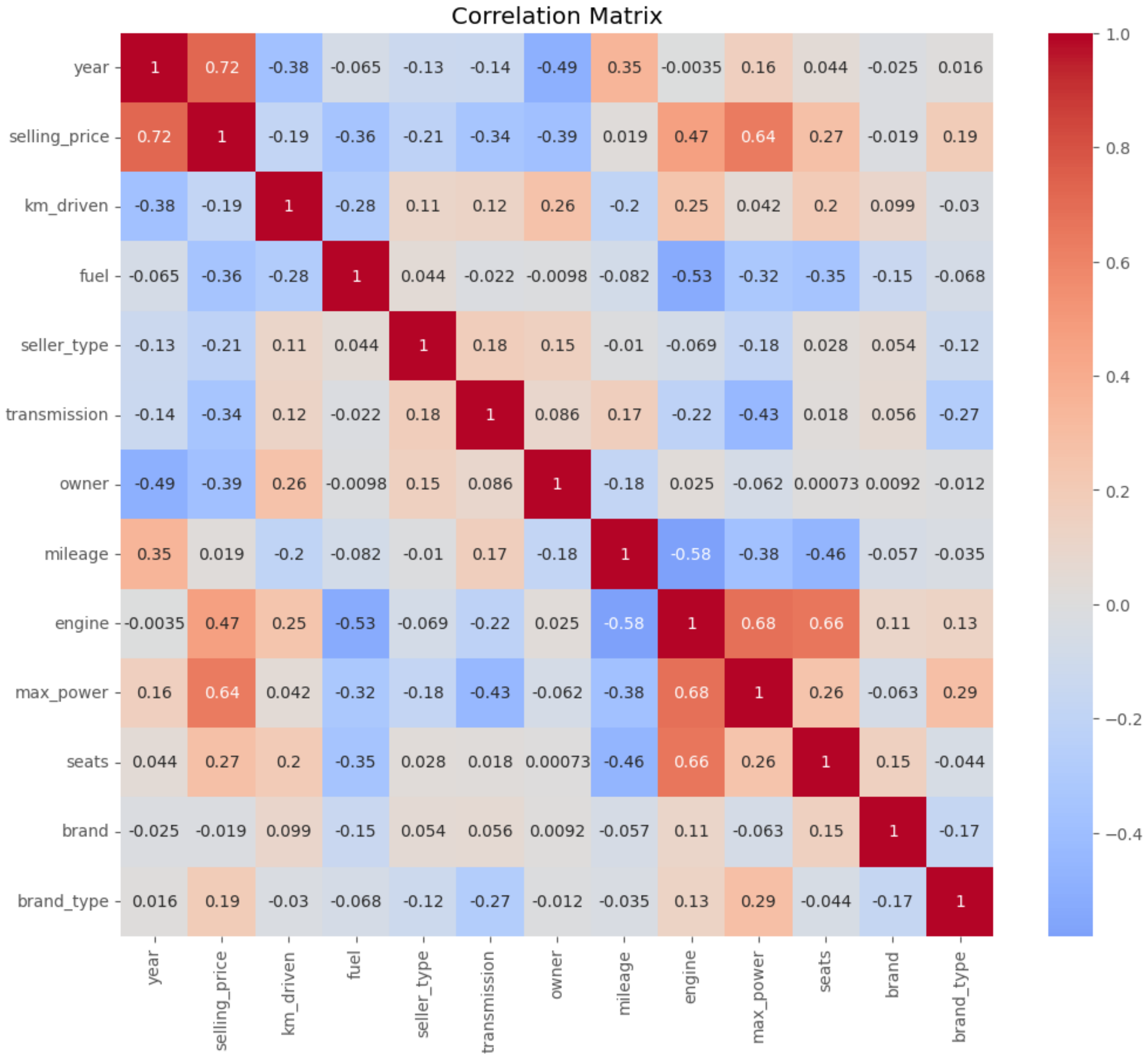


CORRELATION METRICS

In [32]:

```
# Correlation analysis
from sklearn.preprocessing import LabelEncoder
# Create a copy for correlation analysis
```

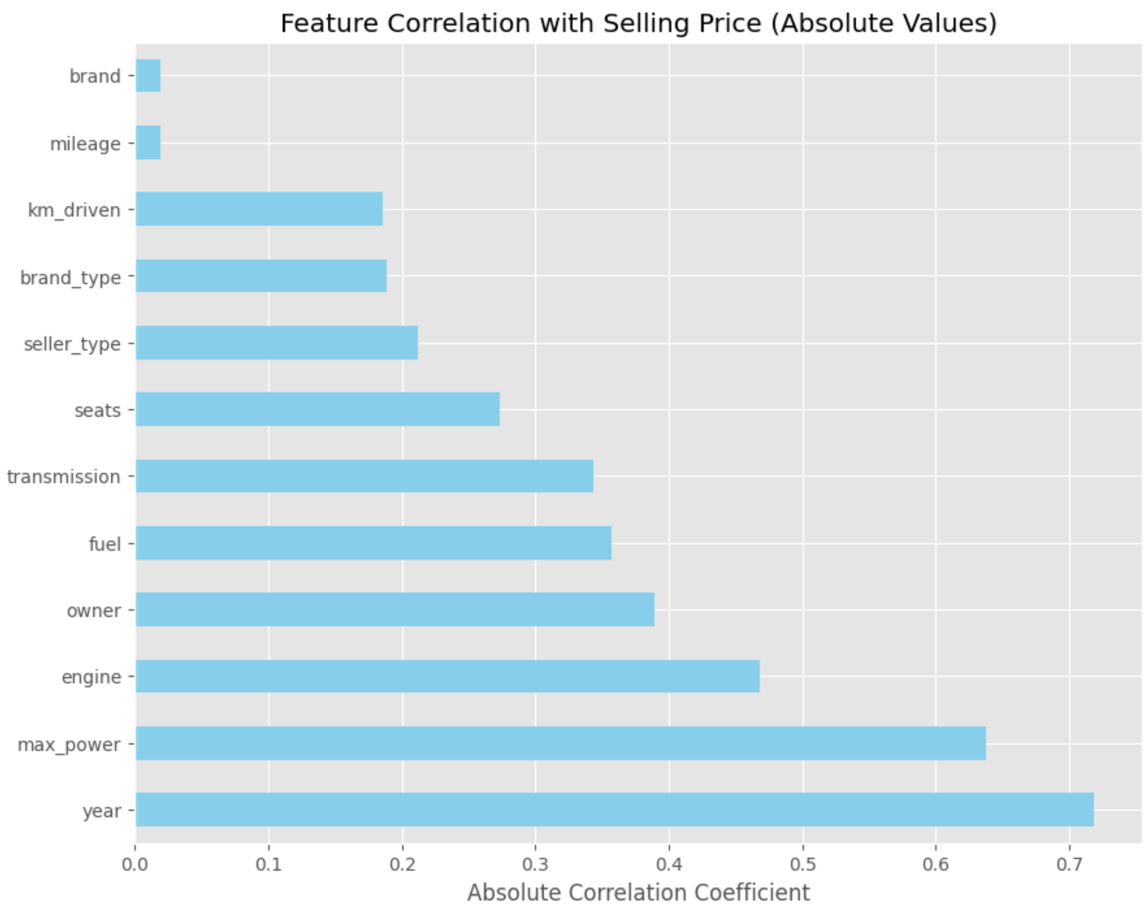
```python
corr_data = data.copy()

# Encode categorical variables
cat_cols = corr_data.select_dtypes(exclude='number').columns
le_dict = {}
for col in cat_cols:
    le = LabelEncoder()
    corr_data[col] = le.fit_transform(corr_data[col].astype(str))
    le_dict[col] = le

# Calculate correlation matrix
corr_matrix = corr_data.corr()

# Plot correlation heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0)
plt.title("Correlation Matrix")
plt.show()
```



In [33]:
```python
# Correlation with target
corr_with_target = corr_matrix['selling_price'].drop('selling_price
corr_sorted = corr_with_target.reindex(corr_with_target.abs().sort_

print("Feature correlations with selling price:")
print(corr_sorted)
```

```
Feature correlations with selling price:
year            0.718678
max_power       0.637513
engine          0.468379
owner          -0.389101
fuel           -0.356654
transmission   -0.343871
seats           0.273511
seller_type    -0.212444
brand_type      0.188333
km_driven      -0.185280
mileage         0.018881
brand          -0.018835
Name: selling_price, dtype: float64
```

In [34]:
```python
# Plot feature importance based on correlation
plt.figure(figsize=(10, 8))
corr_sorted.abs().plot(kind='barh', color='skyblue')
plt.title("Feature Correlation with Selling Price (Absolute Values)'
plt.xlabel("Absolute Correlation Coefficient")
plt.show()
```



FEATURE SELECTION

In [35]:
```python
# Prepare features and target for modeling
feature_cols = ['year', 'max_power', 'engine', 'owner', 'fuel', 'tra
X = data[feature_cols]
y = data["selling_price"]
```

In [36]:
```
X
```

Out[36]:

	year	max_power	engine	owner	fuel	transmission	seats	seller_
0	2014	74.00	1248.0	1	Diesel	Manual	5.0	Indiv
1	2014	103.52	1498.0	2	Diesel	Manual	5.0	Indiv
2	2006	78.00	1497.0	3	Petrol	Manual	5.0	Indiv
3	2010	90.00	1396.0	1	Diesel	Manual	5.0	Indiv
4	2007	88.20	1298.0	1	Petrol	Manual	5.0	Indiv
...	...	...	...	...	...	...	...	
8121	2013	67.10	998.0	2	Petrol	Manual	5.0	Indiv
8122	2014	88.73	1396.0	2	Diesel	Manual	5.0	Indiv
8123	2013	82.85	1197.0	1	Petrol	Manual	5.0	Indiv
8124	2007	110.00	1493.0	4	Diesel	Manual	5.0	Indiv
8125	2009	73.90	1248.0	1	Diesel	Manual	5.0	Indiv

6827 rows × 12 columns

LABEL ENCODING

In [37]:
```
# Dictionary to hold encoders
label_encoders = {}

categorical_cols = [
    'fuel',
    'transmission',
    'seller_type',
    'brand',
    'brand_type'
    ]

for col in categorical_cols:
    le = LabelEncoder()
    X[col] = le.fit_transform(X[col].astype(str))
    label_encoders[col] = le
```

In [38]:
```python
y = pd.qcut(y, q=4, labels=[0,1,2,3])
```

In [39]:
```python
# Split data into train and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size


print(f"Training set shape: {X_train.shape}")
print(f"Testing set shape: {X_test.shape}")
```

```
Training set shape: (5461, 12)
Testing set shape: (1366, 12)
```

In [40]:
```python
X_train = X_train.to_numpy()
X_test = X_test.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

# One-hot encode y_train
k = len(np.unique(y))   # 4 here
Y_train_encoded = np.zeros((y_train.shape[0], k))
Y_train_encoded[np.arange(y_train.shape[0]), y_train] = 1
```

In [41]:
```python
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[41]:
```
((5461, 12), (5461,), (1366, 12), (1366,))
```

In [42]:
```python
Y_train_encoded.shape
```

Out[42]:
```
(5461, 4)
```

LOGISTIC REGRESSION

In [43]:
```python
import numpy as np
import matplotlib.pyplot as plt
import time

class LogisticRegression:
    def __init__(self, k, n, lr=0.001, max_iter=1000, l2_penalty=Fa
                 patience=10, min_delta=1e-4):
        self.k = k          # number of classes
        self.n = n          # number of features
        self.lr = lr
        self.max_iter = max_iter
        self.l2_penalty = l2_penalty
        self.lambda_ = lambda_
        self.momentum = momentum
        self.patience = patience
        self.min_delta = min_delta

    def _xavier_init(self):
        limit = np.sqrt(6 / (self.n + self.k))
```

```python
        W = np.random.uniform(-limit, limit, size=(self.n, self.k))
        b = np.zeros((1, self.k))
        return W, b

    def softmax(self, Z):
        Z = np.array(Z, dtype=float)
        Z = Z - np.max(Z, axis=1, keepdims=True)  # stability trick
        expZ = np.exp(Z)
        return expZ / np.sum(expZ, axis=1, keepdims=True)

    def _predict(self, X):
        return self.softmax(np.dot(X, self.W) + self.b)

    def predict(self, X_test):
        return np.argmax(self._predict(X_test), axis=1)

    def gradient(self, X, Y):
        X = np.array(X, dtype=float)
        Y = np.array(Y, dtype=float)
        m = X.shape[0]

        H = self._predict(X)
        loss = -np.sum(Y * np.log(H + 1e-9)) / m

        grad_W = np.dot(X.T, (H - Y)) / m
        grad_b = np.sum(H - Y, axis=0, keepdims=True) / m

        if self.l2_penalty:
            grad_W += (self.lambda_ / m) * self.W
            loss += (self.lambda_ / (2*m)) * np.sum(self.W**2)

        return loss, grad_W, grad_b

    def fit(self, X, Y):
        X = np.array(X, dtype=float)
        Y = np.array(Y, dtype=float)

        # Xavier initialization
        self.W, self.b = self._xavier_init()
        self.losses = []

        # Initialize velocities
        vW = np.zeros_like(self.W)
        vb = np.zeros_like(self.b)

        start_time = time.time()
        best_loss = float("inf")
        patience_counter = 0

        for i in range(self.max_iter):
            # Full batch gradient descent
            loss, grad_W, grad_b = self.gradient(X, Y)

            # Momentum update
            vW = self.momentum * vW - self.lr * grad_W
            vb = self.momentum * vb - self.lr * grad_b
```

```
            self.W += vW
            self.b += vb

            # Logging every 100 steps
            if i % 100 == 0:
                self.losses.append(loss)
                print(f"Loss at iteration {i}: {loss}")

            # Early stopping check
            if loss + self.min_delta < best_loss:
                best_loss = loss
                patience_counter = 0
            else:
                patience_counter += 1

            if patience_counter >= self.patience:
                print(f"Early stopping at iteration {i}, loss={loss
                break

        print(f"Time taken: {time.time() - start_time:.2f} seconds"

    def plot(self):
        plt.figure(figsize=(8,5))
        plt.plot(np.arange(len(self.losses))*100, self.losses, labe
        plt.xlabel("Iteration")
        plt.ylabel("Loss")
        plt.title("Training Loss over Iterations")
        plt.legend()
        plt.show()
```

In [44]:
```
m = X_train.shape[0]
n = X_train.shape[1]

k,m,n
```

Out[44]: (4, 5461, 12)

In [45]:
```
Y_train_encoded.shape
```

Out[45]: (5461, 4)

In [46]:
```
X_train = X_train.astype(float)
X_test  = X_test.astype(float)
```

Polynomial feature transformation

In [47]:
```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=True)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
```

Standardize features

In [48]:
```
X_mean = X_train.mean(axis=0)
X_std  = X_train.std(axis=0)
X_train_std = (X_train - X_mean) / X_std
X_test_std  = (X_test - X_mean) / X_std
```

Model fitting

In [49]:
```
#define model
model = LogisticRegression(k=k, n=X_train_std.shape[1], lr=0.01, ma
model.fit(X_train_std, Y_train_encoded)
```

```
Loss at iteration 0: 1.698945120380664
Loss at iteration 100: 0.8674392157322122
Loss at iteration 200: 0.7976906614274604
Loss at iteration 300: 0.7662175842558323
Loss at iteration 400: 0.7474235271053361
Loss at iteration 500: 0.7347964456245728
Loss at iteration 600: 0.7257369087323711
Loss at iteration 700: 0.7189573348000714
Loss at iteration 800: 0.7137313135538611
Loss at iteration 900: 0.7096127928741851
Loss at iteration 1000: 0.7063109073786917
Loss at iteration 1100: 0.7036270872286047
Loss at iteration 1200: 0.7014207770553547
Loss at iteration 1300: 0.6995895863213825
Loss at iteration 1400: 0.6980571976154192
Loss at iteration 1500: 0.6967656591605541
Loss at iteration 1600: 0.6956702742584439
Early stopping at iteration 1614, loss=0.695530
Time taken: 0.81 seconds
```

In [54]:
```
model.plot()
```

PREDICTION

In [50]:
```
yhat = model.predict(X_test_std)
```

In [51]:
```
yhat
```

Out[51]:
```
array([0, 3, 1, ..., 1, 0, 0])
```

In [52]:
```
X_test[0]
```

Out[52]:
```
array([2.0110e+03, 6.8000e+01, 1.3990e+03, 2.0000e+00, 0.0000e+00,
       1.0000e+00, 5.0000e+00, 0.0000e+00, 0.0000e+00, 7.7395e+04,
       2.0000e+01, 9.0000e+00])
```

In [53]:
```
X.iloc[0]
```

Out[53]:
```
year                2014.0
max_power             74.0
engine              1248.0
owner                  1.0
fuel                   0.0
transmission           1.0
seats                  5.0
seller_type            1.0
brand_type             0.0
km_driven         145500.0
mileage               23.4
brand                 20.0
Name: 0, dtype: float64
```

Custom Classification Report

In [55]:

```
# ---------------------
# Basic metrics
# ---------------------
def accuracy(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
    return np.sum(y_true == y_pred) / len(y_true)

def precision(y_true, y_pred, cls):
    tp = np.sum((y_pred == cls) & (y_true == cls))
    fp = np.sum((y_pred == cls) & (y_true != cls))
    return tp / (tp + fp) if (tp + fp) > 0 else 0.0

def recall(y_true, y_pred, cls):
    tp = np.sum((y_pred == cls) & (y_true == cls))
    fn = np.sum((y_pred != cls) & (y_true == cls))
    return tp / (tp + fn) if (tp + fn) > 0 else 0.0

def f1_score(y_true, y_pred, cls):
    p = precision(y_true, y_pred, cls)
    r = recall(y_true, y_pred, cls)
    return (2 * p * r) / (p + r) if (p + r) > 0 else 0.0

# ---------------------
# Macro metrics
# ---------------------
def macro_precision(y_true, y_pred, num_classes):
    return np.mean([precision(y_true, y_pred, cls) for cls in range

def macro_recall(y_true, y_pred, num_classes):
    return np.mean([recall(y_true, y_pred, cls) for cls in range(nu

def macro_f1(y_true, y_pred, num_classes):
    return np.mean([f1_score(y_true, y_pred, cls) for cls in range(

# ---------------------
# Weighted metrics
# ---------------------
def weighted_precision(y_true, y_pred, num_classes):
    supports = [np.sum(np.array(y_true) == cls) for cls in range(nu
    total = len(y_true)
    weights = np.array(supports) / total
    metrics = np.array([precision(y_true, y_pred, cls) for cls in ra
    return np.sum(weights * metrics) / num_classes

def weighted_recall(y_true, y_pred, num_classes):
    supports = [np.sum(np.array(y_true) == cls) for cls in range(nu
    total = len(y_true)
    weights = np.array(supports) / total
    metrics = np.array([recall(y_true, y_pred, cls) for cls in rang
    return np.sum(weights * metrics) / num_classes

def weighted_f1(y_true, y_pred, num_classes):
```

```
    supports = [np.sum(np.array(y_true) == cls) for cls in range(nu
    total = len(y_true)
    weights = np.array(supports) / total
    metrics = np.array([f1_score(y_true, y_pred, cls) for cls in ra
    return np.sum(weights * metrics) / num_classes

# ---------------------
# Full report
# ---------------------
def classification_report_custom(y_true, y_pred, num_classes):
    print("Report:")
    print(f"{'':<17}{'precision':>10}{'recall':>10}{'f1-score':>10}

    supports = [np.sum(np.array(y_true) == cls) for cls in range(nu

    for cls in range(num_classes):
        p = precision(y_true, y_pred, cls)
        r = recall(y_true, y_pred, cls)
        f1 = f1_score(y_true, y_pred, cls)
        print(f"{cls:<17}{p:>10.2f}{r:>10.2f}{f1:>10.2f}{supports[c

    acc = accuracy(y_true, y_pred)
    macro_p = macro_precision(y_true, y_pred, num_classes)
    macro_r = macro_recall(y_true, y_pred, num_classes)
    macro_f = macro_f1(y_true, y_pred, num_classes)
    weighted_p = weighted_precision(y_true, y_pred, num_classes)
    weighted_r = weighted_recall(y_true, y_pred, num_classes)
    weighted_f = weighted_f1(y_true, y_pred, num_classes)
    total_samples = len(y_true)

    print(f"\n{'accuracy':<17}{'':>20}{acc:>10.2f}{total_samples:>1
    print(f"{'macro avg':<17}{macro_p:>10.2f}{macro_r:>10.2f}{macro_
    print(f"{'weighted avg':<17}{weighted_p:>10.2f}{weighted_r:>10.
```

In [56]:
```
# prints nicely aligned report
classification_report_custom(y_test, yhat, num_classes=4)
```

```
Report:
                  precision    recall  f1-score   support
0                      0.80      0.86      0.83       366
1                      0.62      0.54      0.58       340
2                      0.59      0.60      0.59       346
3                      0.76      0.79      0.77       314

accuracy                                   0.70      1366
macro avg              0.69      0.70      0.69      1366
weighted avg           0.17      0.17      0.17      1366
```

COMPARING WITH SKLEARN

In [57]:
```
from sklearn.metrics import classification_report
print("Report:", classification_report(y_test,yhat))
```

```
Report:               precision    recall  f1-score   support

           0       0.80      0.86      0.83       366
           1       0.62      0.54      0.58       340
           2       0.59      0.60      0.59       346
           3       0.76      0.79      0.77       314

    accuracy                           0.70      1366
   macro avg       0.69      0.70      0.69      1366
weighted avg       0.69      0.70      0.69      1366
```

Understanding `support` in a Classification Report

In a **classification report**, `support` refers to the **number of true instances of each class** in the dataset.
In other words, it tells you **how many samples actually belong to that class** in your `y_true` labels.

Example

Suppose we have 4 classes (0, 1, 2, 3) and the true labels are:

```
y_true = [0, 0, 1, 2, 2, 2, 3, 3, 3, 3]
```

Then the **support** for each class is:

Class	Support (number of true samples)
0	2
1	1
2	3
3	4

Why Support is Important

- **Weighted averages:** Classes with more samples contribute more to **weighted precision, recall, or f1-score**.
- **Interpreting metrics:** Helps identify if your dataset is **imbalanced**. A class with very low support may have less reliable metrics.

Summary

> `support` = count of true samples for each class in your dataset (`y_true`)

PREDICTION FUNCTION THAT APPLIES PREPROCESSING

In [58]:
```
class CarPricePredictor:
    def __init__(self, model, label_encoders, mean, std):
        self.model = model
        self.label_encoders = label_encoders
        self.mean = mean
        self.std = std

    def preprocess(self, X_raw):
        X = X_raw.copy()
        # Label encode categorical columns
        for col, le in self.label_encoders.items():
            if col in X:
                X[col] = le.transform([X[col]])[0] if isinstance(X,
        # Convert to numpy array
        if isinstance(X, pd.Series):
            X = X.to_numpy().reshape(1, -1)
        # Standardize
        X_std = (X - self.mean) / self.std
        return X_std

    def predict(self, X_raw):
        X_processed = self.preprocess(X_raw)
        return np.argmax(self.model._predict(X_processed), axis=1)
```

WRAPPER FUNCTION WITH PROCESSING FOR MLFLOW

In [59]:
```
class CarPriceWrapper(mlflow.pyfunc.PythonModel):
    def __init__(self, predictor):
        self.predictor = predictor

    def predict(self, context, model_input):
        # Make sure it works with DataFrames or Series
        return self.predictor.predict(model_input)
```

```
/Users/saugatshakya/Library/Python/3.9/lib/python/site-packages/mlfl
ow/pyfunc/utils/data_validation.py:186: UserWarning: Add type hints
to the `predict` method to enable data validation and automatic sign
ature inference during model logging. Check https://mlflow.org/docs/
latest/model/python_model.html#type-hint-usage-in-pythonmodel for mo
re details.
  color_warning(
```

RECALCULATING METRICS FOR MLFLOW

In [60]:
```
acc = accuracy(y_test, yhat)
macro_p = macro_precision(y_test, yhat, num_classes=4)
macro_r = macro_recall(y_test, yhat, num_classes=4)
macro_f = macro_f1(y_test, yhat, num_classes=4)
```

SAMPLE DATA FOR MLFLOW

In [ ]:
```
# Now you can pass raw Pandas DataFrame/Series directly
sample_df = pd.DataFrame([{
    'year': 2014,
    'km_driven': 145500,
    'fuel': 'Diesel',
    'seller_type': 'Individual',
    'transmission': 'Manual',
    'owner': 1,
    'mileage': 23.4,
    'engine': 1248.0,
    'max_power': 74.0,
    'seats': 5.0,
    'brand': 'Maruti',
    'brand_type': 'Mass-Market'
}])
```

MLFLOW MODEL LOGGING

In [70]:
```
import joblib

local_path = "app/model/st125986-a3-model.pkl"
predictor = CarPricePredictor(
    model=model,         # your trained logistic regression
    label_encoders=label_encoders,
    mean=X_mean,
    std=X_std
)
joblib.dump(predictor, local_path)
```

Out[70]: ['app/model/st125986-a3-model.pkl']

In [67]:
```
predictor = CarPricePredictor(
    model=model,         # your trained logistic regression
    label_encoders=label_encoders,
    mean=X_mean,
    std=X_std
)
local_path = "app/model/st125986-a3-model.pkl"

print(f"Model saved locally at {local_path}")
with mlflow.start_run(run_name="logistic_regression") as run:
    # Log parameters and metrics
    mlflow.log_param("model_type", "LogisticRegression")
```

```
    mlflow.log_param("max_iter", model.max_iter)
    mlflow.log_param("lr", model.lr)

    mlflow.log_metric("accuracy", acc)
    mlflow.log_metric("macro_precision", macro_p)
    mlflow.log_metric("macro_recall", macro_r)
    mlflow.log_metric("macro_f1", macro_f)

    # Log model
    mlflow.pyfunc.log_model(
        name="model",
        python_model=CarPriceWrapper(predictor),
        input_example=sample_df
    )

    # Construct proper model URI to register
    model_uri = f"runs:/{run.info.run_id}/model"
    with open(local_path, "wb") as f:
        pickle.dump(CarPriceWrapper(predictor), f)

# Register as a new version
registered_model = mlflow.register_model(
    model_uri=model_uri,
    name="st125986-a3-model"
)

print(f"Registered version: {registered_model.version}")
```

```
Model saved locally at app/model/st125986-a3-model.pkl
2025/10/04 13:27:25 INFO mlflow.pyfunc: Inferring model signature fr
om input example
🏃 View run logistic_regression at: https://mlflow.ml.brain.cs.ait.a
c.th/#/experiments/226583874314941070/runs/9fdb7c99d43c4439897ec691b
f713669
🧪 View experiment at: https://mlflow.ml.brain.cs.ait.ac.th/#/experi
ments/226583874314941070
Registered model 'st125986-a3-model' already exists. Creating a new
version of this model...
2025/10/04 13:27:31 WARNING mlflow.tracking._model_registry.fluent:
Run with id 9fdb7c99d43c4439897ec691bf713669 has no artifacts at art
ifact path 'model', registering model based on models:/m-15cf7e66a0c
94b4c9cdc40f9c274e04a instead
2025/10/04 13:27:31 INFO mlflow.store.model_registry.abstract_store:
Waiting up to 300 seconds for model version to finish creation. Mode
l name: st125986-a3-model, version 5
Created version '5' of model 'st125986-a3-model'.
Registered version: 5
```

INFERENCE

In [ ]:

```
```

In [65]:

```
# Load from MLflow registry
model_uri = "models:/st125986-a3-model/latest"
```

```python
loaded_model = mlflow.pyfunc.load_model(model_uri)

prediction = loaded_model.predict(sample_df)
print("Predicted price:", prediction)
```

```
Downloading artifacts: 100%|██████████| 7/7 [00:00<00:00, 19.51it/s]
Predicted price: [3]
```

🚗 Car Price Prediction – Final Report

📌 Overview

This project implements a **Car Price Prediction System** with Logistic Regression, fully integrated into an **MLOps workflow**.

Key achievements include:

- Experiment tracking and model versioning using **MLflow**.
- Automated **unit testing** for model input/output.
- A **CI/CD pipeline** with GitHub Actions for testing and deployment.
- A **Flask web application** that always serves the latest model from MLflow.

🔎 Experiment Tracking with MLflow

- Experiments were logged on the MLflow tracking server.
- Metrics such as accuracy, precision, recall, and F1-score were stored.
- Each new training run automatically produced a new registered model version.
- The registry ensured that every deployed model was version-controlled and reproducible.

📦 Model Saving and Registration

- A wrapper (`CarPriceWrapper`) combined preprocessing and prediction logic.

- Trained models were:

  - Saved locally as `.pkl` files for reproducibility.
  - Logged and registered in MLflow as **`st125986-a3-model`**.
- Each commit-triggered run produced a new model version in MLflow.

🧪 Unit Testing

Two lightweight unit tests ensured reliability:

1. **Input validation** – confirmed the model accepts the expected input format.
2. **Output validation** – confirmed predictions have the correct shape.

This prevented invalid or broken models from being deployed.

⚙️ CI/CD Pipeline

A GitHub Actions pipeline was created with the following workflow:

1. **On Commit Push**
    - Run unit tests automatically.
    - Block deployment if any test fails.
2. **On Test Success**
    - Log and register a new model version in MLflow.
    - Trigger deployment so the Flask app updates automatically.

This ensures only tested models reach production.

🌐 Flask Web Application

- The Flask app was updated to always **fetch the latest model from MLflow**.
- It exposes a web interface for predictions, hosted at: 🔗 st125986.ml.brain.cs.ait.ac.th
- The full source code is available on GitHub: 🔗 github.com/saugatshakya/predicting_car_price-3

✅ Final Outcome

- **Training workflow**: Train → Log → Register → Save.
- **CI/CD workflow**: Test → Deploy → Serve latest model.
- **Deployment**: Flask app automatically updates to the newest registered model.
- **Result**: A reliable, automated, end-to-end MLOps pipeline for car price

prediction.