CSI4142 - A3: Part 1

Group: 9

Members:

- Jay Ghosh (300243766)
- Alexander Azizi-Martin (300236257)

Introduction

This notebook illustrates a high-level workflow for preparing and modeling a dataset using linear regression. The process begins with basic data validation and duplicate removal. Categorical features are then one-hot encoded, and LOF is employed to numeric outliers. New features are engineered to capture aspects like depreciation and usage patterns. Several variants of the dataset were tested using linear regression, with cross-validation guiding the choice of final model.

Dataset Description

Dataset Name: CAR DETAILS FROM CAR DEKHO [1]

Dataset Author: Nehal Birla, Nishant Verma, Nikhil Kushwaha [1]

Purpose: The dataset was built for a pedagogical purpose: to exemplify the use of linear regression in machine learning. [1]

In [1]:
```
import kagglehub
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.neighbors import LocalOutlierFactor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import mean_squared_error, r2_score
```

In [2]:
```
# Loading dataset
df_path = kagglehub.dataset_download("nehalbirla/vehicle-dataset-from-cardekho")
df = pd.read_csv(f"{df_path}/CAR DETAILS FROM CAR DEKHO.csv")
df.head()
```

```
Warning: Looks like you're using an outdated `kagglehub` version (installed: 0.3.9), please consider upgrading to th
e latest version (0.3.10).
```

Out[2]:

	name	year	selling_price	km_driven	fuel	seller_type	transmission	owner
0	Maruti 800 AC	2007	60000	70000	Petrol	Individual	Manual	First Owner
1	Maruti Wagon R LXI Minor	2007	135000	50000	Petrol	Individual	Manual	First Owner
2	Hyundai Verna 1.6 SX	2012	600000	100000	Diesel	Individual	Manual	First Owner
3	Datsun RediGO T Option	2017	250000	46000	Petrol	Individual	Manual	First Owner
4	Honda Amaze VX i-DTEC	2014	450000	141000	Diesel	Individual	Manual	Second Owner

Dataset Shape

In [3]:
```
df.shape
```

Out[3]:
```
(4340, 8)
```

The dataset has 4340 rows and 8 columns.

Features of the dataset (and what they mean)

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4340 entries, 0 to 4339
Data columns (total 8 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   name           4340 non-null   object
 1   year           4340 non-null   int64 
 2   selling_price  4340 non-null   int64 
 3   km_driven      4340 non-null   int64 
 4   fuel           4340 non-null   object
 5   seller_type    4340 non-null   object
 6   transmission   4340 non-null   object
 7   owner          4340 non-null   object
dtypes: int64(3), object(5)
memory usage: 271.4+ KB
```

Features:

name:

- Type: Categorical
- Purpose: Represents the model or make of a car.

year:

- Type: Numerical
- Purpose: Year of manufacture of the car.

selling_price:

- Type: Numerical
- Purpose: Price at which the car is sold. **The target variable for regression.**

km_driven

- Type: Numerical
- Purpose: Kilometers driven by the vehicle, impacting its depreciation.

fuel

- Type: Categorical
- Purpose: Type of fuel used, e.g. petrol, diesel.

seller_type

- Type: Categorical
- Purpose: Indicates the type of seller, e.g. individual or dealer.

transmission

- Type: Categorical
- Purpose: Indicates vehicle transmission type, e.g. automatic or manual.

owner

- Type: Categorical
- Purpose: Number of previous owners, indicating vehicle usage and condition history.

Section A: Validating and Cleaning

Check 1: Data Type

In [5]:
```python
# Define columns with explicitly expected datatypes
expected_dtypes = {
    "year": "numeric",
    "selling_price": "numeric",
    "km_driven": "numeric",
    "name": "string",
    "fuel": "string",
    "seller_type": "string",
    "transmission": "string",
    "owner": "string"
}

# Checker Code
errors = {}
for col, expected_type in expected_dtypes.items():
  if expected_type == "numeric":
    parsed_col = pd.to_numeric(df[col], errors="coerce")
    failed_mask = parsed_col.isna() & df[col].notna()
  elif expected_type == "string":
```

```
    failed_mask = ~df[col].apply(lambda x: isinstance(x, str))
  errors[col] = df.loc[failed_mask]

# Results
for col, errors_df in errors.items():
  type_to_check = expected_dtypes[col]
  print(f"Report on Data Type Check for '{col}' ({type_to_check})")
  if errors_df.empty:
    print(f"No data type errors found in column '{col}'.")
  else:
    print(f"{len(errors_df)} rows have invalid {type_to_check} values in '{col}'.")
    print("Examples of invalid rows:")
    display(errors_df.head(10))
```

```
Report on Data Type Check for 'year' (numeric)
No data type errors found in column 'year'.
Report on Data Type Check for 'selling_price' (numeric)
No data type errors found in column 'selling_price'.
Report on Data Type Check for 'km_driven' (numeric)
No data type errors found in column 'km_driven'.
Report on Data Type Check for 'name' (string)
No data type errors found in column 'name'.
Report on Data Type Check for 'fuel' (string)
No data type errors found in column 'fuel'.
Report on Data Type Check for 'seller_type' (string)
No data type errors found in column 'seller_type'.
Report on Data Type Check for 'transmission' (string)
No data type errors found in column 'transmission'.
Report on Data Type Check for 'owner' (string)
No data type errors found in column 'owner'.
```

No data type errors, so no cleaning required here.

Check 2: Range Check

In [6]:

```
# Define ranges for numeric columns
ranges = {
    "year": (1900, 2025),
    "selling_price": (1, 1e7),
    "km_driven": (0, 1e6)
}

# Checker Code
range_errors = {}
for col, (min_val, max_val) in ranges.items():
    out_of_range_mask = (df[col] < min_val) | (df[col] > max_val)
    range_errors[col] = df.loc[out_of_range_mask]

# Results
for col, error_df in range_errors.items():
    print(f"Report on Range Check for '{col}' (Range: {ranges[col]})")
    if error_df.empty:
        print(f"No range errors found in column '{col}'.")
    else:
        print(f"{len(error_df)} range errors found in column '{col}'.")
        print("Examples of out-of-range rows:")
        display(error_df.head(10))
```

```
Report on Range Check for 'year' (Range: (1900, 2025))
No range errors found in column 'year'.
Report on Range Check for 'selling_price' (Range: (1, 10000000.0))
No range errors found in column 'selling_price'.
Report on Range Check for 'km_driven' (Range: (0, 1000000.0))
No range errors found in column 'km_driven'.
```

No range errors, so no cleaning required here.

Check 3: Format Check

In [7]:

```
# Define format rules as regex patterns for relevant columns
format_rules = {
    "year": r"^\d{4}$",  # 4 digits
}

# Checker Code
format_errors = {}
for col, pattern in format_rules.items():
    failed_mask = ~df[col].astype(str).str.match(pattern)
    format_errors[col] = df.loc[failed_mask]

# Results
for col, error_df in format_errors.items():
    print(f"Report on Format Check for '{col}' (Pattern: '{format_rules[col]}')")
    if error_df.empty:
        print(f"No format errors found in column '{col}'.")
    else:
```

```
        print(f"{len(error_df)} format errors found in column '{col}'.")
        print("Examples of format errors:")
        display(error_df.head(10))
```

```
Report on Format Check for 'year' (Pattern: '^\d{4}$')
No format errors found in column 'year'.
```

No format errors, so no cleaning required here.

Check 4: Consistency Check

While we do have columns like year, selling_price, km_driven, fuel, seller_type, transmission, and owner, there is no strict, guaranteed set of logical dependencies in this dataset. For example, it is not explicitly required that a specific fuel type must always imply a specific transmission value, nor that a particular owner status dictates a precise range for km_driven. Any rules we could create (for instance, "electric cars can't be manual") tend to be heuristic rather than definitively contradictory. As such, pure consistency checks (strict logic rules that a violation would necessarily mean invalid data) do not meaningfully apply here.

Check 5: Uniqueness Check

In our dataset, columns such as name, year, selling_price, km_driven, fuel, seller_type, transmission, and owner do not include any natural primary key (like a car_id or VIN) that must be uniquely assigned to each record. The combination of, for instance, name and year can recur legitimately for multiple cars of the same model and production year, and no domain rule explicitly states any field or set of fields must be unique across the dataset. Thus, a "uniqueness check" provides little value here, as repeated entries do not necessarily indicate invalid or duplicated data but can simply reflect different cars sharing attributes.

Check 6: Presence Check

In [8]:
```python
# Define which columns must not be null for presence checks
presence_cols = [
    "year",
    "selling_price",
    "km_driven",
    "name",
    "fuel",
    "seller_type",
    "transmission",
    "owner"
]

# Checker Code
presence_errors = {}
for col in presence_cols:
    missing_mask = df[col].isnull()
    presence_errors[col] = df.loc[missing_mask]

# Results
for col, error_df in presence_errors.items():
    print(f"Report on Presence Check for '{col}'")
    if error_df.empty:
        print(f"No missing values found in column '{col}'.")
    else:
        print(f"{len(error_df)} rows have missing (null) values in '{col}'.")
        print("Examples of missing-value rows:")
        display(error_df.head(10))
```

```
Report on Presence Check for 'year'
No missing values found in column 'year'.
Report on Presence Check for 'selling_price'
No missing values found in column 'selling_price'.
Report on Presence Check for 'km_driven'
No missing values found in column 'km_driven'.
Report on Presence Check for 'name'
No missing values found in column 'name'.
Report on Presence Check for 'fuel'
No missing values found in column 'fuel'.
Report on Presence Check for 'seller_type'
No missing values found in column 'seller_type'.
Report on Presence Check for 'transmission'
No missing values found in column 'transmission'.
Report on Presence Check for 'owner'
No missing values found in column 'owner'.
```

No presence check errors, so no cleaning required here.

Check 7: Length Check

In [9]:
```python
# Define the minimum and maximum allowed lengths for each string column
# Format: { column_name: (min_length, max_length) }
length_constraints = {
    "name": (1, 70),
    "fuel": (3, 10),
    "seller_type": (3, 20),
```

```
    "transmission": (3, 10),
    "owner": (5, 20)
}

# Checker Code
length_errors = {}
for col, (min_len, max_len) in length_constraints.items():
    str_col = df[col].astype(str)
    # Identify rows that are too short or too long
    too_short_mask = str_col.str.len() < min_len
    too_long_mask = str_col.str.len() > max_len
    # Combine both to create the final mask
    length_mask = too_short_mask | too_long_mask
    length_errors[col] = df.loc[length_mask]

# Results
for col, error_df in length_errors.items():
    print(f"Report on Length Check for '{col}' (Allowed: {length_constraints[col]})")
    if error_df.empty:
        print(f"No length errors found in column '{col}'.")
    else:
        print(f"{len(error_df)} rows have length errors in '{col}'.")
        print("Examples of invalid length rows:")
        display(error_df.head(10))
```

```
Report on Length Check for 'name' (Allowed: (1, 70))
No length errors found in column 'name'.
Report on Length Check for 'fuel' (Allowed: (3, 10))
No length errors found in column 'fuel'.
Report on Length Check for 'seller_type' (Allowed: (3, 20))
No length errors found in column 'seller_type'.
Report on Length Check for 'transmission' (Allowed: (3, 10))
No length errors found in column 'transmission'.
Report on Length Check for 'owner' (Allowed: (5, 20))
No length errors found in column 'owner'.
```

No length errors, so no cleaning required here.

Check 8: Lookup Check

In [10]:
```
# Define valid values for each categorical column
lookup_valid_values = {
  "fuel": ['Petrol', 'Diesel', 'CNG', 'LPG', 'Electric'],
  "seller_type": ["Individual", "Dealer", "Trustmark Dealer"], # trustmark dealer and dealer are distinct definitio
  "transmission": ["Manual", "Automatic"],
  "owner": ['First Owner', 'Second Owner', 'Fourth & Above Owner', 'Third Owner', 'Test Drive Car']
}

# Checker Code
lookup_errors = {}
for col, valid_list in lookup_valid_values.items():
  invalid_mask = ~df[col].isin(valid_list)
  lookup_errors[col] = df.loc[invalid_mask]

# Results
for col, error_df in lookup_errors.items():
  print(f"Report on Look-up Check for '{col}' (Valid Values: {lookup_valid_values[col]})")
  if error_df.empty:
    print(f"No invalid categories found in column '{col}'.")
  else:
    print(f"{len(error_df)} rows have invalid categories in '{col}'.")
    print("Examples of invalid-category rows:")
    display(error_df.head(10))
```

```
Report on Look-up Check for 'fuel' (Valid Values: ['Petrol', 'Diesel', 'CNG', 'LPG', 'Electric'])
No invalid categories found in column 'fuel'.
Report on Look-up Check for 'seller_type' (Valid Values: ['Individual', 'Dealer', 'Trustmark Dealer'])
No invalid categories found in column 'seller_type'.
Report on Look-up Check for 'transmission' (Valid Values: ['Manual', 'Automatic'])
No invalid categories found in column 'transmission'.
Report on Look-up Check for 'owner' (Valid Values: ['First Owner', 'Second Owner', 'Fourth & Above Owner', 'Third Ow
ner', 'Test Drive Car'])
No invalid categories found in column 'owner'.
```

No lookup check errors, so no cleaning required here.

Check 9: Duplicate Check

In [11]:
```
# Checker Code
duplicate_errors = df[df.duplicated(keep='first')]

# Results
if duplicate_errors.empty:
    print("No exact duplicate errors found.")
else:
    print(f"{len(duplicate_errors)} rows are involved in exact duplicates.")
```

```
    print("Examples of duplicate rows:")
    display(duplicate_errors.head(10))
```

```
763 rows are involved in exact duplicates.
Examples of duplicate rows:
```

	name	year	selling_price	km_driven	fuel	seller_type	transmission	owner
13	Maruti 800 AC	2007	60000	70000	Petrol	Individual	Manual	First Owner
14	Maruti Wagon R LXI Minor	2007	135000	50000	Petrol	Individual	Manual	First Owner
15	Hyundai Verna 1.6 SX	2012	600000	100000	Diesel	Individual	Manual	First Owner
16	Datsun RediGO T Option	2017	250000	46000	Petrol	Individual	Manual	First Owner
17	Honda Amaze VX i-DTEC	2014	450000	141000	Diesel	Individual	Manual	Second Owner
18	Maruti Alto LX BSIII	2007	140000	125000	Petrol	Individual	Manual	First Owner
19	Hyundai Xcent 1.2 Kappa S	2016	550000	25000	Petrol	Individual	Manual	First Owner
20	Tata Indigo Grand Petrol	2014	240000	60000	Petrol	Individual	Manual	Second Owner
21	Hyundai Creta 1.6 VTVT S	2015	850000	25000	Petrol	Individual	Manual	First Owner
22	Maruti Celerio Green VXI	2017	365000	78000	CNG	Individual	Manual	First Owner

Cleaning Duplicate Rows

In [12]:
```
print("Removing duplicate rows...")
initial_len = len(df)
df.drop_duplicates(keep='first', inplace=True)
final_len = len(df)
print(f"Removed {initial_len - final_len} duplicate rows. Updated dataset length: {final_len}")
```

```
Removing duplicate rows...
Removed 763 duplicate rows. Updated dataset length: 3577
```

Verifying No Duplicate Rows

In [13]:
```
# Checker Code
duplicate_errors = df[df.duplicated(keep='first')]

# Results
if duplicate_errors.empty:
    print("No exact duplicate errors found.")
else:
    print(f"{len(duplicate_errors)} rows are involved in exact duplicates.")
    print("Examples of duplicate rows:")
    display(duplicate_errors.head(10))
```

```
No exact duplicate errors found.
```

Check 10: Near Duplicate Errors

In this dataset, attributes such as name, year, selling_price, and km_driven can vary significantly, even for cars of the same model. A single difference in mileage, price, or fuel type may reflect a genuinely distinct listing. Since these columns play a decisive role in identifying a car's unique identity, labeling rows with minor differences as "near duplicates" can lead to misclassification of truly separate entries. Consequently, a near-duplicate check does not meaningfully apply here, as the dataset's attributes carry enough variability and relevance that each row's tiny variation is likely important rather than indicative of redundant data.

We will still check the semantic uniqueness of categorical attributes (except "name"--we don't use "name" in our analysis), as it's not best practice to ignore synonyms of words when creating near duplicates.

In [14]:
```
# Checking unique values for all categorical features
for col in ["fuel", "seller_type", "transmission", "owner"]:
    unique_vals = df[col].unique()
    print(f"Column: {col}")
    print("Unique Values:", unique_vals)
    print(f"Total Unique: {len(unique_vals)}")
```

```
Column: fuel
Unique Values: ['Petrol' 'Diesel' 'CNG' 'LPG' 'Electric']
Total Unique: 5
Column: seller_type
Unique Values: ['Individual' 'Dealer' 'Trustmark Dealer']
Total Unique: 3
Column: transmission
Unique Values: ['Manual' 'Automatic']
Total Unique: 2
Column: owner
Unique Values: ['First Owner' 'Second Owner' 'Fourth & Above Owner' 'Third Owner'
 'Test Drive Car']
Total Unique: 5
```

All unique values for the categorical attributes are distinct. One could suspect closeness in meaning between 'Dealer' and 'Trustmark Dealer', but they are distinct terms used to signify the difference between a "third-party dealer" and a "trusted, official dealer",

respectively.

Section B: One-Hot Encoding

Here, we first list the columns suitable for one-hot encoding (fuel, seller_type, transmission, and owner), then use pd.get_dummies to one-hot encode. We set drop_first=True to avoid perfect multicollinearity ("dummy variable trap") [2], ensuring only (k-1) indicator columns for each categorical feature. We don't one-hot encode the "name" column, as that has too many unique values.

In [15]:
```
# Define our categorical columns
categorical_cols = ["fuel", "seller_type", "transmission", "owner"]
# One-hot encode but also keep original columns
# This is for categorical outlier analysis
dummy_subset = pd.get_dummies(df[categorical_cols], drop_first=False)
df_encoded = pd.concat([df, dummy_subset], axis=1)
# Displaying the transformed dataset
print("Original shape:", df.shape)
print("Encoded shape:", df_encoded.shape)
print("One-hot encoded dataset (head):")
df_encoded.head()
```

```
Original shape: (3577, 8)
Encoded shape: (3577, 23)
One-hot encoded dataset (head):
```

Out[15]:

	name	year	selling_price	km_driven	fuel	seller_type	transmission	owner	fuel_CNG	fuel_Diesel	...	seller_type_Deale
0	Maruti 800 AC	2007	60000	70000	Petrol	Individual	Manual	First Owner	False	False	...	Fals
1	Maruti Wagon R LXI Minor	2007	135000	50000	Petrol	Individual	Manual	First Owner	False	False	...	Fals
2	Hyundai Verna 1.6 SX	2012	600000	100000	Diesel	Individual	Manual	First Owner	False	True	...	Fals
3	Datsun RediGO T Option	2017	250000	46000	Petrol	Individual	Manual	First Owner	False	False	...	Fals
4	Honda Amaze VX i-DTEC	2014	450000	141000	Diesel	Individual	Manual	Second Owner	False	True	...	Fals

5 rows × 23 columns

Removing "Name"

We drop the "name" column, because it's largely unstructured text identifying the car's make/model and doesn't directly contribute numerical or categorical information that a linear regression model can leverage.

In [16]:
```
df = df_encoded.drop(columns=["name"])
# Show new dataframe without name column
display(df.head())
```

	year	selling_price	km_driven	fuel	seller_type	transmission	owner	fuel_CNG	fuel_Diesel	fuel_Electric	...	seller_type_De
0	2007	60000	70000	Petrol	Individual	Manual	First Owner	False	False	False	...	F
1	2007	135000	50000	Petrol	Individual	Manual	First Owner	False	False	False	...	F
2	2012	600000	100000	Diesel	Individual	Manual	First Owner	False	True	False	...	F
3	2017	250000	46000	Petrol	Individual	Manual	First Owner	False	False	False	...	F
4	2014	450000	141000	Diesel	Individual	Manual	Second Owner	False	True	False	...	F

5 rows × 22 columns

Creating DR & DT

We separate the target column (selling_price) from the other features, then split the dataset into two parts: 80% as the Reduced

Training set (DR) and 20% as the final Test set (DT). We set the random state to 42 to ensure reproducibility.

In [17]:
```
# Splitting the total dataset
# Taking 80% for DR, 20% for DT
DR, DT = train_test_split(df, test_size=0.2, random_state=42)
print("DR shape:", DR.shape)
print("DT shape:", DT.shape)
```

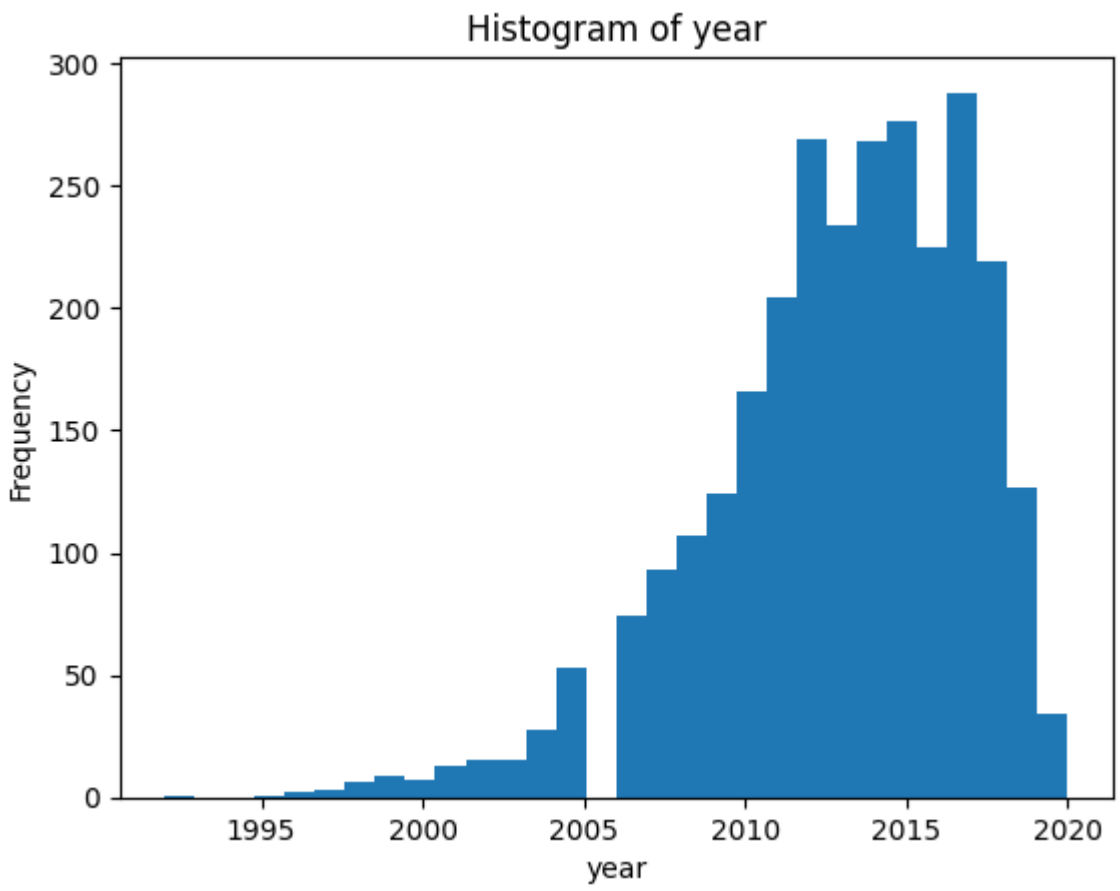
```
DR shape: (2861, 22)
DT shape: (716, 22)
 (2861, 22)
DT shape: (716, 22)
```

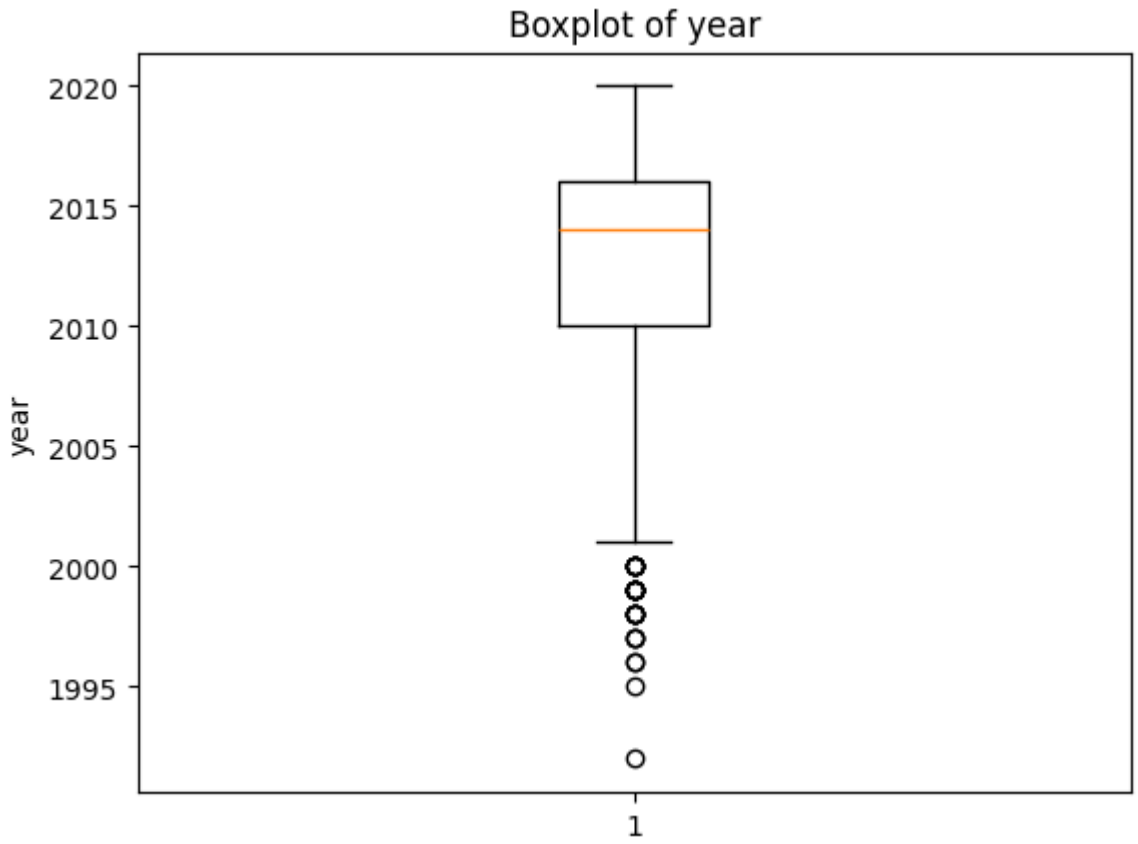
Section C: Outliers and LOF

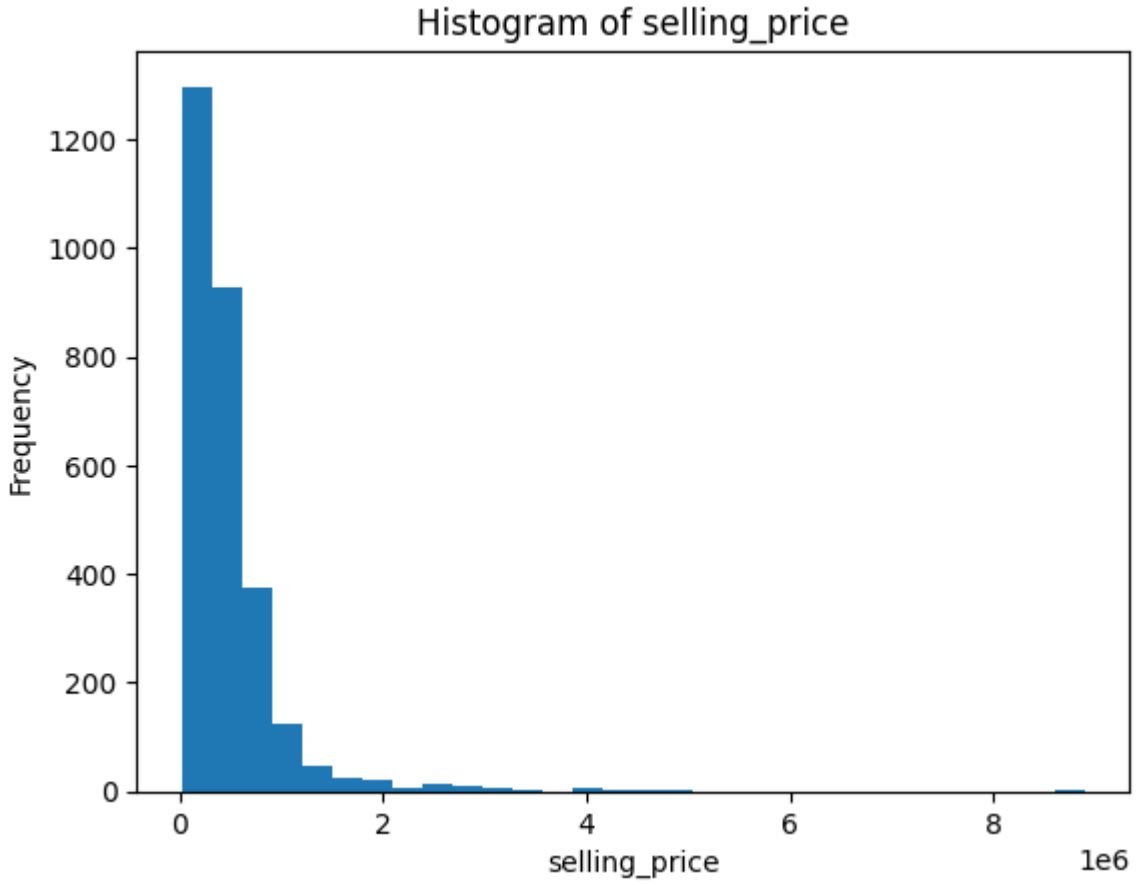
Finding features for LOF

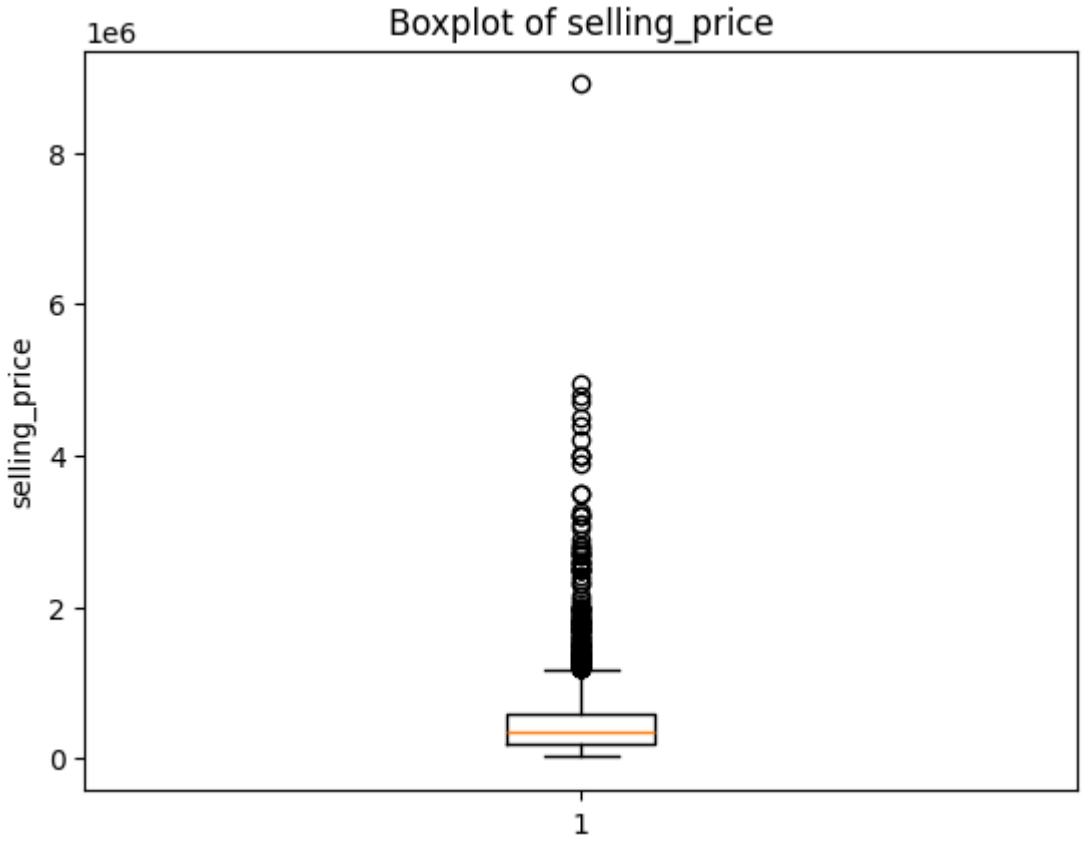
First, we display the distributions of relevant, numerical features for LOF with possible outliers.

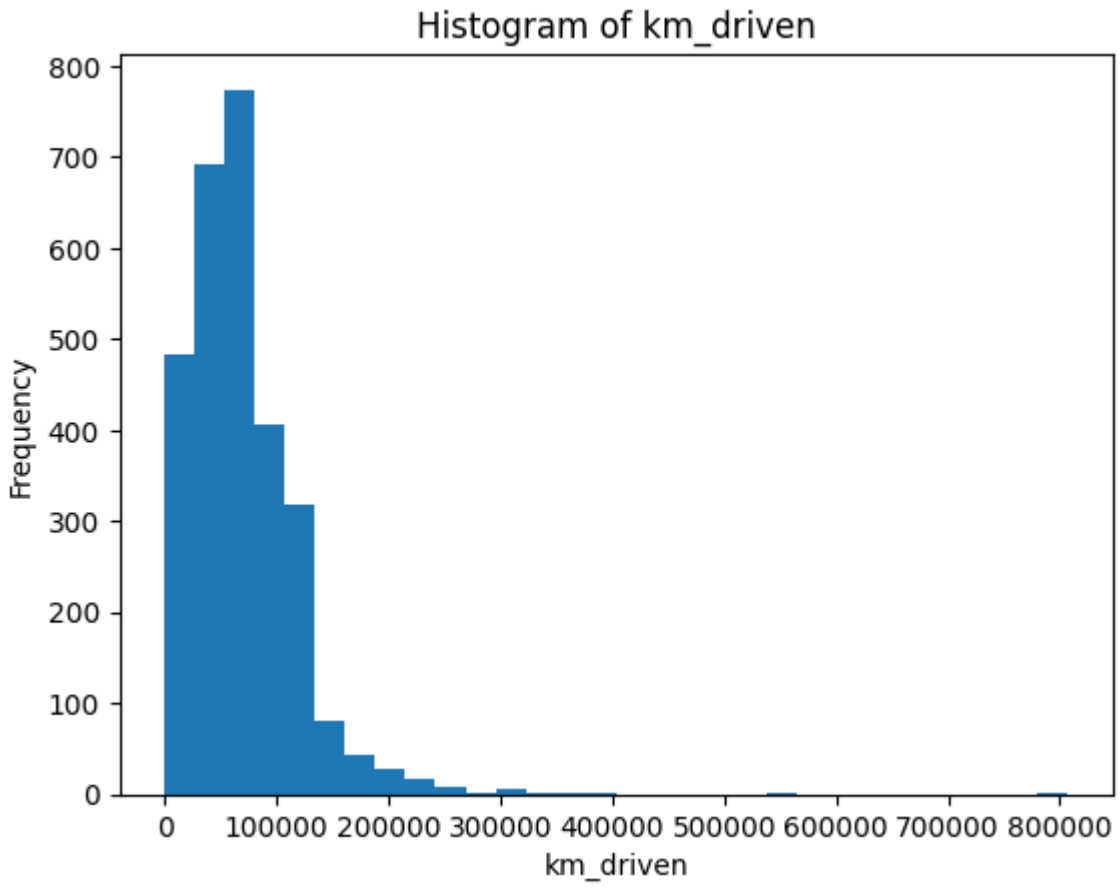
In [18]:
```
# Numeric columns relevant to outlier detection and LOF usage
numeric_cols = ["year", "selling_price", "km_driven"]
# Display distribution for each numeric column
for col in numeric_cols:
    plt.figure()
    plt.hist(DR[col].dropna(), bins=30)
    plt.title(f"Histogram of {col}")
    plt.xlabel(col)
    plt.ylabel("Frequency")
    plt.show()
    plt.figure()
    plt.boxplot(DR[col].dropna())
    plt.title(f"Boxplot of {col}")
    plt.ylabel(col)
    plt.show()
```

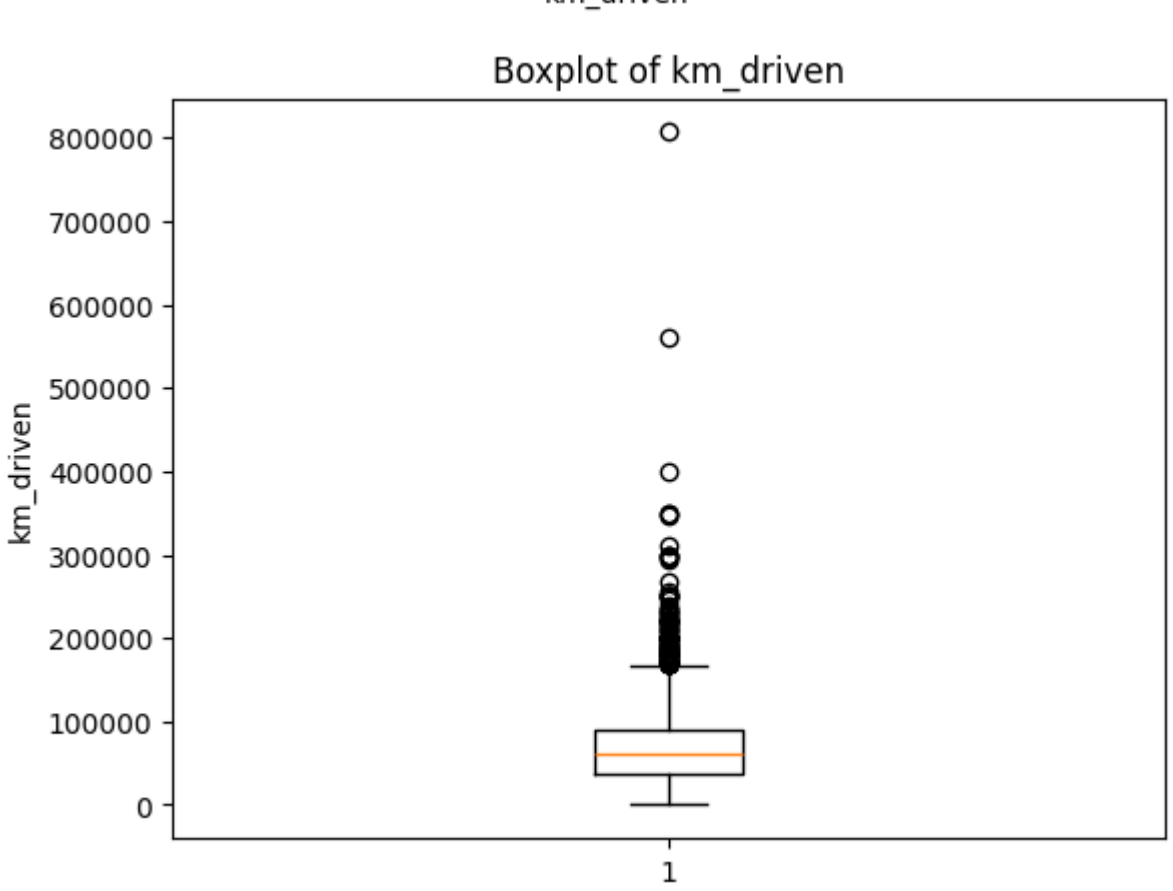



Boxplot of year
2020
2015
2010
2005
2000
1995
year
1


Histogram of selling_price
1200
1000
800
600
400
200
0
Frequency
0
2
4
6
8
1e6
selling_price


Boxplot of selling_price
1e6
8
6
4
2
0
selling_price
1





selling_price and km_driven stand out as the strongest candidates for LOF (Local Outlier Factor) detection because both variables exhibit heavy right-skew and contain a notable fraction of extreme values. LOF identifies points that lie in low-density regions compared to their local neighborhood, making it well suited for spotting anomalously high or low mileage-price pairs. In the context of our dataset, a vehicle with unusually high mileage but an excessively high price, or vice versa, can signal potential data errors or listings that are not representative of the broader market.

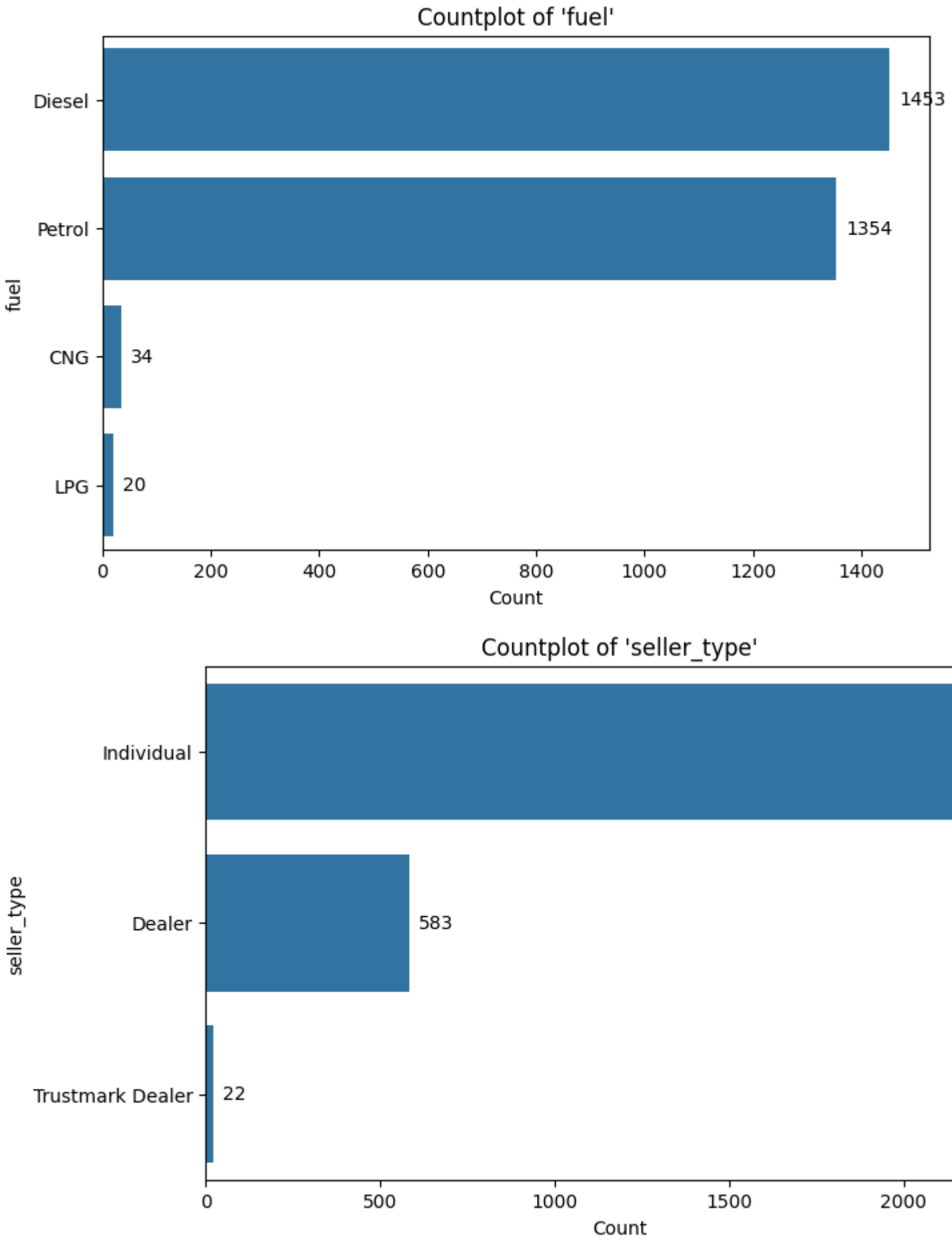
From the boxplots and descriptive statistics, selling_price shows an especially wide distribution, with a small number of listings priced far above the average. These outlying points can inflate model coefficients if they remain unaddressed. Likewise, km_driven forms a dense cluster under 100,000 km, but then extends out to as much as 800,000 km, introducing another long tail that can distort linear relationships. Log-scaling both variables before running LOF can help keep extreme values in proportion, but the fundamental skew still makes them prime targets for outlier detection.

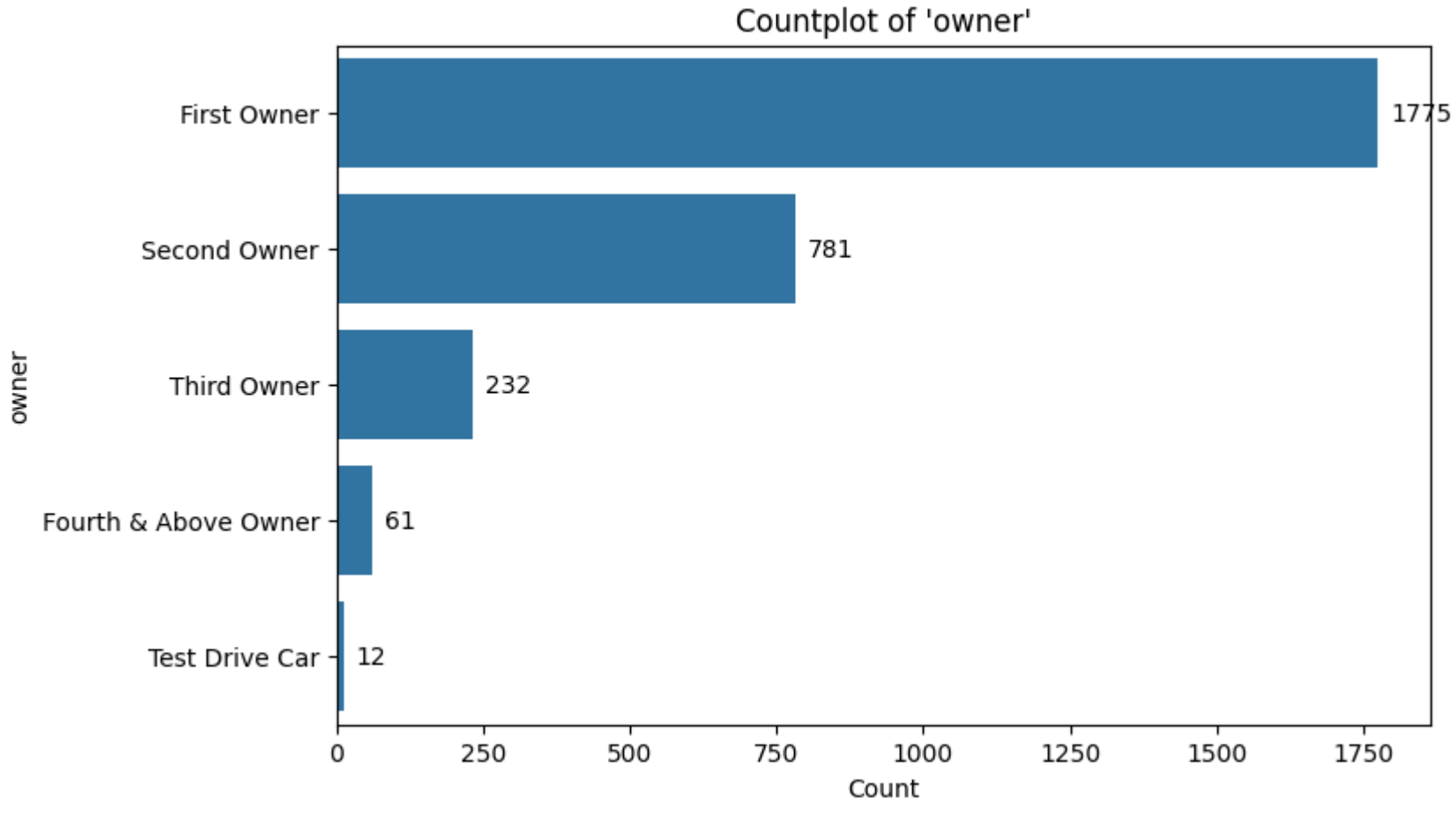
On the other hand, year has a narrower range (roughly 1992 to 2020) and does not necessarily indicate an erroneous entry when it falls at the extremes. Although very old cars might seem unusual, they could be genuine classic or vintage models. Hence, including year in LOF could lead to incorrectly flagging valid older cars as anomalies, making it less critical to treat them as potential outliers without further domain-specific knowledge.

Even though LOF doesn't operate on categorical features, we display their counts to see if there's any feature with extremely low count or not. These would be "categorical outliers".

In [19]:
```
# Plot countplots for each categorical col
for col in categorical_cols:
  plt.figure(figsize=(8, 5))
  graph = sns.countplot(y=DR[col], order=DR[col].value_counts().index)
  plt.title(f"Countplot of '{col}'")
  plt.xlabel("Count")
  plt.ylabel(col)
  for p in graph.patches:
    width = p.get_width()
```

```
  graph.annotate(f'{int(width)}', xy=(width, p.get_y() + p.get_height()/2), xytext=(5, 0), textcoords='offset poi
plt.show()
```





Handling Outliers

Numerical

Before LOF, it makes sense to temporarily log-transform and standard-scale. The log transform addresses skew and large numerical ranges by compressing heavy tails, making the data more "normal", while standard scaling then centers (mean = 0) and normalizes (std = 1) those values, ensuring different features contribute more evenly to LOF.

We set n_neighbors to 30 for a moderate neighborhood size that balances local vs. global density estimation; it's large enough to mitigate issues like duplicates or small clusters but not so large that it blurs actual local anomalies.

It's important to note that when converting data into Z-scores, the original units and meaning are lost, since values are expressed solely as standard deviations from the mean. While this standardization is beneficial for distance-based calculations like LOF, it isn't intuitive when interpreting real-world data. For this reason, we visualize LOF scores using log-transformed units, making it easier to intuitively assess whether points are truly anomalous.

We will drop the outliers discovered by LOF to ensure that erroneous or extreme values do not skew model fitting. By removing these flagged rows, we preserve the typical relationships in the dataset, reducing the risk of overfitting to noise or undermining the model's validity. We don't treat these points as missing values, as we consider them genuinely invalid or unrepresentative rather than incomplete entries that can be reliably imputed.

Categorical

From the above charts, we note that certain categorical features have outliers.

We will remove "Test Drive Car" from the "owner" column due to its extremely low frequency (12 instances). Such rare categories might skew the regression outcomes or introduce instability, as they do not represent meaningful segments. Also, note that even though the "Electric" fuel type doesn't appear in DR, it has exactly occurence in the whole dataset and should be removed. We consider it in the removing outliers section (to remove it from our one-hot encoded categorical features).

Conversely, we will keep "LPG" (20 instances), "CNG" (34 instances), and "Trustmark Dealer" (22 instances) despite their lower counts, as these still represent sufficient data points to contribute meaningfully without significantly biasing the model.

Just like the numerical outliers, we won't treat categorical outliers as missing values and impute them. We consider the categorical outlier points genuinely invalid.

In [20]:
```python
def remove_outliers_and_categorical(d_in):
    """
    - Temporarily log-transforms 'selling_price' and 'km_driven' to handle skew
    - Standard-scales those logged values
    - Uses LOF to detect numeric outliers
    - Plots outliers vs inliers in log space for visualization
    - Keeps only inliers (predictions == 1)
    - Removes one-hot encoded column for owner == 'Test Drive Car' and fuel == 'Electric'
    - Returns the cleaned dataframe
    """
    df_lof = d_in.copy()
    # Log-transform 'selling_price' and 'km_driven' to handle skew
    df_lof["log_selling_price"] = np.log1p(df_lof["selling_price"])
    df_lof["log_km_driven"] = np.log1p(df_lof["km_driven"])
    data_logged = df_lof[["log_selling_price", "log_km_driven"]]
```

```
    # Standard scale the log-transformed features to put them
    # in proportional, similar units for distance calculation
    scaler = StandardScaler()
    data_scaled = scaler.fit_transform(data_logged)

    # Prepare LOF class, fit, and predict
    lof = LocalOutlierFactor(n_neighbors=30)
    predictions = lof.fit_predict(data_scaled)

    # Count & display numeric outliers
    outliers = df_lof[predictions == -1]
    n_outliers = len(outliers)
    print(f"Number of outliers detected (bivariate, log+scaled, auto contamination): {n_outliers}")
    if n_outliers > 0:
        print("Sample outliers:")
        display(outliers.head(5))
    else:
        print("No outliers flagged with these LOF settings.")

    # Results
    plt.figure(figsize=(8, 5))
    # For interpretability, we plot the log-transformed values
    inliers_logged = data_logged[predictions == 1]
    outliers_logged = data_logged[predictions == -1]
    plt.scatter(inliers_logged["log_km_driven"], inliers_logged["log_selling_price"],
                label="Inliers (log-space)", alpha=0.5)
    plt.scatter(outliers_logged["log_km_driven"], outliers_logged["log_selling_price"],
                color="red", label="Outliers (log-space)", marker="x", s=80)
    plt.title("LOF Outliers in Log Space")
    plt.xlabel("log_km_driven")
    plt.ylabel("log_selling_price")
    plt.legend()
    plt.show()

    print("Removing numerical outliers")
    print()
    print(f"Shape before removing numerical outliers: {df_lof.shape}")
    print()
    df_lof = df_lof[predictions == 1].copy()
    print(f"Shape after removing numerical outliers: {df_lof.shape}")
    print()

    print("Removing one-hot columns for categorical outliers")
    print()
    print(f"Shape before removing one-hot columns: {df_lof.shape}")
    print()
    one_hot_cols_to_drop = ["fuel_Electric", "owner_Test Drive Car"]
    df_lof.drop(columns=one_hot_cols_to_drop, inplace=True, errors="ignore")
    print("Remaining columns:", df_lof.columns)
    print()
    print(f"Shape after removing one-hot columns: {df_lof.shape}")
    print()

    # Drop the log columns from the returned dataframe
    print("Removing log transformed features for LOF")
    print()
    df_lof.drop(columns=["log_selling_price", "log_km_driven"], inplace=True, errors="ignore")
    print("Final columns:", df_lof.columns)
    print()
    return df_lof
```

Section D: Exploring Linear Regression

The assignment instructions suggest exploring linear regression; however, linear regression itself is not directly "explorable." Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. It estimates the coefficients that minimize the difference between predicted values and actual observations. Instead of directly exploring linear regression, we will apply this method later to train a model on our completed dataset, using the default LinearRegression implementation from scikit-learn.

Section E: Feature Engineering

New Features

Two especially useful engineered features that we can create are car_age (current_year - year) and km_per_year (km_driven / car_age). car_age captures the idea that older vehicles typically depreciate more, so it often correlates strongly with selling price. km_per_year shows how heavily the car was driven each year, which goes beyond total mileage by accounting for how long the car has been on the road. Both features use simple arithmetic with existing columns and can help a linear model relate mileage, age, and price.

In [21]:
```
def add_aggregated_features(df_in, current_year=2025):
    """
    Adds two new features:
    - car_age = current_year - year
```

```
- km_per_year = km_driven/car_age
Logs target variable, selling_price
Returns a copy of the dataframe with the new columns
"""
df_out = df_in.copy()
# car_age
df_out["car_age"] = current_year - df_out["year"]
# km_per_year, avoiding division by zero
df_out["km_per_year"] = np.where(
  df_out["car_age"] == 0,
  df_out["km_driven"],
  df_out["km_driven"] / df_out["car_age"]
)
return df_out
```

Section F: Empirical Study

Creating the 4 DR variants

We systematically create four different data "variants", each reflecting a different combination of outlier removal and feature engineering. First, we define a list of columns to drop (fuel, seller_type, transmission, owner) because they are encoded elsewhere and no longer needed in these variants. Next, for each variant, we copy the full dataframe (DR) and apply the desired transformations in sequence:

- DR1 (Baseline) is created without removing outliers or adding new features. It simply drops the unneeded columns.
- DR2 (Features) is based on the baseline but adds aggregated features (like car_age or km_per_year) using a function called add_aggregated_features. It still does not remove outliers.
- DR3 (Outliers Removed) introduces outlier removal using Local Outlier Factor (LOF) and possibly additional category-based filters. It does not add the new aggregated features.
- DR4 (Outliers Removed + Features) applies both the LOF outlier removal and the creation of aggregated features. This makes it the most "augmented" variant.

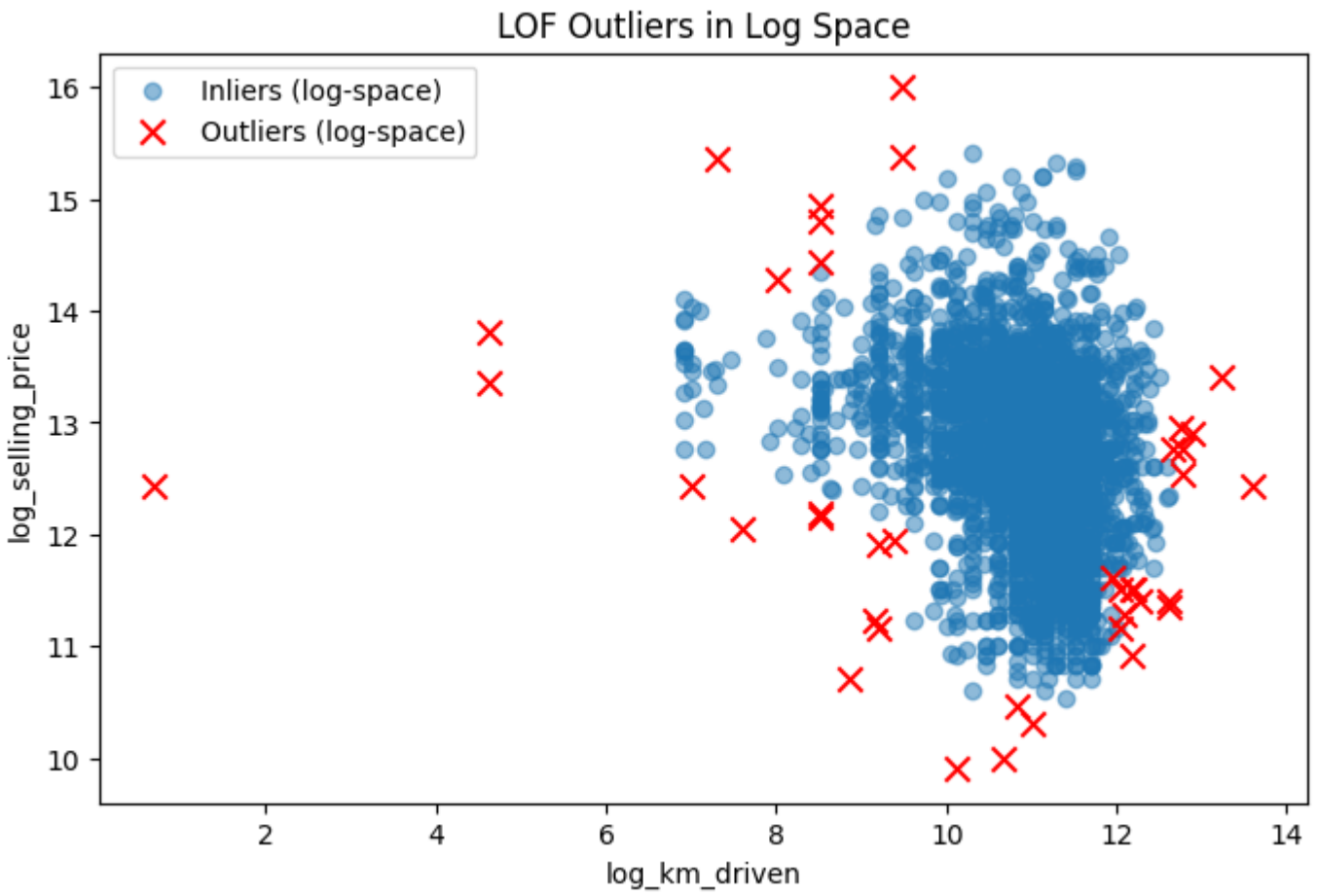
We also have a similarly processed test dataframe (DT) from which we drop the same columns for consistency.

We then print out the shapes of each dataframe so we can confirm how many rows remain after outlier removal and make sure that the transformations ran correctly.

In [22]:
```
# Define columns that we're dropping from DR and DT
# These are the columns that are extra from one-hot encoding
cols_to_drop = ["fuel", "seller_type", "transmission", "owner"]

print("Creating DR1: No outlier removal, no new features.")
DR1 = DR.copy()
DR1.drop(columns=cols_to_drop, inplace=True, errors="ignore")
print("Created DR1")
print("="*50)

print("Creating DR2: No outlier removal, but WITH new features.")
DR2 = DR.copy()
DR2.drop(columns=cols_to_drop, inplace=True, errors="ignore")
DR2 = add_aggregated_features(DR2)
print("Created DR2")
print("="*50)

print("Creating DR3: Outlier removal (LOF + categorical), no new features.")
DR3 = DR.copy()
DR3.drop(columns=cols_to_drop, inplace=True, errors="ignore")
DR3 = remove_outliers_and_categorical(DR3)
print("Created DR3")
print("="*50)

print("Creating DR4: Outlier removal + new features.")
DR4 = DR.copy()
DR4.drop(columns=cols_to_drop, inplace=True, errors="ignore")
DR4 = remove_outliers_and_categorical(DR4)
DR4 = add_aggregated_features(DR4)
print("Created DR4")
print("="*50)

# Also drop these columns from DT for consistency
DT.drop(columns=cols_to_drop, inplace=True, errors="ignore")

print("Shapes after dropping columns:")
print("DR1 shape:", DR1.shape)
print("DR2 shape:", DR2.shape)
print("DR3 shape:", DR3.shape)
print("DR4 shape:", DR4.shape)
print("DT shape:", DT.shape)
```

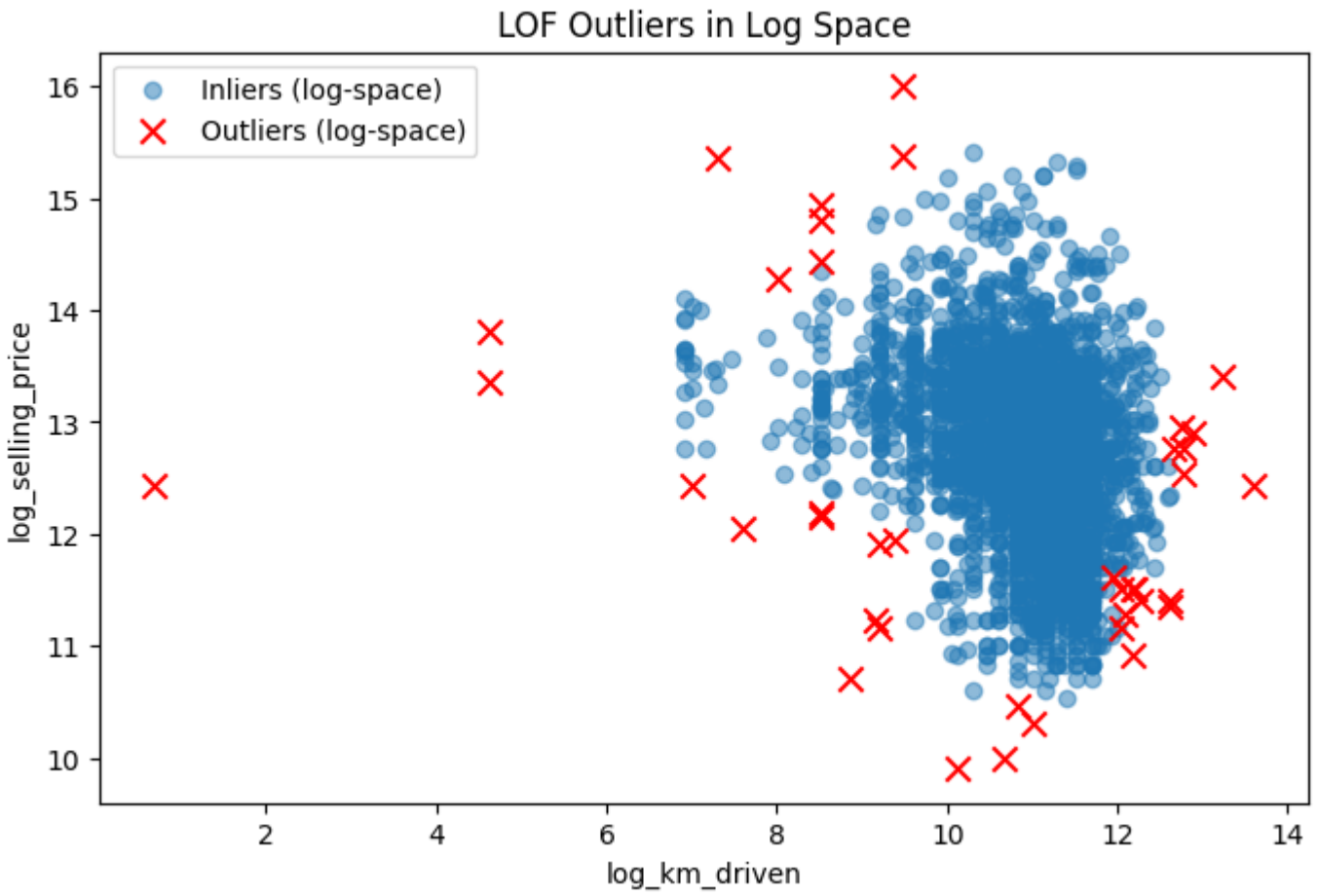
```
Creating DR1: No outlier removal, no new features.
Created DR1
================================================
Creating DR2: No outlier removal, but WITH new features.
Created DR2
================================================
Creating DR3: Outlier removal (LOF + categorical), no new features.
Number of outliers detected (bivariate, log+scaled, auto contamination): 40
Sample outliers:
```

	year	selling_price	km_driven	fuel_CNG	fuel_Diesel	fuel_Electric	fuel_LPG	fuel_Petrol	seller_type_Dealer	seller_type_In
2118	2011	75000	9528	False	False	False	False	True	True	
3898	2010	90000	300000	False	False	False	False	True	False	
3081	2007	100000	170000	False	False	False	False	True	False	
69	2010	280000	350000	False	True	False	False	False	False	
1496	2011	149000	10000	False	True	False	False	False	False	




```
Removing numerical outliers

Shape before removing numerical outliers: (2861, 20)

Shape after removing numerical outliers: (2821, 20)

Removing one-hot columns for categorical outliers

Shape before removing one-hot columns: (2821, 20)

Remaining columns: Index(['year', 'selling_price', 'km_driven', 'fuel_CNG', 'fuel_Diesel',
       'fuel_LPG', 'fuel_Petrol', 'seller_type_Dealer',
       'seller_type_Individual', 'seller_type_Trustmark Dealer',
       'transmission_Automatic', 'transmission_Manual', 'owner_First Owner',
       'owner_Fourth & Above Owner', 'owner_Second Owner', 'owner_Third Owner',
       'log_selling_price', 'log_km_driven'],
      dtype='object')

Shape after removing one-hot columns: (2821, 18)

Removing log transformed features for LOF

Final columns: Index(['year', 'selling_price', 'km_driven', 'fuel_CNG', 'fuel_Diesel',
       'fuel_LPG', 'fuel_Petrol', 'seller_type_Dealer',
       'seller_type_Individual', 'seller_type_Trustmark Dealer',
       'transmission_Automatic', 'transmission_Manual', 'owner_First Owner',
       'owner_Fourth & Above Owner', 'owner_Second Owner',
       'owner_Third Owner'],
      dtype='object')

Created DR3
================================================
Creating DR4: Outlier removal + new features.
Number of outliers detected (bivariate, log+scaled, auto contamination): 40
Sample outliers:
```

	year	selling_price	km_driven	fuel_CNG	fuel_Diesel	fuel_Electric	fuel_LPG	fuel_Petrol	seller_type_Dealer	seller_type_In
2118	2011	75000	9528	False	False	False	False	True	True	
3898	2010	90000	300000	False	False	False	False	True	False	
3081	2007	100000	170000	False	False	False	False	True	False	
69	2010	280000	350000	False	True	False	False	False	False	
1496	2011	149000	10000	False	True	False	False	False	False	




```
Removing numerical outliers

Shape before removing numerical outliers: (2861, 20)

Shape after removing numerical outliers: (2821, 20)

Removing one-hot columns for categorical outliers

Shape before removing one-hot columns: (2821, 20)

Remaining columns: Index(['year', 'selling_price', 'km_driven', 'fuel_CNG', 'fuel_Diesel',
       'fuel_LPG', 'fuel_Petrol', 'seller_type_Dealer',
       'seller_type_Individual', 'seller_type_Trustmark Dealer',
       'transmission_Automatic', 'transmission_Manual', 'owner_First Owner',
       'owner_Fourth & Above Owner', 'owner_Second Owner', 'owner_Third Owner',
       'log_selling_price', 'log_km_driven'],
      dtype='object')

Shape after removing one-hot columns: (2821, 18)

Removing log transformed features for LOF

Final columns: Index(['year', 'selling_price', 'km_driven', 'fuel_CNG', 'fuel_Diesel',
       'fuel_LPG', 'fuel_Petrol', 'seller_type_Dealer',
       'seller_type_Individual', 'seller_type_Trustmark Dealer',
       'transmission_Automatic', 'transmission_Manual', 'owner_First Owner',
       'owner_Fourth & Above Owner', 'owner_Second Owner',
       'owner_Third Owner'],
      dtype='object')

Created DR4
==================================================
Shapes after dropping columns:
DR1 shape: (2861, 18)
DR2 shape: (2861, 20)
DR3 shape: (2821, 16)
DR4 shape: (2821, 18)
DT shape: (716, 18)
```

Running 4-Fold Cross Validation

In [23]:
```python
def cross_validate_variant(df_variant, cv_splits=4, random_state=12):
  """
  - Splits df_variant into X (features) and y (target)
  - Runs 4-fold cross validation
  - Returns average MSE and average R^2
```

```
"""
# Separate features and target
X = df_variant.drop(columns=["selling_price"])
y = df_variant["selling_price"]

# Setup up the Linear regression model
model = LinearRegression()

# Set random state to ensure reproducibility
kf = KFold(n_splits=cv_splits, shuffle=True, random_state=random_state)

# Cross-validation for negative MSE (but we convert to positive)
mse_scores = cross_val_score(
  model,
  X, y,
  cv=kf,
  scoring="neg_mean_squared_error"
)

# Cross-validation for R^2
r2_scores = cross_val_score(
  model,
  X, y,
  cv=kf,
  scoring="r2"
)

# Calculate average metrics
  mean_mse = -mse_scores.mean()
  mean_r2  = r2_scores.mean()
  return mean_mse, mean_r2

# Evaluate each variant
variants = [
  ("DR1 (Baseline)", DR1),
  ("DR2 (Features)", DR2),
  ("DR3 (Outliers Removed)", DR3),
  ("DR4 (Outliers Removed+Features)", DR4)
]
results_dict = {}
for name, dfv in variants:
  mse_val, r2_val = cross_validate_variant(dfv)
  results_dict[name] = {"mean_MSE": mse_val, "mean_R2": r2_val}
```

Choosing The Best System

In [24]:
```
# Convert results dict into a dataframe for clear comparison
results_df = pd.DataFrame.from_dict(results_dict, orient='index')
results_df = results_df.rename(columns={
    "mean_MSE": "Mean MSE",
    "mean_R2": "Mean R^2"
})
display(results_df.round(3))
```

	Mean MSE	Mean R^2
DR1 (Baseline)	1.355402e+11	0.452
DR2 (Features)	1.355462e+11	0.452
DR3 (Outliers Removed)	1.068068e+11	0.472
DR4 (Outliers Removed+Features)	1.067669e+11	0.472

The table compares four different data/model setups (DR1, DR2, DR3, and DR4), showing their mean squared error (MSE) and coefficient of determination (R^2) from cross-validation. MSE measures the average squared distance between predicted and actual values (lower is better), and R^2 indicates how much of the target's variance is explained (higher is better).

DR1 is the baseline model (no outlier removal, no extra features). Its mean MSE of about 1.3554×10^11 and R^2 of 0.452 set a reference point. DR2, which adds features such as car_age and km_per_year but does not remove outliers, has nearly the same performance (1.35546×10^11 MSE and 0.452 R^2). This suggests that simply introducing these new features is not enough to offset the effect of outliers.

DR3 removes outliers while keeping the original feature set. Its mean MSE drops to about 1.06807×10^11 and the R^2 rises to 0.472, demonstrating that excluding a small set of extreme observations can substantially lower average error and increase explained variance. DR4 combines both outlier removal and the added features, yielding a mean MSE of around 1.06767×10^11 and maintaining the R^2 of 0.472. Although the jump from DR3 to DR4 is modest, the slightly lower MSE indicates that the new features still contribute a small but positive improvement once outliers are removed.

By integrating outlier removal (to diminish the undue influence of extreme values) and feature engineering (to capture key aspects like vehicle age and usage), DR4 achieves both lower error and higher explanatory power than the baseline or individual interventions alone. Even if the differences from DR3 are small, DR4 provides the best overall performance in cross-validation, making it the natural choice for final training and testing.

Testing on DT

Below is the final step after cross validating different data variants (DR1, DR2, DR3, DR4). Since DR4 showed the best average cross-validation performance, we now train a new linear regression model on the entire DR4 dataset to give it full exposure to all training examples (rather than the partial subsets used in cross validation). Afterward, we prepare the test set using the same data transformations, but without removing outliers as that only occurred during training, and make predictions on this truly unseen data. By comparing the test MSE and R^2 to the cross-validation metrics, we can assess whether the performance gains observed in training carry over to new data.

In [25]:
```
# Prepare DR4 for final training
X_train = DR4.drop(columns=["selling_price"])
y_train = DR4["selling_price"]

# Train a final linear regression model on DR4
final_model = LinearRegression()
final_model.fit(X_train, y_train)

# Prepare the test set the same way we prepared DR4 (minus outlier removal)
one_hot_cols_to_drop = ["fuel_Electric", "owner_Test Drive Car"]
DT.drop(columns=one_hot_cols_to_drop, inplace=True, errors="ignore")

# Add aggreated features (including log scaling of target variable)
DT_test = add_aggregated_features(DT)

# Split the test set into features and target
X_test = DT_test.drop(columns=["selling_price"])
y_test = DT_test["selling_price"]

# Predict on the test set
y_pred = final_model.predict(X_test)

# Calculate test set metrics
test_mse = mean_squared_error(y_test, y_pred)
test_r2  = r2_score(y_test, y_pred)
print(f"Test MSE: {test_mse:.4f}")
print(f"Test R^2:  {test_r2:.4f}")
```

```
Test MSE: 196729802656.6460
Test R^2:  0.3893
```

The final model, trained on DR4 and evaluated on the unseen test set, produces a mean squared error (MSE) of approximately 1.96×10^11 and an R^2 of about 0.389. These results indicate that the model explains roughly 38.9% of the variance in the test data's selling price, and it experiences a somewhat larger prediction error than suggested by the cross-validation phase.

Section G: Results Analysis

a) Analysis of Obtained Results and Improvement

After experimenting with various data transformations, including outlier removal and feature engineering, DR4 tended to give the strongest cross-validation results, with both its MSE and R^2 looking relatively robust. However, when DR4 was evaluated on the test set, its MSE increased to around 1.96×10^11 and its R^2 fell to roughly 0.389. This noticeable drop in R^2 points to a generalization gap between performance measured via cross-validation and performance measured on truly unseen test data.

In essence, while the additional features and outlier removal steps improved DR4's fit on the training folds, the specific relationships the model learned did not carry over as well to the test set. This discrepancy often arises if the test data have slightly different distributions than those in the training folds, or if the model has partially overfit the training data, even if outliers were removed. The gap indicates

that some of the patterns that boosted cross-validation scores are not as relevant or consistent in the test portion, highlighting the importance of validating transformations and feature engineering approaches specifically against data that the model has not encountered.

b) Impact of Outlier Detection and Feature Aggregation

Removing outliers lowered the average cross-validation MSE from about 1.3554×10^11 (in the baseline, DR1) to around 1.0681×10^11 (in DR3) and raised the mean R^2 from 0.452 to 0.472. This shows that a handful of extreme points had been distorting the model's parameter estimates and artificially inflating errors. Dropping those outliers made the regression fitter more robust and less influenced by high-leverage observations, which improved performance on the training folds.

Simply adding aggregated features, such as car_age and km_per_year (DR2), did not substantially change the MSE or R^2 over the baseline. However, when outlier removal was combined with these features in DR4, the mean MSE nudged down further from 1.0681×10^11 to about 1.0677×10^11, while the higher R^2 of 0.472 was maintained. This modest improvement suggests that most of the boost stemmed from removing outliers, with the new features providing an additional small lift in modeling the relationships tied to mileage and depreciation.

Despite these stronger cross-validation metrics, the final test MSE was around 1.97×10^9, and the test R^2 was about 0.389. This drop indicates that the model's test-set predictions did not match the expectations set by cross-validation. One possible reason is that any outliers present in the test set were not removed, so the final model encountered unusual points in new data that it had not been trained to handle. Another possibility is that the patterns uncovered during training (for instance, in how car_age and km_per_year affect selling_price) differ slightly in the unseen test distribution. These gaps remind us that even careful data transformations and outlier handling cannot fully guarantee generalization to all real-world scenarios.

c) Comparing Test-Set Results to Cross-Validation Results

Cross-validation often appears more optimistic than the final test results because it still relies on subsets drawn from the same overall training distribution. Even if outliers are removed or the dataset is shuffled, the cross-validation folds typically resemble one another closely. Once the model is confronted with truly unseen test data, patterns in vehicle age, mileage, or pricing may differ, uncovering weaknesses the model did not display in cross-validation.

In the case of DR4, its R^2 of about 0.472 in cross-validation dropped to around 0.389 on the test set, signaling that certain assumptions or relationships the model learned during training—like how car_age or km_per_year relate to price—did not hold up as strongly in the unseen data. This can happen if the test set includes more unusually priced cars, higher-mileage vehicles, or simply follows a different distribution of features. Another factor is residual overfitting: even after outlier removal, the model may still learn nuances that do not generalize well. Together, these issues possibly explain why our model that outperforms others in cross-validation might still struggle once exposed to new data outside the controlled conditions of training folds.

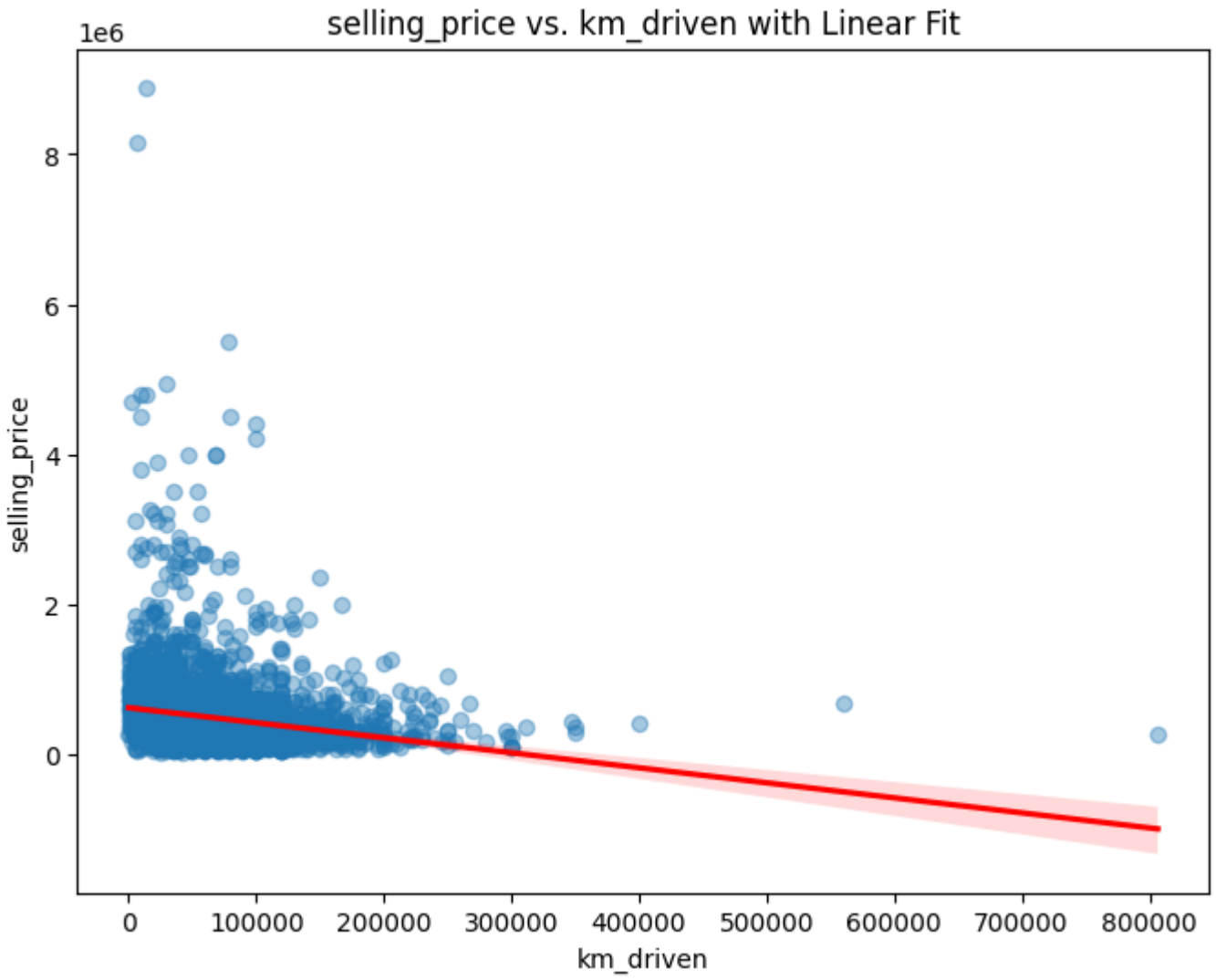
Additional Considerations

Even after performing outlier removal and adding helpful features, a few remaining columns can still exert a large influence on the linear model if their values significantly deviate from the rest of the dataset. Whenever a feature has a very wide numeric range or contains categories with only a handful of samples, ordinary least squares regression may assign disproportionate weight to these points. In practical terms, this can lead the model to "chase" extreme values that do not follow typical patterns, distorting the fitted slope and intercept.

A common approach to addressing this issue is to further transform or remove problematic columns. For instance, log-scaling a feature like km_driven that can reach up to 800,000 km often yields a more normal distribution of values. By logging the mileage, points near the extreme high end are pulled closer in scale to the rest of the data, reducing their leverage and helping the model fit a gentler, more accurate slope. Regularized linear models such as Ridge or Lasso can also help by shrinking large coefficients and mitigating the impact of outliers. If these transformations are insufficient, switching to more robust regressions or entirely removing columns that strongly violate linearity may further improve generalization.

In our model, the km_driven column exemplifies this problem. Some vehicles have extremely high mileages that are poorly explained by a standard linear slope; on a scatter plot, these points clearly stand apart, pulling the regression line away from the denser cluster of points. Transforming km_driven (for example, by taking its logarithm) or dropping the most extreme mileage observations (if they are not representative) could yield a better fit in training and a more stable model.

In [26]:
```
plt.figure(figsize=(8, 6))
sns.regplot(
    x="km_driven",
    y="selling_price",
    data=df,
    scatter_kws={"alpha": 0.4},
    line_kws={"color": "red"}
)
plt.title("selling_price vs. km_driven with Linear Fit")
plt.xlabel("km_driven")
plt.ylabel("selling_price")
plt.show()
```



Conclusion

In this project, we cleaned and transformed a dataset of car listings, focusing on outlier detection, feature engineering, and model training. Data validation confirmed that the dataset's columns matched expected types and ranges, and exact duplicates were removed to avoid redundancy. After one-hot encoding the categorical features, LOF was used to detect extreme numeric outliers, and rare categories were excluded to reduce data sparsity. New features were introduced to capture age and yearly mileage, and transformations were explored to address heavy skew in key columns.

Multiple dataset variants were created and evaluated using linear regression. The approach that combined outlier removal and feature engineering showed stronger performance during cross-validation, producing lower error and a higher measure of explanatory power than the baseline or simpler alternatives. However, the final evaluation on unseen data revealed a decline in model accuracy, indicating that certain patterns learned during training did not generalize fully. Despite this gap, the improvements gained from targeted outlier handling and feature design clearly highlight the benefits of careful data preprocessing for linear regression.

Further refinement could involve transforming or removing especially high-leverage features, such as extreme mileage, or adopting more robust or regularized regression methods to mitigate overfitting. Monitoring residuals and validating model assumptions would also help ensure that linear relationships remain meaningful in practice.

References

[1] https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho

[2] https://www.algosome.com/articles/dummy-variable-trap-regression.html