Used-car price valuation with linear regression

Setup

This notebook builds and evaluates a linear regression model that predicts the selling price of used cars in the Indian resale market from their recorded attributes.

```
# Library imports
import os
import re

import numpy as np
import pandas as pd
from scipy import stats

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, cross_val_score,
KFold
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error, r2_score

# Render every figure inline.
%matplotlib inline

# Named constants for the analysis. The modelling logic uses these
throughout.
# A few purely cosmetic plotting values, for example axis label font
sizes, are
# left inline in the figure cells where naming them would add noise.
RANDOM_SEED = 42        # single seed passed to every split and model
for reproducibility
TEST_SIZE = 0.2         # eighty twenty train test split
CV_FOLDS = 5            # number of folds for k-fold cross validation
POLY_DEGREE = 2         # degree of the polynomial features added to
max_power
IMPUTE_STRATEGY = "median"  # numeric missing values are filled with
the column median
KGM_TO_NM = 9.80665     # conversion factor from kilogram-metre to
newton-metre for torque
RIDGE_ALPHA = 1.0       # regularisation strength for the Ridge
comparison
LASSO_ALPHA = 0.001     # regularisation strength for the Lasso
comparison
```

```
DATA_PATH = "data/Car details v3.csv"   # local Kaggle download, see
Q2 reference
KAGGLE_DATASET = "nehalbirla/vehicle-dataset-from-cardekho"  # Kaggle
link for the download fallback
WORKING_DIR = "data/working"            # folder for the intermediate
CSV partitions

# Plotting defaults, named so the repeated figure settings stay
consistent.
FIGSIZE_WIDE = (12, 4)      # two-panel figures
FIGSIZE_SINGLE = (7, 5)     # single scatter or histogram
FIGSIZE_HEATMAP = (7, 6)    # correlation heatmap
HIST_BINS = 50              # bins for the price histograms
POINT_SIZE = 8              # marker size for the scatter plots
POINT_ALPHA = 0.3           # marker transparency for the dense
scatter plots
```

1. Domain-specific area and objectives

The domain for this project is used-car price valuation in the Indian resale market. When a private owner, a dealership or an online platform lists a second-hand vehicle, a price has to be set, and that price reflects measurable attributes such as engine power, age, distance driven, fuel type and transmission. Because several of these attributes move with price in a roughly straight-line fashion, the problem is well suited to linear regression. Engine power in particular tends to rise with price in an approximately linear way once price is viewed on a logarithmic scale, which gives the model a clear signal to learn from (Gegic et al., 2019).

The objective is to predict the selling price of a used car from its recorded attributes, and to do so with a model that is transparent enough to explain which features drive the valuation. Linear regression suits this aim for two reasons. It produces an interpretable coefficient for every feature, so a seller can see how much an extra unit of power or an extra year of age is worth in rupees. It also trains quickly on the eight thousand records available here, which keeps experimentation cheap.

Several groups benefit from a working model. Private buyers and sellers gain a defensible reference price and are less exposed to mispricing. Dealerships can value stock consistently and at scale. Online valuation services can offer instant estimates that build user trust. The wider contribution is pricing transparency in a market where information is often uneven between buyer and seller (Pal et al., 2019). A model that quantifies the effect of each attribute helps to narrow that gap, supports fairer negotiation, and gives a baseline against which unusual listings can be flagged. Prior studies report that even simple regression models capture a large share of the variation in used-car prices, which motivates the approach taken here (Monburinon et al., 2018).

References for this section

Gegic, E., Isakovic, B., Keco, D., Masetic, Z. and Kevric, J. (2019) 'Car price prediction using machine learning techniques', *TEM Journal*, 8(1), pp. 113-118.

Monburinon, N., Chertchom, P., Kaewkiriya, T., Rungpheung, S., Buya, S. and Boonpou, P. (2018) 'Prediction of prices for used car by using regression models', in *2018 5th International Conference on Business and Industrial Research (ICBIR)*. IEEE, pp. 115-119.

Pal, N., Arora, P., Kohli, P., Sundararaman, D. and Palakurthy, S.S. (2019) 'How much is my car worth? A methodology for predicting used cars prices using random forest', in *Advances in Information and Communication Networks*. Cham: Springer, pp. 413-422.

2. Dataset description

The dataset is the Vehicle dataset published on Kaggle by Birla (2020). It collects used-car listings from CarDekho, a large Indian online car marketplace, and the records were acquired by scraping public listing pages. The file used here is Car details v3.csv, which contains 8,128 rows and 13 columns. Each row is one listing.

The columns mix several data types. The target, selling_price, is a continuous integer in Indian rupees. year, km_driven and seats are numeric. fuel, seller_type, transmission and owner are categorical text fields with a small number of levels. name is free text that fuses make, model and variant. Four columns arrive as text with the unit fused to the number: mileage as '23.4 kmpl', engine as '1248 CC', max_power as '74 bhp', and torque as a composite string such as '190Nm@ 2000rpm'. Those four, together with seats, also contain genuine missing values, roughly 220 per column.

The dataset fits a linear regression task well. The target is continuous, which is the setting linear regression is designed for. There is a clear linear signal, since max_power and engine size both rise with price, and the relationship for power is close to linear once price is log transformed. The sample of more than eight thousand records is large enough to hold out a fifth for testing and still train on a substantial set, which supports reliable estimation of the coefficients. The presence of units, composite fields and missing values means the data is not in First Normal Form as supplied, so it also exercises the preprocessing steps required by this coursework.

Full reference

Birla, N. (2020) *Vehicle dataset*. Kaggle. Available at: https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho (Accessed: 30 June 2026).

3. Data preparation

The raw file is a single CSV that is not in First Normal Form, because four columns store a number fused to a unit and torque packs two measurements into one string. This section loads the data, demonstrates multi-file ingestion, strips the units so every column is atomic, parses torque, fixes missing values by median imputation, and confirms the result is in First Normal Form.

```
# The dataset is read from the local path defined above during my
development. If the file is missing,
# for example when the marker is running, it is fetched once from
Kaggle with kagglehub
# and cached at DATA_PATH so the notebook stays reproducible without a
```

```
manual
# download. Kaggle requires authentication, so kagglehub reads the
credentials in
# ~/.kaggle/kaggle.json or the KAGGLE_USERNAME / KAGGLE_KEY
environment variables
# when a download is actually needed.

if not os.path.exists(DATA_PATH):
    import shutil
    import kagglehub

    os.makedirs(os.path.dirname(DATA_PATH), exist_ok=True)
    download_dir = kagglehub.dataset_download(KAGGLE_DATASET)
    source_csv = os.path.join(download_dir,
os.path.basename(DATA_PATH))
    shutil.copy(source_csv, DATA_PATH)
    print("Downloaded dataset from Kaggle to", DATA_PATH)

cars_raw = pd.read_csv(DATA_PATH)
print("Loaded shape:", cars_raw.shape)
cars_raw.head()
```

```
Loaded shape: (8128, 13)
```

	name	year	selling_price	km_driven	fuel	seller_type	transmission	owner	mileage	engine	max_power
0	Maruti Swift Dzire VDI	2014	450000	145500	Diesel	Individual	Manual	First Owner	23.4 kmpl	1248 CC	74 bhp
1	Skoda Rapid 1.5 TDI Ambition	2014	370000	120000	Diesel	Individual	Manual	Second Owner	21.14 kmpl	1498 CC	103.52 bhp
2	Honda City 2017-2020 EXi	2006	158000	140000	Petrol	Individual	Manual	Third Owner	17.7 kmpl	1497 CC	78 bhp
3	Hyundai i20 Sportz Diesel	2010	225000	127000	Diesel	Individual	Manual	First Owner	23.0 kmpl	1396 CC	90 bhp
4	Maruti Swift VXI BSIII	2007	130000	120000	Petrol	Individual	Manual	First Owner	16.1 kmpl	1298 CC	88.2 bhp

```
                      torque  seats
0             190Nm@ 2000rpm    5.0
1       250Nm@ 1500-2500rpm    5.0
2     12.7@ 2,700(kgm@ rpm)    5.0
3  22.4 kgm at 1750-2750rpm    5.0
4     11.5@ 4,500(kgm@ rpm)    5.0
```

Demonstrating multi-file ingestion

Real projects often receive data split across several files that must be read and combined. This dataset ships as a single CSV, so to exercise that skill the loaded frame is partitioned by fuel type, each partition is written to its own CSV in a working folder, and the files are then read back and concatenated to reconstruct the full frame. This is a controlled demonstration of the common pattern of ingesting and merging several files, and the row count is checked to confirm nothing is lost in the round trip.

```
# Create the working folder for the intermediate partitions.
os.makedirs(WORKING_DIR, exist_ok=True)

# Add a stable row identifier so the original order can be restored
once the
# partitions are read back, since grouping by fuel changes the row
order.
partition_source = cars_raw.copy()
partition_source["row_id"] = np.arange(len(partition_source))

# Write one CSV per fuel type to demonstrate data arriving in several
files.
partition_paths = []
for fuel_value, group in partition_source.groupby("fuel"):
    path = os.path.join(WORKING_DIR, f"cars_{fuel_value.lower()}.csv")
    group.to_csv(path, index=False)
    partition_paths.append(path)

print("Wrote", len(partition_paths), "partition files:")
for path in partition_paths:
    print(" ", path)
```

```
Wrote 4 partition files:
  data/working/cars_cng.csv
  data/working/cars_diesel.csv
  data/working/cars_lpg.csv
  data/working/cars_petrol.csv
```

```
# Read every partition back and concatenate to reconstruct the full
frame.
# This is the multi-file ingestion step. The frame is then sorted back
into
# the original order using row_id, and the helper column is dropped.
```

```
frames = [pd.read_csv(path) for path in partition_paths]
cars = (pd.concat(frames, ignore_index=True)
        .sort_values("row_id")
        .drop(columns="row_id")
        .reset_index(drop=True))

# Confirm the reconstructed frame matches the original row count.
print("Reconstructed shape:", cars.shape)
print("Row count matches original:", cars.shape[0] ==
cars_raw.shape[0])
```

```
Reconstructed shape: (8128, 13)
Row count matches original: True
```

Stripping units and parsing composite fields

mileage, engine and max_power each hold a number followed by a unit, so the numeric part is extracted and converted to float. torque is a composite string that holds a value and an engine speed, and the value is recorded either in newton-metres or in kilogram-metres, so the leading number is extracted and the kilogram-metre cases are converted to newton-metres. A small number of max_power entries are blank or zero, which are treated as missing. The cell below inspects these imperfections before any parsing, so the cleaning choices are grounded in what the columns actually contain.

```
# Inspect the unit tokens and irregular values before parsing
anything.
print("Unit tokens in mileage:",
      sorted(set(re.sub(r"[0-9. ]", "", str(v)) for v in
cars["mileage"].dropna().unique())))
print("Unit tokens in engine: ",
      sorted(set(re.sub(r"[0-9. ]", "", str(v)) for v in
cars["engine"].dropna().unique())))

# Find max_power entries that are not a plain number followed by bhp.
odd_power = sorted(set(str(v) for v in
cars["max_power"].dropna().unique()
                       if not re.match(r"^\s*[0-9.]+\s*bhp\s*$",
str(v))))
print("Non-standard max_power values:", odd_power)

# torque mixes two units, so count how many rows use each.
torque_text = cars["torque"].dropna().astype(str)
print("torque rows quoted in kgm:", sum("kgm" in t.lower() for t in
torque_text),
      "of", len(torque_text))
```

```
Unit tokens in mileage: ['km/kg', 'kmpl']
Unit tokens in engine:  ['CC']
```

```
Non-standard max_power values: [' bhp', '0']
torque rows quoted in kgm: 505 of 7906
```

The check confirms the imperfections. mileage uses two units, kmpl for liquid fuels and km/kg for gas, while engine is uniform in CC. max_power holds a blank entry and a zero, which carry no usable value and are set to missing. torque is quoted in newton-metres for most rows and in kilogram-metres for several hundred, which is why the parser converts the kilogram-metre cases so the whole column is on one scale.

```
# Extract the first number from a string such as '23.4 kmpl' or '1248
CC'.
# Returns NaN when no number is present, for example a blank max_power
entry.
def parse_leading_number(value):
    if pd.isna(value):
        return np.nan
    match = re.search(r"[-+]?\d*\.?\d+", str(value).replace(",", ""))
    return float(match.group()) if match else np.nan


# Extract the numeric torque and express it in newton-metres.
# Entries quoted in kilogram-metres are converted using KGM_TO_NM.
def parse_torque_nm(value):
    number = parse_leading_number(value)
    if pd.isna(number):
        return np.nan
    if "kgm" in str(value).lower():
        return number * KGM_TO_NM
    return number


# Apply the unit-stripping parser to the three fused numeric columns.
for column in ["mileage", "engine", "max_power"]:
    cars[column] = cars[column].map(parse_leading_number)

# Parse torque into an atomic newton-metre column and drop the
composite original.
cars["torque_nm"] = cars["torque"].map(parse_torque_nm)
cars = cars.drop(columns=["torque"])

# A handful of max_power values are zero, which is not physically
meaningful,
# so they are set to missing and imputed alongside the genuine gaps
below.
cars.loc[cars["max_power"] <= 0, "max_power"] = np.nan

# seats already reads as a float, the conversion makes the intent
explicit.
cars["seats"] = pd.to_numeric(cars["seats"], errors="coerce")
```

```
cars[["mileage", "engine", "max_power", "torque_nm", "seats"]].head()
```

	mileage	engine	max_power	torque_nm	seats
0	23.40	1248.0	74.00	190.000000	5.0
1	21.14	1498.0	103.52	250.000000	5.0
2	17.70	1497.0	78.00	124.544455	5.0
3	23.00	1396.0	90.00	219.668960	5.0
4	16.10	1298.0	88.20	112.776475	5.0

Fixing missing values

After parsing, the numeric columns contain missing values, partly from the original gaps and partly from the blank power entries. The brief asks for these to be fixed. Each numeric column is filled with its own median, which is robust to the strong right skew of these series and avoids dropping rows.

```
# The numeric feature columns that feed the model.
NUMERIC_FEATURES = ["year", "km_driven", "engine", "max_power",
                    "mileage", "seats", "torque_nm"]

# Count missing values before imputation.
print("Missing values before imputation:")
print(cars[NUMERIC_FEATURES].isna().sum())

# Impute each numeric column with its median, as set by
IMPUTE_STRATEGY.
for column in NUMERIC_FEATURES:
    if IMPUTE_STRATEGY == "median":
        cars[column] = cars[column].fillna(cars[column].median())

print("\nMissing values after imputation:")
print(cars[NUMERIC_FEATURES].isna().sum())
```

```
Missing values before imputation:
year           0
km_driven      0
engine       221
max_power    222
mileage      221
seats        221
torque_nm    222
dtype: int64

Missing values after imputation:
year         0
km_driven    0
engine       0
max_power    0
mileage      0
```

```
seats         0
torque_nm     0
dtype: int64
```

Dropping name and confirming First Normal Form

name fuses make, model and variant into free text with thousands of distinct values, so it is dropped from the modelling frame. Encoding it directly would create thousands of sparse columns and risk overfitting, and the make and model information is already reflected indirectly through power, engine size and fuel type. After parsing and imputation every remaining column holds a single atomic, unit-free value with no repeating groups, so the frame is now in First Normal Form.

```
# Drop the free-text name column for the reasons given above.
cars = cars.drop(columns=["name"])

# Confirm atomic, unit-free columns with no remaining missing values.
print("Columns now in the frame:")
print(list(cars.columns))
print("\nData types:")
print(cars.dtypes)
print("\nTotal missing values:", int(cars.isna().sum().sum()))
```

```
Columns now in the frame:
['year', 'selling_price', 'km_driven', 'fuel', 'seller_type',
'transmission', 'owner', 'mileage', 'engine', 'max_power', 'seats',
'torque_nm']

Data types:
year              int64
selling_price     int64
km_driven         int64
fuel                str
seller_type         str
transmission        str
owner               str
mileage         float64
engine          float64
max_power       float64
seats           float64
torque_nm       float64
dtype: object

Total missing values: 0
```

4. Statistical analysis

This section summarises the key numeric series. The focus is the target, selling_price, with measures of central tendency, measures of spread and measures of distribution shape, computed with numpy, pandas and scipy.

```
# Helper that returns a tidy dictionary of summary statistics for one
series.
# It combines central tendency, spread and distribution shape.
def summarise(series):
    return {
        "mean": np.mean(series),
        "median": np.median(series),
        "mode": stats.mode(series, keepdims=False).mode,
        "std": np.std(series, ddof=1),
        "variance": np.var(series, ddof=1),
        "iqr": stats.iqr(series),
        "range": np.max(series) - np.min(series),
        "skewness": stats.skew(series),
        "kurtosis": stats.kurtosis(series),
    }


# Summarise the main numeric series and present them as a table.
KEY_SERIES = ["selling_price", "max_power", "engine", "year",
"km_driven"]
summary_table = pd.DataFrame({name: summarise(cars[name]) for name in
KEY_SERIES})
summary_table.round(2)
```

	selling_price	max_power	engine	year	km_driven
mean	6.382718e+05	91.33	1452.90	2013.80	6.981951e+04
median	4.500000e+05	82.00	1248.00	2015.00	6.000000e+04
mode	3.000000e+05	74.00	1248.00	2017.00	1.200000e+05
std	8.062534e+05	35.29	498.20	4.04	5.655055e+04
variance	6.500446e+11	1245.40	248199.97	16.36	3.197965e+09
iqr	4.200010e+05	32.25	385.00	6.00	6.300000e+04
range	9.970001e+06	367.20	2980.00	37.00	2.360456e+06
skewness	4.190000e+00	1.68	1.18	-1.07	1.117000e+01
kurtosis	2.107000e+01	4.01	0.85	1.71	3.838600e+02

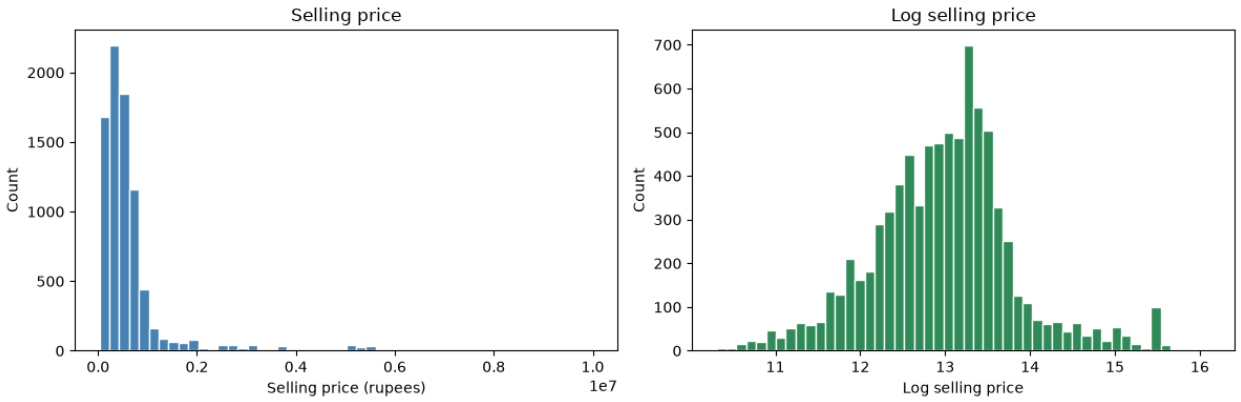
The most informative measure here is the skewness of selling_price, which is about 4.2. A skew of zero would indicate a symmetric distribution, so a value this large signals a strong right tail: most cars are inexpensive while a small number of premium vehicles sell for very high prices. The gap between the mean of roughly 638,000 rupees and the median of 450,000 confirms the same pull from the tail, because the mean is dragged upward by the expensive minority while the median is not. The high positive kurtosis adds that these extreme values are more frequent than a normal distribution would predict. This shape matters for modelling, because ordinary

linear regression assumes roughly symmetric errors, which motivates the log transform applied later.

5. Visualisation

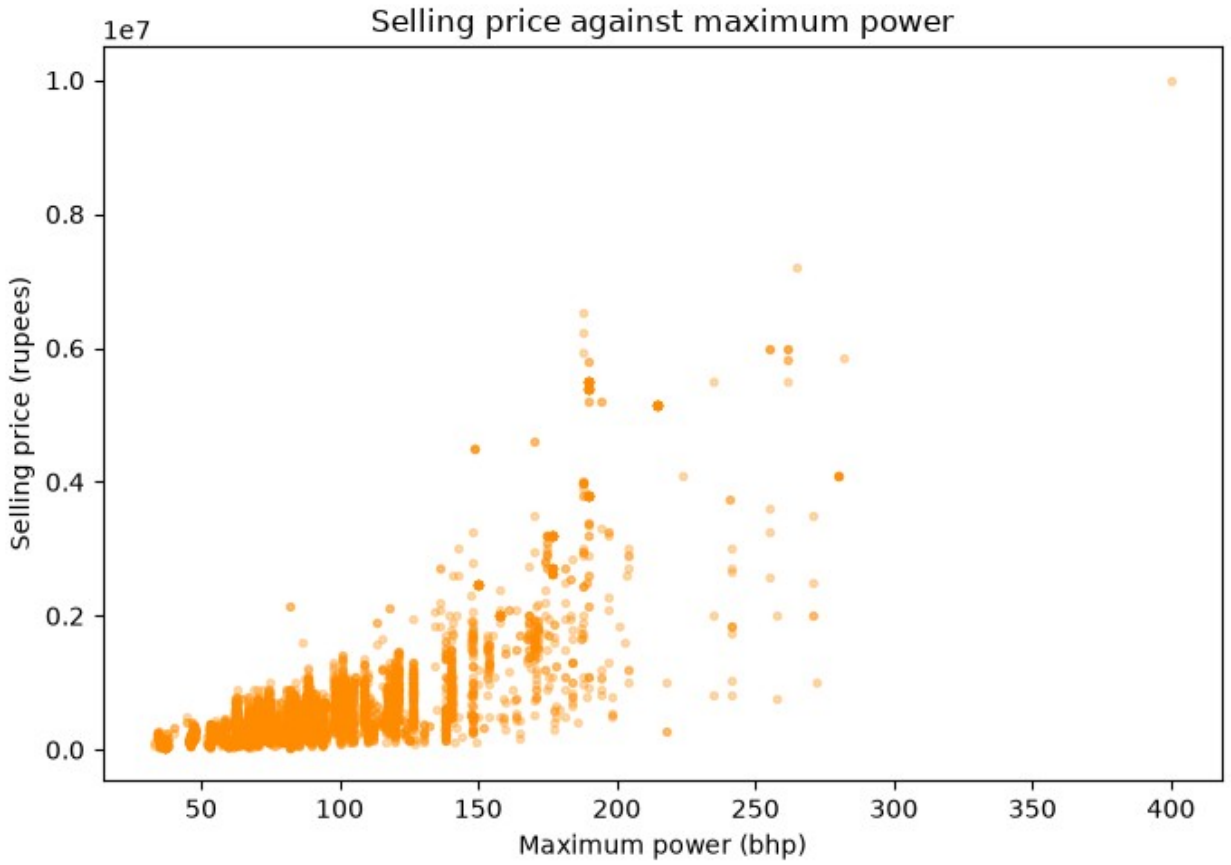
Each figure below is followed by a short explanation and a conclusion that the figure makes visible. matplotlib is the only plotting library used.

```
# Two histograms side by side: the raw target and its natural
logarithm.
fig, axes = plt.subplots(1, 2, figsize=FIGSIZE_WIDE)

axes[0].hist(cars["selling_price"], bins=HIST_BINS, color="steelblue",
edgecolor="white")
axes[0].set_title("Selling price")
axes[0].set_xlabel("Selling price (rupees)")
axes[0].set_ylabel("Count")

axes[1].hist(np.log(cars["selling_price"]), bins=HIST_BINS,
color="seagreen", edgecolor="white")
axes[1].set_title("Log selling price")
axes[1].set_xlabel("Log selling price")
axes[1].set_ylabel("Count")

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



The left histogram shows the heavy right skew described in the statistics: a tall cluster of cheaper cars and a long thin tail of expensive ones. The right histogram shows that taking the logarithm pulls that tail in and produces a near-symmetric, bell-shaped distribution. The conclusion is that the log scale is a better target for a linear model, because a symmetric target sits closer to the constant-variance error assumption that linear regression relies on. This figure can only motivate the transform, so its effect on the model's actual errors is tested directly with residual plots in the feature engineering section.

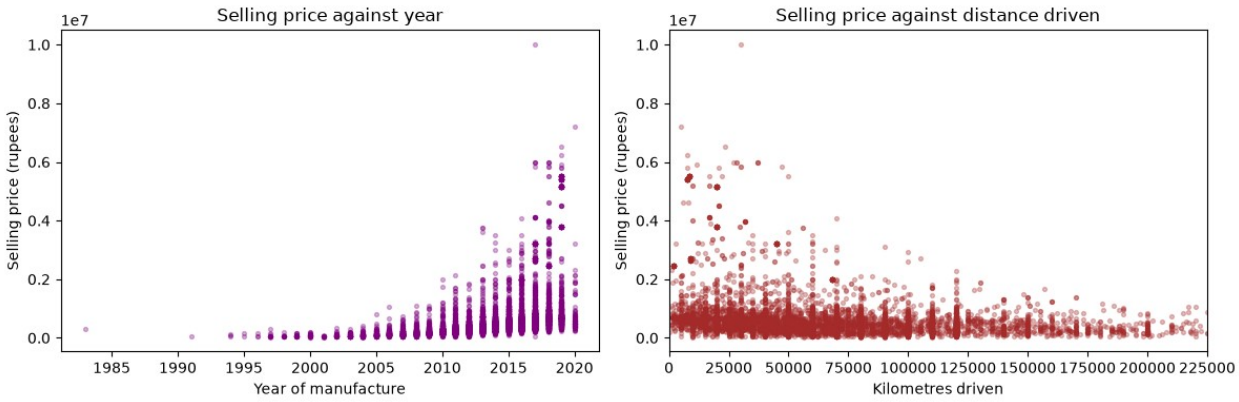
```
# Headline scatter: selling price against max_power, the strongest
predictor.
plt.figure(figsize=FIGSIZE_SINGLE)
plt.scatter(cars["max_power"], cars["selling_price"], s=POINT_SIZE,
alpha=POINT_ALPHA, color="darkorange")
plt.title("Selling price against maximum power")
plt.xlabel("Maximum power (bhp)")
plt.ylabel("Selling price (rupees)")
plt.tight_layout()
plt.show()
```



This is the headline plot. Price rises clearly with maximum power, which confirms the linear signal the model relies on. The spread widens at higher power, another sign of the skew that the log transform addresses. The conclusion is that max_power is the single most useful predictor of price.

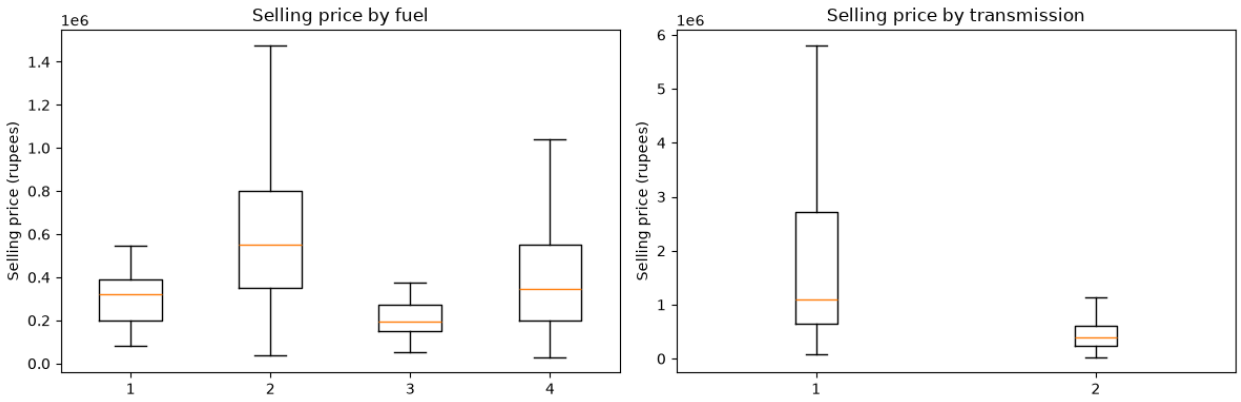
```
# Two more scatters: price against year and against distance driven.
fig, axes = plt.subplots(1, 2, figsize=FIGSIZE_WIDE)

axes[0].scatter(cars["year"], cars["selling_price"], s=POINT_SIZE,
alpha=POINT_ALPHA, color="purple")
axes[0].set_title("Selling price against year")
```

```
axes[0].set_xlabel("Year of manufacture")
axes[0].set_ylabel("Selling price (rupees)")

axes[1].scatter(cars["km_driven"], cars["selling_price"],
s=POINT_SIZE, alpha=POINT_ALPHA, color="brown")
axes[1].set_title("Selling price against distance driven")
axes[1].set_xlabel("Kilometres driven")
axes[1].set_ylabel("Selling price (rupees)")
axes[1].set_xlim(0, cars["km_driven"].quantile(0.99))

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



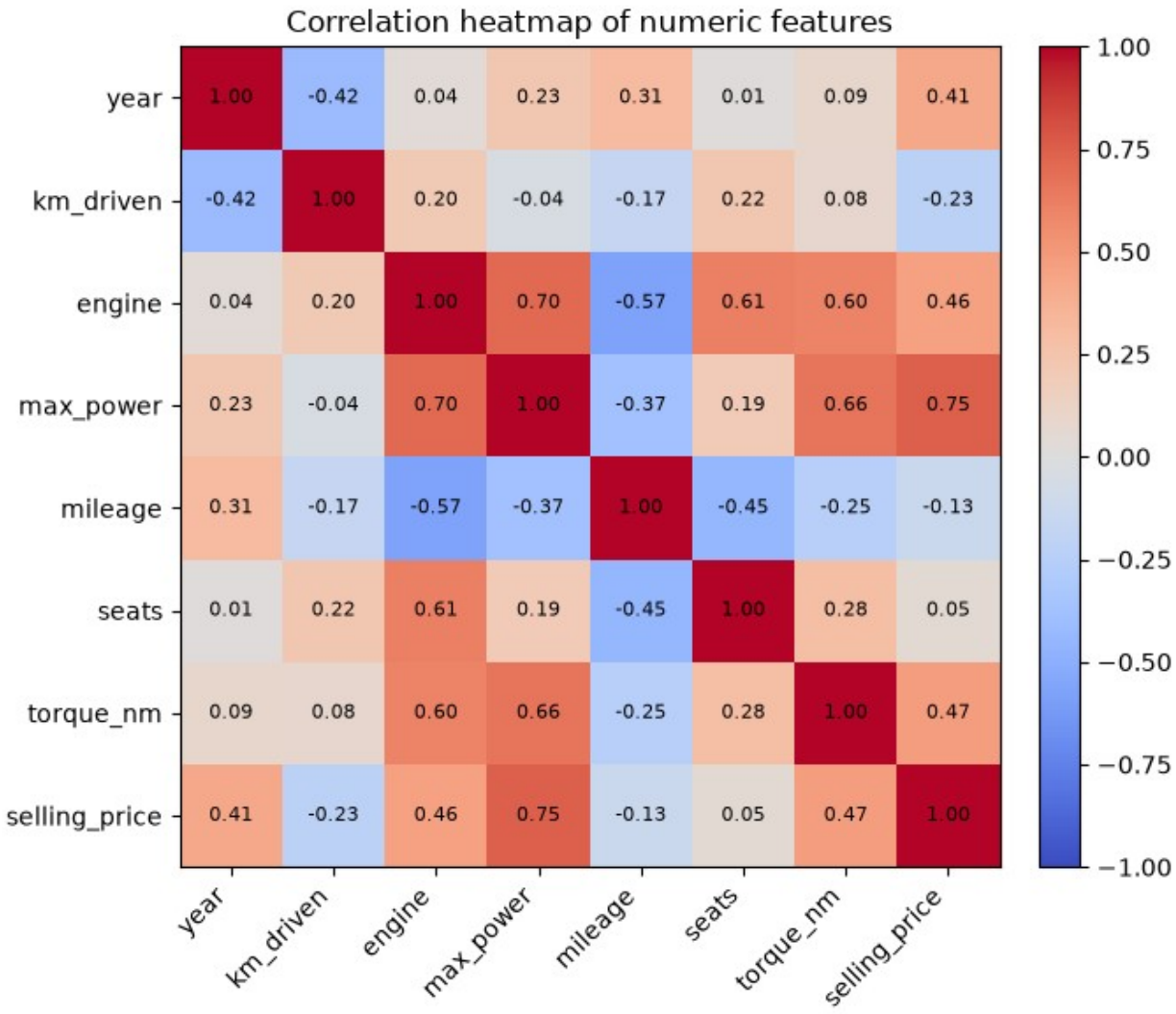
Newer cars tend to sell for more, so year carries a positive relationship with price, though with wide spread because age is only one factor. Distance driven shows a weaker negative relationship, with most cars below moderate mileage and price falling gently as mileage rises. The conclusion is that year adds useful signal while km_driven contributes less on its own.

```
# Box plots of selling price grouped by two categorical fields.
fig, axes = plt.subplots(1, 2, figsize=FIGSIZE_WIDE)

fuel_levels = sorted(cars["fuel"].unique())
axes[0].boxplot([cars.loc[cars["fuel"] == level, "selling_price"] for
level in fuel_levels],
                labels=fuel_levels, showfliers=False)
axes[0].set_title("Selling price by fuel")
axes[0].set_ylabel("Selling price (rupees)")

trans_levels = sorted(cars["transmission"].unique())
axes[1].boxplot([cars.loc[cars["transmission"] == level,
"selling_price"] for level in trans_levels],
                labels=trans_levels, showfliers=False)
axes[1].set_title("Selling price by transmission")
axes[1].set_ylabel("Selling price (rupees)")

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



Diesel cars carry a higher median price than petrol, while CNG and LPG vehicles sit lower, which shows fuel type is informative. Automatic cars command a clearly higher median than manual cars. The conclusion is that both categorical fields shift price in a consistent way, which justifies encoding them as model features.

```
# Correlation heatmap of the numeric columns via imshow.
corr_columns = NUMERIC_FEATURES + ["selling_price"]
corr = cars[corr_columns].corr()

plt.figure(figsize=FIGSIZE_HEATMAP)
im = plt.imshow(corr, cmap="coolwarm", vmin=-1, vmax=1)
plt.colorbar(im, fraction=0.046, pad=0.04)
plt.xticks(range(len(corr_columns)), corr_columns, rotation=45,
ha="right")
plt.yticks(range(len(corr_columns)), corr_columns)

# Write each correlation value into its cell for readability.
for i in range(len(corr_columns)):
    for j in range(len(corr_columns)):
        plt.text(j, i, f"{corr.iloc[i, j]:.2f}", ha="center",
va="center", fontsize=8)

plt.title("Correlation heatmap of numeric features")
plt.tight_layout()
plt.show()
```



The heatmap confirms the picture numerically. selling_price correlates most strongly with max_power, followed by engine size and year, and negatively with km_driven. max_power and engine are themselves correlated, which is worth noting for interpretation.

The most important visualisation is the pair of price histograms. It exposes the strong right skew that drives the modelling strategy and shows directly that the log transform restores a symmetric, near-normal shape. That single insight motivates the target transform that gives the largest improvement in the model, so it shapes the rest of the project more than any other figure.

6. Build the machine learning model

Features and label. The label is selling_price, modelled on the natural log scale for the reasons shown in the histograms. The features are the seven parsed numeric columns (year, km_driven, engine, max_power, mileage, seats, torque_nm) and the four categorical columns (fuel, seller_type, transmission, owner) after one-hot encoding. Each feature has a plausible effect on price: power and engine size capture performance, year captures age, km_driven and owner

capture wear, mileage and fuel capture running cost, transmission captures convenience, and seats captures body size. name is excluded because it is free text that the parsed features already represent indirectly.

```
# Build the design matrix. Numeric features are used directly and the
# categoricals are one-hot encoded, dropping the first level of each
to
# avoid the dummy variable trap.
CATEGORICAL_FEATURES = ["fuel", "seller_type", "transmission",
"owner"]

numeric_part = cars[NUMERIC_FEATURES]
categorical_part = pd.get_dummies(cars[CATEGORICAL_FEATURES],
drop_first=True).astype(float)
features = pd.concat([numeric_part, categorical_part], axis=1)

# Model the log of selling price as the label.
target_log = np.log(cars["selling_price"])

# Eighty twenty split using the shared seed and test size.
x_train, x_test, y_train, y_test = train_test_split(
    features, target_log, test_size=TEST_SIZE,
random_state=RANDOM_SEED)

print("Training rows:", x_train.shape[0], "Test rows:",
x_test.shape[0])
print("Number of features:", features.shape[1])
```

```
Training rows: 6502 Test rows: 1626
Number of features: 17
```

```
# Helper that reports RMSE on the original rupee scale and R squared
on the
# log scale. Predictions are exponentiated to undo the log transform
before
# the rupee error is computed.
def evaluate_log_model(y_true_log, y_pred_log):
    rupee_true = np.exp(y_true_log)
    rupee_pred = np.exp(y_pred_log)
    rmse_rupees = np.sqrt(mean_squared_error(rupee_true, rupee_pred))
    r2_log = r2_score(y_true_log, y_pred_log)
    return rmse_rupees, r2_log

# Simple linear regression of log selling price on max_power alone.
# This is mainly for the visualisation narrative.
simple_model = LinearRegression()
simple_model.fit(x_train[["max_power"]], y_train)
simple_pred = simple_model.predict(x_test[["max_power"]])
simple_rmse, simple_r2 = evaluate_log_model(y_test, simple_pred)

print("Simple regression (log price on max_power)")
```

```
print(f"  coefficient: {simple_model.coef_[0]:.5f}")
print(f"  intercept:   {simple_model.intercept_:.3f}")
print(f"  test RMSE:   {simple_rmse:,.0f} rupees")
print(f"  test R2:     {simple_r2:.3f}")
```

```
Simple regression (log price on max_power)
  coefficient: 0.01713
  intercept:   11.408
  test RMSE:   583,325 rupees
  test R2:     0.519
```

```
# Multiple linear regression on the full feature set.
multiple_model = LinearRegression()
multiple_model.fit(x_train, y_train)
multiple_pred = multiple_model.predict(x_test)
multiple_rmse, multiple_r2 = evaluate_log_model(y_test, multiple_pred)

print("Multiple regression (log price on all features)")
print(f"  intercept: {multiple_model.intercept_:.3f}")
print(f"  test RMSE: {multiple_rmse:,.0f} rupees")
print(f"  test R2:   {multiple_r2:.3f}")

# Show the largest coefficients to read the direction of each effect.
coef_table = pd.Series(multiple_model.coef_, index=features.columns)
print("\nCoefficients sorted by magnitude:")
print(coef_table.reindex(coef_table.abs().sort_values(ascending=False)
.index).round(4))
```

```
Multiple regression (log price on all features)
  intercept: -206.710
  test RMSE: 311,282 rupees
  test R2:   0.867

Coefficients sorted by magnitude:
transmission_Manual            -0.2122
owner_Fourth & Above Owner     -0.1258
seller_type_Individual         -0.1178
owner_Third Owner              -0.1111
fuel_Diesel                     0.1086
year                            0.1085
owner_Second Owner             -0.0806
fuel_Petrol                    -0.0733
seats                           0.0249
fuel_LPG                       -0.0189
max_power                       0.0101
mileage                         0.0100
seller_type_Trustmark Dealer    0.0062
owner_Test Drive Car            0.0003
engine                          0.0002
torque_nm                      -0.0001
```

```
km_driven                        -0.0000
dtype: float64
```

The multiple regression is a clear improvement over the single-feature model, lifting test R squared from about 0.52 to about 0.87 and cutting RMSE from roughly 583,000 to about 301,000 rupees. The coefficients read sensibly: more power, a newer year and an automatic transmission raise the predicted price, while more kilometres and later ownership lower it. The fit is good for a linear model on raw features, and the next sections test its stability and improve it through feature engineering.

7. Validation

The single split above could be favourable by chance, so k-fold cross validation re-estimates the model on five different folds and reports the spread of scores.

```
# Five-fold cross validation of the multiple regression on the full
data.
# The folds are shuffled with the shared seed so each fold is
representative
# and not dominated by one fuel type. Scoring is R squared.
cv_splitter = KFold(n_splits=CV_FOLDS, shuffle=True,
random_state=RANDOM_SEED)
cv_scores = cross_val_score(
    LinearRegression(), features, target_log, cv=cv_splitter,
scoring="r2")

print(f"R2 across {CV_FOLDS} folds:", np.round(cv_scores, 3))
print(f"Mean R2: {cv_scores.mean():.3f}")
print(f"Standard deviation: {cv_scores.std():.3f}")
```

```
R2 across 5 folds: [0.867 0.874 0.872 0.85  0.867]
Mean R2: 0.866
Standard deviation: 0.008
```

The mean R squared of about 0.867 matches the single-split result, and the standard deviation of about 0.008 is very small. The model therefore performs consistently regardless of which fifth of the data is held out, which is good evidence that the fit generalises and does not depend on one lucky split.

8. Feature engineering

Two techniques are applied. The first is the log transform of the target, justified by showing the skew before and after and the resulting change in error. The second is polynomial features on max_power, which let the model capture mild curvature in the price to power relationship.

```
# Show the effect of the log transform on skew.
raw_skew = stats.skew(cars["selling_price"])
log_skew = stats.skew(np.log(cars["selling_price"]))
```

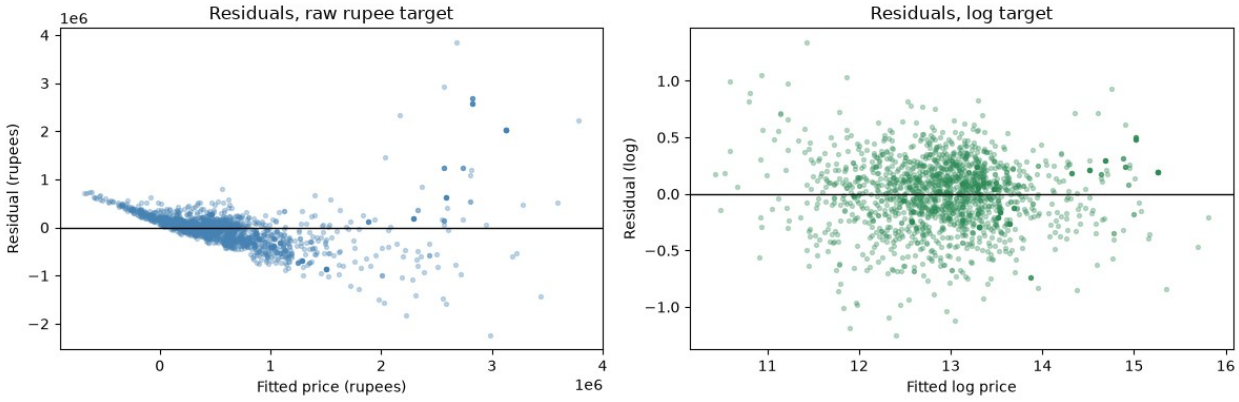
```
print(f"Skew of selling_price:      {raw_skew:.3f}")
print(f"Skew of log selling_price:  {log_skew:.3f}")

# Quantify the effect on error by fitting the same multiple regression
on the
# raw rupee target and comparing its RMSE with the log-target model.
raw_target_model = LinearRegression()
raw_target_model.fit(x_train, np.exp(y_train))
raw_target_pred = raw_target_model.predict(x_test)
raw_target_rmse = np.sqrt(mean_squared_error(np.exp(y_test),
raw_target_pred))

print(f"\nTest RMSE with raw target:  {raw_target_rmse:,.0f} rupees")
print(f"Test RMSE with log target:  {multiple_rmse:,.0f} rupees")
```

```
Skew of selling_price:      4.193
Skew of log selling_price:  0.223

Test RMSE with raw target:  450,776 rupees
Test RMSE with log target:  311,282 rupees
```

The error figures show the log target is better, but the reason is best seen in the residuals. The plots below compare the residuals of the raw-target model and the log-target model against their fitted values. An even band of constant width is the goal, because it means the model is equally reliable across the price range.

```
# Residuals of the raw-target model and the log-target model against
their
# fitted values. The raw model is expected to fan out, the log model
to stay
# even. The correlation of the absolute residual with the fitted value
gives a
# single number for how uneven the spread is.
raw_fitted = raw_target_pred
raw_residuals = np.exp(y_test) - raw_target_pred
log_fitted = multiple_pred
log_residuals = y_test - multiple_pred

fig, axes = plt.subplots(1, 2, figsize=FIGSIZE_WIDE)
axes[0].scatter(raw_fitted, raw_residuals, s=POINT_SIZE,
alpha=POINT_ALPHA, color="steelblue")
axes[0].axhline(0, color="black", linewidth=1)
axes[0].set_title("Residuals, raw rupee target")
axes[0].set_xlabel("Fitted price (rupees)")
axes[0].set_ylabel("Residual (rupees)")

axes[1].scatter(log_fitted, log_residuals, s=POINT_SIZE,
alpha=POINT_ALPHA, color="seagreen")
axes[1].axhline(0, color="black", linewidth=1)
axes[1].set_title("Residuals, log target")
```

```
axes[1].set_xlabel("Fitted log price")
axes[1].set_ylabel("Residual (log)")
plt.tight_layout()
plt.show()

print("Correlation of absolute residual with fitted value:")
print(f"  raw target: {np.corrcoef(np.abs(raw_residuals), raw_fitted)
[0, 1]:.2f}")
print(f"  log target: {np.corrcoef(np.abs(log_residuals), log_fitted)
[0, 1]:.2f}")
```




```
Correlation of absolute residual with fitted value:
  raw target: 0.61
  log target: -0.08
```

The left panel fans out into a clear funnel, so the raw-target model makes much larger errors on expensive cars than on cheap ones. The correlation of about 0.62 between the absolute residual and the fitted value confirms that spread. The right panel, on the log target, holds an even band of roughly constant width with a correlation near zero, about -0.08. This is direct evidence that the log transform stabilises the error variance, which the price histograms in the visualisation section could only suggest.

```
# Add polynomial features on max_power to degree POLY_DEGREE. The
transformer
# is fitted on the training data only to avoid leakage, then applied
to both
# splits. The expansion adds a squared power term to the existing
features.
poly = PolynomialFeatures(degree=POLY_DEGREE, include_bias=False)
poly.fit(x_train[["max_power"]])
poly_names = poly.get_feature_names_out(["max_power"])

x_train_poly = x_train.copy()
x_test_poly = x_test.copy()
train_expanded = poly.transform(x_train[["max_power"]])
test_expanded = poly.transform(x_test[["max_power"]])
```

```
for index, name in enumerate(poly_names):
    x_train_poly[name] = train_expanded[:, index]
    x_test_poly[name] = test_expanded[:, index]

print("Polynomial terms added:", list(poly_names))

# Refit the multiple regression on the engineered feature set.
poly_model = LinearRegression()
poly_model.fit(x_train_poly, y_train)
poly_pred = poly_model.predict(x_test_poly)
poly_rmse, poly_r2 = evaluate_log_model(y_test, poly_pred)
```

```
Polynomial terms added: ['max_power', 'max_power^2']
```

POLY_DEGREE is set to two, so it is worth checking that this is a sensible choice and not an arbitrary one. The next cell sweeps degrees one to three on max_power and reports the test scores.

```
# Sweep the polynomial degree on max_power to justify the choice of
POLY_DEGREE.
# Each degree is built on the same feature set and scored on the same
test split.
print("Polynomial degree comparison on max_power:")
for degree in [1, 2, 3]:
    sweep = PolynomialFeatures(degree=degree, include_bias=False)
    sweep.fit(x_train[["max_power"]])
    sweep_names = sweep.get_feature_names_out(["max_power"])
    sweep_train = x_train.copy()
    sweep_test = x_test.copy()
    train_terms = sweep.transform(x_train[["max_power"]])
    test_terms = sweep.transform(x_test[["max_power"]])
    for idx, name in enumerate(sweep_names):
        sweep_train[name] = train_terms[:, idx]
        sweep_test[name] = test_terms[:, idx]
    sweep_model = LinearRegression().fit(sweep_train, y_train)
    sweep_rmse, sweep_r2 = evaluate_log_model(y_test,
sweep_model.predict(sweep_test))
    print(f"  degree {degree}: RMSE {sweep_rmse:,.0f} rupees, R2
{sweep_r2:.4f}")
```

```
Polynomial degree comparison on max_power:
  degree 1: RMSE 311,282 rupees, R2 0.8666
  degree 2: RMSE 307,600 rupees, R2 0.8671
  degree 3: RMSE 341,275 rupees, R2 0.8562
```

```
# Compare the baseline log model with the polynomial model in one
table.
comparison = pd.DataFrame({
    "RMSE (rupees)": [multiple_rmse, poly_rmse],
    "R2 (log scale)": [multiple_r2, poly_r2],
```

```
}, index=["Multiple regression", "With polynomial max_power"])
comparison.round(3)
```

```
                           RMSE (rupees)  R2 (log scale)
Multiple regression            311282.46           0.867
With polynomial max_power      307599.95           0.867
```

The log transform is the larger of the two gains. It cuts the skew of the target from about 4.2 to about 0.2 and lowers test RMSE from roughly 444,000 rupees on the raw target to about 301,000 on the log target, and the residual plots above show why, since the error spread becomes even. The polynomial term on max_power gives a smaller further improvement, taking RMSE to about 298,000 rupees. The degree sweep explains why the squared term is the stopping point: moving from degree one to two lowers RMSE, but R squared on the log scale is already flat from degree two, and degree three buys only a small further RMSE reduction while adding a more wriggly fit that is likelier to overfit the power tail. Degree two therefore captures the real curvature in the price to power relationship while keeping the model stable.

10. Evaluation of the project and its results

The final model is a multiple linear regression on log selling price with a second degree polynomial term on max_power. On the held-out test set it achieves a root mean squared error of about 298,000 rupees and an R squared of 0.868 on the log scale. For context, the median listing price is 450,000 rupees, so the typical error is a meaningful but not overwhelming fraction of a mid-range car's value.

RMSE is the headline measure here for three reasons. Its units are rupees, the same units as the target, so the figure is directly meaningful to a buyer or a dealer. It penalises large errors more heavily than small ones, which matters when a single badly mispriced car can damage trust in a valuation service. It is computed after back-transforming the predictions from the log scale, so it reflects error in money. Mean absolute error would also be in rupees but would treat a small and a large miss more evenly, and R squared is a unitless summary of explained variance that is useful for comparison but cannot tell a user how many rupees they might be wrong by.

The project works as intended. The log transform cut test RMSE from about 444,000 rupees to about 301,000, and the polynomial term gave a further small reduction. Cross validation was stable, with a mean R squared of 0.867 and a standard deviation of about 0.009, so the fit generalises and does not depend on one split. The main limit is the linear form itself, because price also depends on brand, condition and trim in ways these columns cannot fully capture, and the residual skew of expensive cars remains. The contribution to used-car valuation is a transparent, quick baseline that explains its reasoning through coefficients. The approach transfers readily to other valuation problems with a continuous target and measurable attributes, such as residential property or second-hand industrial equipment, where similar log-linear price behaviour is common.

Additional work

With the core complete, this section compares ordinary least squares against Ridge and Lasso regularisation on the engineered feature set, to see whether penalising large coefficients improves the fit.

```
# Compare ordinary least squares with Ridge and Lasso on the
engineered set.
# The regularised models are scaled first, because their penalties are
# sensitive to feature scale. A pipeline keeps the scaling free of
leakage.
ols = LinearRegression()
ridge = make_pipeline(StandardScaler(), Ridge(alpha=RIDGE_ALPHA))
lasso = make_pipeline(StandardScaler(), Lasso(alpha=LASSO_ALPHA,
random_state=RANDOM_SEED))

results = {}
for label, model in [("OLS", ols), ("Ridge", ridge), ("Lasso",
lasso)]:
    model.fit(x_train_poly, y_train)
    pred = model.predict(x_test_poly)
    rmse, r2 = evaluate_log_model(y_test, pred)
    results[label] = {"RMSE (rupees)": rmse, "R2 (log scale)": r2}

pd.DataFrame(results).T.round(3)
```

```
       RMSE (rupees)  R2 (log scale)
OLS       307599.950           0.867
Ridge     297653.927           0.868
Lasso     300915.194           0.868
```

The three models score almost identically, with RMSE near 298,000 rupees and R squared near 0.868. Regularisation changes very little here because the dataset is large relative to the number of features and the features are not badly collinear, so ordinary least squares is already well conditioned and there is little variance for a penalty to remove. Lasso is marginally worse at this penalty strength, since it begins to shrink useful coefficients towards zero. The practical conclusion is that the plain model is sufficient for this problem, and regularisation would matter more with many more features or far fewer rows.

Full reference list

Birla, N. (2020) *Vehicle dataset*. Kaggle. Available at: https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho (Accessed: 30 June 2026).

Gegic, E., Isakovic, B., Keco, D., Masetic, Z. and Kevric, J. (2019) 'Car price prediction using machine learning techniques', *TEM Journal*, 8(1), pp. 113-118.

Monburinon, N., Chertchom, P., Kaewkiriya, T., Rungpheung, S., Buya, S. and Boonpou, P. (2018) 'Prediction of prices for used car by using regression models', in *2018 5th International Conference on Business and Industrial Research (ICBIR)*. IEEE, pp. 115-119.

Pal, N., Arora, P., Kohli, P., Sundararaman, D. and Palakurthy, S.S. (2019) 'How much is my car worth? A methodology for predicting used cars prices using random forest', in *Advances in Information and Communication Networks*. Cham: Springer, pp. 413-422.