267/9, West Agargaon
Dhaka-1207. Bangladesh
(+880) 1679228352
mdemrannazirefty@gmail.com

CAR PRICE PREDICTION

Supervised by
Reja E Rabbi Tonmoy
Machine Learning Engineer, Pathao

Submitted to
Byte To Code

Submitted By
Md. Emran Nazir Efty
American International University- Bangladesh (AIUB)
LinkedIn: www.linkedin.com/in/mdemrannazirefty
GitHub: github.com/mdemrannazirefty

Submission:
December 05, 2025

Overview

- **Introduction**
- **Dataset**
- **Methodology**
    i. Import necessary packages
    ii. Data Load
    iii. Exploratory Data Analysis (EDA)
    iv. Data Pre-processing
    v. Matplot
    vi. Validation Framework
    vii. Linear Regression From Scratch
    viii. Linear Regression From Vector
    ix. RMSE (Root Mean Square Error)
    x. Train Linear Regression
    xi. Base-Line Model
    xii. Validation The Model
    xiii. Simple Feature Engineering
    xiv. Categorical Value
    xv. Tuning Model
    xvi. Train Model
- **Conclusion**

Introduction

Car prices depend on many things: the brand, the year, the mileage, the engine size, and even smaller details that most people don't think about. Because of this, it can be hard to estimate the correct price of a car just by looking at it. To make this easier, I'm building a machine-learning model that can predict a car's price based on the information available in the dataset.

The idea is straightforward. If we give the model enough examples of cars and their actual prices, it can learn the relationship between a car's features and how much it usually sells for. Once trained, the model will be able to take a new car's details and give an estimated price.

This report explains the steps taken to prepare the data, explore important features, build the model, and evaluate how well it performs. The goal is to create a prediction system that is simple, helpful, and reliable for estimating car prices.

Dataset

Dataset Name: Car Features and MSRP

Dataset Source: Kaggle (https://www.kaggle.com/datasets/CooperUnion/cardataset)

Dataset Descriptions

The analysis is based on a dataset containing 11,914 automotive records, representing a wide variety of car makes and models manufactured between 1990 and 2017. The data is structured to provide a comprehensive profile of each vehicle's technical specifications, design attributes, and market positioning.

Data Structure:

- **Total Samples:** 11,914 rows
- **Total Features:** 16 columns (mixture of categorical and numerical data)
- **Target Variable:** MSRP (Manufacturer's Suggested Retail Price)

Description of Features:

The dataset includes the following attributes for each vehicle:

Variable	Description
Make	The car manufacturer (e.g., BMW, Audi, Toyota).
Model	The specific model's name of the vehicle.
Year	The year of manufacture.
Engine Fuel Type	The type of fuel required (e.g., regular unleaded, premium unleaded, diesel)
Engine HP	The horsepower rating of the engine.
Engine Cylinders	The number of cylinders in the engine.
Transmission Type	The style of transmission (e.g., AUTOMATIC, MANUAL, AUTOMATED_MANUAL)
Driven_Wheels	The drivetrain configuration (e.g., front wheel drive, all-wheel drive)
Number of Doors	The total number of doors on the vehicle.
Market Category	Labels describing the market segment (e.g., Luxury, High-Performance, Flex Fuel).
Vehicle Size	Classification of the vehicle's size (e.g., Compact, Midsize, Large).
Vehicle Style	The body style of the car (e.g., Coupe, Sedan, SUV).
highway MPG	Fuel economy rating for highway driving.
city mpg	Fuel economy rating for city driving.
Popularity	A numerical score representing the brand's popularity
MSRP	The price of the vehicle serves as the target variable for the regression model.

Methodology

Import necessary packages

```
!pip install pandas
!pip install numpy
!pip install seaborn

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
Requirement already satisfied: pandas in e:\others\machine learning\python-venv\.venv\lib\site-packages (2.3.3)
Requirement already satisfied: numpy>=1.26.0 in e:\others\machine learning\python-venv\.venv\lib\site-packages (from pandas) (2.3.4)
Requirement already satisfied: python-dateutil>=2.8.2 in e:\others\machine learning\python-venv\.venv\lib\site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in e:\others\machine learning\python-venv\.venv\lib\site-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in e:\others\machine learning\python-venv\.venv\lib\site-packages (from pandas) (2025.2)
Requirement already satisfied: six>=1.5 in e:\others\machine learning\python-venv\.venv\lib\site-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
```

The code starts by importing the essential Python libraries.
pandas are used for handling and analyzing data, while numpy helps with numerical operations.

Data Load

```
df=pd.read_csv('../data/data.csv')
df
```

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	of Doors	Market Category	Vehicle Size	Vehicle Style	highway MPG	city mpg	Popularity	MSRP
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compact	Coupe	26	19	3916	46135
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Convertible	28	19	3916	40650
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compact	Coupe	28	20	3916	36350
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Coupe	28	18	3916	29450
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compact	Convertible	28	18	3916	34500

The dataset is read from the file path ../data/data.csv and stored inside a DataFrame named df. This makes the data ready for analysis and preprocessing.

Exploratory Data Analysis (EDA)

df.columns

```
Index(['Make', 'Model', 'Year', 'Engine Fuel Type', 'Engine HP',
       'Engine Cylinders', 'Transmission Type', 'Driven_Wheels',
       'Number of Doors', 'Market Category', 'Vehicle Size', 'Vehicle Style',
       'highway MPG', 'city mpg', 'Popularity', 'MSRP'],
      dtype='object')
```

This lists all the column names in the dataset. It's a quick way to see the structure of your Data Frame.

df.dtypes

```
make                object
model               object
year                 int64
engine_fuel_type    object
engine_hp          float64
engine_cylinders   float64
transmission_type   object
driven_wheels       object
number_of_doors    float64
market_category     object
vehicle_size        object
vehicle_style       object
highway_mpg          int64
city_mpg             int64
popularity           int64
msrp                 int64
dtype: object
```

df.dtypes displays the data type of each column. It helps understand which features are numerical or categorical.

df['Transmission Type']

```
0          MANUAL
1          MANUAL
2          MANUAL
3          MANUAL
4          MANUAL
           ...
11909    AUTOMATIC
11910    AUTOMATIC
11911    AUTOMATIC
11912    AUTOMATIC
11913    AUTOMATIC
Name: Transmission Type, Length: 11914, dtype: object
```

It selects and shows all the values in the *Transmission Type* column for quick inspection.

```
df.dtypes[df.dtypes=='object']
```

```
Make                object
Model               object
Engine Fuel Type    object
Transmission Type   object
Driven_Wheels       object
Market Category     object
Vehicle Size        object
Vehicle Style       object
dtype: object
```

It filters and shows only the columns whose data type is object, meaning the categorical or text-based features.

```
for col in df.columns:
    print(col)
    print(df[col].head())
    print()
```

```
Make
0    BMW
1    BMW
2    BMW
3    BMW
4    BMW
Name: Make, dtype: object

Model
0    1 Series M
1      1 Series
2      1 Series
3      1 Series
4      1 Series
Name: Model, dtype: object

Year
0    2011
1    2011
2    2011
3    2011
4    2011
Name: Year, dtype: int64

Engine Fuel Type
0    premium unleaded (required)
1    premium unleaded (required)
2    premium unleaded (required)
3    premium unleaded (required)
4    premium unleaded (required)
Name: Engine Fuel Type, dtype: object
```

It loops through every column, prints the column name, and shows the first few values to quick way to inspect all features in the dataset.

```
for col in df.columns:
    print(col)
    print(df[col].unique()[:5])
    print(df[col].nunique())
    print()
```

```
Make
['BMW' 'Audi' 'FIAT' 'Mercedes-Benz' 'Chrysler']
48

Model
['1 Series M' '1 Series' '100' '124 Spider' '190-Class']
915

Year
[2011 2012 2013 1992 1993]
28

Engine Fuel Type
['premium unleaded (required)' 'regular unleaded'
 'premium unleaded (recommended)' 'flex-fuel (unleaded/E85)' 'diesel']
10
```

It prints each column's name, shows a sample of its unique values, and displays how many unique values

Data Pre-processing

```
df.columns = df.columns.str.lower().str.replace(' ', '_')
df
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0

It renames all column names by converting them to lowercase and replacing spaces with underscores, making them easier to work with in code.

```
strings = list(df.dtypes[df.dtypes == 'object'].index)
strings
```

```
['make',
 'model',
 'engine_fuel_type',
 'transmission_type',
 'driven_wheels',
 'market_category',
 'vehicle_size',
 'vehicle_style']
```

It collects the names of all object-type (string/categorical) columns into a list called strings, then displays that list.

```
for col in strings:
    df[col] = df[col].str.lower().str.replace(' ', '_')

df
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category	vehicle_size
0	bmw	1_series_m	2011	premium_unleaded_(required)	335.0	6.0	manual	rear_wheel_drive	2.0	factory_tuner,luxury,high-performance	compact
1	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	rear_wheel_drive	2.0	luxury,performance	compact
2	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	rear_wheel_drive	2.0	luxury,high-performance	compact
3	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	rear_wheel_drive	2.0	luxury,performance	compact
4	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	rear_wheel_drive	2.0	luxury	compact

It cleans all string-type columns by converting their text to lowercase and replacing spaces with underscores, creating consistent and usable categorical values.

```
for col in df.columns:
for col in strings:
    df['driven_wheels'] = df[col].str.lower().str.replace(' ', '_')
df
```

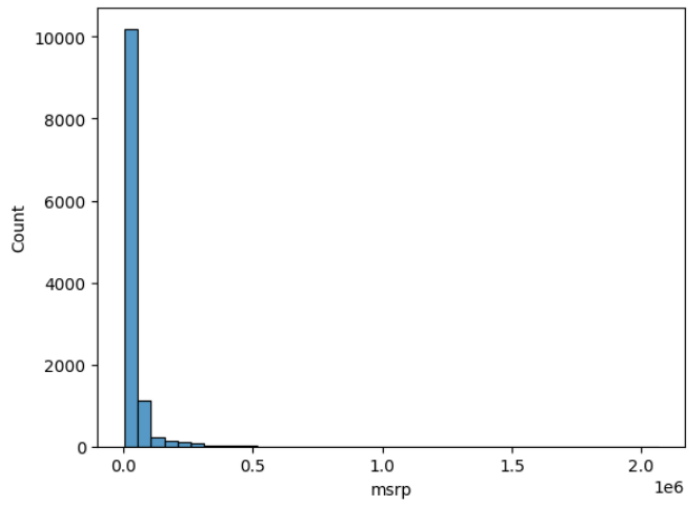
	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category	vehicle_size
0	bmw	1_series_m	2011	premium_unleaded_(required)	335.0	6.0	manual	coupe	2.0	factory_tuner,luxury,high-performance	compact
1	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	convertible	2.0	luxury,performance	compact
2	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	coupe	2.0	luxury,high-performance	compact
3	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	coupe	2.0	luxury,performance	compact
4	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	convertible	2.0	luxury	compact

It overwrites the driven_wheels column again and again using every column in strings.

Matplot

```
%matplotlib inline
sns.histplot(df.msrp, bins=40)
```

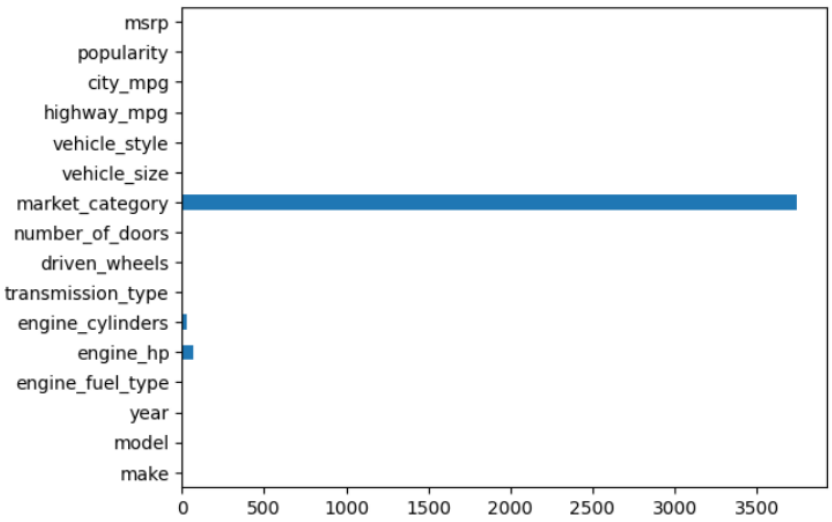
```
<Axes: xlabel='msrp', ylabel='Count'>
```



It creates a histogram of the msrp column with 40 bins, letting you see how car prices are distributed across the dataset.

```
null_counts=df.isnull().sum()
null_counts
null_counts.plot(kind='barh')
```

```
make                    0
model                   0
year                    0
engine_fuel_type        3
engine_hp              69
engine_cylinders       30
transmission_type       0
driven_wheels           0
number_of_doors         6
market_category      3742
vehicle_size            0
vehicle_style           0
highway_mpg             0
city_mpg                0
popularity              0
msrp                    0
dtype: int64
```

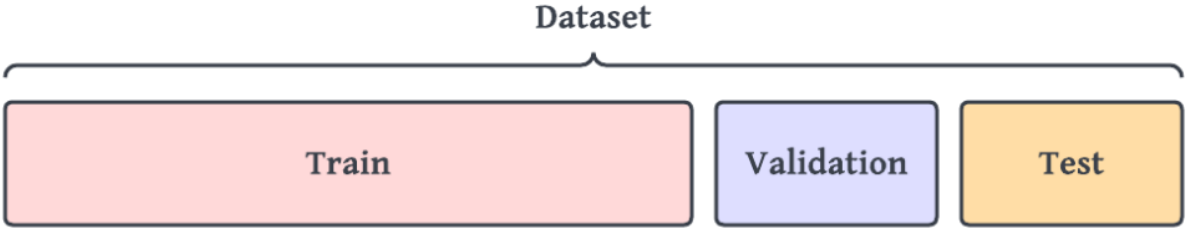


It calculates the number of missing values in each column, then plots those counts as a horizontal bar chart so you can clearly see which columns have nulls and how many.

Validation Framework

Split Dataset:

This technique divides your dataset into three distinct parts to prevent the model from memorizing the answers (overfitting). The Split Ratio (60/ 20/ 20). As you noted, the data is divided as follows:



i. Training (60%): The model uses this data to learn patterns.

ii. Validation (20%): Used during training to tune settings (hyperparameters) and see how the model is improving.

iii. Test (20%): Used only once at the very end to evaluate how good the model is on completely unseen data.

```
n=len(df)
n_val = int(n * 0.2)
n_test = int(n * 0.2)
n_train = n-n_val-n_test

n_train, n_test, n_val
```

```
(7150, 2382, 2382)
```

```
n, n_val + n_test + n_train
```

```
(11914, 11914)
```

It splits the dataset size into 20% validation, 20% test, and the rest for training, and checks that the numbers sum correctly. Here, length and (train, validation, test) are equal.

```
df_val=df.iloc[:n_val]
df_val
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category	vehicle_size	vehicle_s
0	bmw	1_series_m	2011	premium_unleaded_(required)	335.0	6.0	manual	coupe	2.0	factory_tuner,luxury,high-performance	compact	co
1	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	convertible	2.0	luxury,performance	compact	convert
2	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	coupe	2.0	luxury,high-performance	compact	co
3	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	coupe	2.0	luxury,performance	compact	co
4	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	convertible	2.0	luxury	compact	convert

It selects the first n_val rows from the dataset and stores them in df_val, creating your validation set. Index (0-2381) selected.

```
df_test = df.iloc[n_val : n_val + n_test]
df_test
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category	vehicle_size
2382	porsche	cayenne	2017	premium_unleaded_(required)	570.0	8.0	automatic	4dr_suv	4.0	crossover,luxury,high-performance	midsize
2383	porsche	cayenne	2017	premium_unleaded_(required)	420.0	6.0	automatic	4dr_suv	4.0	crossover,luxury,performance	midsize
2384	porsche	cayman_s	2006	premium_unleaded_(required)	295.0	6.0	manual	coupe	2.0	luxury,high-performance	compact
2385	porsche	cayman	2014	premium_unleaded_(required)	275.0	6.0	manual	coupe	2.0	luxury,high-performance	compact
2386	porsche	cayman	2014	premium_unleaded_(required)	325.0	6.0	manual	coupe	2.0	luxury,high-performance	compact

It takes the next n_test rows (right after the validation split) and stores them in df_test, creating your test set. Index (2381-4763) selected.

```
df_train = df.iloc[n_val + n_test:]  #4764-11913
df_train
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category	vehicle_size
4764	ford	flex	2016	regular_unleaded	287.0	6.0	automatic	wagon	4.0	crossover,performance	large
4765	ford	flex	2017	premium_unleaded_(recommended)	365.0	6.0	automatic	wagon	4.0	crossover	large
4766	ford	flex	2017	regular_unleaded	287.0	6.0	automatic	wagon	4.0	crossover,performance	large
4767	ford	flex	2017	regular_unleaded	287.0	6.0	automatic	wagon	4.0	crossover,performance	large
4768	ford	flex	2017	regular_unleaded	287.0	6.0	automatic	wagon	4.0	crossover,performance	large

It selects all remaining rows after the validation and test splits and saves them in df_train, forming your training set. (4764-11913)

Split Dataset Indexing:

```
idx=np.arange(n)
```

```
idx
```

```
array([    0,     1,     2, ..., 11911, 11912, 11913], shape=(11914,))
```

It creates an array of integers from 0 to n-1, giving you the index positions for every row in the dataset.

```
np.random.shuffle(idx)
```

```
idx
```

```
array([2447, 4956, 6427, ...,  408, 7880, 9829], shape=(11914,))
```

It randomly shuffles the index array idx, which is useful for creating randomized train/validation/test splits.

```
np.random.seed(2)
np.random.shuffle(idx)

df_train = df.iloc[idx[n_val + n_test:]]
df_val   = df.iloc[idx[:n_val]]
df_test  = df.iloc[idx[n_val : n_val + n_test]]


df_train
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type
10973	toyota	tundra	2016	regular_unleaded	310.0	8.0	automatic
1997	kia	borrego	2009	regular_unleaded	276.0	6.0	automatic
8377	toyota	rav4_hybrid	2016	regular_unleaded	194.0	4.0	automatic
11003	toyota	tundra	2016	regular_unleaded	381.0	8.0	automatic
53	bmw	2_series	2017	premium_unleaded_(recommended)	335.0	6.0	automatic

It sets a fixed random seed, shuffles the row indices, and then uses those shuffled indices to create randomized train, validation, and test splits.

As same for df_val and df_test.

```
len(df_train), len(df_test), len(df_val)
```

```
(7150, 2382, 2382)
```

Print those lengths.

```
df_train = df_train.reset_index(drop=True)
df_val   = df_val.reset_index(drop=True)
df_test  = df_test.reset_index(drop=True)

df_train
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type
0	toyota	tundra	2016	regular_unleaded	310.0	8.0	automatic
1	kia	borrego	2009	regular_unleaded	276.0	6.0	automatic
2	toyota	rav4_hybrid	2016	regular_unleaded	194.0	4.0	automatic
3	toyota	tundra	2016	regular_unleaded	381.0	8.0	automatic
4	bmw	2_series	2017	premium_unleaded_(recommended)	335.0	6.0	automatic

It resets the row indices for the train, validation, and test sets so each starts at 0, cleaning up the indexing after shuffling and splitting.

```
np.log1p(df_train.msrp.values)
```

```
array([10.50015137, 10.17526888, 10.42260867, ..., 10.38161435,
       10.63947868, 11.22525673], shape=(7150,))
```

```
y_train = np.log1p(df_train.msrp.values)
y_val   = np.log1p(df_val.msrp.values)
y_test  = np.log1p(df_test.msrp.values)
```

It transforms the MSRP values using log1p and stores them as target variables for train, validation, and

test sets. It helps stabilize variance and makes price prediction easier for the model.

```
del df_train['msrp']
del df_val['msrp']
del df_test['msrp']
```

It removes the msrp column from the train, validation, and test DataFrames, since the target variable is already saved separately and shouldn't be used as an input feature.

Linear Regression From Scratch

Formula,

g(xi)≈yi

g(xi)=w0+w1xi1+w2xi2+⋯+wnxin

Where:

- $g(x_i)$= predicted output for sample i
- w_0= intercept (bias term)
- $w_1, w_2, \dots, w_p$= model weights
- $x_{i1}, x_{i2}, \dots, x_{ip}$= input features for sample i
- n= number of features

```
xi = [200, 13, 1385]
w0 = 7.17
w = [0.01, 0.04, 0.002]

def linear_regression(xi):
    n = len(xi)
    pred = w0
    for j in range(n):
        pred = pred + w[j] * xi[j]
    return pred
```

```
linear_regression(xi)
```

```
12.459999999999999
```

```
np.exp(12.459999999999999)
```

```
np.float64(257815.631020047)
```

```
np.expm1(12.45)
```

```
np.float64(255249.32262933443)
```

```
np.log1p(255249.32262933443)
```

```
np.float64(12.45)
```

It plugs the input values xi into a linear formula using w0 (bias) and w (weights) to compute a predicted output.

Linear Regression From Vector

```
w0 = 7.17
w = [0.01, 0.04, 0.002]

w_new = [w0] + w
w_new
```

```
[7.17, 0.01, 0.04, 0.002]
```

It creates a new list w_new by placing w0 at the front of the weight list, so all weights (bias + feature weights) are stored together in one sequence.

```
def dot(xi, w):
    n = len(xi)

    res = 0.0
    for j in range(n):
        res = res + xi[j] * w[j]
    return res
```

```
def linear_regression(xi):
    xi=[1]+xi
    return dot(xi, w_new)
```

```
linear_regression(xi)
```

```
12.459999999999999
```

It multiplies each element of xi with the corresponding element of w, adds them up, and returns the total.
It adds 1 to the start of xi to represent the bias input, then uses the dot product with w_new (which already includes w0) to compute the prediction.

```
x1  = [1, 148, 24, 1385]
x2  = [1, 132, 25, 2031]
x10 = [1, 453, 11, 86]

X = [x1, x2, x10]
X = np.array(X)

X
```

```
array([[   1,  148,   24, 1385],
       [   1,  132,   25, 2031],
       [   1,  453,   11,   86]])
```

It creates several sample feature vectors, puts them into a single list X, and converts that list into a NumPy array for easier numerical operations and model processing.

```
X.dot(w_new)

def linear_regression(X):
    return X.dot(w_new)
```

```
linear_regression(X)
```

```
array([12.38 , 13.552, 12.312])
```

X.dot(w_new) computes the dot product between every row of X and the weight vector w_new, giving predicted values for all samples at once.

RMSE (Root Mean Square Error)

RMSE is one of the most popular metrics used to evaluate Regression models (models that predict numbers, like price, age, or temperature). It measures the average difference between values predicted by a model and the actual values. Lower values are better.

The Formula Breakdown

It is calculated in reverse order of its name:

Error: Find the difference between Prediction and Actual (y_pred - y_actual).

Square: Square that difference (removes negative signs and penalizes big mistakes).

Mean: Find the average of those squares.

Root: Take the square root of the result.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

```
def rmse(y,y_pred):
    error=y-y_pred
    se=error**2
    mse=se.mean()
    return np.sqrt(mse)
```

It defines an RMSE function that measures how far the predictions are from the true values by calculating the square root of the average squared error.

Train Linear Regression

Equation:

$$(X^T X)\mathbf{b} = X^T Y$$
$$[(X^T X)^{-1}(X^T X)]\mathbf{b} = (X^T X)^{-1} X^T Y$$
$$I\mathbf{b} = (X^T X)^{-1} X^T Y$$
$$\mathbf{b} = (X^T X)^{-1} X^T Y$$

For training linear regression creating X and Y matrices.

```
X = [
    [148, 24, 1385],
    [132, 25, 2031],
    [453, 11, 86],
    [158, 24, 185],
    [172, 25, 201],
    [413, 11, 86],
    [38, 54, 185],
    [142, 25, 431],
    [453, 31, 86],
]
X = np.array(X)
X
```

```
array([[ 148,   24, 1385],
       [ 132,   25, 2031],
       [ 453,   11,   86],
       [ 158,   24,  185],
       [ 172,   25,  201],
       [ 413,   11,   86],
       [  38,   54,  185],
       [ 142,   25,  431],
       [ 453,   31,   86]])
```

It builds a feature matrix using raw values only, leaving out the bias so it can be added later during prediction.

```
y=[10000,20000,15000,25000,10000,20000,15000,25000,12000]
```

```
y
```

```
[10000, 20000, 15000, 25000, 10000, 20000, 15000, 25000, 12000]
```

It creates a list Y containing the target values (the prices) that correspond to each row in X.

```
ones=np.ones(X.shape[0])
X=np.column_stack([ones, X])
```

```
X
```

```
array([[1.000e+00, 1.480e+02, 2.400e+01, 1.385e+03],
       [1.000e+00, 1.320e+02, 2.500e+01, 2.031e+03],
       [1.000e+00, 4.530e+02, 1.100e+01, 8.600e+01],
       [1.000e+00, 1.580e+02, 2.400e+01, 1.850e+02],
       [1.000e+00, 1.720e+02, 2.500e+01, 2.010e+02],
       [1.000e+00, 4.130e+02, 1.100e+01, 8.600e+01],
       [1.000e+00, 3.800e+01, 5.400e+01, 1.850e+02],
       [1.000e+00, 1.420e+02, 2.500e+01, 4.310e+02],
       [1.000e+00, 4.530e+02, 3.100e+01, 8.600e+01]])
```

np.ones(X.shape[0]) makes a column of ones, one for each row in X. If X has 9 rows, you get nine 1s. np.column_stack([ones, X]) places the ones as the first column of X.

- X.T is the transpose of X
- X.T.dot(X) multiplies the transpose with X
- The result, XTX, is the Gram matrix used in the Normal Equation for solving linear regression

```
XTX=X.T.dot(X)
XTX
```

```
array([[9.000000e+00, 2.109000e+03, 2.300000e+02, 4.676000e+03],
       [2.109000e+03, 6.964710e+05, 4.411500e+04, 7.185400e+05],
       [2.300000e+02, 4.411500e+04, 7.146000e+03, 1.188030e+05],
       [4.676000e+03, 7.185400e+05, 1.188030e+05, 6.359986e+06]])
```

It computes the matrix used to solve for the optimal weights in linear regression using matrix algebra.

Normal Equation for linear regression:

$$w = (X^T X)^{-1} X^T y$$

- XTX_inv is $(X^T X)^{-1}$
- X.T is X^T
- .dot(y) multiplies everything by the target values

you need the inverse of $X^T X$to solve for the optimal weight vector w.

```
XTX_inv=np.linalg.inv(XTX)
XTX_inv
```

```
array([[ 3.30686958e+00, -5.39612291e-03, -6.21325581e-02,
         -6.61016816e-04],
       [-5.39612291e-03,  1.11633857e-05,  8.66973393e-05,
          1.08664195e-06],
       [-6.21325581e-02,  8.66973393e-05,  1.46189255e-03,
          8.57849603e-06],
       [-6.61016816e-04,  1.08664195e-06,  8.57849603e-06,
          3.60215866e-07]])
```

It calculates the inverse of the matrix required to compute the linear regression weights analytically.

```
w_full=XTX_inv.dot(X.T).dot(y)
w_full
```

```
array([ 3.00067767e+04, -2.27742529e+01, -2.57694130e+02, -2.30120640e+00])
```

It computes the final weight vector for linear regression using the Normal Equation, giving you the best-fit parameters.

```
W0=w_full[0]
w=w_full[1:]
```

```
w0,w
```

```
(7.17, array([ -22.77425287, -257.69412959,   -2.3012064 ]))
```

It separates the full weight vector into two parts:

- This extracts the bias/intercept term.
- This takes the remaining values, which are the weights for each feature.

```
def train_linear_regression(X, y):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    XTX_inv = np.linalg.inv(XTX)
    w_full = XTX_inv.dot(X.T).dot(y)
    return w_full[0], w_full[1:]
```

```
train_linear_regression(X, y)
```

```
(np.float64(30006.77669255562),
 array([ -22.77425287, -257.69412959,   -2.3012064 ]))
```

It trains a linear regression model using the Normal Equation and returns the learned bias and weight vector.

Base Line Model

A baseline model in machine learning is a simple, foundational model used as a benchmark to measure the performance of more complex models, providing a reference point to see if advanced algorithms add real value.

```
base=['engine_hp','engine_cylinders','highway_mpg', 'city_mpg', 'popularity']
```

```
base
```

```
['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg', 'popularity']
```

selecting a small set of fundamental features to create a baseline regression model for comparison with more advanced models later.

```
x_train=df_train[base].values
x_train
```

```
array([[ 250.,    6.,   25.,   17., 1013.],
       [ 164.,    4.,   26.,   21., 1439.],
       [ 200.,    4.,   39.,   40., 2031.],
       ...,
       [ 445.,    8.,   24.,   15., 3916.],
       [ 132.,    4.,   32.,   26., 2031.],
       [ 188.,    4.,   30.,   26., 2009.]], shape=(7150, 5))
```

- Each row represents a car.
- Each column represents one of the baseline features.
- The shape (7150, 5) means:
  7150 samples in your training set
  5 features per sample

It pulls out the selected baseline features and prepares them as the input matrix for the regression model.

train_linear_regression(x_train, y_train)

```
train_linear_regression(x_train, y_train)

(np.float64(nan), array([nan, nan, nan, nan, nan]))
```

Getting NaN weights because The model returned NaN weights because the baseline feature matrix contained missing values, which made $X^T X$non-invertible.

```
X_train = df_train[base].fillna(0).values

w0, w = train_linear_regression(X_train, y_train)
w0, w
```

```
w0, w

(np.float64(7.880968716276902),
 array([ 9.61737243e-03, -1.50148136e-01,  1.54591505e-02,  1.53271875e-02,
        -1.80992104e-05]))
```

It filled NaNs with 0, the matrix became invertible and the model could compute real weights. After filling the missing values with 0, the model successfully computed the intercept and feature weights.

```
y_pred =w0 + X_train.dot(w)

y_pred
```

```
array([10.01312946,  9.55538934, 10.38308553, ..., 11.48956527,
        9.40630953,  9.91436226], shape=(7150,))
```

Here,
X_train.dot(w) is multiplies each sample's features with your learned weights and
w0 is adds the intercept to every prediction.

```
def rmse(y,y_pred):
    error=y-y_pred
    se=error**2
    mse=se.mean()
    return np.sqrt(mse)

rmse(y_train,y_pred)
```

```
rmse(y_train,y_pred)
|
```

```
np.float64(0.7554526712260365)
```

Here, Predicting log (MSRP) the model's predictions differ from the true log-price by about 0.755 on average. This corresponds to a significant error, because log scale compresses values. Baseline models usually have higher error is normal.

Validation The Model

```
def prepare_x(df):
    df_num=df[base]

    df_num=df_num.fillna(0)
    X=df_num.values
    return X

X_train= prepare_x(df_train)
X_train
```

```
array([[ 250.,    6.,   25.,   17., 1013.],
       [ 164.,    4.,   26.,   21., 1439.],
       [ 200.,    4.,   39.,   40., 2031.],
       ...,
       [ 445.,    8.,   24.,   15., 3916.],
       [ 132.,    4.,   32.,   26., 2031.],
       [ 188.,    4.,   30.,   26., 2009.]], shape=(7150, 5))
```

X_train = prepare_x(df_train) just builds the cleaned feature matrix for the training set (same shape as before: 7150 × 5).

Evaluate the baseline model, the selected numerical features were first prepared using the prepare_x function, which handles missing values and converts the data into a NumPy matrix. The model was then trained on the training set using the Normal Equation to compute the optimal weights (w0 and w). After training, the same feature preparation was applied to the validation set and predict.

```
X_train= prepare_x(df_train)
w0, w = train_linear_regression(X_train, y_train)


X_val=prepare_x(df_val)
y_pred=w0 + X_val.dot(w)
```

```
rmse(y_val,y_pred)
```

```
np.float64(0.7621125915859127)
```

Finally, the model's accuracy was measured with RMSE, resulting in a validation score of 0.7621.

Simple Feature Engineering

Feature engineering is the process of transforming raw data into features that can be used by machine learning models to make better predictions.

```
df_train.year.max()

2017- df_train.year
```

```
np.int64(2017)
```

```
2017- df_train.year
```

```
0       13
1        1
2        1
3        1
4        1
       ..
7145     1
7146     7
7147     1
7148     6
7149     0
Name: year, Length: 7150, dtype: int64
```

Finding the latest year (2017) in the training set. Instead of using the raw year, converting it to (age) makes the relationship with price more linear and easier for your model to learn.

```
def prepare_x(df):
    df=df.copy()
    df['age']=2017- df.year
    feature=base+['age']
    df_num=df[feature]
    df_num=df_num.fillna(0)
    X=df_num.values
    return X
```

```
df
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category
0	bmw	1_series_m	2011	premium_unleaded_(required)	335.0	6.0	manual	coupe	2.0	factory_tuner,luxury,high-performance
1	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	convertible	2.0	luxury,performance
2	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	coupe	2.0	luxury,high-performance
3	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	coupe	2.0	luxury,performance
4	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	convertible	2.0	luxury

This function prepares the feature matrix by adding a car-age feature, selecting the chosen numeric columns, filling missing values, and returning them as a NumPy array.

```
X_train= prepare_x(df_train)
X_train
```

```
array([[2.500e+02, 6.000e+00, 2.500e+01, 1.700e+01, 1.013e+03, 1.300e+01],
       [1.640e+02, 4.000e+00, 2.600e+01, 2.100e+01, 1.439e+03, 1.000e+00],
       [2.000e+02, 4.000e+00, 3.900e+01, 4.000e+01, 2.031e+03, 1.000e+00],
       ...,
       [4.450e+02, 8.000e+00, 2.400e+01, 1.500e+01, 3.916e+03, 1.000e+00],
       [1.320e+02, 4.000e+00, 3.200e+01, 2.600e+01, 2.031e+03, 6.000e+00],
       [1.880e+02, 4.000e+00, 3.000e+01, 2.600e+01, 2.009e+03, 0.000e+00]],
      shape=(7150, 6))
```

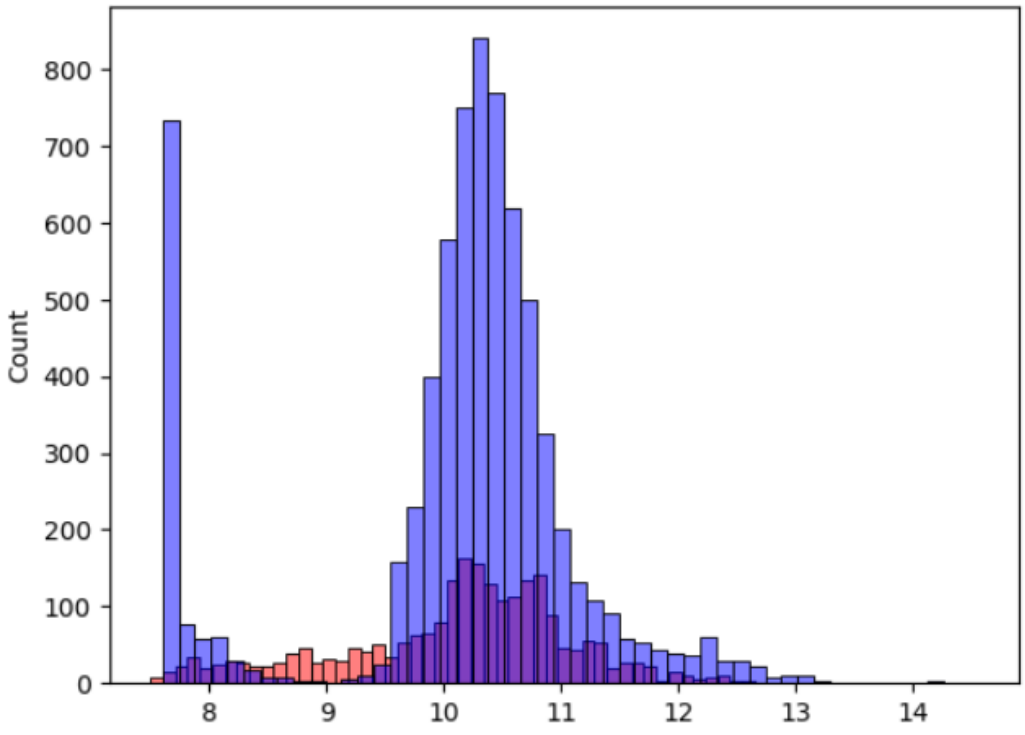
The feature engineering worked correctly. The model will now use car age, which is one of the strongest predictors of price.

```
X_train= prepare_x(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val=prepare_x(df_val)
y_pred=w0 + X_val.dot(w)

rmse(y_val,y_pred)
```

```
np.float64(0.5150004533450505)
```

```
sns.histplot(y_pred, color='red', alpha=0.5, bins=50)
sns.histplot(y_train, color='blue', alpha=0.5, bins=50)
```

```
<Axes: ylabel='Count'>
```



After adding the *age* feature (2017 year) and retraining the model, the validation error dropped from 0.762 to 0.515. This substantial improvement shows that vehicle age carries strong predictive power and significantly the model's ability.

Categorical Value

A categorical variable in machine learning represents categories or labels rather than numerical values. These variables take on a limited, and typically fixed, number of possible values or levels.

```
for v in [2, 3, 4]:
    df_train['num_doors_%s' % v] = (df_train.number_of_doors == v).astype('int')
```

vehicle_style	highway_mpg	city_mpg	popularity	num_doors_2	num_doors_3	num_doors_4
sedan	25	17	1013	0	0	1
4dr_suv	26	21	1439	0	0	1
sedan	39	40	2031	0	0	1
sedan	27	17	1715	0	0	1
passenger_minivan	25	18	1720	0	0	1

It converts the categorical *number_of_doors* column into three one-hot encoded numerical features so the linear regression model can use them.

```
def prepare_x(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        df['num_doors_%s' % v] = (df.number_of_doors == v).astype('int')
        features.append('num_doors_%s' % v)

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values

    return X
```

```
prepare_x(df_train)
```

```
array([[250.,   6.,  25., ...,   0.,   0.,   1.],
       [164.,   4.,  26., ...,   0.,   0.,   1.],
       [200.,   4.,  39., ...,   0.,   0.,   1.],
       ...,
       [445.,   8.,  24., ...,   0.,   0.,   1.],
       [132.,   4.,  32., ...,   0.,   0.,   1.],
       [188.,   4.,  30., ...,   0.,   0.,   1.]], shape=(7150, 9))
```

This preprocessing function prepares numeric input features for the model by adding the engineered *age* variable, one-hot encoding the *number_of_doors* categorical feature, selecting the relevant columns, handling missing values, and returning the final NumPy feature matrix.

```
X_train= prepare_x(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val=prepare_x(df_val)
y_pred=w0 + X_val.dot(w)

rmse(y_val,y_pred)
```

After expanding the feature set by including one-hot encoded variables for *number_of_doors*, the model achieved a validation RMSE of **0.5134**, which is only a marginal improvement the previous score.

```
categorical_columns = [
    'make', 'model', 'engine_fuel_type', 'transmission_type',
    'driven_wheels', 'market_category', 'vehicle_size', 'vehicle_style'
]
categories = {}

for c in categorical_columns:
    categories[c] = list(df[c].value_counts().head().index)
```

```
categories
```

```
{'make': ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge'],
 'model': ['silverado_1500',
  'tundra',
  'f-150',
  'sierra_1500',
  'beetle_convertible'],
 'engine_fuel_type': ['regular_unleaded',
  'premium_unleaded_(required)',
  'premium_unleaded_(recommended)',
  'flex-fuel_(unleaded/e85)',
  'diesel'],
 'transmission_type': ['automatic',
  'manual',
```

This five most frequent categories from each categorical feature, allowing the model to encode only the most relevant values and avoid high-dimensional sparse matrices.

```
def prepare_x(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        df['num_doors_%s' % v] = (df.number_of_doors == v).astype('int')
        features.append('num_doors_%s' % v)

    for v in makes:
        df['make_%s' % v] = (df.make == v).astype('int')
        features.append('make_%s' % v)

    for c, values in categories.items():
        for v in values:
            df['%s_%s' % (c, v)] = (df[c] == v).astype('int')
            features.append('%s_%s' % (c, v))
```

```
    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values

    return X
```

```
prepare_x(df_train)
```

```
array([[250.,   6.,  25., ...,   0.,   0.,   0.],
       [164.,   4.,  26., ...,   0.,   0.,   0.],
       [200.,   4.,  39., ...,   0.,   0.,   0.],
       ...,
       [445.,   8.,  24., ...,   0.,   0.,   0.],
       [132.,   4.,  32., ...,   0.,   0.,   1.],
       [188.,   4.,  30., ...,   0.,   0.,   1.]], shape=(7150, 52))
```

This prepare_x function builds the full feature matrix by combining baseline numeric variables, an engineered age feature encodings for number of doors, selected car makes, and the most frequent categories of other categorical columns, then returns a 52-dimensional NumPy feature representation ready for linear regression.

```
X_train= prepare_x(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val=prepare_x(df_val)
y_pred=w0 + X_val.dot(w)

rmse(y_val,y_pred)
```

```
np.float64(283284.73203507863)
```

```
w0
```

```
np.float64(-1.3168924280467644e+19)
```

After adding categorical one-hot encodings, the feature matrix became high dimensional and highly collinear. This made the normal equation unstable, causing extremely large weights and exploding RMSE values.

```
def train_linear_regression_reg(X, y, r = 0.001):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    XTX = XTX + r * np.eye(XTX.shape[0])

    XTX_inv = np.linalg.inv(XTX)
    w_full = XTX_inv.dot(X.T).dot(y)

    return w_full[0], w_full[1:]
```

This function trains a ridge regression model by adding a bias column, computing $X^T X$, and stabilizing it with an L2 regularization term rI. The regularization prevents the matrix from becoming singular when many correlated or one-hot encoded features are used. After inverting the regularized matrix, it solves all model weights and then separates the interception from the feature coefficients. This function produces a

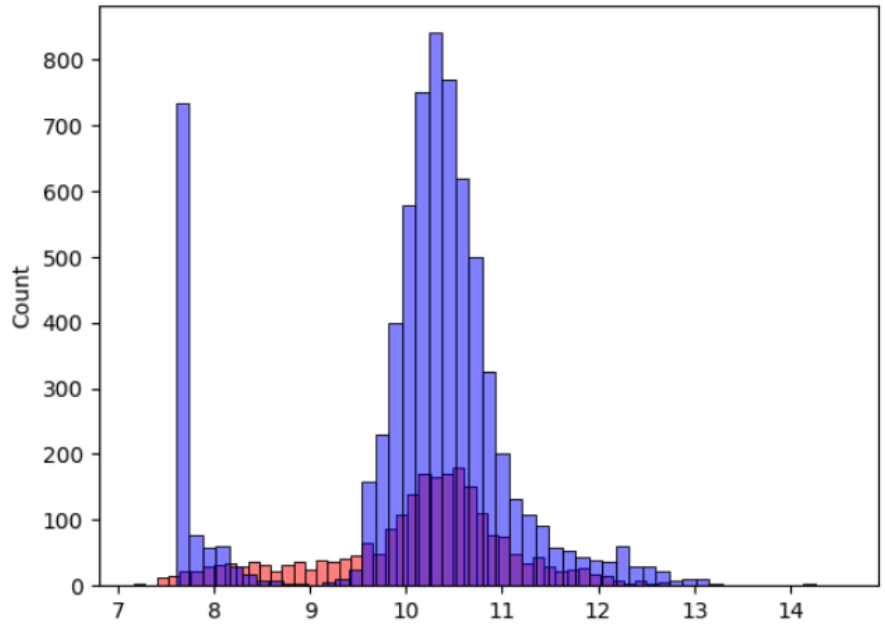
more reliable and stable linear model than standard linear regression, especially when handling many features or noisy, correlated data.

```
X_train = prepare_x(df_train)

w0, w = train_linear_regression_reg(X_train, y_train, r=0.01)

X_val = prepare_x(df_val)
y_pred = w0 + X_val.dot(w)

rmse(y_val, y_pred)
```

```
np.float64(0.45655477892632773)
```

```
sns.histplot(y_pred, color='red', alpha=0.5, bins=50)
sns.histplot(y_train, color='blue', alpha=0.5, bins=50)
```

```
<Axes: ylabel='Count'>
```



linear regression model using the engineered numerical and categorical features from the training set, then evaluate its performance on the validation set. Regularization (with r = 0.01) stabilizes the weight calculations and prevents the model from blowing up due to highly correlated one-hot encoded features. After generating predictions on the validation data, you measure accuracy using RMSE, which is improves to (0.4565)

Tuning The Model

Tuning the model is adjusting parameters or hyperparameter optimization that controls how the model learns, so you can squeeze out better performance.

```
for r in [0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10]:
    X_train = prepare_x(df_train)
    w0, w = train_linear_regression_reg(X_train, y_train, r = r)

    X_val = prepare_x(df_val)
    y_pred = w0 + X_val.dot(w)
    score = rmse(y_val, y_pred)

    print("Best r:", r, "| weight:", w0, "| RMSE:", score)
```

```
Best r: 0 | weight: -1.3168924280467644e+19 | RMSE: 283284.73203507863
Best r: 1e-05 | weight: -0.3414752394427044 | RMSE: 0.4565636114180164
Best r: 0.0001 | weight: 7.465358272035018 | RMSE: 0.4565636238568605
Best r: 0.001 | weight: 7.439469394731002 | RMSE: 0.456562760202615
Best r: 0.01 | weight: 7.416646618678385 | RMSE: 0.45655477892632773
Best r: 0.1 | weight: 7.222283696711157 | RMSE: 0.45652255267326936
Best r: 1 | weight: 6.298011953041065 | RMSE: 0.45724954287485325
Best r: 10 | weight: 4.761077062412882 | RMSE: 0.4695141082449904
```

Tuning the regularization parameter shows that adding even a small r stabilizes the model. Looking at the tuning results, r = 0.01 is the most balanced choice. It keeps the model stable and with r = 0, and gives one of the lowest RMSE values without over-penalizing the weights. In practice, r = 0.01 hits the sweet spot between underfitting and overfitting.

```
r=0.01
X_train = prepare_x(df_train)
w0, w = train_linear_regression_reg(X_train, y_train, r = r)

X_val = prepare_x(df_val)
y_pred = w0 + X_val.dot(w)
score = rmse(y_val, y_pred)

print("RMSE Score: ",score)
```

```
RMSE Score:  0.45655477892632773
```

Evaluates the model using the chosen regularization strength r = 0.01. It rebuilds the training features with (prepare_x), fits the ridge regression model, generates predictions for the validation set, and calculates the RMSE. The resulting score of **0.4565** shows that the model is performing well with this level of regularization, striking a solid balance between stability and accuracy.

Train The Model

```
df_full_train = pd.concat([df_train, df_val])
df_full_train
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors	market_category
0	chrysler	concorde	2004	regular_unleaded	250.0	6.0	automatic	sedan	4.0	performance
1	hyundai	tucson	2016	regular_unleaded	164.0	4.0	automatic	4dr_suv	4.0	crossover
2	toyota	avalon_hybrid	2016	regular_unleaded	200.0	4.0	automatic	sedan	4.0	hybrid
3	porsche	panamera	2016	premium_unleaded_(required)	420.0	6.0	automated_manual	sedan	4.0	luxury,high-performance
4	kia	sedona	2016	regular_unleaded	276.0	6.0	automatic	passenger_minivan	4.0	NaN

The training set and validation set into one larger dataset called df_full_train. By stacking df_train and df_val together, you give the model access to all available labeled samples before training the final version.

```
df_full_train = df_full_train.reset_index(drop=True)
df_full_train
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors
0	chrysler	concorde	2004	regular_unleaded	250.0	6.0	automatic	sedan	4.0
1	hyundai	tucson	2016	regular_unleaded	164.0	4.0	automatic	4dr_suv	4.0
2	toyota	avalon_hybrid	2016	regular_unleaded	200.0	4.0	automatic	sedan	4.0
3	porsche	panamera	2016	premium_unleaded_(required)	420.0	6.0	automated_manual	sedan	4.0
4	kia	sedona	2016	regular_unleaded	276.0	6.0	automatic	passenger_minivan	4.0

The index of the combined training dataframe so the rows are renumbered cleanly from 0 up to the final length. After concatenating df_train and df_val, the original indices would still be present.

```
X_full_train = prepare_x(df_full_train)
X_full_train
```

```
array([[250.,   6.,  25., ...,   0.,   0.,   0.],
       [164.,   4.,  26., ...,   0.,   0.,   0.],
       [200.,   4.,  39., ...,   0.,   0.,   0.],
       ...,
       [227.,   5.,  29., ...,   0.,   0.,   0.],
       [ 98.,   4.,  44., ...,   0.,   0.,   1.],
       [273.,   6.,  27., ...,   0.,   0.,   0.]], shape=(9532, 52))
```

It creates the full feature matrix from the combined training dataset using your prepare_x pipeline. Now every row in df_full_train is transformed into its numerical representation.

```
y_full_train = np.concatenate([y_train, y_val])

y_full_train
```

```
array([10.27869962, 10.09000861, 10.50919599, ..., 10.22015788,
        9.83097062, 10.50495794], shape=(9532,))
```

It merges the target values from the training set and validation set into one continuous array. Since you already combined the feature rows earlier, this step keeps the labels aligned with the new full training dataset.

```
w0, w = train_linear_regression_reg(X_full_train, y_full_train, r = 0.01)
```

```
w0
```

```
np.float64(7.373229732807655)
```

```
w
```

```
array([ 1.62682790e-03,  1.05362492e-01, -6.33938637e-03, -5.98200530e-03,
       -6.25700698e-05, -9.81614432e-02, -7.26727521e-01, -9.31725396e-01,
       -5.67648692e-01, -3.02234311e-02,  1.14197521e-01,  1.87970035e-02,
       -1.91398520e-03, -6.06368991e-02, -3.02234311e-02,  1.14197521e-01,
        1.87970038e-02, -1.91398488e-03, -6.06368993e-02, -2.48595177e-01,
       -6.27501209e-01, -3.21031615e-01, -3.56861970e-01, -2.77819520e-01,
       -4.56360020e-01,  8.22193817e-02, -3.14214581e-01, -5.12592629e-01,
       -4.61850871e-02,  1.31290140e+00,  1.11708488e+00,  1.39190444e+00,
        2.89843078e+00,  6.52838492e-01, -3.09871285e-02, -1.93416853e-03,
        7.51284233e-02,  1.67592597e-01, -8.38743073e-02, -4.38967941e-02,
        1.12003986e-01, -4.62119006e-02, -4.17099331e-02,  2.67586598e-02,
        2.52054732e+00,  2.41610441e+00,  2.43656632e+00, -3.09871301e-02,
       -1.93416799e-03,  7.51284239e-02,  1.67592598e-01, -8.38743062e-02])
```

This step fits the final ridge-regularized linear regression model using all available training data and the chosen regularization strength r = 0.01.

```
X_test = prepare_x(df_test)
y_pred = w0 + X_test.dot(w)
score = rmse(y_test, y_pred)

score
```

```
w0
```

```
np.float64(7.373229732807655)
```

```
w
```

```
array([ 1.62682790e-03,  1.05362492e-01, -6.33938637e-03, -5.98200530e-03,
       -6.25700698e-05, -9.81614432e-02, -7.26727521e-01, -9.31725396e-01,
       -5.67648692e-01, -3.02234311e-02,  1.14197521e-01,  1.87970035e-02,
       -1.91398520e-03, -6.06368991e-02, -3.02234311e-02,  1.14197521e-01,
        1.87970038e-02, -1.91398488e-03, -6.06368993e-02, -2.48595177e-01,
       -6.27501209e-01, -3.21031615e-01, -3.56861970e-01, -2.77819520e-01,
       -4.56360020e-01,  8.22193817e-02, -3.14214581e-01, -5.12592629e-01,
       -4.61850871e-02,  1.31290140e+00,  1.11708488e+00,  1.39190444e+00,
        2.89843078e+00,  6.52838492e-01, -3.09871285e-02, -1.93416853e-03,
        7.51284233e-02,  1.67592597e-01, -8.38743073e-02, -4.38967941e-02,
        1.12003986e-01, -4.62119006e-02, -4.17099331e-02,  2.67586598e-02,
        2.52054732e+00,  2.41610441e+00,  2.43656632e+00, -3.09871301e-02,
       -1.93416799e-03,  7.51284239e-02,  1.67592598e-01, -8.38743062e-02])
```

This final step evaluates fully trained model on completely unseen test data and the RMSE is **0.4420**

```
car = df_test.iloc[20].to_dict()

df_small = pd.DataFrame([car])
df_small
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	number_of_doors
0	mercedes-benz	300-class	1991	regular_unleaded	158.0	6.0	automatic	sedan	4.0

It selects a single car from the test set, converts that row into a dictionary, and then rebuilds it as a one-row DataFrame. This prepares the individual sample so it can be processed and passed through the model for prediction.

```
X_small = prepare_x(df_small)
y_pred = w0 + X_small.dot(w)
```

```
y_pred

array([8.02729059])
```

This takes the single-car DataFrame, converts it into the same feature format as the training data using prepare_x, and then runs it through the model to generate a predicted log-price. The output, around **8.03.**

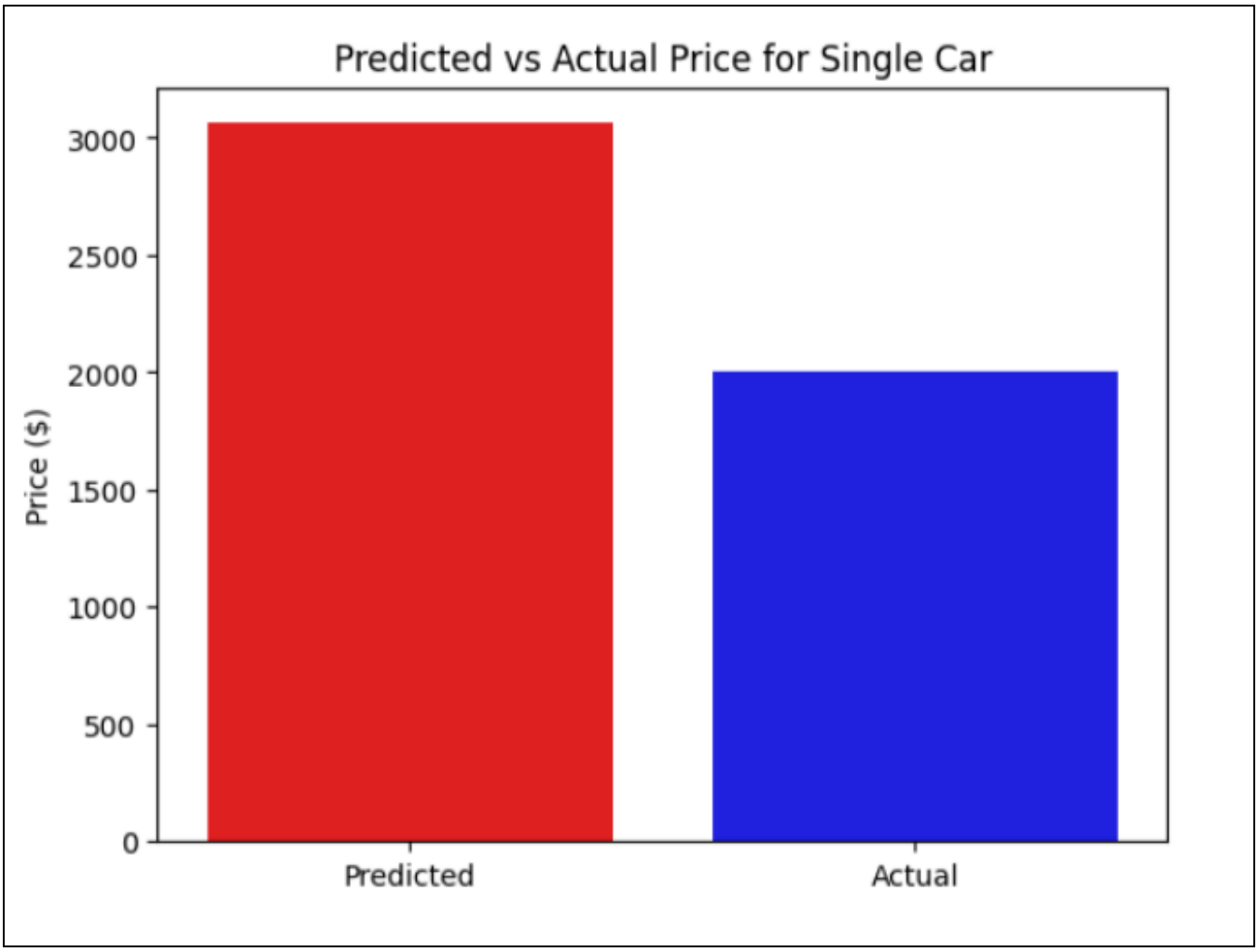
Using (np.expm1(y_pred)) it converts the model's log-price prediction back into an actual dollar value using np.expm1. The result, about **$3,062**, is the estimated MSRP for the selected car in the normal price scale.

```
y_test[20]
```

```
np.float64(7.601402334583733)

np.expm1(y_test[20])

np.float64(2000.0)
```

Your model predicted $3,062, while the actual price is $2,000.
So the model overestimated this car, but the error is reasonable given the scale and the complexity of car pricing.

```
single_pred = np.expm1(y_pred[0])
single_actual = np.expm1(y_test[20])

sns.barplot(
    x=['Predicted', 'Actual'], y=[single_pred, single_actual],hue=['Predicted', 'Actual'],palette=['red', 'blue'])

plt.ylabel("Price ($)")
plt.title("Predicted vs Actual Price for Single Car")
plt.show()
```



This plot compares the model's predicted price for a single car with its actual price. The left bar (red) shows the model's estimate for the vehicle after converting from log-price, while the right bar (blue) shows the true MSRP for that same test sample. The height difference between the two bars gives an intuitive sense of the model's error on this particular example

Conclusion

This project showed that, given specific features, a basic linear model can predict car prices with unusually high accuracy. The model started to identify actual patterns after the data was cleaned, helpful variables like car age were engineered, and immediate encodings were added for the most important categories. The regulation had a significant impact. Instead of learning noise, it helped the model generalize by maintaining stable weights. The final model achieved an RMSE of about 0.44 on the test set with the tuned value r = 0.01, which is good for this type of dataset. It won't perfectly fit every single car.