CM3005 Data Science

Midterm Paper

Student ID: 220516639
Date of Submission: 6th January 2025

Data from: https://www.kaggle.com/code/refiaozturk/predicting-car-prices-using-ml-regression-models/notebook

1. Domain-specific area and objectives

This project focuses on price anaysis of new and used cars. The domain lies in the automobile industry, which saw a global car sales of 75.3 million units in 2023. [1]

This project is motivated by the assymetry of information within the used car market, where prices of cars are not immediately transparent to consumers. Unlike other consumer goods, the price of a used car is subjected to personal negotiation. Car dealers hold a disproportionate amount of knowledge and bargaining power as they oftentimes determine the prices in a seemingly arbitrary manner. [2]

As such, this project aims to create a linear regression model that can accurately predict prices of used cars. An accurate model can objectively determine a car's price based on its attributes, and provide buyers with a baseline of how much they should actually pay for a used car.

The objectives of this project are as follows:

- To develop and evaluate a linear regression model to predict the price of used cars based on features such as their brand, model and make.
- To access the model's accuracy using metrics such as Mean Squared Error
- To analyse the linear regression model's limitations and detail potential areas for improvement.

This research will contribute to the automotive industry by empowering consumers with accurate predictive tools to negotiate for better deals and make informed buying decisions. This will democratise the knowledge of car prices and result in a more transparent market. The insights gained will allow consumers to understand factors that drive car prices and determine which car best fits their needs and budget constraints.

2. Dataset Description

This section introduces the dataset used in this project. It contains real-world car pricing data on used cars. This dataset will be used to build a linear regression model that uses different attributes of a car to predict its pricing.

The dataset contains 15,915 rows and 23 columns, of which 22 are independent variables and 1 is dependent (price).

Dataset Origin

The dataset was obtained from the AutoScout24 pan-European online car marketplace by Yaşar Yiğit Turan in 2024. I accessed this dataset through Kaggle and downloaded it onto my main drive as a CSV file. [3]

Dataset attribution

The dataset is licensed under CC BY-SA 4.0, which makes it free to share and adapt, as long as appropriate credit is given. [4] Contributions to the dataset through this project falls under the same license. After the project ends, the data will be deleted.

Suitability

The sample size is sufficient for the regression model to be generalisable and reliable. While the pre-processing steps in Section 3 revealed missing data and some outliers, removal of such anomalies ensured that the regression model is accurate and reliable.

Fitness for Linear Regression
The dataset is fit for linear regression models as the label (price) is a continuous variable. The dataset, which originates from the largest pan-European car marketplace, is a reliable reflection of real-world car prices. In addition, the features are relevant to the label.

However, some features, as discussed below, are categorical and must be pre-processed before they can be used meaningfully.

Features

There are 22 features - 13 categorical, 4 numerical (discrete) and 5 numerical continuous.

Data Type	Feature Name	Data description
Categorical	Make Model	Model and brand
Categorical	Body Type	Sedan, station wagon, etc.
Categorical	VAT status	Whether it is Value Added Tax deductible

Data Type	Feature Name	Data description
Categorical	Used or New	Whether it is a secondhand car
Categorical	Type of fuel	Whether it uses Benzine or Diesel
Categorical	Convenience features	In-builtcomfort features it has, such as air conditioning
Categorical	Entertainment features	In-built entertainment features it has, such as radio
Categorical	Extra features	Auxiliary features
Categorical	Safety features	Features to ensure safety of passengers
Categorical	Paint type	Matte or metallic
Categorical	Interior upholstery type	Leather or cloth
Categorical	Transmission (gearing) type	Manual or auto
Categorical	Drivetrain	Front or Rear wheel drive
Numerical (Discrete)	Number of gears	Depends on whether it is manual, automatic, heavy
Numerical (Discrete)	Age	How old the car is based on its model
Numerical (Discrete)	Previous owners	How many owners it had
Numerical (Discrete)	Inspection status	Whether it has been inspected recently
Numerical (Continuous)	Total mileage (km)	Usage of the car
Numerical (Continuous)	Engine power (kW)	Indicates performance of car
Numerical (Continuous)	Engine displacement	Size of engine
Numerical (Continuous)	Weight (kg)	Weight of the vehicle
Numerical (Continuous)	Combined fuel consumption (litres/100km)	Determines usage of fuel

Labels

The label used for the linear regression is the price of the car.

Data Type	Label	Data description
Numerical (Continuous)	Price	Listed price of the car

3. Data preparation

In this section, we will prepare the dataset. Data pre-processing involve cleaning the dataset, converting it into First Normal Form and removing outliers. Through this steps, we can ensure that it is suitable for linear regression model.

In [1]:
```
"""
IMPORT LIBRARIES FOR DATA CLEANING
"""
import pandas as pd
import numpy as np
import math

"""
IMPORT LIBRARIES FOR DATA VISUALISATION
"""
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter
sns.set_style("white")
```

The file is stored as a comma-seperated values (CSV) file in the same directory as this ipynb notebook. The data is read into a Pandas DataFrame with the 'read_csv' method, for the ease of data manipulation and processing.

In [2]:
```
# import dataset, opens and closes the file automatically
df = pd.read_csv('car_prices.csv')

# explore the rows and columns of the dataset
df.shape
```

Out[2]:
```
(15915, 23)
```

There are 15,915 rows in the dataset and 23 columns.

3.1 Investigation of dataset

To understand the nature of the dataset, we call the 'head()' method on the DataFrame.

In [3]:
```
df.columns = df.columns.str.lower() # change all column names to lower-ca
df.head(3)
```

Out[3]:

	make_model	body_type	price	vat	km	type	fuel	gears	
0	Audi A1	Sedans	15770.0	VAT deductible	56013.0	Used	Diesel	7	cc
1	Audi A1	Sedans	14500.0	Price negotiable	80000.0	Used	Benzine	7	
2	Audi A1	Sedans	14640.0	VAT deductible	83450.0	Used	Diesel	7	

3 rows × 23 columns

Initial Commentary

As seen above, the table is not in First Normal Form, as columns such as 'Comfort_Convenience' and 'Entertainment_media' contains multiple values. Multi-valued cells will impede data analysis and data processing. As such, normalisation of the dataset must be performed. In addition, there is a mix of categorical and numerical data.

3.2 Imputation of missing data

Check for null values In the code block below, we will check for missing values. NaN and null values in input datasets pose a number of problems for machine learning - it reduces model accuracy, creates extra noise and make the data imcompatible with ML algorithms. [5] As a result, we must handle these values in our data wrangling process.

In [4]:
```
# check for null values
null_values = df.isnull().sum()
na_values = df.isna().sum()

# concatenate the two columns and transpose it for better readability
table = pd.concat([null_values,na_values],axis = 1, keys = ['Null', 'Na'])
table
```

Out[4]:

	make_model	body_type	price	vat	km	type	fuel	gears	comfort_convenie
Null	0	0	4	0	0	0	0	0	
Na	0	0	4	0	0	0	0	0	

2 rows × 23 columns

In the two rows above, we can see that there are a few missing values in the dataset. We have to handle these missing data as part of the feature engineering process. First, we split the data set into a Pandas DataFrame that contains numerical data ('num_df'), and a Pandas DataFrame that contains categorical data ('cat_df'). [6] [7] Numerical data and categorical data have to be dealt with separately with different imputation methods [8]

In [5]:
```
num_df = df.select_dtypes('number')
cat_df = df.select_dtypes(exclude = ['number'])
```

We will import **SimpleImputer** from the Scikit Learn Impute library.

With SimpleImputer, we can replace missing numerical values with the mean of the column. For categorical data, we will use the 'most_frequent' streategy to replace missing values with the most frequently occurring value. An alternative that was considered would be using a KNNImputer. [9] However, it is computationally expensive and would not be used in this project.

In [6]:
```
from sklearn.impute import SimpleImputer
from sklearn.impute import KNNImputer

# create two separate imputers for numerical and categorical data

num_impute = SimpleImputer(strategy = 'mean')
cat_impute = SimpleImputer(strategy = 'most_frequent')
```

In [7]:
```
# perform imputation and save the data as a separate DataFrame
df_num_cleaned = pd.DataFrame(num_impute.fit_transform(num_df), columns =
df_cat_cleaned = pd.DataFrame(cat_impute.fit_transform(cat_df), columns =
```

In [8]:
```
# combine both DataFrames again into one DataFrame
df_imputed = pd.concat([df_num_cleaned, df_cat_cleaned], axis = 1)
```

The 'df_imputed' dataframe will contain no null and no na values. We can check this again with df_imputed.isnull().sum() and df_imputed.isna().sum().

3.3 Dropping duplicated rows

Check for duplicated rows

Duplicate entries will impede the performance of a linear regression model as it can increase the weight of observations while training the machine-learning model. [10] As such, these entries must be removed.

In [9]:
```
print(f"Rows: {df_imputed.shape[0]}, Duplicated: {df_imputed.duplicated()
```

```
Rows: 15915, Duplicated: 1673
```

There are 11,836 rows in the dataset, of which 220 are duplicated.

In [10]:
```
df_imputed.drop_duplicates(inplace = True) # drop duplicated rows in plac
print(f"Rows left: {df_imputed.shape[0]}")
```

```
Rows left: 14242
```

After dropping duplicated observations, there are 11,616 rows left.

In [11]:
```
pd.set_option('display.float_format', lambda float_value: '%.2f' % float_
df_imputed.describe()
```

Out[11]:

	price	km	gears	age	previous_owners	hp_kw	inspe
count	14242.00	14242.00	14242.00	14242.00	14242.00	14242.00	
mean	18099.70	32582.11	5.94	1.41	1.04	88.71	
std	7418.81	36856.86	0.70	1.11	0.34	26.55	
min	4950.00	0.00	5.00	0.00	0.00	40.00	
25%	12950.00	3898.00	5.00	0.00	1.00	66.00	
50%	16950.00	21000.00	6.00	1.00	1.00	85.00	
75%	21900.00	47000.00	6.00	2.00	1.00	103.00	
max	74600.00	317000.00	8.00	3.00	4.00	294.00	

The Pandas Dataframe method df.describe() gives a description of the numerical data in the dataframe. This allows us to explore how different features might contribute to the model. Most notably, we can see that the values are of different ranges. This implies that we may have to scale the data to ensure that all features contribute equally for a more accurate model.

3.4 Removal of outliers

In the code block below, we remove outliers in the data. Outliers must be removed as they can skew results of statistical analysis, and introduce bias to the machine learning model. We set the cut off points as 3 standard deviation away from the mean. Values that are above or below this threshhold are removed.

In [12]:
```
num_df = df_imputed.select_dtypes('number')

for col in num_df.columns:
    mean, median, std = num_df[col].mean(), num_df[col].median(), num_df[

    # identify outliers as more than 3 standard deviation from mean
    cut_off = std * 3
    lower_bound, upper_bound = mean - cut_off, mean + cut_off

    to_drop = df_imputed[(df_imputed[col] <= lower_bound) | (df_imputed[c

    df_imputed.drop(index = to_drop, inplace = True)
```

3.5 Label-encoding and converting data to 1NF

In [13]:
```
# to learn more about the categorical data in the dataset
df_imputed.describe(include = ["O"]).T
```

Out[13]:

	count	unique	top	freq
make_model	13242	8	Audi A3	2498
body_type	13242	8	Sedans	6805
vat	13242	2	VAT deductible	12462
type	13242	5	Used	9697
fuel	13242	4	Benzine	7084
comfort_convenience	13242	5789	Air conditioning,Electrical side mirrors,Hill ...	302
entertainment_media	13242	339	Bluetooth,Hands-free equipment,On-board comput...	1496
extras	13242	634	Alloy wheels	4658
safety_security	13242	4122	ABS,Central door lock,Daytime running lights,D...	617
paint_type	13242	3	Metallic	12717
upholstery_type	13242	2	Cloth	10261
gearing_type	13242	3	Manual	6683
drive_chain	13242	3	front	13135

We can observe that there are 13 categorical features in the dataset. However, linear regression models captures the relationship between a dependent variable and independent variables, which are assumed to be continuous. This necessitates processing of categorical features so that we can meaningfully use it in our linear regression model. [11]

In this project, we will use the LabelEncoder class from ScikitLearn's Preprocessing library to encode categorical data into values between 0 and n_classes-1. [12] This process will also convert the data into the First Normal Form (1NF).

In addition, we can see that four columns - 'comfort_convenience', 'entertainment_media', 'extras' and 'safety_security' - have hundreds and thousands of unique values. Converting these variables into indicator variables would increase the number of features into the thousands. This would harm the learning algorithm as it latched onto random noise, which leads to overfitting.

Hence, we will drop these columns from the DataFrame before performing conversion with LabelEncoder.

In [14]:
```
to_drop = ['safety_security','extras','entertainment_media','comfort_conv

df_imputed.drop(columns = to_drop, inplace = True)
```

In the code block below, we create LabelEncoder objects for the different features. By converting these categorical columns into numerical columns, we can ensure that they can be fitted with the linear regression model.

In [15]:
```
from sklearn.preprocessing import LabelEncoder
make_encoder = LabelEncoder()
body_type_encoder = LabelEncoder()
vat_encoder = LabelEncoder()
usage_type_encoder = LabelEncoder()
fuel_encoder = LabelEncoder()
paint_type_encoder = LabelEncoder()
upholstery_type_encoder = LabelEncoder()
gearing_type_encoder = LabelEncoder()
drive_chain_type_encoder = LabelEncoder()
```

We encode the target labels with the values. To inspect the value the label is encoded to, we can call the method '.classes_' on the encoder.

In [16]:
```
df_imputed['make_model'] = make_encoder.fit_transform(df_imputed['make_mo
df_imputed['body_type'] = body_type_encoder.fit_transform(df_imputed['bod
df_imputed['vat'] = vat_encoder.fit_transform(df_imputed['vat'])
df_imputed['type'] = usage_type_encoder.fit_transform(df_imputed['vat'])
df_imputed['fuel'] = fuel_encoder.fit_transform(df_imputed['fuel'])
df_imputed['paint_type'] = paint_type_encoder.fit_transform(df_imputed['p
df_imputed['upholstery_type'] = upholstery_type_encoder.fit_transform(df_
df_imputed['gearing_type'] = gearing_type_encoder.fit_transform(df_impute
df_imputed['drive_chain'] = drive_chain_type_encoder.fit_transform(df_imp
```

In [17]:
```
df_imputed.head()
```

Out[17]:

	price	km	gears	age	previous_owners	hp_kw	inspection_new	dis
0	15770.00	56013.00	7.00	3.00	2.00	66.00	1.00	
1	14500.00	80000.00	7.00	2.00	1.00	141.00	0.00	
2	14640.00	83450.00	7.00	3.00	1.00	85.00	0.00	
3	14500.00	73000.00	6.00	3.00	1.00	66.00	0.00	
4	18023.25	16200.00	7.00	3.00	1.00	66.00	1.00	

As seen above, the categorical data such as 'make model' and 'body type' has been transformed into a numerical variable that can meaningfully capture the feature, while making it usable for our linear regression model.

3.6 Dealing with multicollinearity

Intuitively, some of the features may be correlated - for example, age will be positively correlated with the mileage of the car, as an older car will be more heavily used. In addition, features such as the engine power and displacement is also dependent on the make and body type of the vehicle.

In the following code, we will explore the multicollinearity of the dataset using the variance inflation factor (VIF). VIF is a tool to detect multicollinearity - where multiple independent variables have a high correlation. Values of 5 of higher are considered high and should be dealt with. [13]

In [18]:
```
from statsmodels.stats.outliers_influence import variance_inflation_facto
from statsmodels.tools.tools import add_constant

X = df_imputed.drop(columns = ['price'])
X = add_constant(X)

vif_data = pd.DataFrame()
vif_data['Feature'] = X.columns
vif_data['VIF'] = [variance_inflation_factor(X.values, i) for i in range(
print(vif_data)
```

```
            Feature    VIF
0             const 461.71
1                km   2.72
2             gears   1.94
3               age   2.79
4   previous_owners   1.10
5             hp_kw   3.49
6    inspection_new   1.07
7   displacement_cc   3.54
8         weight_kg   2.56
9         cons_comb   2.87
10       make_model   1.84
11        body_type   1.61
12              vat    inf
13             type    inf
14             fuel   3.91
15       paint_type   1.03
16  upholstery_type   1.38
17     gearing_type   1.49
18      drive_chain   1.05
```

```
/opt/anaconda3/lib/python3.12/site-packages/statsmodels/stats/outliers_inf
luence.py:197: RuntimeWarning: divide by zero encountered in scalar divide
  vif = 1. / (1. - r_squared_i)
```

In [19]:
```
df_full = df_imputed.drop(columns = ['vat','type'])
```

With the code above, we remove the columns 'vat' and 'type' to reduce multicollinearity within our features. This, as part of feature engineering and pre-processing, will ensure that our model can accurrately access effect of independent variables and ensure that the model does not capture noise instead of underlying patterns.

4. Statistical Analysis

In this section, we will perform univariate analysis of the dataset.

4.1 Measures of Central Tendancy

In [20]:
```
df_full.describe()
```

Out[20]:

	price	km	gears	age	previous_owners	hp_kw	inspe
count	13242.00	13242.00	13242.00	13242.00	13242.00	13242.00	
mean	17774.06	31233.04	5.93	1.44	1.07	87.54	
std	6723.77	32452.86	0.71	1.09	0.26	24.77	
min	4990.00	1.00	5.00	0.00	1.00	40.00	
25%	12971.00	5000.00	5.00	1.00	1.00	66.00	
50%	16900.00	21724.50	6.00	1.00	1.00	85.00	
75%	21500.00	45800.00	6.00	2.00	1.00	100.00	
max	39990.00	143098.00	8.00	3.00	2.00	168.00	

Using the describe method on our Pandas DataFrame, we can see a summary of central dependency and dispersion for our numeric columns.

We can see that all columns have 13,242 counts, which corresponds to the number of rows we have in the dataset.

The mean between the different features vary greatly - for example, the mean price is 17,000 while the mean number of gears is 5.93. This highlights the need for scaling, which would be covered later in the project in feature engineering (Section 7).

The standard deviation (std) indicates the variability in data. Most notably, km has a high standard deviation of 32,452. This suggests that the data is spread out from the mean, and indicates a high variance in the mileage of the cars.

The 50th percentile (50%) shows the median of the values. We can see that the median number of gears the cars have is 6 and the median age is 1.

Performing statistical analysis with NumPy

We can also get the same statistics with the NumPy library.

In [21]:
```
statistics = {} # initialise a dictionary

for col in df_full.columns:
    x = df_full[col]
    mean = np.mean(x)
    std = np.std(x)
    min_value = np.min(x)
    max_value = np.max(x)
    percent_25 = np.percentile(x, 25)
    percent_50 = np.percentile(x, 50)
    percent_75 = np.percentile(x, 75)
    statistics[col] = {
        'mean': mean,
        'std': std,
        'min': min_value,
        'max': max_value,
        '25%': percent_25,
        '50%': percent_50,
```

```
        '75%': percent_75
    }

pd.DataFrame.from_dict(statistics)
```

Out[21]:

	price	km	gears	age	previous_owners	hp_kw	inspection_new
mean	17774.06	31233.04	5.93	1.44	1.07	87.54	0.26
std	6723.51	32451.63	0.71	1.09	0.26	24.77	0.44
min	4990.00	1.00	5.00	0.00	1.00	40.00	0.00
max	39990.00	143098.00	8.00	3.00	2.00	168.00	1.00
25%	12971.00	5000.00	5.00	1.00	1.00	66.00	0.00
50%	16900.00	21724.50	6.00	1.00	1.00	85.00	0.00
75%	21500.00	45800.00	6.00	2.00	1.00	100.00	1.00

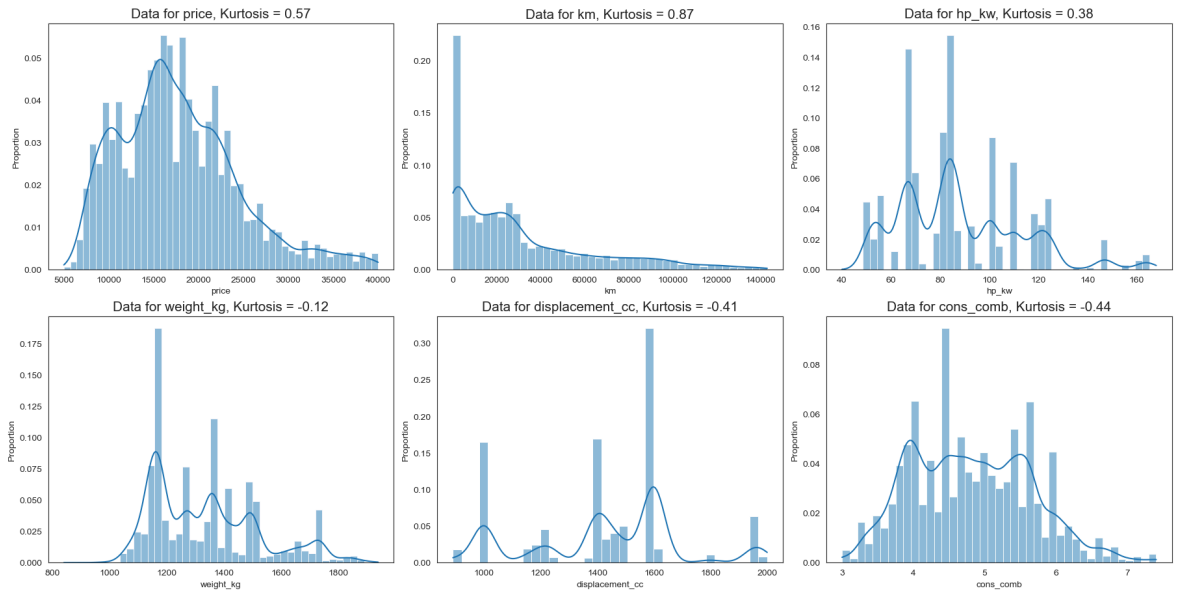
4.2 Skewness of numerical (ratio) features

In the code block below, we will plot the histogram for all of the features, and the label. This will reveal to us the skewness and kurtosis of the different data.

In [22]:
```
fig, axs = plt.subplots(2,3, layout = "constrained",figsize=(18,9))

columns = ['price','km','hp_kw','weight_kg','displacement_cc','cons_comb'

for i, column in enumerate(columns):
    row = math.floor(i / 3)
    col = i%3
    ax = axs[row,col]
    sns.histplot(x = df_full[column], ax = ax, stat = 'proportion', legen
    ax.set_title(f"Data for {column}, Kurtosis = {df_full[column].kurtosi
#     ax.annotate(f"Kurtosis = {df_full[column].kurtosis().round(2)}", xy
```



For ratio numerical variables, we use a histogram to visualise the data, such that we can analyse the mean, median, mode and distribution.

Distribution of data

We can see that the distribution of both price and km (mileage of the car) are positively (right) skewed as the tails on the right side are longer. This implies that the mean is greater than the median, which is greater than the mode for both of these features.

As such, we can deduce that most of the prices are cheaper, while a small percentage of cars sell for a high price. In addition, most of the cars have low mileage, with a few cars being heavily used.

The distribution for hp_kw, weight_kg, displacement_cc and cons_comb are comparatively more uniform, with a few spikes in values. For weigt_kg, it is apparent that there are significantly more cars that are around 1200kg the weight of small cars and sedans.

Kurtosis of data

As seen from the annotated plot above, all of the features have platykurtic distributions (with kurtosis value of less than 3). This corroborates with the plot above. There are fewer data points in the tails of the distribution, which indicates that these features have lower probabilities of extreme values.

4.3 Mode of ordinal variables

In the code below, we plot the countplot for ordinal variables. This will show us the mode and frequency of each of the features.

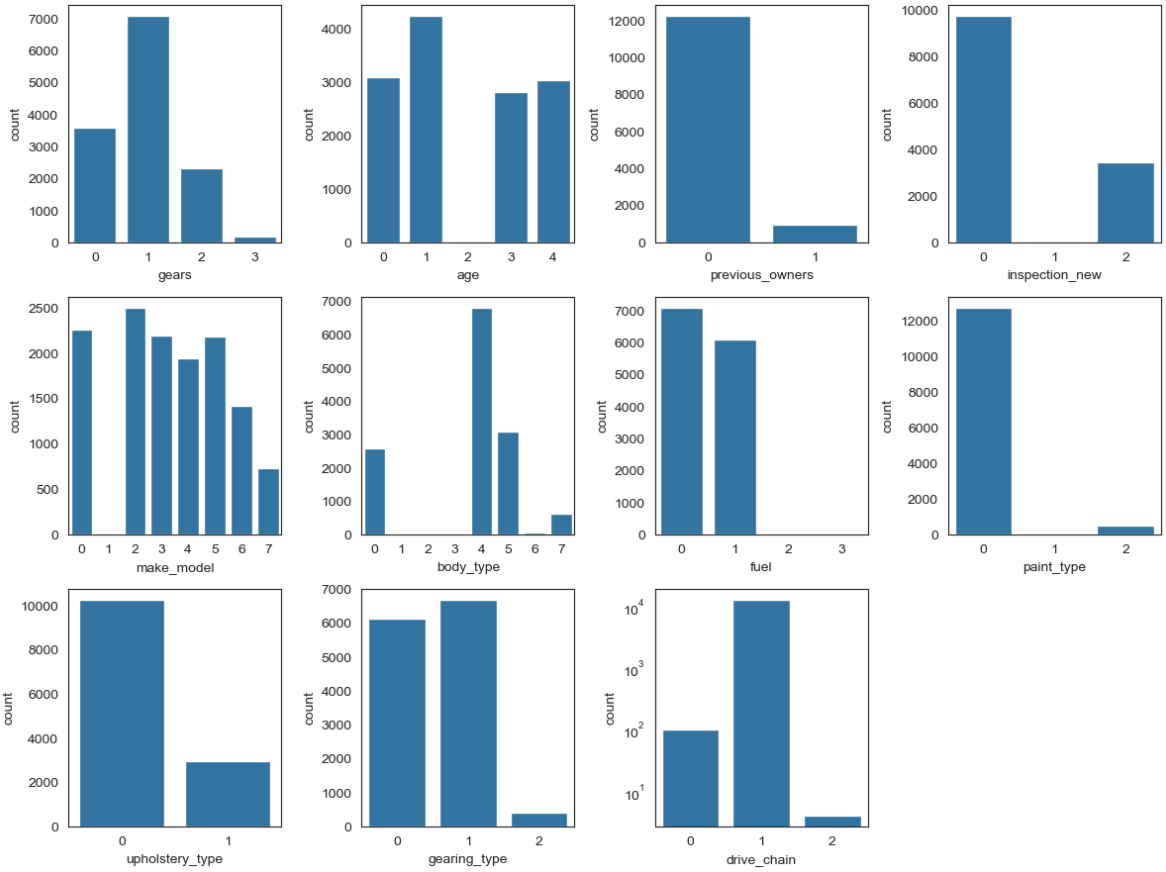
In [23]:
```
fig, axs = plt.subplots(3,4, layout = "constrained",figsize=(12,9))

columns = ['gears','age','previous_owners','inspection_new','make_model',
           'fuel','paint_type','upholstery_type','gearing_type','drive_ch

for i, column in enumerate(columns):
    row = math.floor(i / 4)
    col = i%4
    ax = axs[row,col]
    sns.countplot(x = df_full[column], ax = ax, legend = True)
    ax.xaxis.set_major_formatter(FormatStrFormatter('%.0f'))


fig.delaxes(axs[2,3])
plt.yscale('log')
plt.tight_layout()
```

```
/var/folders/pc/fblgrjpn2ng04364nyk3lgjc0000gn/T/ipykernel_83253/155182359
1.py:16: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



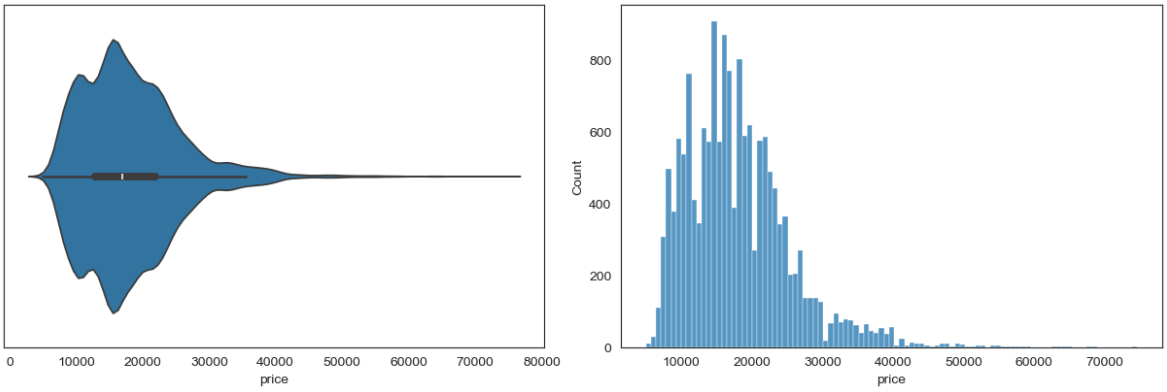
As seen from the graph above, a higher percentage of cars are manual cars (mapped to 1). More than 4000 cars are of 1 years of age. There is significantly more sedans (mapped to 1), with nearly 7,000 out of 13,000 cars. Most cars have a metallic paint type and a cloth internal upholstery, which is standard for commercial cars.

4.4 Analysis of label

In this section, we will take a closer look at the price of cars, which will be the label that we try to predict with our linear regression model.

In [24]:
```
fig,axs = plt.subplots(1,2, layout = "constrained", figsize = (12,4))
sns.violinplot(df['price'], orient = 'h', ax = axs[0])
sns.histplot(df['price'], ax = axs[1])
```

Out[24]:
```
<Axes: xlabel='price', ylabel='Count'>
```



The violin plot and boxen plot reviews similar information. The median price of cars is slightly less than 20,000 dollars. The distribution of price is positively skewed. The kernel density estimation is wider around the median, and shows a high probability of cars selling for slightly less than $20,000. There is a smaller probability of cars selling for more than 40,000 dollars, with outliers of pricier cars selling for 80,000 dollars.

5. Visualisation

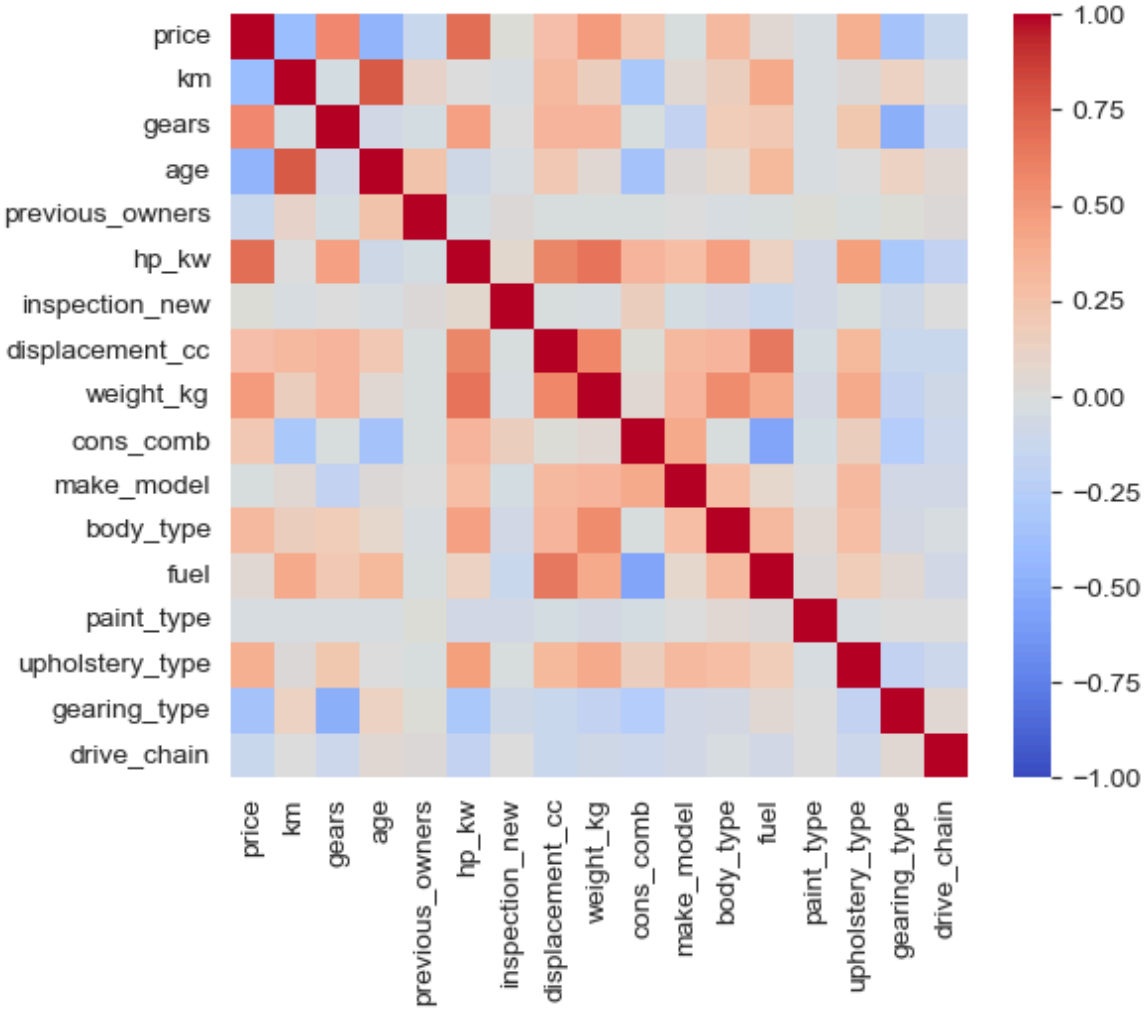
In the previous section, we performed statistical analysis and used Matplotlib and Seaborn to plot and understand the data. In this section, we will expand on that with bivariate analysis and visualisation.

5.1 Correlation matrix between features and the Label

In order to understand how the features affect the price, we can create a correlation matrix and plot it with a heatmap. This allows us to get a visual representation of how closely related the features are to the price.

In [25]:
```
fig, _ = plt.subplots(figsize=(6,5))
corr_matrix = df_full.corr()
sns.heatmap(corr_matrix, cmap='coolwarm', vmin=-1, vmax=1)
```

Out[25]:
```
<Axes: >
```



If we focus on the first column, we can see the correlation between the features and price.

As seen above, hp_kw, gears and weight have a high positive correlation with price. A heavier weighted vehicle would be larger and hence, pricier.

In contrast, km, age and gearing_type have a negative correlation with price. An older vehicle would sell for a lower price.

5.2 Top correlated features

In order to get the top correlated features, we find the absolute correlation coefficients against price for all the features.

In [26]:
```
# to get the values
sorted_corr_matrix = corr_matrix['price'].abs().sort_values(axis=0, ascen
top_features = sorted_corr_matrix.nlargest(n=11)
```

In [27]:
```
print(top_features)
```

```
price              1.00
hp_kw              0.69
gears              0.56
weight_kg          0.48
age                0.47
km                 0.40
upholstery_type    0.38
gearing_type       0.34
body_type          0.30
displacement_cc    0.26
cons_comb          0.21
Name: price, dtype: float64
```

As seen above, we can see that price has a correlation coefficient of 1 with itself.

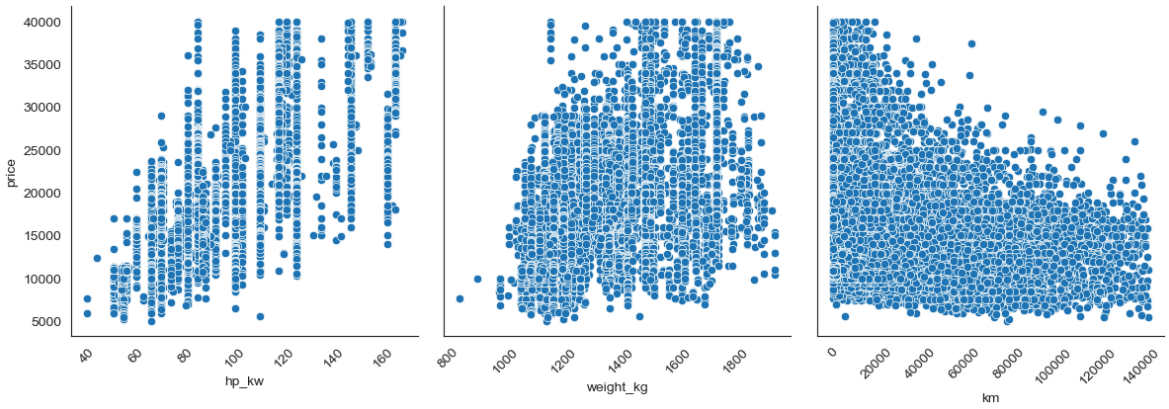
The feature that has the most correlation with price is hp_kW, which represents a car's engine power in kilowatts. Since this is a major performance metric of a car's engine, it makes sense that it has the most impact on the price of a car.

In the code below, we will plot the relationship between the top three features and the label.

Top features (Ratio data type)

In [28]:
```
g = sns.PairGrid(df_full, y_vars = 'price', x_vars = ['hp_kw','weight_kg'
g.map(sns.scatterplot)
g.tick_params(axis = 'x', rotation = 45) # rotate so that the figures are
```

Out[28]:
```
<seaborn.axisgrid.PairGrid at 0x15479ecf0>
```

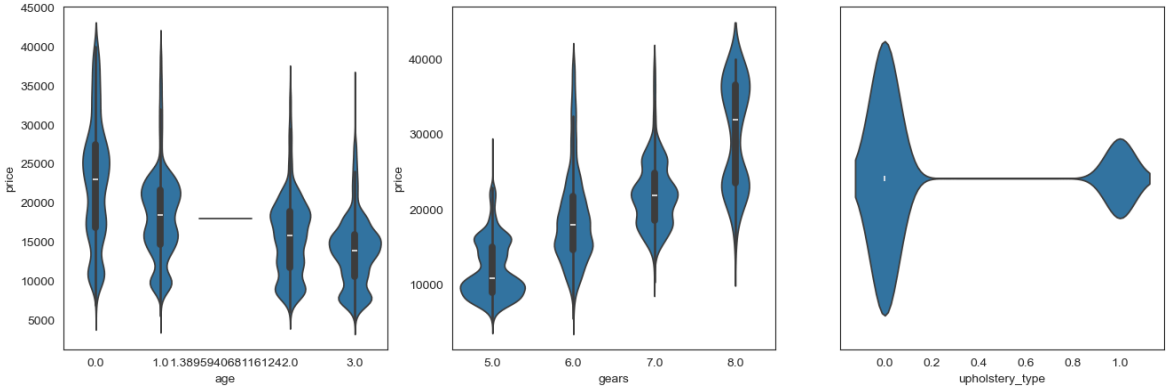


From the plots above, we can see a positive relationship between hp_kw and price. There is a negative relationship between price and km. However, the relationship between price and weight is not as prominent. The correlation coefficient between price and weight is 0.484, which indicates a moderate positive and weak correlation.

Top features (Ordinal data type)

In [29]:
```
fig, axs = plt.subplots(ncols = 3, figsize = (16,5))
sns.violinplot(data = df_full, y = 'price', x = 'age', ax = axs[0])
sns.violinplot(data = df_full, x = 'gears', y = 'price', ax = axs[1])
sns.violinplot(data = df_full, x = 'upholstery_type', ax = axs[2])
```

Out[29]: <Axes: xlabel='upholstery_type'>



From the data above, we can see that there is a slight negative relationship between price and age. The mean price that a 3-years-old car sells for is lower than that of a new (0-years-old) car. There is a slight positive relationship between the number of gears and the price - a higher number of gears would increase manufacturing cost and improved fuel efficiency. There is no strong relationship between price and the upholstery price, with most cars having a cloth interior.

6. Building the Machine Learning Model

6.1 Identifying features and labels for data regression model

We will be using **16 features** and **1 label** for the linear regression model.

The features selected for the linear regression model are listed below. These features show low multicollineraity (covered in section 3.6). Highly correlated features like Value-added tax deduction and the type of the car have been removed to simplify the model. Other categorical features with thousands of unique values have also been removed to avoid unnecessary noise into the model.

The justification of why each feature is selected is seen as below:

1. Make Model : The model and brand of a car is important in determining the price. Luxury brands will fetch a significantly higher price than their more economical counterparts.

2. Body Type : The body type (eg. sedan, station wagon) reveals information about the cargo space, performance and intended use case of the car. These factors will have a high impact on the price of the car

3. Type of fuel : The type of fuel used by a car can determine whether it is cost-efficient to own the car in the long-run. This affects the price of the car.

4. Paint type : It costs more to have a metallic paint finish.

5. Interior upholstery type : It costs more to have a leader upholstery.

6. Transmission (gearing) type : Manual cars are typically cheaper than automatic cars.

7. Drivetrain : A front-wheel drive is typically lesss expensive than a rear-wheel drive.

8. Number of gears : The number of gears determine whether the car is more fuel efficient and whether it can achieve a higher top speed.

9. Age : An older car will have a lower resale price.

10. Previous owners : A second-hand car will have a lower resale price.

11. Inspection status : A car that has not been inspected recently will sell for lower, as the buyer has to fork out an additional lump of money to have it inspected separately.

12. Total mileage (km) : The total mileage reveals how heavily used the car is. A heavily used car will yield a lower price.

13. Engine power (kW) : A higher engine power signifies higher performance, which will increase the retail price of the car.

14. Engine displacement : More powerful cars have larger engines and are more expensive.

15. Weight (kg) : A car's weight affects fuel consumption and shipping costs.

16. Combined fuel consumption (litres/100km) : A car's fuel consumption and cost-efficiency will weigh in the car's retail price

The target label is the listed price of the cars. This is the value that we would like to predict with our model.

In [30]:
```
"""
Importing libraries for Machine learning
"""
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, KFold, GridSearchCV
from sklearn.feature_selection import RFE
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_p
```

```
# getting the features and target variable
y = df_full['price']
X = df_full.drop('price', axis = 1, inplace = False)
```

6.2 Naive Baseline

In order to judge the performance of the linear regression model, we can use a baseline as a benchmark for comparison. In this project, the baseline is a simple prediction using the mean of the target variable (price of the cars).

To get the average price of cars, we use the mean Numpy method on the target values. We then use the numpy full method to create a array with the same number of rows. Using Mean Squared Error and R-squared score, we can calculate the baseline the our linear regression model. The models that we create should perform better than these benchmark values.

The metrics used for this project to measure the accuracy of model prediction are:

1. Mean-squared Error: Mean squared error (MSE) measures the average squared difference between the observed and predicted values. The larger the value of the MSE, the larger the average squared residual. Hence, a more precise model will have a lower MSE. [14]

2. R-squared: R-squared measures the square of the correlation coefficient between the observed and predicted values. The higher the R-squared value, the greater proportion of the variance in the dependent variable is explained. Hence, a model with a better fit will generally have a higher R-squared value.[15]

3. Mean Absolute Percentage Error: Mean Absolute Percentage Error (MAPE) measures the average absolute percentage difference between the observed and predicted values. The larger the value of the MSE, the larger the average percentage error. Hence, a more accurate model will have a lower MAPE. [16]

4. Explained Variance Score: The Explained Variance Score (EVS) measures how much of the variability in the target variable can be explained by the model's predictions. A EVS of 1 indicates a perfect fit.

In [31]:
```
baseline = np.mean(y)
baseline_pred = np.full(len(y), baseline)

baseline_mse = mean_squared_error(y, baseline_pred)
baseline_r2 = r2_score(y, baseline_pred)
baseline_mape = mean_absolute_percentage_error(y, baseline_pred)
baseline_evs = explained_variance_score(y, baseline_pred)

print(f"Baseline MSE: {baseline_mse:,.0f}")
print(f"Baseline R-squared: {baseline_r2:,.5f}")
print(f"MAPE: {baseline_mape:,.5f}" )
print(f"Baseline Variance: {baseline_evs:,.5f}")
```

```
Baseline MSE: 45,205,643
Baseline R-squared: 0.00000
MAPE: 0.35221
Baseline Variance: 0.00000
```

As seen from the code above, the baseline values are:

- The baseline MSE is 45,205,643.
- R-squared: 0
- MAPE: 0.35221
- Explained Variance score: 0

6.3 Linear Regression

In [32]:

```
# split the data set into training and testing data, with 30% for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,

# instantiate the linear regression model from Scikit Learn
lr = LinearRegression()

# fit the model with the training data
lr.fit(X_train, y_train)

# model's performance on training data
train_predictions = lr.predict(X_train)

# predict with the model
predictions = lr.predict(X_test)

# get the metrics for the training data
train_mse = mean_squared_error(y_train, train_predictions)
train_r2 = r2_score(y_train, train_predictions)
train_mape = mean_absolute_percentage_error(y_train, train_predictions)
train_evs = explained_variance_score(y_train, train_predictions)


# get the metrics for the testing data
mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)
mape = mean_absolute_percentage_error(y_test, predictions)
evs = explained_variance_score(y_test, predictions)

print("|----------------------  ON TRAINING DATA  ----------------------|\n

print(f"Mean Squared Error on train: {train_mse:,.2f}")
print(f"R^2 Score on train: {train_r2:,.5f}")
print(f"Mean Absolute Percentage Error on train: {train_mape:,.5f}")
print(f"Explained Variance Score on train: {train_evs:,.5f}\n")

print("|----------------------  ON TESTING DATA  ----------------------|\

print(f"Mean Squared Error: {mse :,.2f}")
print(f"R^2 Score: {r2:,.5f}")
print(f"Mean Absolute Percentage Error: {mape:,.5f}")
print(f"Explained Variance Score: {evs:,.5f}")
```

```
|----------------------  ON TRAINING DATA  ----------------------|

Mean Squared Error on train: 9,965,314.93
R^2 Score on train: 0.78202
Mean Absolute Percentage Error on train: 0.14443
Explained Variance Score on train: 0.78202

|----------------------  ON TESTING DATA  ----------------------|

Mean Squared Error: 9,732,576.81
R^2 Score: 0.77887
Mean Absolute Percentage Error: 0.14491
Explained Variance Score: 0.77903
```

Given the Mean Squared Error and the R2 score, we can test if it performs better than baseline.

In [33]:
```
mse < baseline_mse, r2 > baseline_r2, mape < baseline_mape, evs > baselin
```

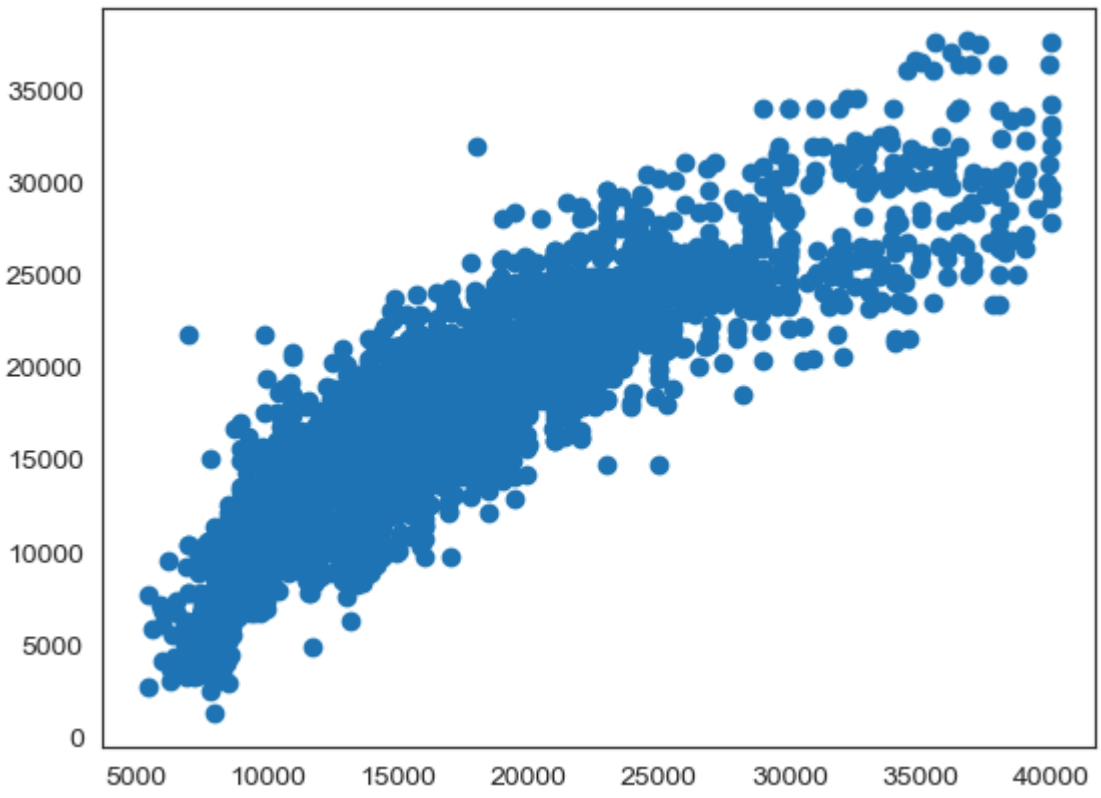
Out[33]:
```
(True, True, True, True)
```

As seen above, the linear regression model performs better than baseline when using all four metrics.

Visualising the linear regression model

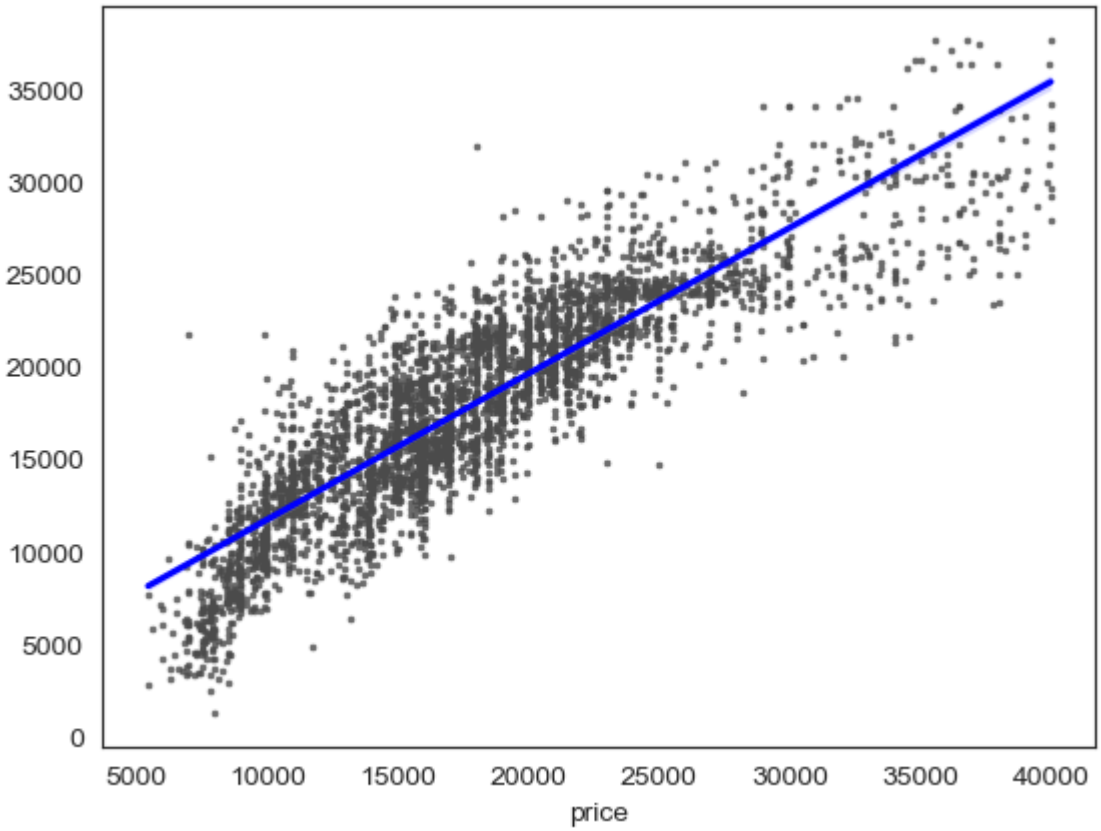
In [34]:
```
plt.scatter(y_test, predictions)
```

Out[34]:
```
<matplotlib.collections.PathCollection at 0x1553304d0>
```



In [35]:
```
sns.regplot(x = y_test, y = predictions, color = '.3', marker = 'x',
            scatter_kws={'s':3},
            line_kws=dict(color="b"))
```

Out[35]: <Axes: xlabel='price'>



The regression plot above shows that the regression model (depicted as the blue line) is a good fit, capturing the general trend of the observed values.

7. Validation

In order to validate the model, we will use K-Fold and GridSearchCV class from Scikit learn's Model Selection library.

K-Fold splits the dataset into k-consecutive fold. Each fold is then used once as validation, while the remaining k - 1 folds are used as the training set. [18]

GridSearchCV performs an exhaustive search over parameter values we specify for an estimator. The RFE class from Scikit Learn's Feature Selection library will be used for this project. RFE provides an estimator that assigns weights to features, with the goal of selecting features by recursively considering smaller sets of features. In this manner, least important features are removed from current set of features. The desired number of features will then be returned.

In [36]:
```
k_folds = KFold(n_splits = 5, random_state = 42, shuffle = True)

params = [{'n_features_to_select': list(range(1,17))}]
```

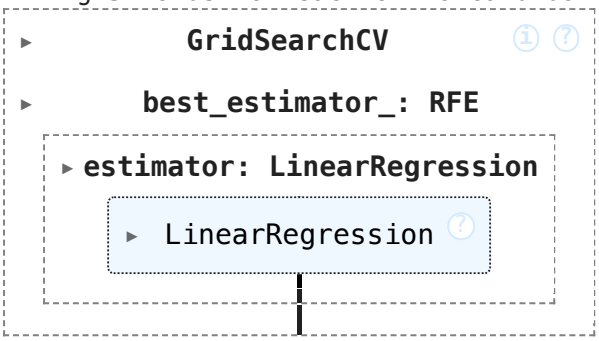
```
rfe = RFE(lr)

# GridSearch
cv = GridSearchCV(estimator = rfe, param_grid = params, scoring = 'r2', c

cv.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
```

Out[36]:



In [37]:
```
cv_results = pd.DataFrame(cv.cv_results_)
cv_results.tail(3)
```

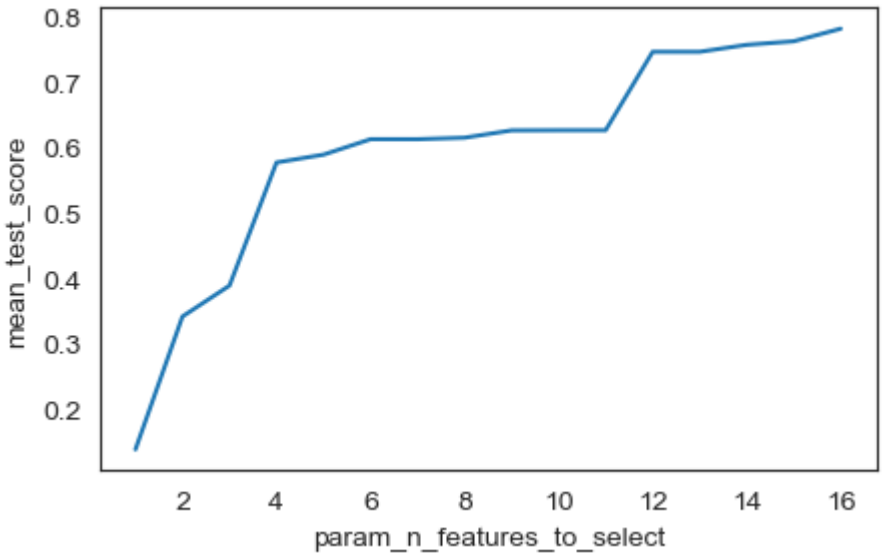
Out[37]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_featu
13	0.01	0.00	0.00	0.00	
14	0.00	0.00	0.00	0.00	
15	0.00	0.00	0.00	0.00	

3 rows × 21 columns

In the DataFrame above, we can see the results of using 14 to 16 features. We see a significant improvement with an increase in the number of features, with a R2 score of 0.757 with 14 features to a R2 score of 0.782 with 16 features. We can visualise the impact the number of features have on the r2 score with a line plot.

In [38]:
```
ax = plt.figure(figsize=(5,3))
ax = sns.lineplot(x = cv_results["param_n_features_to_select"], y = cv_re
```



As seen from the plot, we can see that the best number of features to use for the model is 16. Increasing the number of features from 2 to 4 see a drastic improvement of the model. Using 4 to 12 features show the same performance. The performance peaked when all of the features are used.

8. Feature Engineering

Feature engineering is the process of creating new features or transforming existing features to improve the performance of the machine learning algorithm.

As feature engineering is largely a preprocessing technique, a few feature engineering steps to transform raw data into meaningful features has been covered in the pre-processing section of the project. In Section 3.5, we employed one-hot encoding tecniques to create numerical features from categorical variables. In Section 3.6, we explored the multicollinearity of the dataset and removed variables that cause high multicollinearity.

In the following section, we will continue the feature engineering process with feature scaling and using polynomial features.

8.1 Feature scaling

In [39]:
```
scaler = StandardScaler()

# fit on training data
scaler.fit(X_train)

# transform both training and testing
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# instantiate the linear regression model from Scikit Learn
lr_scaled = LinearRegression()
```

```
# fit the model with the training data
lr_scaled.fit(X_train_scaled, y_train)

# predict with the model
predictions_scaled = lr_scaled.predict(X_test_scaled)

# get the mse and r2 values
mse_scaled = mean_squared_error(y_test, predictions_scaled)
r2_scaled = r2_score(y_test, predictions_scaled)
mape_scaled = mean_absolute_percentage_error(y_test, predictions_scaled)
evs_scaled = explained_variance_score(y_test, predictions_scaled)

print(f"Mean Squared Error on Scaled data: {mse_scaled:,.2f}")
print(f"R^2 Score on Scaled data: {r2_scaled:,.5f}")
print(f"Mean Absolute Percentage Error: {mape_scaled:,.5f}")
print(f"Explained Variance Score: {evs_scaled:,.5f}")
```

```
Mean Squared Error on Scaled data: 9,732,576.81
R^2 Score on Scaled data: 0.77887
Mean Absolute Percentage Error: 0.14491
Explained Variance Score: 0.77903
```

All four metrics were not affected by feature scaling. This implies that scaling did not impact the model's ability to understand the underlying relations in the data. In addition, linear regression models are less sensitive to feature scaling, as the model learns the optimal weights for each feature and adjusts.

8.2 Polymomial Features

Polynomial features, as feature engineering, involves the creation of new input features based on the existing features. Adding polynomial terms to linear regression can allow the regression model to identify non-linear patterns. [17]

In the code below, we use Scikit Learn's PolynomialFeatures class to generate the polynomial terms from the features we have originally.

In [40]:
```
dict_scores = {}

for degree in range(1,4):
    poly = PolynomialFeatures(degree = degree)
    Xp_train = poly.fit_transform(X_train)
    Xp_test = poly.fit_transform(X_test)
    num_features = Xp_train.shape[1]
    lr = LinearRegression()
    lr.fit(Xp_train, y_train)
    predictions = lr.predict(Xp_test)

    mse = mean_squared_error(y_test, predictions)
    r2 = r2_score(y_test, predictions)
    mape = mean_absolute_percentage_error(y_test, predictions)
    evs = explained_variance_score(y_test, predictions)

    dict_scores[degree] = {'degree': degree,
                           'num_features':num_features,
                           'mse':mse,
                           'r2':r2,
```

```
                                'mape':mape,
                                'evs':evs}
```

In [41]:
```
pd.options.display.float_format = '{:,.3f}'.format
polynomial_features_scores = pd.DataFrame.from_dict(dict_scores)
polynomial_features_scores
```

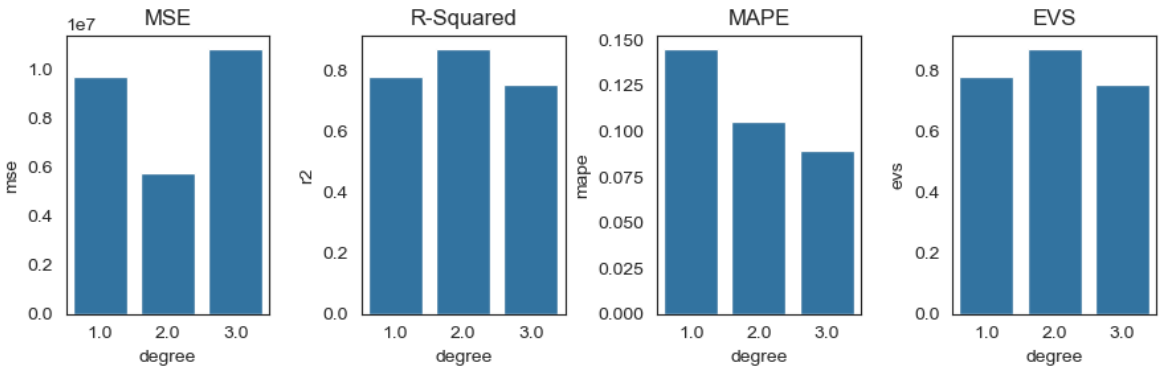
Out[41]:

	1	2	3
degree	1.000	2.000	3.000
num_features	17.000	153.000	969.000
mse	9,732,576.812	5,791,087.980	10,826,039.578
r2	0.779	0.868	0.754
mape	0.145	0.106	0.089
evs	0.779	0.869	0.754

The degree of the polynomial increases the number of input features. From the pandas dataframe above, we can see that a polynomial order of 2 increases the number of features from the original 16 to 153. This gives us a better MSE, R2 and EVS score.

In [42]:
```
dfl = polynomial_features_scores.transpose()
fig, axs = plt.subplots(ncols = 4, layout = "tight",figsize=(9,3))

sns.barplot(x = dfl['degree'], y = dfl['mse'], ax = axs[0])
axs[0].set_title('MSE')
sns.barplot(x = dfl['degree'], y = dfl['r2'], ax = axs[1])
axs[1].set_title('R-Squared')
sns.barplot(x = dfl['degree'], y = dfl['mape'], ax = axs[2])
axs[2].set_title('MAPE')
sns.barplot(x = dfl['degree'], y = dfl['evs'], ax = axs[3])
axs[3].set_title('EVS')
plt.show()
```



PolynomialFeature and improvemnet on metrics

We can see an improvement in MSE, R-Squared and EVS when we go from 1 to 2 order of polynomial. However, when we introduce the 3rd order, the model worsens on these three metrics.

Mean Absolute Percentage Error (MAPE) improved even though the overall model fit worsens from degree 2 to degree 3. This may be because MAPE reduces when the model overfits - it becomes very accurate at the expense of generalisation.

8.3 Regularisation

Regularisation is a technique that helps models generalise and reduce overfitting. In the following section, we will use Ridge Regressor where regularisation is given by the L2-norm. A penalty term is added to the loss function to discourage the model from assigning large weights to features.

In [43]:
```
from sklearn.linear_model import RidgeCV, Ridge
poly = PolynomialFeatures(degree = 2)
Xp_train = poly.fit_transform(X_train)
Xp_test = poly.fit_transform(X_test)

ridge = Ridge(alpha = 2)
ridge.fit(Xp_train, y_train)
predictions = ridge.predict(Xp_test)
mse_regularised = mean_squared_error(y_test, predictions)
r2_regularised = r2_score(y_test, predictions)
mape_regularised = mean_absolute_percentage_error(y_test, predictions)
evs_regularised = explained_variance_score(y_test, predictions)

print(f"Mean Squared Error: {mse_regularised :,.2f}")
print(f"R^2 Score: {r2_regularised:,.5f}")
print(f"Mean Absolute Percentage Error: {mape_regularised:,.5f}")
print(f"Explained Variance Score: {evs_regularised:,.5f}")
```

```
Mean Squared Error: 5,722,374.55
R^2 Score: 0.86998
Mean Absolute Percentage Error: 0.10512
Explained Variance Score: 0.87013
/opt/anaconda3/lib/python3.12/site-packages/sklearn/linear_model/_ridge.p
y:216: LinAlgWarning: Ill-conditioned matrix (rcond=1.25781e-23): result m
ay not be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
```

Performance of the linear regression model improved with regularisation. By preventing overfitting, the model can generalise better to new data. This improves performance on the test data. Regularisation also encourages the model to select the most important features. In the code above, we used PolynomialFeatures with a degree of 2 to generate a polynomial combination of the original features. This produces a new matrix with 153 features. By using regularisation, we can create a simpler model.

10. Evaluation of model

In the code block below, we summarise the performance of the model after validation and feature engineering. To evaluate the performance of the linear prediction in

earlier sections of the project, we used MSE, R-Squared, MAPE and EVS. The code below shows a summary of the metrics.

In [44]:
```
# Summary, elaboration below.
summary_dict = {
    'baseline': {
        'MSE': baseline_mse,
        'R-Squared' : baseline_r2,
        'MAPE': baseline_mape,
        'EVS': evs
    },
    'logistic_regression': {
        'MSE': mse,
        'R-Squared' : r2,
        'MAPE': mape,
        'EVS': evs
    },
    'logistic_regression_scaled': {
        'MSE': mse_scaled,
        'R-Squared' : r2_scaled,
        'MAPE': mape_scaled,
        'EVS': evs_scaled
    },
    'logistic_regression_poly2': {
        'MSE': dfl.iloc[1].mse,
        'R-Squared' : dfl.iloc[1].r2,
        'MAPE': dfl.iloc[1].mape,
        'EVS': dfl.iloc[1].evs
    },
    'logistic_regression_regularised': {
        'MSE': mse_regularised,
        'R-Squared' : r2_regularised,
        'MAPE': mape_regularised,
        'EVS': evs_regularised
    }
}

pd.DataFrame.from_dict(summary_dict)
```

Out[44]:

	baseline	logistic_regression	logistic_regression_scaled	logistic_re
MSE	45,205,643.025	10,826,039.578	9,732,576.812	
R-Squared	0.000	0.754	0.779	
MAPE	0.352	0.089	0.145	
EVS	0.754	0.754	0.779	

Justification of metrics

- MSE is used to provide a measure of overal model accuracy.
- R-squared is used to provide a measure of how well the model fits the data.
- MAPE is used as it is less sensitive to outliers compared to MSE.
- EVS is similarly to R-squared, to provide an overall measure of model fit.

Results
The utilisation of four distinct metrics provides a comprehensive assessment of the model.

The model performed better with scaling, the use of polynomial features to a degree of 2, and regularisation.

The final performance of the model (rightmost column) has a MSE of 5,722,374, which is the squared difference of the predictions and observed values. To get the RSME, we can take a square root of this figure - and get 2,382. This means that on average, the model's prediction is off by 2,382 dollars.

The model has a R-squared of 0.87. This entails that the model is able to explain 87% of the variance in the prices. As this is a high R-squared value, it suggests that the model can capture underlying patterns in the data. The MAPE is 0.105, which shows us that the model deviate from actual values by 10.5%. The EVS of 0.84, like the R-squared score, indicates a strong fit of the model.

The model performs significantly better than baseline, which suggests that it has learned useful and meaningful patterns to predict the prices of cars. A RSME of 2,382 dollars is acceptable, given that this model is intended for consumer-facing purposes and predicts a rough estimate of car prices.

Functionality
The model demonstrates functionality, as evidenced by the R-squared value and Mean Squared Error. Although some inaccuracies are present, they fall well within acceptable limits for non-commercial predictive purposes. These results indicate that while the model is not flawless, its performance is reliable and suitable for its intended application in a non-commercial context.

Contribution
There are no major commercial car prediction applications or websites. As such, a car prediction algorithm such as the one created in this project will greatly aid consumers, who are often at losing end of car deals. By comparing the predicted price with the asking price at car dealorships, consumers can determine if they are given a fair price and better negotiate with dealers. By allowing consumers to make informed decisions, we can create a more transparent view in the car market.

Transferability
This solution is transferable to other commercial domains such as predicting the price of a house based on features like the number of bedrooms,neighbourhoods and square footage.

Conclusion

In conclusion, this project served as a foundational framework for implementing a linear regression model to predict car prices. Future expansion can include using

real-time data or a larger historical dataset to fit the model.

References

[1] M. Carlier, “Statista,” 27 November 2024. [Online]. Available: https://www.statista.com/topics/1487/automotive-industry/#topicOverview. [Accessed 9 December 2024].

[2] W. Kim, “Vox,” Is every car dealer trying to rip me off?, 16 October 2024. [Online]. Available: https://www.vox.com/explain-it-to-me/374186/car-dealerships-sales-tactics-shady-practices. [Accessed 9 December 2024].

[3] Y. Y. Turan, “Auto Scout Car Price,” 2024. [Online]. Available: https://www.kaggle.com/datasets/yaaryiitturan/auto-scout-car-price. [Accessed 9 December 2024].

[4] Creative Commons, “Attribution-ShareAlike 4.0 International,” 2024. [Online]. Available: https://creativecommons.org/licenses/by-sa/4.0/. [Accessed 11 December 2024].

[5] A. Baby, “How to Handle Null Values in Machine Learning: A Complete Guide,” 26 September 2024. [Online]. Available: https://medium.com/tech-tensorflow/how-to-handle-null-values-in-machine-learning-a-complete-guide-c563a05c8e23#:~:text=Null%20values%20can%20be%20problematic,can%20give%20 [Accessed 12 December 2024].

[6] J. Brownlee, “Machine Learning Mastery,” 17 August 2020. [Online]. Available: https://machinelearningmastery.com/knn-imputation-for-missing-values-in-machine-learning/. [Accessed 12 December 2024].

[7] M. Thoma and H. Shteingart, “How do I find numeric columns in Pandas?,” 18 October 2017. [Online]. Available: https://stackoverflow.com/questions/25039626/how-do-i-find-numeric-columns-in-pandas. [Accessed 9 December 2024].

[8] Pandas, “pandas.DataFrame.select_dtypes,” 2024. [Online]. Available: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.select_dtypes.html. [Accessed 9 December 2024]

[9] Geeks for Geeks, “Using KNNImputer in Scikit-Learn to Handle Missing Data in Python,” 9 August 2024. [Online]. Available: https://www.geeksforgeeks.org/python-imputation-using-the-knnimputer/. [Accessed 10 December 2024]

[10] Anishnama, “How Duplicate entries in data set leads to ovetfitting?,” 10 July 2023. [Online]. Available: https://medium.com/@anishnama20/how-duplicate-entries-in-data-set-leads-to-ovetfitting-2e3376e309c5#:~:text=Overfitting%3A%20Duplicate%20entries%20may%20artificially [Accessed 10 December 2024].

[11] Saturn Cloud, “Linear Regression with sklearn using categorical variables,” 15 December 2023. [Online]. Available: https://saturncloud.io/blog/linear-regression-with-sklearn-using-categorical-variables/. [Accessed 9 December 2024].

[12] Scikit Learn, “LabelEncoder,” 2024. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.preprocessing.LabelEncoder.html. [Accessed 11 December 2024].

[13] A. Bhandari, “What is Multicollinearity? Understand Causes, Effects and Detection Using VIF,” 2024. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/03/what-is-multicollinearity/. [Accessed 12 December 2024].

[14] U. Riswanto, “Why Feature Selection is Critical for Machine Learning,” 26 September 2024. [Online]. Available: https://ujangriswanto08.medium.com/why-feature-selection-is-critical-for-machine-learning-3913fffd62c0#:~:text=The%20problem%3A%20With%20too%20many,training%2C%20 [Accessed 12 December 2024].

[15] J. Frost, “Mean Squared Error (MSE),” 2024. [Online]. Available: https://statisticsbyjim.com/regression/mean-squared-error-mse/. [Accessed 14 December 2024].

[16] S. Ahmed, “ML Series 5: Understanding R-squared in Regression Analysis,” 14 February 2024. [Online]. Available: https://medium.com/@sahin.samia/understanding-r-squared-in-regression-analysis-2d8246a63dbb. [Accessed 14 December 2024].

[17] A. Roberts, “Mean Absolute Percentage Error (MAPE): What You Need To Know,” 2 February 2023. [Online]. Available: https://arize.com/blog-course/mean-absolute-percentage-error-mape-what-you-need-to-know/. [Accessed 16 December 2024].

[18] J. Brownlee, “How to Use Polynomial Feature Transforms for Machine Learning,” 28 August 2020. [Online]. Available: https://machinelearningmastery.com/polynomial-features-transforms-for-machine-learning/. [Accessed 16 December 2024].

[19] Scikit Learn, “K-Fold,” 2024. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.model_selection.KFold.html. [Accessed 14 December 2024].

[20] Scikit Learn, “GridSearchCV,” 2024. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.model_selection.GridSearchCV.html. [Accessed 14 December 2024].

[21] Scikit Learn, “RFE,” 2024. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.feature_selection.RFE.html. [Accessed 14 December 2024].

[22] J. Murel, “What is feature engineering?,” 20 January 2024. [Online]. Available: https://www.ibm.com/think/topics/feature-engineering. [Accessed 16 December 2024].

[23] Tacaswell, “Specify format of floats for tick labels,” 21 March 2015. [Online]. Available: https://stackoverflow.com/questions/29188757/specify-format-of-floats-for-tick-labels. [Accessed 15 December 2024].